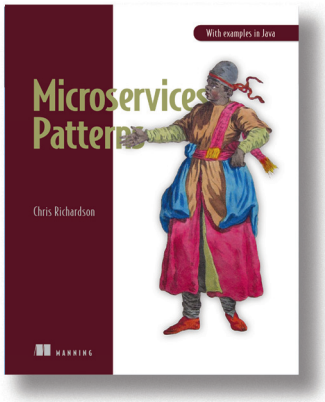
Author Picks

FREE

Microservices Stability: Design For Failure, Deploy With Confidence

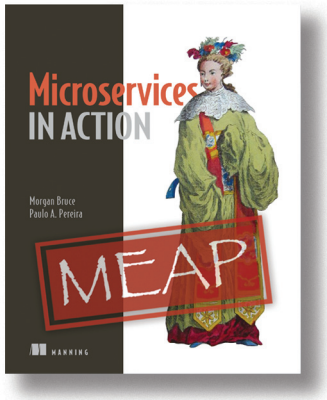
Chapters selected by Paulo Pereira

manning



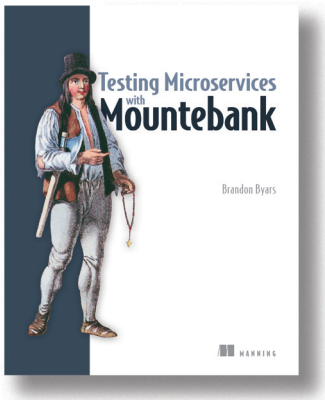
Microservices Patterns
with Examples in Java
by Chris Richardson

ISBN 9781617294549
520 pages
$39.99
October 2018



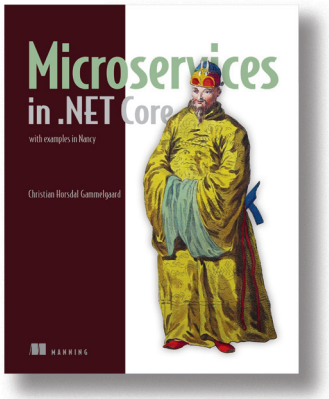
Microservices in Action
by Morgan Bruce, and Paulo A. Pereira

ISBN 9781617294457
392 pages
$39.99
October 2018



Testing Microservices
with Mountebank
by Brandon Byars

ISBN 9781617294778
240 pages
$39.99
December 2018



Microservices in .NET Core
with examples in Nancy
by Christian Horsdal Gammelgaard

ISBN 9781617293375
344 pages
$39.99
January 2017

Microservices Stability: Design For Failure, Deploy With Confidence

Chapters Selected by Paulo Pereira

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Candace Gillhoolley, cagi@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617296994
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 24 23 22 21 20 19

contents

introduction

A microservice application is a collection of specialised and autonomous services working together to perform more intricate operations. This type of architecture reduces friction while developing complex systems that allow for quick time to market and experimentation. The flexibility, adaptability and lower friction of microservice approach come with the downside of increased complexity of how you deploy, verify, and observe your application in operation. Having multiple and independent components is not only challenging from a design standpoint, but also due to the fact that microservice applications are distributed systems. To understand a bit more about the challenges of distributed systems, here is a list of the fallacies of distributed systems as proposed by Peter Deutsch and James Gosling:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

We need to keep in mind calls over the network will fail; they are not instant and there is limited bandwidth. The network is also insecure and topology changes constantly, so this brings additional challenges when dealing with microservice applications.

Potential points of failure for an application increase when introducing new services—or features in the existing services, and care must be taken to make sure any defects introduced in a component don't affect the whole application.

With some care taken at the design stage and proper testing approach the deployment of microservice applications can happen with confidence, minimizing risks and enabling frictionless change.

This ebook includes chapters from three Manning books covering microservice applications: Microservices in Action, The Tao of Microservices, and Microservices in .NET Core.

Designing reliable services

Understanding the impact of individual service availability on the overall application reliability is a good motivation to design microservices that can defend against faults in their dependencies.

Chapter 6 from *Microservices in Action*
by Morgan Bruce and Paulo A. Pereira

Designing reliable services

This chapter covers

- The impact of service availability on application reliability
- Designing microservices that defend against faults in their dependencies
- Applying retries, rate limits, circuit breakers, health checks, and caching to mitigate interservice communication issues
- Applying safe communication standards across many services

No microservice is an island; each one plays a small part in a much larger system. Most services that you build will have other services that rely on them—upstream collaborators—and in turn themselves will depend on other services—downstream collaborators—to perform useful functions. For a service to reliably and consistently perform its job, it needs to be able to trust these collaborators.

This is easier said than done. Failures are inevitable in any complex system. An individual microservice might fail for a variety of reasons. Bugs can be introduced into code. Deployments can be unstable. Underlying infrastructure might let you down:

resources might be saturated by load; underlying nodes might become unhealthy; even entire data centers can fail. As we discussed in chapter 5, you can't even trust that the network between your services is reliable—believing otherwise is a well-known fallacy of distributed computing.[1] Lastly, human error can lead to major failures. For example, I'm writing this chapter a week after an engineer's mistake in running a maintenance script led to a severe outage in Amazon S3, affecting thousands of well-known websites.

It's impossible to eliminate failure in microservice applications—the cost of that would be infinite! Instead, your focus needs to be on designing microservices that are tolerant of dependency failures and able to gracefully recover from them or mitigate the impact of those failures on their own responsibilities.

In this chapter, we'll introduce the concept of service availability, discuss the impact of failure in microservice applications, and explore approaches to designing reliable communication between services. We'll also discuss two different tactics—frameworks and proxies—for ensuring all microservices in an application interact safely. Using these techniques will help you maximize the reliability of your microservice application—and keep your users happy.

6.1 Defining reliability

Let's start by figuring out how to measure the reliability of a microservice. Consider a simple microservice system: a service, *holdings*, calls two dependencies, *transactions* and *market-data*. Those services in turn call further dependencies. Figure 6.1 illustrates this relationship.

For any of those services, you can assume that they spent some time performing work successfully. This is known as *uptime*. Likewise, you can safely assume—because failure is inevitable—that they spent some time failing to complete work. This is known as *downtime*. You can use uptime and downtime to calculate *availability*: the percentage of operational time during which the service was working correctly. A service's availability is a measure of how reliable you can expect it to be.

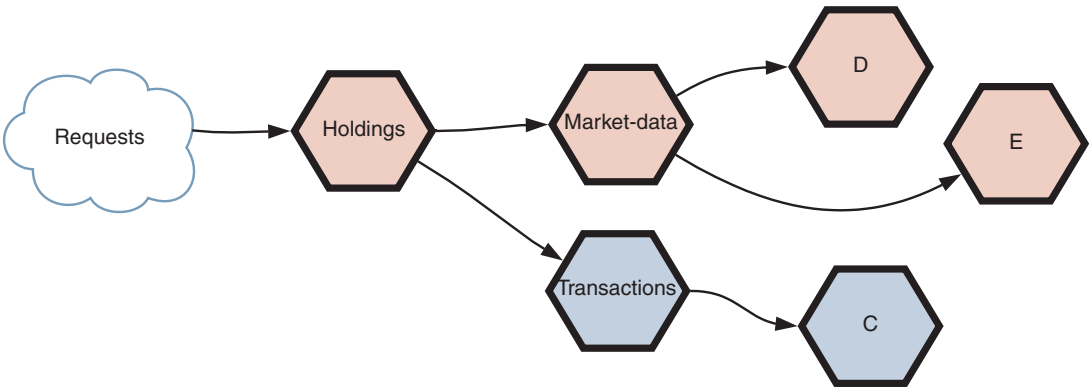


Figure 6.1 A simple microservice system, illustrating dependencies between collaborating services

[1] Peter Deutsch originally posited the eight fallacies of distributed computing in 1994. A good overview is available here: http://mng.bz/9T5F.

A typical shorthand for high availability is "nines:" for example, two nines is 99%, whereas five nines is 99.999%. It'd be highly unusual for critical production-facing services to be less reliable than this.

To illustrate how availability works, imagine that calls from holdings to market-data are successful 99.9% of the time. This might sound quite reliable, but downtime of 0.1% quickly becomes pronounced as volumes increase: only one failure per 1,000 requests, but 1,000 failures per million. These failures will directly affect your calling service unless you can design that service to mitigate the impact of dependency failure.

Microservice dependency chains can quickly become complex. If those dependencies can fail, what's the probability of failure within your whole system? You can treat your availability figure as the probability of a request being successful—by multiplying together the availability figures for the parts of the chain, you can estimate the failure rate across your entire system.

Say you expand the previous example to specify that you have six services with the same success rate for calls. For any request to your system, you can expect one of four outcomes: all services work correctly, one service fails, multiple services fail, or all services fail.

Because calls to each microservice are successful 99.9% of the time, combined reliability of the system will be $0.999^6 = 0.994 = 99.4\%$. Although this is a simple model, you can see that the whole application will always be less reliable than its independent components; the maximum availability you can achieve is a product of the availability of a service's dependencies.

To illustrate, imagine that service D's availability is degraded to 95%. Although this won't affect transactions—because it's not part of that call hierarchy—it will reduce the reliability of both market-data and holdings. Figure 6.2 illustrates this impact.

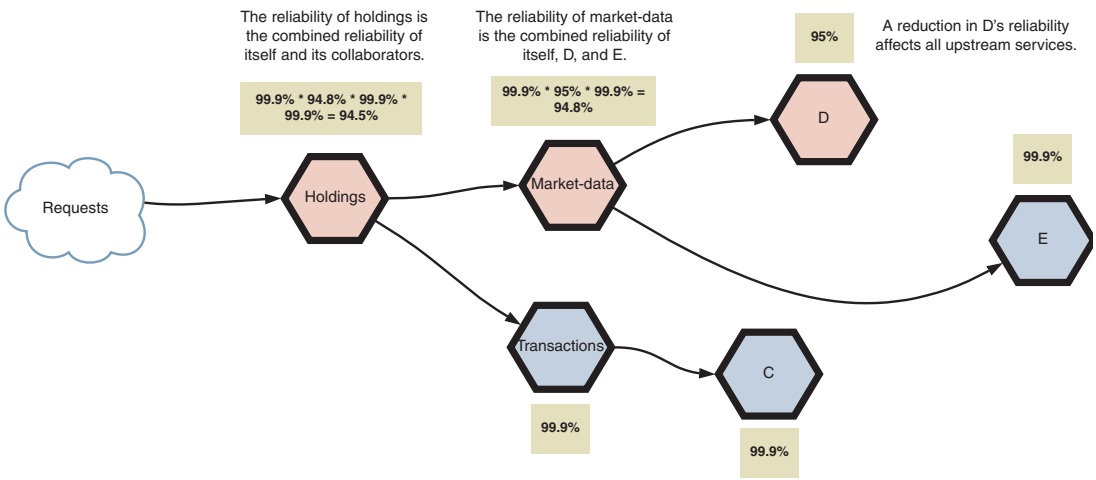


Figure 6.2 **The impact of service dependency availability on reliability in a microservice application**

It's crucial to maximize service availability—or isolate the impact of unreliability—to ensure the availability of your entire application. Measuring availability won't tell you how to make your services reliable, but it gives you a target to aim for or, more specifically, a goal to guide both the development of services and the expectations of consuming services and engineers.

> **NOTE** How do you monitor availability? We'll explore approaches to monitoring service availability in a microservice application in part 4 of this book.

If you can't trust the network, your hardware, other services, or even your own services to be 100% reliable, how can you maximize availability? You need to design defensively to meet three goals:

- Reduce the incidence of avoidable failures
- Limit the cascading and system-wide impact of unpredictable failures
- Recover quickly—and ideally automatically—when failures do occur

Achieving these goals will ultimately maximize the uptime and availability of your services.

6.2 What could go wrong?

As we've stated, failure is inevitable in a complex system. Over the lifetime of an application, it's incredibly likely that any catastrophe that could happen, will happen. Consequently, you need to understand the different types of failures that your application might be susceptible to. Understanding the nature of these risks and their likelihood is fundamental to both architecting appropriate mitigation strategies and reacting rapidly when incidents do occur.

Balancing risk and cost

It's important to be pragmatic: you can neither anticipate nor eliminate every possible cause of failure. When you're designing for resilience, you need to balance the risk of a failure against what you can reasonably defend against given time and cost constraints:

- The cost to design, build, deploy, and operate a defensive solution
- The nature of your business and expectations of your customers

To put that in perspective, consider the S3 outage I mentioned earlier. You could defend against that error by replicating data across multiple regions in AWS or across multiple clouds. But given that S3 failures of that magnitude are exceptionally rare, that solution wouldn't make economic sense for many organizations because it would significantly increase operational costs and complexity.

As a responsible service designer, you need to identify possible types of failure within your microservice application, rank them by anticipated frequency and impact, and decide how you'll mitigate their impact. In this section, we'll walk you through some

common failure scenarios in microservice applications and how they arise. We'll also explore cascading failures—a common catastrophic scenario in a distributed system.

6.2.1 Sources of failure

Let's examine a microservice to understand where failure might arise, using one of SimpleBank's services as an example. You can assume a few things about the market-data service:

- The service will run on hardware—likely a virtualized host—that ultimately depends on a physical data center.
- Other upstream services depend on the capabilities of this service.
- This service stores data in some store—for example, a SQL database.
- It retrieves data from third-party data sources through APIs and file uploads.
- It may call other downstream SimpleBank microservices.

Figure 6.3 illustrates the service and its relationship to other components.

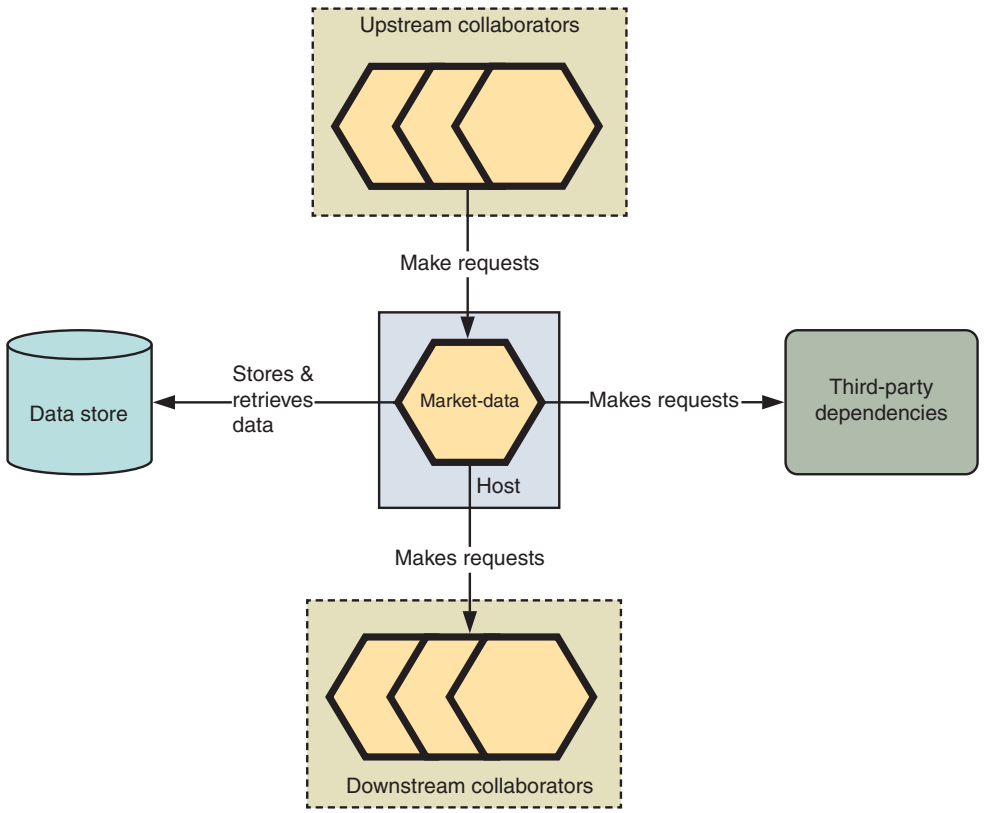


Figure 6.3 Relationships between the market-data microservice and other components of the application

Every point of interaction between your service and another component indicates a possible area of failure. Failures could occur in four major areas:

- *Hardware*—The underlying physical and virtual infrastructure on which a service operates
- *Communication*—Collaboration between different services and/or third parties
- *Dependencies*—Failure within dependencies of a service
- *Internal*—Errors within the code of the service itself, such as defects introduced by engineers

Let's explore each category in turn.

HARDWARE

Regardless of whether you run your services in a public cloud, on-premise, or using a PaaS, the reliability of the services will ultimately depend on the physical and virtual infrastructure that underpins them, whether that's server racks, virtual machines, operating systems, or physical networks. Table 6.1 illustrates some of the causes of failure within the hardware layer of a microservice application.

Table 6.1 Sources of failure within the hardware layer of a microservice application

Source of failure	Frequency	Description
Host	Often	Individual hosts (physical or virtual) may fail.
Data center	Rare	Data centers or components within them may fail.
Host configuration	Occasionally	Hosts may be misconfigured— for example, through errors in provisioning tools.
Physical network	Rare	Physical networking (within or between data centers) may fail.
Operating system and resource isolation	Occasionally	The OS or the isolation system—for example, Docker—may fail to operate correctly.

The range of possible failures at this layer of your application are diverse and unfortunately, often the most catastrophic because hardware component failure may affect the operation of multiple services within an organization.

Typically, you can mitigate the impact of most hardware failures by designing appropriate levels of redundancy into a system. For example, if you're deploying an application in a public cloud, such as AWS, you'd typically spread replicas of a service across multiple zones—geographically distinct data centers within a wider region—to reduce the impact of failure within a single center.

It's important to note that hardware redundancy can incur additional operational cost. Some solutions may be complex to architect and run—or just plain expensive. Choosing the right level of redundancy for an application requires careful consideration of the frequency and impact of failure versus the cost of mitigating against potentially rare events.

COMMUNICATION

Communication between services can fail: network, DNS, messaging, and firewalls are all possible sources of failure. Table 6.2 details possible communication failures.

Table 6.2 Sources of communication failure within a microservice application

Source of failure	Description
Network	Network connectivity may not be possible.
Firewall	Configuration management can set security rules inappropriately.
DNS errors	Hostnames may not be correctly propagated or resolved across an application.
Messaging	Messaging systems—for example, RPC—can fail.
Inadequate health checks	Health checks may not adequately represent instance state, causing requests to be routed to broken instances.

Communication failures can affect both internal and external network calls. For example, connectivity between the market-data service and the external APIs it relies on could degrade, leading to failure.

Network and DNS failures are reasonably common, whether caused by changes in firewall rules, IP address assignment, or DNS hostname propagation in a system. Network issues can be challenging to mitigate, but because they're often caused by human intervention (whether through service releases or configuration changes), the best way to avoid many of them is to ensure that you test configuration changes robustly, and that they're easy to roll back if issues occur.

DEPENDENCIES

Failure can occur in other services that a microservice depends on, or within that microservice's internal dependencies (such as databases). For example, the database that market-data relies on to save and retrieve data might fail because of underlying hardware failure or hitting scalability limits—it wouldn't be unheard of for a database to run out of disk space!

As we outlined earlier, such failures have a drastic effect on overall system availability. Table 6.3 outlines possible sources of failure.

Table 6.3 Sources of dependency-related failure

Source of failure	Description
Timeouts	Requests to services may time out, resulting in erroneous behavior.
Decommissioned or nonbackwardscompatible functionality	Design doesn't take service dependencies into account, unexpectedly changing or removing functionality.
Internal component failures	Problems with databases or caches prevent services from working correctly.
External dependencies	Services may have dependencies outside of the application that don't work correctly or as expected—for example, third-party APIs.

In addition to operational sources of failure, such as timeouts and service outages, dependencies are prone to errors caused by design and build failures. For example, a service may rely on an endpoint in another service that's changed in a nonbackwards-compatible way or, even worse, removed completely without appropriate decommissioning.

SERVICE PRACTICES

Lastly, inadequate or limited engineering practices when developing and deploying services may lead to failure in production. Services may be poorly designed, inadequately tested, or deployed incorrectly. You may not catch errors in testing, or a team may not adequately monitor the behavior of their service in production. A service might scale ineffectively: hitting memory, disk, or CPU limits on its provisioned hardware such that performance is degraded—or the service becomes completely unresponsive.

Because each service contributes to the effectiveness of the whole system, one poor quality service can have a detrimental effect on the availability of swathes of functionality. Hopefully the practices we outline throughout this book will help you avoid this—unfortunately common—source of failure!

6.2.2 Cascading failures

You should now understand how failure in different areas can affect a single microservice. But the impact of failure doesn't stop there. Because your applications are composed of multiple microservices that continually interact with each other, failure in one service can spread across an entire system.

Cascading failures are a common mode of failure in distributed applications. A cascading failure is an example of *positive feedback*: an event disturbs a system, leading to some effect, which in turn increases the magnitude of the initial disturbance. In this case, positive means that the effect increases—not that the outcome is beneficial.

You can observe this phenomenon in several real-world domains, such as financial markets, biological processes, or nuclear power stations. Consider a stampede in a herd of animals: panic causes an animal to run, which in turn spreads panic to other animals, which causes them to run, and so on. In a microservice application, overload can cause a domino effect: failure in one service increases failure in upstream services, and in turn their upstream services. At worst, the result is widespread unavailability.

Let's work through an example to illustrate how overload can result in a cascading failure. Imagine that SimpleBank built a UI to show a user their current financial holdings (or positions) in an account. That might look something like figure 6.4.

Each financial position is the sum of the transactions—purchases and sales of a stock—made to date, multiplied by the current price. Retrieving these values relies on collaboration between three services:

- *Market-data*—A service responsible for retrieving and processing price and market information for financial instruments, such as stocks
- *Transactions*—A service responsible for representing transactions occurring within an account
- *Holdings*—A service responsible for aggregating transactions and market-data to report financial positions

Holdings *as at 2017-07-23*

BHP Billiton Ltd BHP	Quantity **1000**	Value **$91,720**
Google GOOGL	Quantity **103**	Value **$14,023**
ABC Company ABC	Quantity **24**	Value **$1.20**

Figure 6.4 **A user interface that reports financial holdings in an account**

Figure 6.5 outlines the production configuration of these services. For each service, load is balanced across multiple replicas.

Suppose that holdings are being retrieved 1,000 times per second (QPS). If you have two replicas of your holdings service, each replica will receive 500 QPS (figure 6.6).

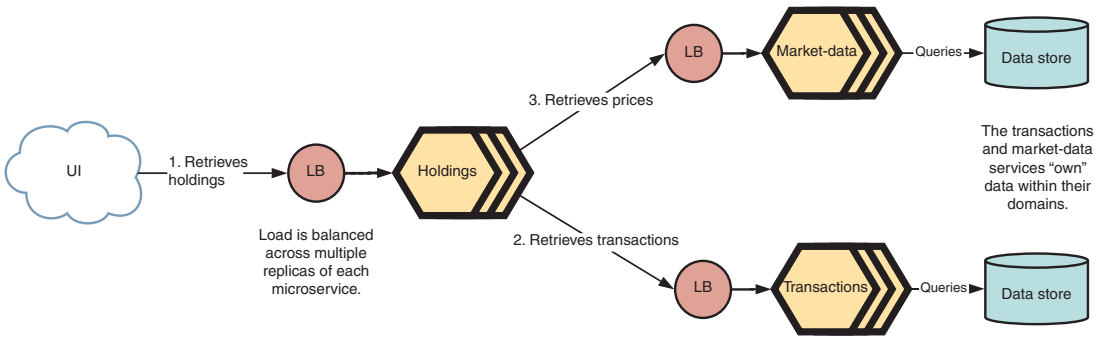


Figure 6.5 **Production configuration and collaboration between services to populate the "current financial holdings" user interface**

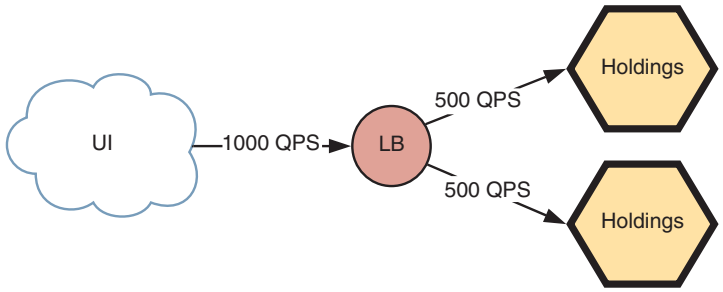


Figure 6.6 **Queries made to a service are split across multiple replicas.**

Your holdings service subsequently queries transactions and market-data to construct the response. Each call to holdings will generate two calls: one to transactions and one to market-data.

Now, let's say a failure occurs that takes down one of your transactions replicas. Your load balancer reroutes that load to the remaining replica, which now needs to service 1,000 QPS (figure 6.7).

But that reduced capacity is unable to handle the level of demand to your service. Depending on how you've deployed your service—the characteristics of your web server—the change in load might first lead to increased latency as requests are queued. In turn, increased latency might start exceeding the maximum wait time that the holdings service expects for that query. Alternatively, the transactions service may begin dropping requests.

It's not unreasonable for a service to retry a request to a collaborator when it fails. Now, imagine that the holdings service will retry any request to transactions that times out or fails. This will further increase the load on your remaining transactions resource, which now needs to handle both the regular request volume and the heightened retry volume (figure 6.8). In turn, the holdings service takes longer to respond while it waits on its collaborator.

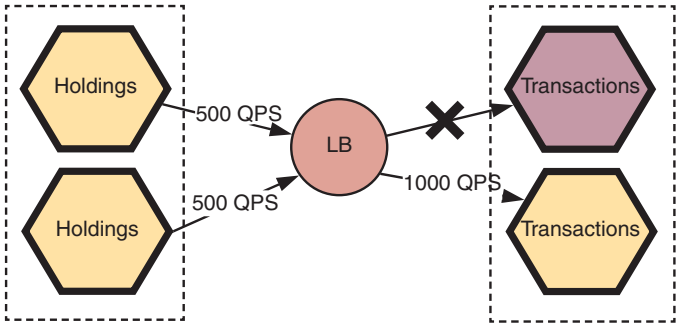


Figure 6.7 One replica of a collaborating service fails, sending all load to the remaining instance.

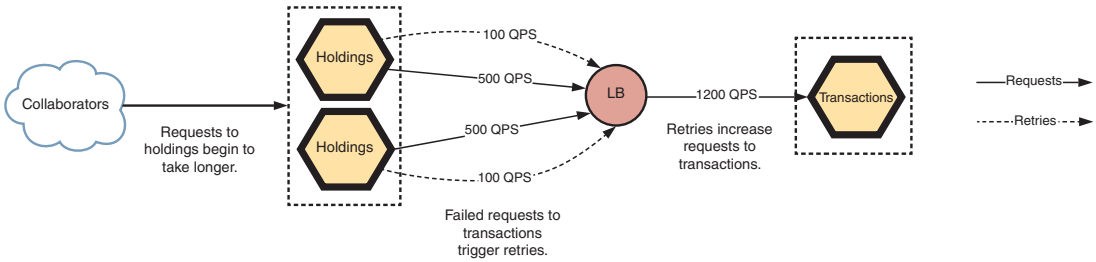


Figure 6.8 Overload on transactions causes some requests to fail, in turn causing holdings to retry those requests, which starts to degrade holdings' response time.

Upstream dependencies are unable to service requests that rely on transactions.

Increased failure in upstream dependencies leads to retries, repeating the cycle of failure.

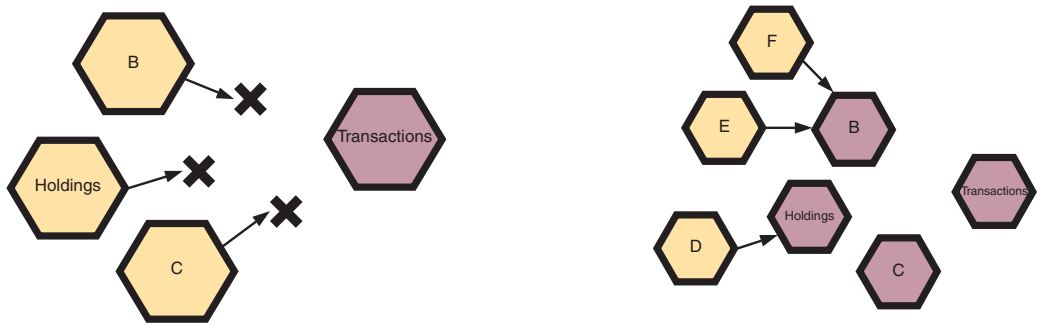


Figure 6.9 Overload in a service leads to complete failure. Unhealthy retry behavior is repeated across dependency chains as service performance progressively degrades, leading to further overloads.

This feedback loop—failed requests lead to a higher volume of subsequent requests, leading to a higher rate of failure—continues to escalate. Your whole system is unable to complete work, as other services that rely on transactions or holdings begin to fail. Your initial failure in a single service causes a domino effect, worsening response times and availability across several services. At worst, the cumulative impact on the transactions service causes it to fail completely. Figure 6.9 illustrates this final stage of a cascading failure.

Cascading failures aren't only caused by overload—although this is one of the most common root causes. In general, increased error rates or slower response times can lead to unhealthy service behavior, increasing the chance of failure across multiple services that depend on each other.

You can use several approaches to limit the occurrence of cascading failures in micro-service applications: circuit breakers; fallbacks; load testing and capacity planning; back-off and retries; and appropriate timeouts. We'll explore these approaches in the next section.

6.3 Designing reliable communication

Earlier, we emphasized the importance of collaboration in a microservice application. Dependency chains of multiple microservices will achieve most useful capabilities in an application. When one microservice fails, how does that impact its collaborators and ultimately, the application's end customers?

If failure is inevitable, you need to design and build your services to maximize availability, correct operation, and rapid recovery when failure does occur. This is fundamental to achieving resiliency. In this section, we'll explore several techniques for ensuring that services behave appropriately—maximizing correct operation—when their collaborators are unavailable:

- Retries
- Fallbacks, caching, and graceful degradation
- Timeouts and deadlines

- Circuit breakers
- Communication brokers

Before we start, let's get a simple service running that we can use to illustrate the concepts in this section. You can find these examples in the book's Github repository (http://mng.bz/7eN9). Clone the repository to your computer and open the chapter-6 directory. This directory contains some basic services—holdings and market-data—which you'll run inside Docker containers (figure 6.10). The holdings service exposes a `GET /holdings` endpoint, which makes a JSON API request to retrieve price information from market-data.

To run these, you'll need `docker-compose` installed (directions online: https://docs.docker.com/compose/install/). If you're ready to go, type the following at the command line:

```
$ docker-compose up
```

This will build Docker images for each service and start them as isolated containers on your machine. Now let's dive in!

6.3.1 Retries

In this section, we'll explore how to use retries when failed requests occur. To understand these techniques, let's start by examining communication from the perspective of your upstream service, holdings.

Imagine that a call from the holdings service to retrieve prices fails, returning an error. From the perspective of the calling service, it's not clear yet whether this failure is isolated—repeating that call is likely to succeed, or systemic—the next call has a high likelihood of failing. You expect calls to retrieve data to be *idempotent*—to have no effect on the state of the target system and therefore be repeatable.[2]

As a result, your first instinct might be to retry the request. In Python, you can use an open source library—`tenacity`—to decorate the appropriate method of your API client (the `MarketDataClient` class in holdings/clients.py) and perform retries if the method throws an exception. The following listing shows the class with retry code added.

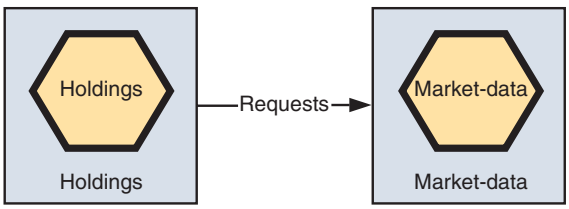


Figure 6.10 Docker containers for working with microservice requests

[2] Requests that effect some system change aren't typically idempotent. One strategy for guaranteeing "exactly once" semantics is to implement idempotency keys. See Brandur Leach, "Designing robust and predictable APIs with idempotency," February 22, 2017, https://stripe.com/blog/idempotency.

Listing 6.1 Adding a retry to a service call

```
import requests
import logging
from tenacity import retry, stop, before

class MarketDataClient(object):

    logger = logging.getLogger(__name__)
    base_url = 'http://market-data:8000'

    def _make_request(self, url):
        response = requests.get(f"{self.base_url}/{url}",
                                headers={'content-type': 'application/json'})
        return response.json()

    @retry(stop=stop_after_attempt(3),
           before=before_log(logger, logging.DEBUG))

    def all_prices(self):
        return self._make_request("prices")
```

Imports relevant functions from the library

Retries the query up to three times

Logs each retry before execution

Let's call the holdings service to see how it behaves. In another terminal window, make the following request:

```
curl -I http://{DOCKER_HOST}/holdings
```

This will return a 500 error, but if you follow the logs from the market-data service, you can see a request being made to `GET /prices` three times, before the holdings service gives up.

If you read the previous section, you should be wary at this point. Failure might be isolated or persistent, but the holdings service can't know which one is the case based on one call.

If the failure is isolated and transient, then a retry is a reasonable option. This helps to minimize direct impact to end users—and explicit intervention from operational staff—when abnormal behavior occurs. It's important to consider your budget for retries: if each retry takes a certain number of milliseconds, then the consuming service can only absorb so many retries before it surpasses a reasonable response time.

But if the failure is persistent—for example, if the capacity of market-data is reduced—then subsequent calls may worsen the issue and further destabilize the system. Suppose you retry each failed request to market-data five times. Every failed request you make to this service potentially results in another five requests; the volume of retries continues to grow. The entire service is doing less useful work as it attempts to service a high volume of retries. At worst, retries suffocate your market-data service, magnifying your original failure. Figure 6.11 illustrates this growth of requests.

Figure 6.11 **Growth of load on your unstable market-data service resulting from failed requests being retried**

How can you use retries to improve your resiliency in the face of intermittent failures without contributing to wider system failure if persistent failures occur? First, you could use a variable time between successive retries to try to spread them out evenly and reduce the frequency of retry-based load. This is known as an *exponential back-off* strategy and is intended to give a system under load time to recover. You can change the retry strategy you used earlier, as shown in the following listing. Afterwards, by curling the `/holdings` endpoint, you can observe the retry behavior of the service.

Listing 6.2 **Changing your retry strategy to exponential back-off**

```
@retry(wait=wait_exponential(multiplier=1, max=5),
       stop=stop_after_delay(5))
def all_prices(self):
    return self._make_request("prices")
```

Waits 2 ^ x * 1 second between each retry

Stops after five seconds

Unfortunately, exponential back-off can lead to another instance of curious emergent behavior. Imagine that a momentary failure interrupts several calls to market-data, leading to retries. Exponential back-off can cause the service to schedule those retries together so they further amplify themselves, like the ripples from throwing a stone in a pond.

Instead, back-off should include a random element—jitter—to spread out retries to a more constant rate and avoid thundering herds of synchronized retries.[3] The following listing shows how to adjust your strategy again.

[3] A great article by Marc Brooker about exponential back-off and the importance of jitter is available on the AWS Architecture Blog, March 4, 2015, http://mng.bz/TRk5.

Listing 6.3 Adding jitter to an exponential back-off

```
@retry(wait=wait_exponential(multiplier=1, max=5) + wait_random(0, 1),
       stop=stop_after_delay(5))
def all_prices(self):
    return self._make_request("prices")
```

Exponentially backs off, adding a random wait between zero and one second

Stops after five seconds

This strategy will ensure that retries are less likely to happen in synchronization across multiple waiting clients.

Retries are an effective strategy for tolerating intermittent dependency faults, but you need to use them carefully to avoid exacerbating the underlying issue or consuming unnecessary resources:

- Always limit the total number of retries.
- Use exponential back-off with jitter to smoothly distribute retry requests and avoid compounding load.
- Consider which error conditions should trigger a retry and, therefore, which retries are unlikely to, or will never, succeed.

When your service meets retry limits or can't retry a request, you can either accept failure or find an alternative way to serve the request. In the next section, we'll explore fallbacks.

6.3.2 Fallbacks

If a service's dependencies fail, you can explore four fallback options:

- Graceful degradation
- Caching
- Functional redundancy
- Stubbed data

GRACEFUL DEGRADATION

Let's return to the problem with the holdings service: if market-data fails, the application may not be able to provide valuations to end customers. To resolve this issue, you might be able to design an acceptable degradation of service. For example, you could show holding quantities without valuations. This limits the richness of your UI but is better than showing nothing—or an error. You can see techniques like this in other domains. For example, an e-commerce site could still allow purchases to be made, even if the site's order dispatch isn't functioning correctly.

CACHING

Alternatively, you could cache the results of past queries for prices, reducing the need to query the market-data service at all. Say a price is valid for five minutes. If so, the holdings service could cache pricing data for up to five minutes, either locally or in a

dedicated cache (for example, Memcached or Redis). This solution would both improve performance and provide contingency in the event of a temporary outage.

Let's try out this technique. You'll use a library called `cachetools`, which provides an implementation of a time-to-live cache. As you did earlier with retries, you'll decorate your client method, as shown in the following listing.

Listing 6.4 Adding in-process caching to a client call

```
import requests
import logging
from cachetools import cached, TTLCache

class MarketDataClient(object):

    logger = logging.getLogger(__name__)
    cache = TTLCache(maxsize=10, ttl=5*60)
    base_url = 'http://market-data:8000'

    def _make_request(self, url):
        response = requests.get(f"{self.base_url}/{url}",
                                headers={'content-type': 'application/json'})
        return response.json()

    @cached(cache)
    def all_prices(self):
        logger.debug("Making request to get all_prices")
        return self._make_request("prices")
```

Instantiates a cache

Decorates your method to store results using your cache

Subsequent calls made to `GET /holdings` should retrieve price information from the cache, rather than by making calls to market-data. If you used an external cache instead, multiple instances could use the cache, further reducing load on market-data and providing greater resiliency for all holdings replicas, albeit at the cost of maintaining an additional infrastructural component.

Functional redundancy

Similarly, you might be able to fall back to other services to achieve the same functionality. Imagine that you could purchase market data from multiple sources, each covering a different set of instruments at a different cost. If source A failed, you could instead make requests to source B (figure 6.12).

Functional redundancy within a system has many drivers: external integrations; algorithms for producing similar results with varying performance characteristics; and even older features that remain operational but have been superseded. In a globally distributed deployment, you could even fall back on services hosted in another region.[4]

Only some failure scenarios would allow the use of an alternative service. If the cause of failure was a code defect or resource overload in your original service, then rerouting to another service would make sense. But a general network failure could affect multiple services, including ones you might try rerouting to.

[4] At the ultimate end of this scale, Netflix can serve a given customer from any of their global data centers, conveying an impressive degree of resilience.

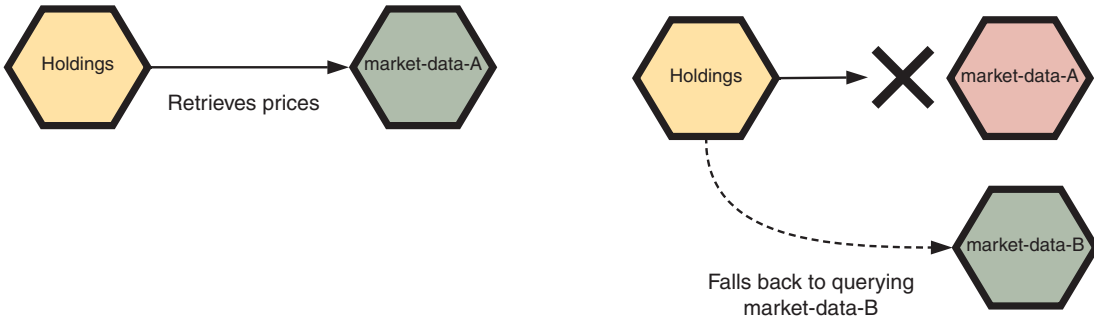


Figure 6.12 **If service failure occurs, you may be able to serve the same capability with other services.**

STUBBED DATA

Lastly, although it wouldn't be appropriate in this specific scenario, you could use stubbed data for fallbacks. Picture the "recommended to you" section on Amazon: if the backend was unable for some reason to retrieve those personalized recommendations, it'd be more graceful to fall back to a nonpersonalized data set than to show a blank section on the UI.

6.3.3 Timeouts

When the holdings service sends a request to market-data, that service consumes resources waiting for a reply. Setting an appropriate deadline for that interaction limits the time those resources are consumed.

You can set a timeout within your HTTP request function. For HTTP calls, you want to timeout if you haven't received any response, but not if the response itself is slow to download. Try the following listing to add a timeout.

Listing 6.5 Adding a timeout to an HTTP call

```
def _make_request(self, url):
    response = requests.get(f"{self.base_url}/{url}",
                            headers={'content-type': 'application/json'},
                            timeout=5)

    return response.json()
```

Sets a timeout of five seconds before receiving data from market-data

In a computational sense, network communication is slow, so the speed of failures is important. In a distributed system, some errors might happen almost instantly. For example, a dependent service may rapidly fail in the event of an internal bug. But many failures are slow. For example, a service that's overloaded by requests may respond sluggishly, in turn consuming the resources of the calling service while it waits for a response that may never come.

Slow failures illustrate the importance of setting appropriate deadlines—timing out in a reasonable timeframe—for communication between microservices. If you don't set upper bounds, it's easy for unresponsiveness to spread through entire microservice dependency chains. In fact, lack of deadlines can extend the impact of issues because a server consumes resources while it waits forever for an issue to be resolved.

Picking a deadline can be difficult. If they're too long, they can consume unnecessary resources for a calling service if a service is unresponsive. If they're too short, they can cause higher levels of failure for expensive requests. Figure 6.13 illustrates these constraints.

For many microservice applications, you set deadlines at the level of individual interactions; for example, a call from holdings to market-data may always have a deadline of 10 seconds. A more elegant approach is to apply an absolute deadline across an entire operation and propagate the remaining time across collaborators.

Without propagating deadlines, it can be difficult to make them consistent across a request. For example, holdings could waste resources waiting for market-data far beyond the overall deadline imposed by a higher level of the stack, such as an API gateway.

Imagine a chain of dependencies between multiple services. Each service takes a certain amount of time to do its work and expects its collaborators to take some time. If any of those times vary, static expectations may no longer be correct (figure 6.14).

If your service interactions are over HTTP, you could propagate deadlines using a custom HTTP header, such as `X-Deadline: 1000`, passing that value to set read timeout values on subsequent HTTP client calls. Many RPC frameworks, such as gRPC, explicitly implement mechanisms for propagating deadlines within a request context.

6.3.4 Circuit breakers

You can combine some of the techniques we've discussed so far. You can consider an interaction between holdings and market-data as analogous to an electrical circuit. In electrical wiring, circuit breakers perform a protective role—preventing spikes in current from damaging a wider system. Similarly, a circuit breaker is a pattern for pausing requests made to a failing service to prevent cascading failures.

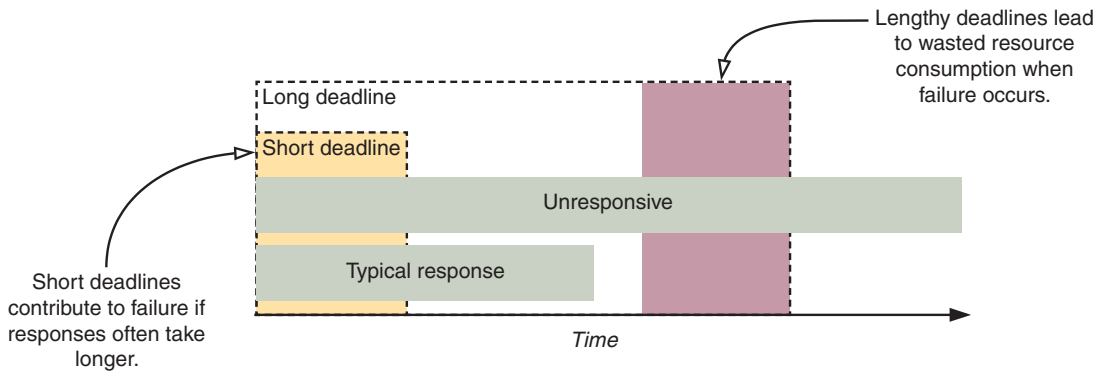


Figure 6.13 Choosing the right deadline requires balancing time constraints to maximize the window of successful requests.

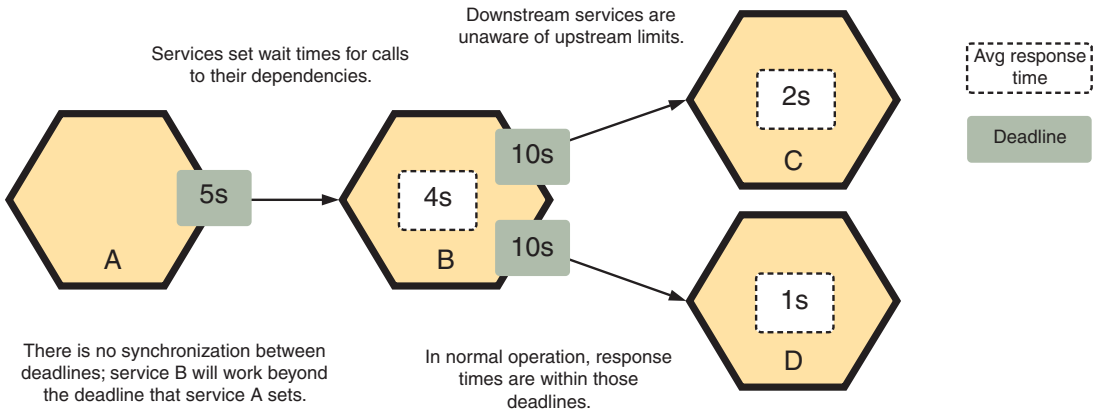


Figure 6.14 Services may set expectations about how long they expect calls to collaborators to take; varying widely because of failure or latency can exacerbate the impact of those failures.

How does it work? Two principles, both of which we touched on in the previous section, inform the design of a circuit breaker:

1. Remote communication should fail quickly in the event of an issue, rather than wasting resources waiting for a response that might never come.
2. If a dependency is failing consistently, it's better to stop making further requests until that dependency recovers.

When making a request to a service, you can track the number of times that request succeeds or fails. You might track this number within each running instance of a service or share that state (using an external cache) across multiple services. In this normal operation, we consider the circuit to be closed.

If the number of failures seen or the rate of failures within a certain time window passes a threshold, then the circuit is opened. Rather than attempting to send requests to your collaborating service, you should short-circuit requests and, where possible, perform an appropriate fallback—returning a stubbed message, routing to a different service, or returning a cached response. Figure 6.15 illustrates the lifecycle of a request using a circuit breaker.

Setting the time window and threshold requires careful consideration of both the expected reliability of the target service and the volume of interactions between services. If requests are relatively sparse, then a circuit breaker may not be effective, because a large time window might be required to obtain a representative sample of requests. For service interactions with contrasting busy and quiet periods, you may want to introduce a minimum throughput to ensure a circuit only reacts when load is statistically significant.

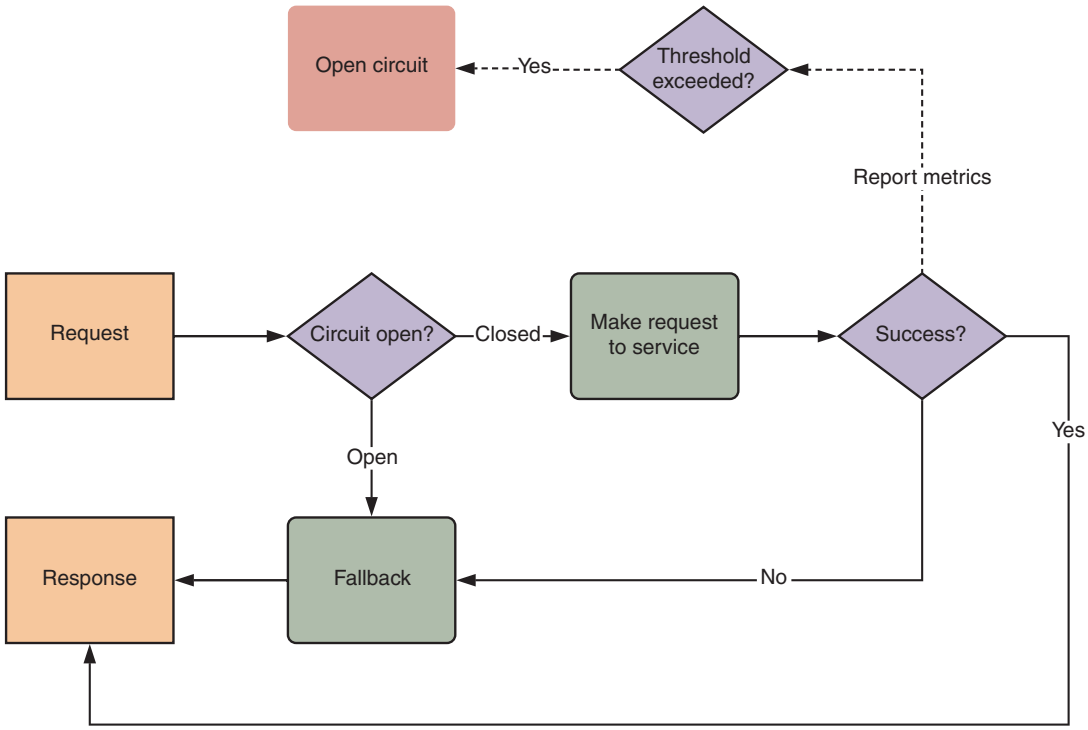


Figure 6.15 **A circuit breaker controls the flow of requests between two services and opens when the number of failed requests surpasses a threshold.**

> **NOTE** You should monitor when circuits are opened and closed, as well as potentially alerting the team responsible, especially if the circuit is frequently opened. We'll discuss this further in part 4.

Once the circuit has opened, you probably don't want to leave it that way. When availability returns to normal, the circuit should be closed. The circuit breaker needs to send a trial request to determine whether the connection has returned to a healthy state. In this trial state, the circuit breaker is *half open*: if the call succeeds, the circuit will be closed; otherwise, it will remain open. As with other retries, you should schedule these attempts with an exponential back-off with jitter. Figure 6.16 shows the three distinct states of a circuit breaker.

Several libraries are available that provide implementations of the circuit breaker pattern in different languages, such as Hystrix (Java), CB2 (Ruby), or Polly (.NET).

> **TIP** Don't forget that closed is the good state for a circuit breaker! The use of open and closed to represent, respectively, negative and positive states may seem counterintuitive but reflects the real-world behavior of an electrical circuit.

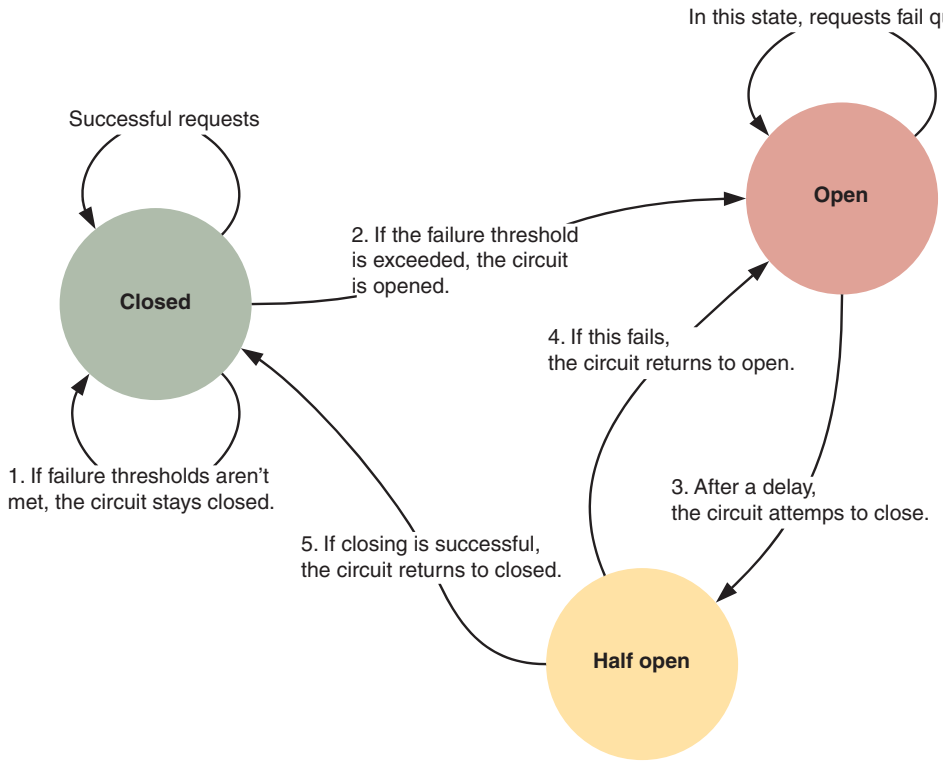


Figure 6.16 **A circuit breaker transitions between three stages: open, closed, and half open.**

6.3.5 Asynchronous communication

So far, we've focused on failure in synchronous, point-to-point communication between services. As we outlined in the first section, the more services in a chain, the lower overall availability you can guarantee for that path.

Designing asynchronous service interactions, using a communication broker like a message queue, is another technique you can use to maximize reliability. Figure 6.17 illustrates this approach.

Where you don't need immediate, consistent responses, you can use this technique to reduce the number of direct service interactions, in turn increasing overall availability—albeit at the expense of making business logic more complex. As we mentioned elsewhere in this book, a communication broker becomes a single point of failure that will require careful attention for you to scale, monitor, and operate effectively.

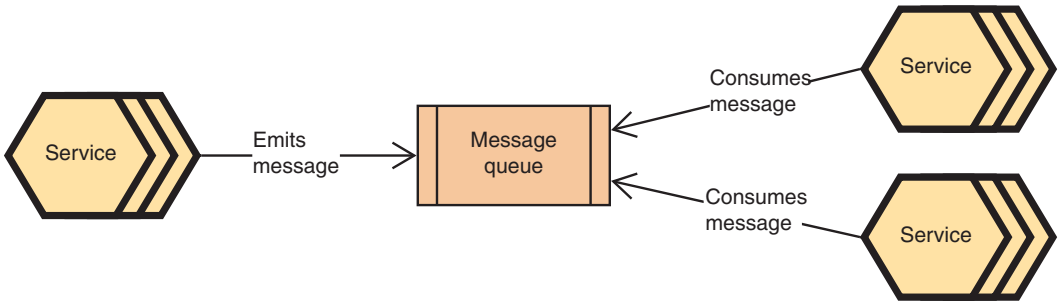


Figure 6.17 Using a message queue to decouple services from direct interaction

6.4 Maximizing service reliability

In the previous sections, we explored techniques to ensure a service can tolerate faults in interactions with its collaborators. Now, let's consider how you can maximize availability and fault tolerance within an individual service. In this section, we'll explore two techniques—health checks and rate limits—as well as methods for validating the resilience of services.

6.4.1 Load balancing and service health

In production, you deploy multiple instances of your market-data service to ensure redundancy and horizontal scalability. A load balancer will distribute requests from other services between these instances. In this scenario, the load balancer plays two roles:

1. Identifying which underlying instances are healthy and able to serve requests
2. Routing requests to different underlying instances of the service

A load balancer is responsible for executing or consuming the results of *health checks*. In the previous section, you could ascertain the health of a dependency at the point of interaction—when requests were being made. But that's not entirely adequate. You should have some way of understanding the application's readiness to serve requests at any time, rather than when it's actively being queried.

Every service you design and deploy should implement an appropriate health check. If a service instance becomes unhealthy, that instance should no longer receive traffic from other services. For synchronous RPC-facing services, a load balancer will typically query each instance's health check endpoint on an interval basis. Similarly, asynchronous services may use a heartbeat mechanism to test connectivity between the queue and consumers.

> **TIP** It's often desirable for repeated or systematic instance failures, as detected by health checks, to trigger alerts to an operations team—a little human intervention can be helpful. We'll explore that further in part 4 of this book.

You can classify health checks based on two criteria: liveness and readiness. A liveness check is typically a simple check that the application has started and is running correctly. For example, an HTTP service should expose an endpoint—commonly `/health`, `/ping`, or `/heartbeat`—that returns a 200 OK response once the service is running (figure 6.18). If an instance is unresponsive, or returns an error message, the load balancer will no longer deliver requests there.

In contrast, a readiness check indicates whether a service is ready to serve traffic, because being alive may still not indicate that requests will be successful. A service might have many dependencies—databases, third-party services, configuration, caches—so you can use a readiness check to see if these constituent components are in the correct state to serve requests. Both of the example services implement a simple HTTP liveness check, as shown in the following listing.

Listing 6.6 Flask handler for an HTTP liveness check

```
@app.route('/ping', methods=["GET"])
def ping():
    return 'OK'
```

Health checks are binary: either an instance is available or it isn't. This works well with typical round-robin load balancing, where requests are distributed to each replica in turn. But in some circumstances the functioning of a service may be degraded and exhibit increased latency or error rates without a health check reflecting this status. As such, it can be beneficial to use load balancers that are aware of latency and able to route requests to instances that are performing better, or those that are under less load, to achieve more consistent service behavior. This is a typical feature of a microservice proxy, which we'll touch on later in this chapter.

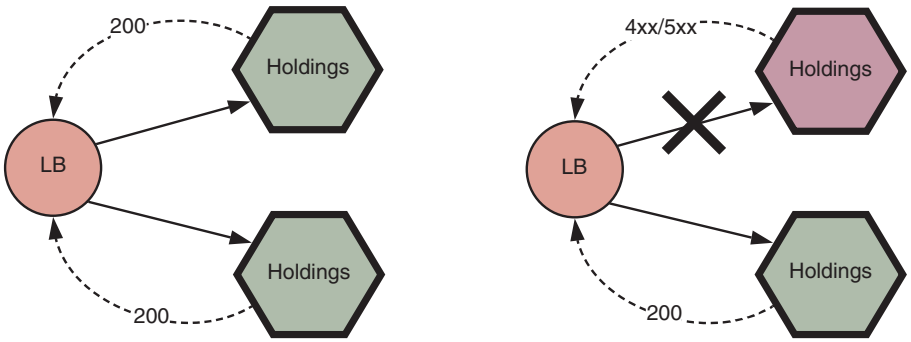


Figure 6.18 Load balancers continuously query service instances to check their health. If an instance is unhealthy, the load balancer will no longer route requests to that instance until it recovers.

6.4.2 Rate limits

Unhealthy service usage patterns can sometimes arise in large microservice applications. Upstream dependencies might make several calls, where a single batch call would be more appropriate, or available resources may not be distributed equitably among all callers. Similarly, a service with third-party dependencies could be limited by restrictions that those dependencies impose.

An appropriate solution is to explicitly limit the rate of requests or total requests available to collaborating services in a timeframe. This helps to ensure that a service—particularly when it has many collaborators—isn't overloaded. The limiting might be indiscriminate (drop all requests above a certain volume) or more sophisticated (drop requests from infrequent service clients, prioritize requests for critical endpoints, and drop low-priority requests). Table 6.4 outlines different rate-limiting strategies.

Table 6.4 Common rate-limiting strategies

Strategy	Description
Drop requests above volume	Drop consumer requests above a specified volume
Prioritize critical traffic	Drop requests to low-priority endpoints to prioritize resources for critical traffic
Drop uncommon clients	Prioritize frequent consumers of the service over infrequent users
Limit concurrent requests	Limit the overall number of requests an upstream service can make over a time period

Rate limits can be shared with a service's clients at design time or, better, at runtime. A service might return a header to a consumer that indicates the remaining volume of requests available. On receipt, the upstream collaborator should take this into account and adjust its rate of outbound requests. This technique is known as back pressure.

6.4.3 Validating reliability and fault tolerance

Applying the tactics and patterns we've covered will put you on a good path toward maximizing availability. But it's not enough to plan and design for resiliency: you need to validate that your services can tolerate faults and recover gracefully.

Thorough testing provides assurance that your chosen design is effective when both predicted and unpredictable failures occur. Testing requires the application of *load testing* and *chaos testing*. Although it's likely you're familiar with code testing—such as unit and acceptance testing to validate implementation, usually in a controlled environ-ment—you might not know that load and chaos testing are intended to validate service limits by closely replicating the turbulence of production operation. Although testing isn't the primary focus of this book, it's useful to understand how these different testing techniques can help you build a robust microservice application.

LOAD TESTING

As a service developer, you can usually be confident that the number of requests made to your service will increase over time. When developing a service, you should

1. Model the expected growth and shape of service traffic to ensure that you understand the likely usage of your service
2. Estimate the capacity required to service that traffic
3. Validate the deployed capacity of the service by load testing against those limits
4. Use business and service metrics as appropriate to re-estimate capacity

Imagine you're considering how much capacity the market-data service requires. First, what do you know about the service's usage patterns? You know that holdings queries the service, but it may be called from elsewhere too—pricing data is used throughout SimpleBank's product.

Let's assume that queries to market-data grow roughly in line with the number of active users on the platform, but you may experience spikes (for example, when the market opens in the morning). You can plan capacity based on predictions of your business growth. Table 6.5 outlines a simple estimation of the QPS that you can expect this service to receive over a three-month period.

Table 6.5 Estimate of calls to a service per second based on growth in average active users over a three-month period

			Jun	*Jul*	*Aug*
Total Users			4000	5600	7840
Expected Growth			40%	40%	40%
Active Users	*Average*	20%	800	1120	1568
	Peak	70%	2800	3920	5488
Service Calls	*Average*				
	Per User/Minute	30	24000	33600	47040
	Per User/Second	0.5	400	560	784
	Peak				
	Per User/Minute	30	84000	117600	164640
	Per User/Second	0.5	1400	1960	2744

Identifying the qualitative factors that drive growth in service utilization is vital to good design and optimizing capacity. Once you've done that, you can determine how much capacity to deploy. For example, the table suggests you need to be able to ser-

vice 400 requests per second in normal operation, growing by 40% month on month, with spikes in peak usage to 1,400 requests per second.

> **TIP** An in-depth review of capacity and scale planning techniques is outside the scope of this book, but a great overview is available in Abbott and Fisher's *The Art of Scalability* (Addison-Wesley Professional, 2015) (ISBN 978-0134032801).

Once you've established a baseline capacity for your service, you can then iteratively test that capacity against expected traffic patterns. Along with validating the traffic limits of a microservice configuration, load testing can identify potential bottlenecks or design flaws that aren't apparent at lower levels of load. Load testing can provide you with highly effective insight into the limitations of your services.

At the level of individual services, you should automate the load testing of each service as part of its delivery pipeline—something we'll explore in part 3 of this book. Along with this systematic load testing, you should perform exploratory load testing to identify limits and test your assumptions about the load that services can handle.

You also should load test services together. This can aid in identifying unusual load patterns and bottlenecks based on service interaction. For example, you could write a load test that exercises all the services in the `GET /holdings` example.

Chaos testing

Many failures in a microservice application don't arise from within the microservices themselves. Networks fail, virtual machines fail, databases become unresponsive—failure is everywhere! To test for these types of failure scenarios, you need to apply chaos testing.

Chaos testing pushes your microservice application to fail in production. By introducing instability and failure, it accurately mimics real system failures, as well as training an engineering team to be able to react to those failures. This should ultimately build your confidence in the system's capability to withstand real chaos because you'll be gradually improving the resiliency of your system and reducing the possible number of events that would cause operational impact.

As explained on the "Principles of Chaos Engineering" website (https://principlesofchaos.org/), you can think of chaos testing as "the facilitation of experiments to uncover systemic weaknesses." The website lays out this approach:

1. Define a measurable steady state of normal system operation.
2. Hypothesize that behavior in an experimental and control group will remain steady; the system will be resilient to the failure introduced.
3. Introduce variables that reflect real-world failure events—for example, removing servers, severing network connections, or introducing higher levels of latency.
4. Attempt to disprove the hypothesis you defined in (2).

Recall how the holdings, transactions, and market-data services were deployed in figure 6.5. In this case, you expect steady operation to return holdings data within a reasonable response time. A chaos test could introduce several variables:

1. Killing nodes running market-data or transactions, either partially or completely
2. Reducing capacity by killing holdings instances at random
3. Severing the network connection—for example, between holdings and downstream services or between services and their data stores

Figure 6.19 illustrates these options.

Companies with mature chaos testing practices might even perform testing on both a systematic and random basis against live production environments. This might sound terrifying; real outages can be stressful enough, let alone actively working to make them happen. But without taking this approach, it's incredibly difficult to know that your system is truly resilient in the ways that you expect. In any organization, you should start small, by introducing a limited set of possible failures, or only running scheduled, rather than random, tests. Although you can also perform chaos tests in a staging environment, you'll need to carefully consider whether that environment is truly representative of or equivalent to your production configuration.

TIP Chaos Toolkit (http://chaostoolkit.org/) is a great tool to start with if you'd like to practice chaos engineering techniques.

Ultimately, by regularly and systematically validating your system against chaotic events and resolving the issues you encounter, you and your team will be able to achieve a significant level of confidence in your application's resilience to failure.

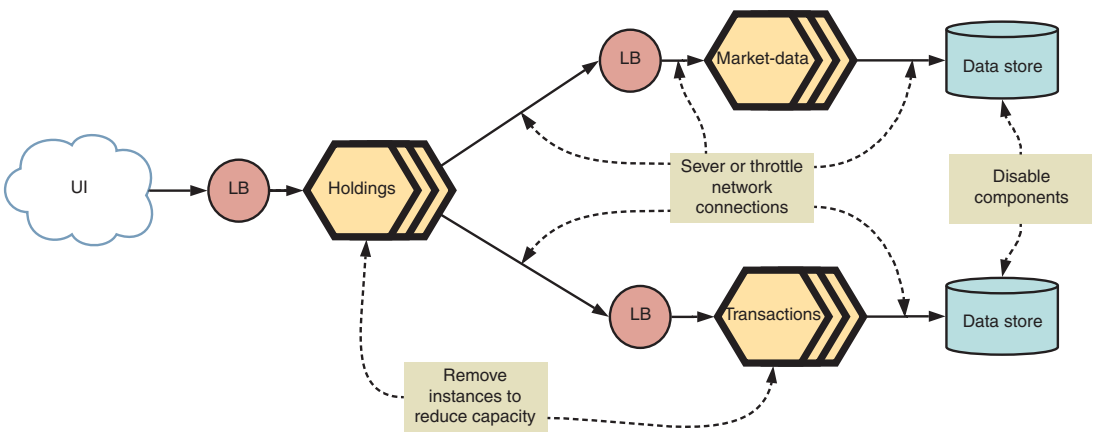


Figure 6.19 Potential variables to introduce in a chaos test to reflect real-world failure events

6.5 Safety by default

Critical paths in your microservice application will only be as resilient and available as their weakest link. Given the impact that individual services can have overall availability, it's imperative to avoid emergencies where introducing new services or changes in a service dependency chain significantly degrade that measure. Likewise, you don't want to find out that crucial functionality can't tolerate faults *when that fault happens.*

When applications are technically heterogeneous, or distinct teams deliver underlying services, it can be exceptionally difficult to maintain consistent approaches to reliable interaction. We touched on this back in chapter 2 when we discussed isolation and technical divergence. Teams are under different delivery pressures and different services have different needs—at worst, developers might forget to follow good resiliency practices.

Any change in service topology can have a negative impact. Figure 6.20 illustrates two examples: adding a new collaborator downstream from market-data might decrease market-data's availability, whereas adding a new consumer might reduce the overall capacity of the market-data service, reducing service for existing consumers.

Frameworks and proxies are two different technical approaches to applying communication standards across multiple services that make it easy for engineers to fall into doing the right thing by ensuring services communicate resiliently and safely by default.

6.5.1 Frameworks

A common approach for ensuring services always communicate appropriately is to mandate the use of specific libraries implementing common interaction patterns like circuit breakers, retries, and fallbacks. Standardizing these interactions across all services using a library has the following advantages:

1. Increases the overall reliability of your application by avoiding roll-your-own approaches to service interaction
2. Simplifies the process of rolling out improvements or optimizations to communication across any number of services
3. Clearly and consistently distinguishes network calls from local calls within code
4. Can be extended to provide supporting functionality, such as collecting metrics on service interactions

This approach tends to be more effective when a company uses one language (or few languages) for writing code; for example, Hystrix, which we mentioned earlier, was intended to provide a standardized way—across all Java-based services in Netflix's organization—of controlling interactions between distributed services.

NOTE Standardizing communication is a crucial element of building a microservice chassis, which we'll explore in the next chapter.

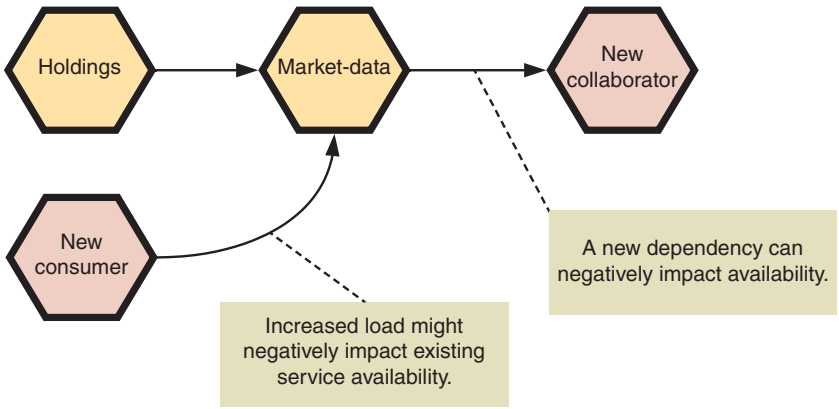


Figure 6.20 **Availability impact of new services in a dependency chain**

6.5.2 Service mesh

Alternatively, you could introduce a *service mesh*, such as Linkerd (https://linkerd.io) or Envoy (www.envoyproxy.io), between your services to control retries, fallbacks, and circuit breakers, rather than making this behavior part of each individual service. A service mesh acts as a proxy. Figure 6.21 illustrates how a service mesh handles communication between services.

Instead of services communicating directly with other services, service communication passes through the service mesh application, typically deployed as a separate process on the same host as the service. You then can configure the proxy to manage that traffic appropriately—retrying requests, managing timeouts, or balancing load across different services. From the caller's perspective, the mesh doesn't exist—it makes HTTP or RPC calls to another service as normal.

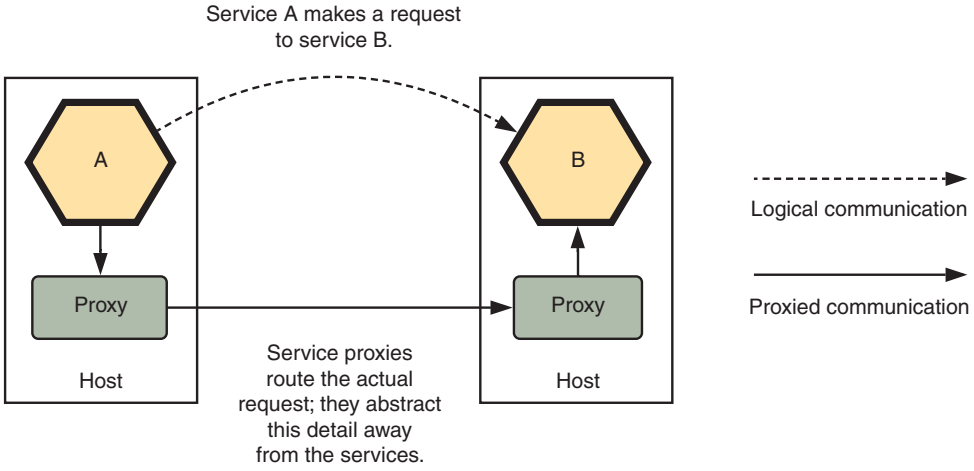


Figure 6.21 **Communication between services using a service mesh**

Although this may make the treatment of service interaction less explicit to an engineer working on a service, it can simplify defensive communication in applications that are heterogeneous. Otherwise, consistent communication can require significant time investment to achieve across different languages, because ecosystems and libraries may have unequal capabilities or support for resiliency features.

6.6 Summary

- Failure is inevitable in complex distributed systems—you have to consider fault tolerance when you're designing them.
- The availability of individual services affects the availability of the wider application.
- Choosing the right level of risk mitigation for an application requires careful consideration of the frequency and impact of failure versus the cost of mitigating against potentially rare events.
- Most failures occur in one of four areas: hardware, communication, dependencies, or internally.
- Cascading failures result from positive feedback and are a common failure mode in a microservice application. They're most commonly caused by server overload.
- You can use retries and deadlines to mitigate against faults in service interactions. You need to apply retries carefully to avoid exacerbating failure in other services.
- You can use fallbacks—such as caching, alternative services, and default results—to return successful responses, even when service dependencies fail.
- You should propagate deadlines between services to ensure they're consistent across a system and to minimize wasted work.
- Circuit breakers between services protect against cascading failures by failing quickly when a high threshold of errors is encountered.
- Services can use rate limits to protect themselves from spikes in load beyond their capacity to service.
- Individual services should expose health checks for load balancers and monitoring to be able to use.
- You can effectively validate resiliency by practicing both load and chaos testing.
- You can apply standards—whether through proxies or frameworks—to help engineers "fall into the pit of success" and build services that tolerate faults by default.

Deployment

To bring our designs to life in a sane, safe, and reproducible way, we need to have some infrastructure in place. Adopting Continuous Delivery to get all types of changes into production will allow safer, quicker and a more sustainable way of getting new features out while measuring and reducing risk. In microservice applications, failure cannot be totally eliminated. It is essential to acknowledge this fact as it will help informing the decisions made in order to distribute failure using microservices.

Chapter 5 from *The Tao of Microservices*
by Richard Rodger

Deployment

This chapter covers

- Understanding the nature of failure in complex systems
- Developing a simple mathematical model of failure
- Using frequent low-cost failure to avoid infrequent high-cost failure
- Using continuous delivery to measure and manage risk
- Understanding the deployment patterns for microservices

The organizational decision to adopt the microservice architecture often represents an acceptance that change is necessary and that current work practices aren't delivering. This is an opportunity not only to adopt a more capable software architecture but also to introduce a new set of work practices for that architecture.

You can use microservices to adopt a scientific approach to risk management. Microservices make it easier to measure and control risk, because they give you small units of control. The reliability of your production system then becomes quantifiable, allowing you to move beyond ineffective manual sign-offs as the pri-

mary risk-reduction strategy. Because traditional processes regard software as a bag of features that are either broken or fixed, and don't incorporate the concept of failure thresholds and failure rates, they're much weaker protections against failure.[5]

5.1 Things fall apart

All things fail catastrophically. There's no gradual decline. There's decline, certainly; but when death comes, it comes quickly. Structures need to maintain a minimum level of integrity before they fall apart. Cross that threshold, and the essence is gone.

This is more than poetic symbolism. Disorder always increases.[6] Systems can tolerate some disorder and can even convert chaos into order in the short term; but in the long run, we're all dead, because disorder inevitably forces the system over the threshold of integrity into failure.

What is *failure*? From the perspective of enterprise software, this question has many answers. Most visible are the technical failures of the system to meet uptime requirements, feature requirements, and acceptable performance and defect levels. Less visible, but more important, are failures to meet business goals.

Organizations obsess about technical failures, often causing business failures as a result. The argument of this chapter is that it's better to accept many small failures in order to prevent large-scale catastrophic failures. It's better for 5% of users to see a broken web page than for the business to go bankrupt because it failed to compete in the marketplace.

The belief that software systems can be free from defects and that this is possible through sheer professionalism is pervasive in the enterprise. There's an implicit assumption that perfect software can be built at a reasonable cost. This belief ignores the basic dynamics of the law of diminishing marginal returns: the cost of fixing the next bug grows ever higher and is unbounded. In practice, all systems go into production with known defects. The danger of catastrophe comes from an institutional consensus that it's best to pretend that this isn't the case.

Can the microservice architecture speak to this problem? Yes, because it makes it easier to reduce the risk of catastrophic failure by allowing you to make small changes that have low impact. The introduction of microservices also provides you, as an architect, with the opportunity to reframe the discussion around acceptable failure rates and risk management. Unfortunately, there's no forcing function, and microservice deployments can easily become mired in the traditional risk management approach of enterprise operations. It's therefore essential to understand the possibilities for risk reduction that the architecture creates.

---

[5] To be bluntly cynical, traditional practices are more about territorial defense and blame avoidance than building effective software.

[6] There are more ways to be disorganized than there are ways to be organized. Any given change is more likely to move you further into disorder.

5.2 Learning from history

To understand how software systems fail and how you can improve deployment, you need to understand how other complex systems fail. A large-scale software system is not unlike a large-scale engineering system. There are many components interacting in many ways. With software, you have the additional complication of deployment—you keep changing the system. With something like a nuclear power plant, at least you only build it once. Let's start by examining just such a complex system in production.

5.2.1 Three Mile Island

On March 28, 1979, the second unit of the nuclear power plant located on Three Mile Island near Harrisburg, Pennsylvania, suffered a partial meltdown, releasing radioactive material into the atmosphere.[7]

The accident was blamed on operator error. From a complex systems perspective, this conclusion is neither fair nor useful. With complex systems, failure is inevitable. The question isn't, "Is nuclear energy safe?" but rather, "What levels of accidents and contamination can we live with?" This is also the question we should ask of software systems.

To understand what happened at Three Mile Island, you need to know how a reactor works at a high level, and at a low level where necessary. Your skills as a software architect will serve you well in understanding the explanation that follows. The reactor heats water, turning it into steam. The steam drives a turbine that spins to produce electricity. The reactor heats the water using a controlled fission reaction. The nuclear fuel, uranium, emits neutrons that collide with other uranium atoms, releasing even more neutrons. This chain reaction must be controlled through the absorption of excess neutrons; otherwise, bad things happen.

The uranium fuel is stored in a large, sealed, stainless steel containment vessel, about the height of a three-story building. The fuel is stored as vertical rods, about the height of a single story. Interspersed are control rods made of graphite, that absorb neutrons; to control the reaction, you raise and lower the control rods. The reaction can be completely stopped by lowering all the control rods fully; this is known as *scramming*. This is an obvious safety feature: if there's a problem, pretty much any problem, drop the rods![8] Nuclear reactors are designed with many such automatic safety devices (ASDs) that activate without human intervention on the basis of input signals from sensors. I'm sure you can already see the opportunity for unintended cascading behavior in the ASDs.

The heat from the core (all the stuff inside the containment vessel, including the rods) is extracted using water. This coolant water is radioactive, so you can't use it directly to drive the turbine. You have to use a heat exchanger to transfer the heat to a

[7] For full details, see John G. Kemeny et al., *Report of the President's Commission on the Accident at Three Mile Island* (U.S. Government Printing Office, 1979), http://mng.bz/hwAm.

[8] The technical term *scram* comes from the early days of research reactors. If anything went wrong, you dropped the rods, shouted "Scram!", and ran. Very fast.

set of completely separate water pipes; that water, which isn't radioactive, drives the turbine. You have a primary coolant system with radioactive water and a secondary coolant system with "normal" water. Everything is under high pressure and at a high temperature, including the turbine, which is cooled by the secondary system. The secondary water must be very pure and contain almost no microscopic particles, to protect the turbine blades, which are precision engineered. Observe how complexity lives in the details: a simple fact—that water drives the turbine—hides the complexity that it must be "special" purified water. Time for a high-level diagram: see figure 5.1.

Now let's go a little deeper. That special purified water for the secondary system doesn't happen by magic: you need something called a *condensate polisher* to purify the water, using filters. Like many parts of the system, the condensate polisher's valves, which allow water to enter and leave, are driven by compressed air. That means the plant, in addition to having water pipes for the primary and secondary cooling systems, also has compressed air pipes for a pneumatic system. Where does the special purified water come from? Feed pumps are used to pump water from a local water source—in this case, the Susquehanna River—into the cooling system. There are also emergency tanks, with emergency feed pumps, in case the main feed pumps fail. The valves for these are also driven by the pneumatic system.

We must also consider the core, which is filled with high-temperature radioactive water under high pressure.[9] High-pressure water is extremely dangerous and can damage the containment vessel, and the associated pipework, leading to a dreaded loss-of-containment accident (LOCA). You don't want holes in the containment vessel. To alleviate water pressure in the core, a *pressurizer* is used. This is a large water tank connected to the core and filled about half and half with water and steam. The pressurizer also has a drain, which allows water to be removed from the core. The steam at the top of the pressurizer tank is compressible and acts as a shock absorber. You can control the core pressure by controlling the volume of the water in the lower half of the pressurizer. But you must never, ever allow the water level to reach 100% (referred to as *going solid*): if you do, you'll have no steam and no shock absorber, and the result

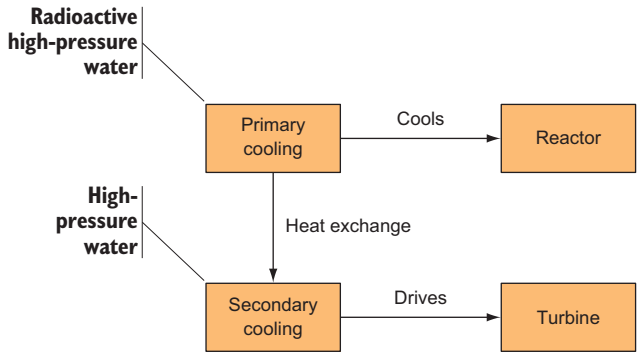


Figure 5.1 High-level components of a nuclear reactor

[9] What fun!

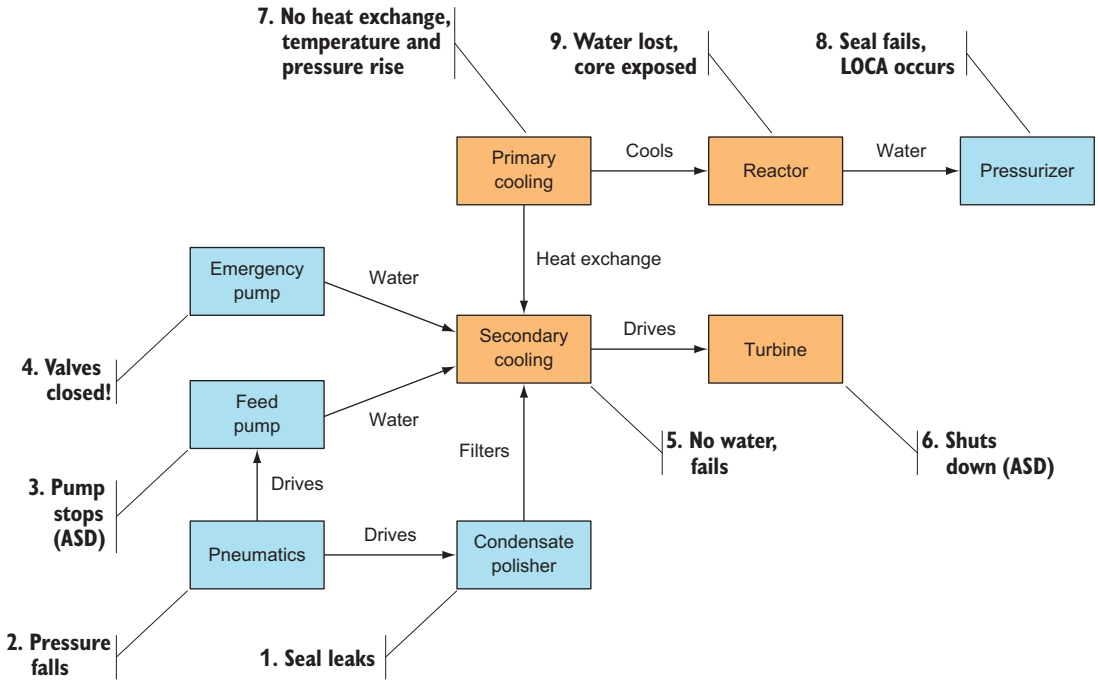


Figure 5.2 A small subset of interactions between high and low levels in the reactor

will be a LOCA because pipes will burst. This fact is drilled into operators from day one. Figure 5.2 shows the expanded diagram.

THE TIMELINE OF THE ACCIDENT

At 4:00 a.m., the steam turbine *tripped*: it stopped automatically because the feed pumps for the secondary cooling system that cools the turbine had stopped. With no water entering the turbine, the turbine was in danger of overheating—and it's programmed to stop under these conditions. The feed pumps had stopped because the pneumatic air system that drives the valves for the pumps became contaminated with water from the condensate polisher. A leaky seal in the condensate polisher had allowed some of the water to escape into the pneumatic system. The end result was that a series of ASDs, operating as designed, triggered a series of ever-larger failures. More was to come.

With the turbine down and no water flowing in the secondary coolant system, no heat could be extracted from the primary coolant system. The core couldn't be cooled. Such a situation is extremely dangerous and, if not corrected, will end with a meltdown.

There was an ASD for this scenario: emergency feed pumps take water from an emergency tank. The emergency pumps kick in automatically. Unfortunately, in this case, the pipes to the emergency pumps were blocked, because two valves had been

accidentally left closed during maintenance. Thus the emergency pumps supplied no water. The complexity of the system as a whole, and its interdependencies, are apparent here; not just the machinery, but also its management and maintenance, are part of the dependency relationship graph.

The system had now entered a cascading failure mode. The steam turbine boiled dry. The reactor scrammed automatically, dropping all the control rods to stop the fission reaction. This didn't reduce the heat to safe levels, however, because the decay products from the reaction still needed to cool down. Normally this takes several days and requires a functioning cooling system. With no cooling, very high temperatures and pressures built up in the containment vessel, which was in danger of breaching.

Naturally, there are ASDs for this scenario. A relief valve, known as the pilot-operated relief valve (PORV), opens under high pressure and allows the core water to expand into the pressurizer vessel. But the PORV is unreliable: valves for high-pressure radioactive water are, unsurprisingly, unreliable, failing about 1 in 50 times. In this case, the PORV opened in response to the high-pressure conditions but then failed to close fully after the pressure was relieved. It's important for operators to know the status of the PORV, and this one had recently been fitted with a status sensor and indicator. This sensor also failed, though, leading the operators to believe that the PORV was closed. The reactor was now under a LOCA, and ultimately more than one third of the primary cooling water drained away. The actual status of the PORV wasn't noticed until a new shift of operators started.

As water drained away, pressure in the core fell, but by too much. Steam pockets formed. These not only block water flow but also are far less efficient at removing heat. So, the core continued to overheat. At this point, Three Mile Island was only *13 seconds* into the accident, and the operators were completely unaware of the LOCA; they saw only a transient pressure spike in the core. Two minutes into the event, pressure dropped sharply as core coolant turned to steam. At this point, the fuel rods in the core were in danger of becoming exposed, because there was barely sufficient water to cover them. Another ASD kicked in—injection of cold, high-pressure water. This is a last resort to save the core by keeping it covered. The problem is that too much cold water can crack the containment vessel. Also, and far worse, too much water makes the pressurizer go solid. Without a pressure buffer, pipes will crack. So the operators, as they had been trained to do, slowed the cold-water injection rate.[10]

The core became partially exposed as a result, and a partial meltdown occurred. Although the PORV was eventually closed and the water was brought under control, the core was badly damaged. Chemical reactions inside the core led to the release of hydrogen gas, which caused a series of explosions; ultimately, radioactive material was released into the atmosphere.[11]

---

[10] Notice that the operators were using a mental model that had diverged from reality. Much the same happens with the operation of software systems under high load.

[11] An excellent analysis of this accident, and many others, can be found in the book *Normal Accidents* (Princeton University Press, 1999) by Charles Perrow. This book also develops a failure model for complex systems that's relevant to software systems.

LEARNING FROM THE ACCIDENT

Three Mile Island is one of the most-studied complex systems accidents. Some have blamed the operators, who "should" have understood what was happening, "should" have closed the valves after maintenance, and "should" have left the high-pressure cold-water injection running.[12] Have you ever left your home and not been able to remember whether you locked the front door? Imagine having 500 front doors—on any given day, in any given reactor, some small percentage of valves will be in the wrong state.

Others blame the accident on sloppiness in the management culture, saying there "should" have been lock sheets for the valves. But since then, adding more paperwork to track work practices has only *reduced* valve errors in other reactors, not eliminated them. Still others blame the design of the reactor: too much complexity, coupling, and interdependence. A simpler design has fewer failure modes. But it's in the nature of systems engineering that there's always hidden complexity, and the final version of initially simple designs becomes complex as a result.

These judgments aren't useful, because they're all obvious and true to some degree. The real learning is that complex systems are fragile and will fail. No amount of safety devices and procedures will solve this problem, because the safety devices and procedures are *part* of the problem. Three Mile Island made this clear: the interactions of all the components of the system (including the humans) led to failure.

There's a clear analogy to software systems. We build architectures that have a similar degree of complexity and the same kinds of interactions and tight couplings. We try to add redundancy and fail-safes but find that these measures fail anyway, because they haven't been sufficiently tested. We try to control risk with detailed release procedures and strict quality assurance. Supposedly, this gives us predictable and safe deployments; but in practice, we still end up having to do releases on the weekend, because strict procedures aren't that effective. In one way, we're worse than nuclear reactors—with every release, we change fundamental core components!

You can't remove risk by trying to contain complexity. Eventually, you'll have a LOCA.

5.2.2 A model for failure in software systems

Let's try to understand the nature of failure in software systems using a simple model. You need to quantify your exposure to risk so that you can understand how different levels of complexity and change affect a system.

A software system can be thought of as a set of components, with dependency relationships between those components. The simplest case is a single component. Under what conditions does the component, and thus the entire system, fail?

To answer that question, we should clarify the term *failure*. In this model, failure isn't an absolute binary condition, but a quantity you can measure. *Success* might be 100%

[12] Or should they? Doing so might have cracked the containment vessel, causing a far worse accident. Expert opinion is conflicted on this point.

uptime over a given period, and *failure* might be any uptime less than 100%. But you might be happy with a failure rate of 1%, setting 99% uptime as the threshold of success. You could count the number of requests that have correct responses: out of every 1,000 requests, perhaps 10 fail, yielding a failure rate of 1%. Again, you might be happy with this. Loosely, we can define the *failure rate* as the proportion of some quantity (continuous or discrete) that fails to meet a specific threshold value. Remember that you're building as simple a model as you can, so what the failure rate is a failure *of* is excluded from the model. You only care about the rate and meeting the threshold. *Failure* means failure to meet the threshold, not failure to operate.

For a one-component system, if the component has a failure rate of 1%, then the system as a whole has a failure rate of 1% (see figure 5.3). Is the system failing?

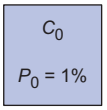


Figure 5.3 A single-component system, where P_0 is the failure rate of component C_0

If the acceptable failure threshold is 0.5%, then the system is failing. If the acceptable failure threshold is 2%, then the system is *not* failing: it's succeeding, and your work is finished.

This model reflects an important change of perspective: accepting that software systems are in a constant state of low-level failure. There's always a failure rate. Valves are left closed. The system as a whole fails only when a threshold of pain is crossed. This new perspective is different from the embedded organizational assumption that software can be perfect and operate without defects. The obsession with tallying defective features seems quaint from this viewpoint. Once you gain this perspective, you can begin to understand how the operational costs of the microservice architecture are outweighed by the benefit of superior risk management.

A TWO-COMPONENT SYSTEM

Now, consider a two-component system (see figure 5.4). One component depends on the other, so both must function correctly for the system to succeed. Let's set the failure threshold at 1%. Perhaps this is the proportion of failed purchases, or maybe you're counting many different kinds of errors; this isn't relevant to the model. Let's also make the simplifying assumption that both components fail independently of each other.[13] The failure of one doesn't make the other more likely to fail. Each component has its own failure rate. In this system, a given function can succeed only if *both* components succeed.

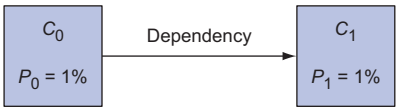


Figure 5.4 A two-component system, where P_i is the failure rate of component C_i

Because the components fail independently, the rules of probability say that you can multiply the probabilities. There are four cases: both fail, both succeed, the first

[13] This assumption is important to internalize. Components are like dice: they don't affect each other, and they have no memory. If one component fails, it doesn't make another component more likely to fail. It may *cause* the other component to fail, but that's different, because the failure has an external cause. We're concerned with internal failure, *independent* of other components.

fails and the second succeeds, or the first succeeds and the second fails. You want to know the failure rate of the system; this is the same as asking for the probability that a given transaction will fail. Of the four cases, three are failing, and one is success. This makes the calculation easier: multiply the success probabilities together to get the probability for the case where the entire system succeeds. The failure probability is found by subtracting the success probability from 1.[14] Keeping the numbers simple, assume that each component has the same failure probability of 1%. The gives an overall failure probability of 1 – (99% x 99%) = 1 - 98.01% = 1.99%.

Despite the fact that both components are 99% reliable, the system as a whole is only 98% reliable and fails to meet the success threshold of 99%. You can begin to see that meeting an overall level of system reliability—where that system is composed of components, all of which are essential to operation—is harder than it looks. Each component must be a lot more reliable than the system as a whole.

Multiple components

You can extend this model to any number of components, as long as the components depend on each other in a serial chain. This is a simplification from the real software architectures we know and love, but let's work with this simple model to build some understanding of failure probabilities. Using the assumption of failure independence, where you can multiply the probabilities together, yields the following formula for the overall probability of failure for a system with an arbitrary number of components in series.

$$P_F = 1 - \prod_{i=1}^{n} (1 - P_i)$$

Here, P_F is the probability of system failure, n is the number of components, and P_i is the probability that component i fails.

If you chart this formula against the number of components in the system, as shown in figure 5.5, you can see that the probability of failure grows quickly with the number of components. Even though each component is reliable at 99% (we've given each component the same reliability to keep things simple), the system is unreliable. For example, reading from the chart, a 10-component system has just under a 10% failure rate. That's a long way from the desired 1%.

The model demonstrates that intuitions about reliability can often be incorrect. A convoy of ships is as slow as its slowest ship, but a software architecture isn't as unreliable as its most unreliable component—it's *much more unreliable*, because the other components can fail, too.

The system in the Three Mile Island reactor definitely wasn't linear. It consisted of a complicated set of components with many interdependencies. Real software is much

[14] The system can be in only one of two states: success or failure. The probabilities of both must sum to 1. This means you can find one state if you can find the other, so you get to choose the one with the easier formula.

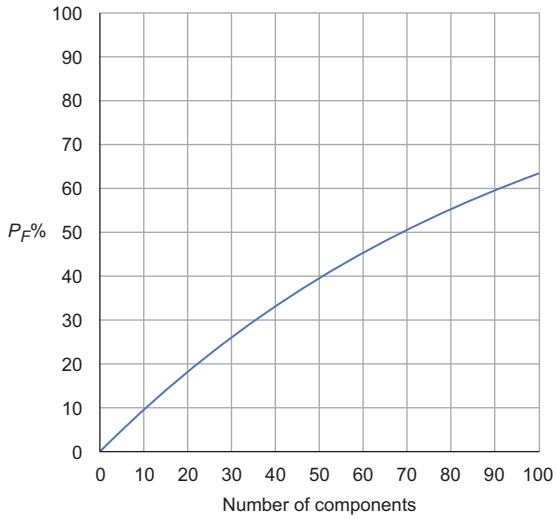


Figure 5.5 **Probability of system failure against the number of components, where all components are 99% reliable**

like Three Mile Island, and software components tend to be even more tightly coupled, with no tolerance for errors. Let's extend the model to see how this affects reliability. Consider a system with four components, one of which is a subcomponent that isn't on the main line (see figure 5.6). Three have a serial dependency, but the middle component also depends on the fourth.

Again, each of the four components has the same 99% reliability. How reliable is the system as a whole? You can solve the serial case with the formula introduced earlier. The reliability of the middle component must take into account its dependency on the fourth component. This is a serial system as well, contained inside the main system. It's a two-component system, and you've seen that this has a reliability of 100% - 1.99% = 98.01%. Thus, the failure probability of the system as a whole is 1 – (99% x 98.01% x 99%) = 1 – 96.06% = 3.94%.

What about an arbitrary system with many dependencies, or systems where multiple components depend on the same subcomponent? You can make another simplifying assumption to handle this case: assume that all components are necessary, and there are no redundancies. Every component must work. This seems unfair, but think of how the

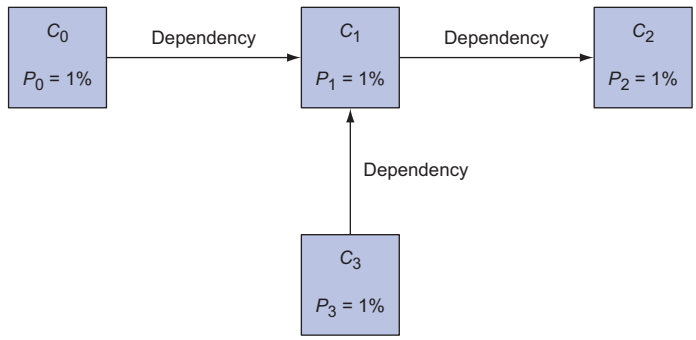


Figure 5.6 **A nonlinear four-component system**

Three Mile Island accident unfolded. Supposedly redundant systems such as the emergency feed pumps turned out to be important as standalone components. Yes, the reactor could work without them, but it was literally an accident waiting to happen.

If all components are necessary, then the dependency graph can be *ignored*. Every component is effectively on the main line. It's easy to overlook subcomponents or assume that they don't affect reliability as much, but that's a mistake. Interconnected systems are much more vulnerable to failure than you may think, because there are many subcomponent relationships. The humans who run and build the system are one such subcomponent relationship—you can only blame "human error" for failures if you consider humans to be part of the system. Ignoring the dependency graph only gives you a first-order approximation of the failure rate, using the earlier formula; but given how quickly independent probabilities compound, that estimate is more than sufficient.

5.2.3 Redundancy doesn't do what you think it does

You can make your systems more reliable by adding redundancy. Instead of *one* instance of a component that might fail, have many. Keeping to the simple model where failures are independent, this makes the system much more reliable. To calculate the failure probability of a set of redundant components, you multiply the individual failure probabilities, because all must fail in order for the entire assemblage to fail.[15] Now, you find that probability theory is your friend. In the single-component system, adding a second redundant component gives you a failure rate of 1% x 1% = 0.01%.

It seems that all you need to do is add lots of redundancy, and your problems go away. Unfortunately, this is where the simple model breaks down. There are few failure modes in a software system where failure of one instance of a component is independent of other components of the same kind. Yes, individual host machines can fail,[16] but most failures affect all software components equally. The data center is down. The network is down. The same bug applies to all instances. High load causes instances to fall like dominoes, or to flap.[17] A deployment of a new version fails on production traffic.

Simple models are also useful when they break, because they can reveal hidden assumptions. Load balancing over multiple instances doesn't give you strong redundancy, it gives you capacity. It barely moves the reliability needle, because multiple instances of the same component are *not* independent.[18]

---

[15] The failure probability formula, in this case, is $P_F = \prod_{i=1}^{n} P_i$.

[16] Physical power supplies fail all the time, and so do hard drives; network engineers will keep stepping on cables from now until the end of the universe; and we'll never solve the Halting Problem (it's mathematically impossible to prove that any given program will halt instead of executing forever—you can thank Mr. Alan Turing for that), so there will always be input that triggers infinite loops.

[17] *Flapping* occurs when services keep getting killed and restarted by the monitoring system. Under high load, newly started services are still *cold* (they have empty caches), and their tardiness in responding to requests is interpreted as failure, so they're killed. Then more services are started. Eventually, there are no services that aren't either starting or stopping, and work comes to a halt.

[18] The statement that multiple instances of the same software component don't fail independently is proposed as an empirical fact from the observed behavior of real systems; it isn't proposed as a mathematical fact.

Automatic safety devices are unreliable

Another way to reduce the risk of component failure is to use ASDs. But as you saw in the story of Three Mile Island, these bring their own risks. In the model, they're additional components that can themselves fail.

Many years ago, I worked on a content-driven website. The site added 30 or 40 news stories a day. It wasn't a breaking news site, so a small delay in publishing a story was acceptable. This gave me the brilliant idea to build a 60-second cache. Most pages could be generated once and cached for 60 seconds. Once expired, any news updates appeared on the regenerated pages, and the next 60-second caching period would begin.

This seemed like a cheap way to build what was effectively an ASD for high load. The site would be able to handle things like election day results without needing to increase server capacity much.

The 60-second cache was implemented as an in-memory cache on each web server. It was load tested, and everything appeared to be fine. But in production, servers kept crashing. Of course, there was a memory leak; and, of course, it didn't manifest unless we left the servers running for at least a day, storing more than 1,440 copies of each page, for each article, in memory. The first week we went live was a complete nightmare—we babysat dying machines on a 24/7 rotation.

5.2.4 Change is scary

Let's not throw out the model just yet. Software systems aren't static, and they suffer from catastrophic events known as *deployments*. During a deployment, many components are changed simultaneously. In many systems, this can't be done without downtime. Let's model this as a simultaneous change of a random subset of components. What does this do to the reliability of the system?

By definition, the reliability of a component is the measured rate of failure in production. If a given component drops only 1 work item in 100, it has 99% reliability. Once a deployment is completed and has been live for a while, you can measure production to get the reliability rate. But this isn't much help in advance. You want to know the probability of failure of the new system *before* the changes are made.

Our model isn't strong enough to provide a formula for this situation. But you can use another technique: Monte Carlo simulation. You run lots of simulations of the deployment and add up the numbers to see what happens. Let's use a concrete example. Assume you have a four-component system, and the new deployment consists of updates to all four components. In a static state, before deployment, the reliability of the system is given by the standard formula: $0.99^4 = .9605 = 96.1\%$.

To calculate the reliability after deployment, you need to estimate the *actual* reliabilities of each component. Because you don't know what they are, you have to *guess* them. Then you run the formula using the guesses.

If you do this many times, you'll be able to plot the distribution of system reliability. You can say things like, "In 95% of simulations, the system has at least 99% reliability.

Deploy!" Or, "In only 1% of simulations, the system has at least 99% reliability. Unplanned downtime ahead!" Bear in mind that these numbers are just for discussion; you'll need to decide on numbers that reflect your organization's risk tolerance.

How do you guess the reliability of a component? You need to do this in a way that makes the simulation useful. Reliability isn't normally distributed, like a person's height.[19] Reliability is skewed because components are mostly reliable—most components come in around 99% and can't go much higher. There's a lot of space below 99% in which to fail. Your team is doing unit testing, staging, code reviews, and so on. The QA department has to sign off on releases, and the head of QA is pretty strict. There's a high probability that your components are, in fact, reliable; but you can't test for everything, and production is a crueler environment than a developer's laptop or a staging system.

You can use a skewed probability distribution[20] to model "mostly reliable." The chart in figure 5.7 shows how the failure probabilities are distributed. To make a guess, pick a random number between 0 and 1, and plot its corresponding probability. You can see that most guesses will give a low failure probability.

For each of the four components, you get a reliability estimate. Multiply these together in the usual manner. Now, do this many times; over many simulation runs, you can chart the reliability of the system. Figure 5.8 shows the output from a sample exercise.[21] Although the system is often fairly reliable, it has mostly poor reliability compared to a static system. In only 0.15% of simulations does the system have reliability of 95% or more.

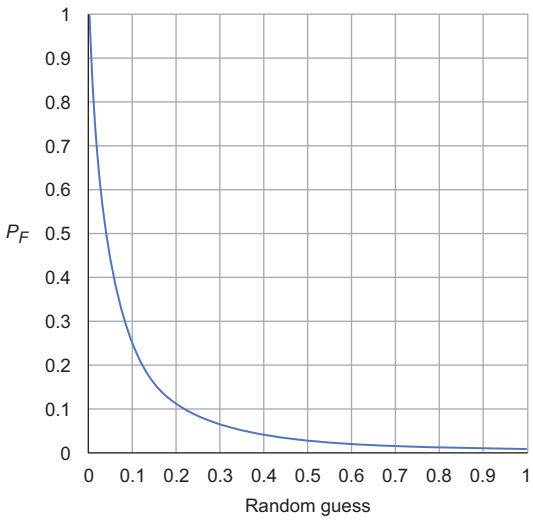


Figure 5.7 A skewed estimator of failure probability

[19] The normal distribution assumes that any given instance will be close to the average and has as much chance of being above average as below.

[20] The Pareto distribution is used in this example, because it's a good model for estimating failure events.

[21] In the sample exercise, 1,000 runs were executed and then categorized into 5% intervals.

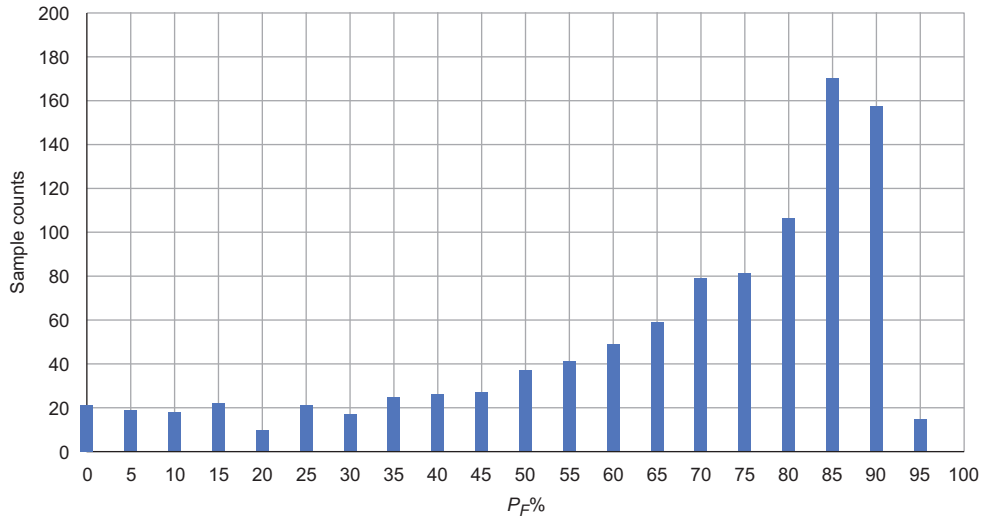


Figure 5.8 **Estimated reliability of the system when four components change simultaneously**

The model shows that simultaneous deployment of multiple components is inherently risky: it almost always fails the first time. That's why, in practice, you have *scheduled* downtime or end up frantically working on the weekend to complete a deployment. You're really making multiple repeated deployment attempts, trying to resolve production issues that are almost impossible to predict.

The numbers don't work in your favor. You're playing a dangerous game. Your releases may be low frequency, but they have high risk.[22] And it seems as though microservices must introduce even more risk, because you have many more components. Yet, as you'll discover in this chapter, microservices also provide the flexibility for a solution. If you're prepared to accept high-frequency releases of single components, then you'll get much lower risk exposure.

I've labored the mathematics to make a point: no software development methodology can defy the laws of probability at reasonable cost. Engineering, not politics, is the key to risk management.

5.3 The centre cannot hold

The collective delusion of enterprise software development is that perfect software can be delivered complete and on time, and deployed to production without errors, by force of management. Any defects are a failure of professionalism on the part of the team. Everybody buys into this delusion. Why?

This book does *not* take the trite and lazy position that it's all management's fault. I don't pull punches when it comes to calling out bad behavior, but you must be careful to see organizational behavior for what it is: rational.

[22] The story of the deployment failure suffered by Knight Capital, in chapter 1, is a perfect example of this danger.

We can analyze corporate politics using Game Theory.[23] Why does nobody point out the absurdities of enterprise software development, even when there are mountains of evidence? How many more books must be written on the subject? Fortunately, we live in an age where the scale of the software systems we must build is slowly forcing enterprise software development to face reality.

Traditional software development processes are an unwanted Nash equilibrium in the game of corporate politics. They're a kind of prisoner's dilemma.[24] If all stakeholders acknowledged that failure rates must exist and used that as a starting point, then continuous delivery would be seen as a natural solution. But nobody is willing to do so; it would be a career-limiting move. Failure isn't an option! So we're stuck with a collective delusion because we can't communicate honestly. This book aims to give you some solid principles to start that honest communication.

> **WARNING** It isn't advisable to push for change unless there's a forcing function. Wait until failure is inevitable under the old system, and then be the white knight. Pushing for change when you have no leverage is indeed a career-limiting move.

5.3.1 The cost of perfect software

The software that controlled the space shuttle was some of the most perfect software ever written. It's a good example of how expensive such software truly is and calls out the absurdity of the expectations for enterprise software. It's also a good example of how much effort is required to build redundant software components.

The initial cost estimate for the shuttle software system was $20 million. The final bill was $200 million. This is the first clue that defect-free software is an order of magnitude more expensive than even software engineers estimate. The full requirements specification has 40,000 pages—for a mere 420,000 lines of code. By comparison, Google's Chrome web browser has more than 5 million lines of code. How perfect is the shuttle software? On average, there was one bug per release. It wasn't completely perfect!

The shuttle's software development process was incredibly strict. It was a traditional process with detailed specifications, strict testing, and code reviews, and bureaucratic signatures were needed for release. Many stakeholders in the enterprise software development process believe that this level of delivery is what they're going to get.

---

[23] The part of mathematics that deals with multiplayer games and the limitations of strategies to maximize results.

[24] A *Nash equilibrium* is a state in a game where no player can improve their position by changing strategy unilaterally. The prisoner's dilemma is a compact example: two guilty criminals who robbed a bank together are captured by the police and placed in separate cells where they can't communicate. If they both stay silent, then they both get a one-year sentence for possession of stolen cash, but the police can't prove armed robbery. The police offer a deal separately to each criminal: confess and betray your accomplice, and instead of three years for armed robbery, you'll only get two years, because you cooperated. The only rational strategy is for each criminal to betray the other and take the two years, because their partner might betray them. Because they can't communicate, they can't agree to remain silent.

It's the business of business to make return-on-investment decisions. You spend money to make money, but you must have a business case. This system breaks down if you don't understand your cost model. It's the job of the software architect to make these costs clear and to provide alternatives, where the cost of software development is matched to the expected returns of the project.

5.4 Anarchy works

The most important question in software development is, "What is the acceptable error rate?" This is the first question to ask at the start of a project. It drives all other questions and decisions. It also makes clear to all stakeholders that the process of software development is about controlling, not conquering, failure.

The primary consequence is that large-scale releases can never meet the acceptable error rate. Reliability is so compromised by the uncertainty of a large release that large releases must be rejected as an engineering approach. This is just mathematics, and no amount of QA can overcome it.

Small releases are less risky. The smaller, the better. Small releases have small uncertainties, and you can stay beneath the failure threshold. Small releases also mean frequent releases. Enterprise software must constantly change to meet market forces. These small releases must go all the way to production to fully reduce risk; collecting them into large releases takes you back to square one. That's just how the probabilities work.

A system that's under constant failure isn't fragile. Every component expects others to fail and is built to be tolerant of failure. The constant failure of components exercises redundant systems and backups so you know they work. You have an accurate measure of the failure rate of the system: it's a known quantity that can be controlled. The rate of deployment can be adjusted as risks grow and shrink.

How does the simple risk model work under these conditions? You may be changing only one component at a time, but aren't you still subject to large amounts of risk? You know that your software development process won't deliver updated components that are as stable as those that have been baked into production for a while.

Let's say updated components are 80% reliable on first deployment. The systems we've looked at definitely won't meet a reliability threshold of 99%. Redeploying a single component still isn't a small enough deployment. This is an engineering and process problem that we'll address in the remainder of this chapter: how to make changes to a production software system while maintaining the desired risk tolerance.

5.5 Microservices and redundancy

An individual component of a software system should never be run as a single instance. A single instance is vulnerable to failure: the component could crash, the machine it's running on could fail, or the network connection to that machine could be accidentally misconfigured. No component should be a single point of failure.

To avoid a single point of failure, you can run multiple instances of the component. Then, you can handle load, and you're more protected against some kinds of failure. You aren't protected against software defects in the component, which affect all instances, but such defects can usually be mitigated by automatic restarts.[25] Once a component has been running in production for a while, you'll have enough data to get a good measure of its reliability.

How do you deploy a new version of a component? In the traditional model, you try, as quickly as possible, to replace all the old instances with a full set of new ones. The blue-green deployment strategy, as it's known, is an example of this. You have a running version of the system; call this the *blue* version. You spin up a new version of the system; call this the *green* version. Then, you choose a specific moment to redirect all traffic from blue to green. If something goes wrong, you can quickly switch back to blue and assess the damage. At least you're still up.

One way to make this approach this less risky is to initially redirect only a small fraction of traffic to green. If you're satisfied that everything still works, redirect greater volumes of traffic until green has completely taken over.

The microservice architecture makes it easy to adopt this strategy and reduce risk even further. Instead of spinning up a full quota of new instances of the green version of the service, you can spin up one. This new instance gets a small portion of all production traffic while the existing blues look after the main bulk of traffic. You can observe the behavior of the single green instance, and if it's badly behaved, you can decommission it: a small amount of traffic is affected, and there's a small increase in failure, but you're still in control. You can fully control the level of exposure by controlling the amount of traffic you send to that single new instance.

Microservice deployments consist of nothing more than introducing a single new instance. If the deployment fails, rollback means decommissioning a single instance. Microservices give you well-defined primitive operations on your production system: add/remove a service instance.[26] Nothing more is required. These primitive operations can be used to construct any deployment strategy you desire. For example, blue-green deployments break down into a list of add and remove operations on specific instances.

Defining a primitive operation is a powerful mechanism for achieving control. If everything is defined in terms of primitives, and you can control the composition of the primitives, then you can control the system. The microservice instance is the primitive and the unit with which you build your systems. Let's examine the journey of that unit from development to production.

---

[25] Restarts don't protect you against nastier kinds of defects, such as poison messages.

[26] More formally, we might call these primitive operations *activate* and *deactivate*, respectively. How the operations work depends entirely on the underlying deployment platform.

5.6 Continuous delivery

The ability to safely deploy a component to production at any time is powerful because it lets you control risk. *Continuous delivery* (CD) in a microservice context means the ability to create a specific version of a microservice and to run one or more instances of that version in production, on demand. The essential elements of a CD pipeline are as follows:

- A *version-controlled local development environment* for each service, supported by unit testing, and the ability to test the service against an appropriate local subset of the other services, using mocking if necessary.
- A *staging environment* to both validate the microservice and build, reproducibly, an artifact for deployment. Validation is automated, but scope is allowed for manual verification if necessary.
- A *management system*, used by the development team to execute combinations of primitives against staging and production, implementing the desired deployment patterns in an automated manner.
- A *production environment* that's constructed from deployment artifacts to the fullest extent possible, with an audit history of the primitive operations applied. The environment is self-correcting and able to take remedial action, such as restarting crashed services. The environment also provides intelligent load balancing, allowing traffic volumes to vary between services.
- A *monitoring and diagnostics system* that verifies the health of the production system after the application of each primitive operation and allows the development team to introspect and trace message behavior. Alerts are generated from this part of the system.

The pipeline assumes that the generation of defective artifacts is a common occurrence. The pipeline attempts to filter them out at each stage. This is done on a per-artifact basis, rather than trying to verify an update to the entire system. As a result, the verification is both more accurate and more credible, because confounding factors have been removed.

Even when a defective artifact makes it to production, this is considered a normal event. The behavior of the artifact is continuously verified in production after deployment, and the artifact is removed if its behavior isn't acceptable. Risk is controlled by progressively increasing the proportion of activity that the new artifact handles.

CD is based on the reality of software construction and management. It delivers the following:

- *Lower risk of failure* by favoring low-impact, high-frequency, single-instance deployments over high-impact, low-frequency, multiple-instance deployments.
- *Faster development* by enabling high-frequency updates to the business logic of the system, giving a faster feedback loop and faster refinement against business goals.

- *Lower cost of development*, because the fast feedback loop reduces the amount of time wasted on features that have no business value.
- *Higher quality*, because less code is written overall, and the code that's written is immediately verified.

The tooling to support CD and the microservice architecture is still in the early stages of development. Although an end-to-end CD pipeline system is necessary to fully gain the benefits of the microservice architecture, it's possible to live with pipeline elements that are less than perfect.

At the time of writing, all teams working with this approach are using multiple tools to implement different aspects of the pipeline, because comprehensive solutions don't exist. The microservice architecture requires more than current platform-as-a-service (PaaS) that vendors offer. Even when comprehensive solutions emerge, they will present trade-offs in implementation focus.[27]

You'll probably continue to need to put together a context-specific toolset for each microservice system you build; as we work through the rest of this chapter, focus on the desirable properties of these tools. You'll almost certainly also need to invest in developing some of your own tooling—at the very least, integration scripts for the third-party tools you select.

5.6.1 Pipeline

The purpose of the CD pipeline is to provide feedback to the development team as quickly as possible. In the case of failure, that feedback should indicate the nature of the failure; it must be easy to see the failing tests, the failing performance results, or the failed integrations. You also should be able to see a history of the verifications and failures of each microservice. This isn't the time to roll your own tooling—many capable continuous integration (CI) tools are available.[28] The key requirement is that your chosen tool be able to handle many projects easily, because each microservice is built separately.

The CI tool is just one stage of the pipeline, usually operating before a deployment to the staging systems. You need to be able to trace the generation of microservices throughout the pipeline. The CI server generates an artifact that will be deployed into production. Before that happens, the source code for the artifact needs to be marked and tagged so that artifact generation can be *hermetic*—you must be able to reproduce any build from the history of your microservice. After artifact generation, you must be able to trace the deployment of the artifact over your systems from staging to production. This tracing must be not only at the system level, but also within the system, tracing the number of instances run, and when. Until third-party tooling solves this problem, you'll have to build this part of the pipeline diagnostics yourself; it's an essential and worthwhile investment for investigating failures.

---

[27] The Netflix suite (http://netflix.github.io) is a good example of a comprehensive, but opinionated, toolchain.

[28] Two quick mentions: if you want to run something yourself, try Hudson (http://hudson-ci.org); if you want to outsource, try Travis CI (http://travis-ci.org).

The unit of deployment is a microservice, so the unit of movement through the pipeline is a microservice. The pipeline should prioritize the focus on the generation and validation of artifacts that represent a microservice. A given version of a microservice is instantiated as a fixed artifact that never changes. Artifacts are *immutable*: the same version of a microservice always generates the same artifact, at a binary-encoding level. It's natural to store these artifacts for quick access.[29] Nonetheless, you need to retain the ability to hermetically rebuild any version of a microservice, because the build process is an important element of defect investigation.

The development environment also needs to make the focus on individual microservices fluid and natural. In particular, this affects the structure of your source code repositories. (We'll look at this more deeply in chapter 7.) Local validation is also important, as the first measure of risk. Once a developer is satisfied that a viable version of the microservice is ready, the developer initiates the pipeline to production.

The staging environment reproduces the development-environment validation in a controlled environment so that it isn't subject to variances in local developer machines. Staging also performs scaling and performance tests and can use multiple machines to simulate production, to a limited extent. Staging's core responsibility is to generate an artifact that has an estimated failure risk that's within a defined tolerance.

Production is the live, revenue-generating part of the pipeline. Production is updated by accepting an artifact and a deployment plan and applying that deployment plan under measurement of risk. To manage risk, the deployment plan is a progressive execution of deployment primitives—activating and deactivating microservice instances. Tooling for production microservices is the most mature at present because it's the most critical part of the pipeline. Many orchestration and monitoring tools are available to help.[30]

5.6.2 Process

It's important to distinguish *continuous delivery* from *continuous deployment*. Continuous deployment is a form of CD where commits, even if automatically verified, are pushed directly and immediately to production. CD operates at a coarser grain: sets of commits are packaged into immutable artifacts. In both cases, deployments can effectively take place in real time and occur multiple times per day.

CD is more suited to the wider context of enterprise software development because it lets teams accommodate compliance and process requirements that are difficult to change within the lifetime of the project. CD is also more suited to the microservice architecture, because it allows the focus to be on the microservice rather than code.

If we view "continuous delivery" as meaning continuous delivery of microservice instances, this understanding drives other virtues. Microservices should be kept small

[29] Amazon S3 isn't a bad place to store them. There are also more-focused solutions, such as JFrog Artifactory (www.jfrog.com/artifactory).

[30] Common choices are Kubernetes (https://kubernetes.io), Mesos (http://mesos.apache.org), Docker (www.docker.com), and so forth. Although these tools fall into a broad category, they operate at different levels of the stack and aren't mutually exclusive. The case study in chapter 9 uses Docker and Kubernetes.

so that verification—especially human verification, such as code reviews—is possible with the desired time frames of multiple deployments per day.

5.6.3 Protection

The pipeline protects you from exceeding failure thresholds by providing measures of risk at each stage of production. It isn't necessary to extract a failure-probability prediction from each measure.[31] You'll know the feel of the measures for your system, and thus you can use a scoring approach just as effectively.

In development, the key risk-measuring tools are code reviews and unit tests. Using modern version control for branch management [32] means you can adopt a development workflow where new code is written on a branch and then merged into the mainline. The merge is performed only if the code passes a review. The review can be performed by a peer, rather than a senior developer: peer developers on a project have more information and are better able to assess the correctness of a merge. This workflow means code review is a normal part of the development process and very low friction. Microservices keep the friction even lower because the units of review are smaller and have less code.

Unit tests are critical to risk measurement. You should take the perspective that unit tests must pass before branches can be merged or code committed on the mainline. This keeps the mainline potentially deployable at all times, because a build on staging has a good chance of passing. Unit tests in the microservice world are concerned with demonstrating the correctness of the code; other benefits of unit testing, such as making refactoring safer, are less relevant.

Unit tests aren't sufficient for accurate risk measurement and are very much subject to diminishing marginal returns: moving from 50% test coverage to 100% reduces deployment risk much less than moving from 0% to 50%. Don't get suckered into the fashion for 100% test coverage—it's a fine badge of honor (literally!) for open source utility components but is superstitious theater for business logic.

On the staging system, you can measure the behavior of a microservice in terms of its adherence to the message flows of the system. Ensuring that the correct messages are sent by the service, and that the correct responses are given, is also a binary pass/fail test, which you can score with a 0 or 1. The service must meet expectations fully. Although these message interactions are tested via unit tests in development, they also need to be tested on the staging system, because this is a closer simulation of production.

Integrations with other parts of the system can also be tested as part of the staging process. Those parts of the system that aren't microservices, such as standalone databases, network services such a mail servers, external web service endpoints, and others, are simulated or run in small scale. You can then measure the microservice's behavior with respect to them. Other aspects of the service, such as performance,

[31] You could use statistical techniques such as Bayesian estimation to do this if desired.

[32] Using a distributed version control system such as Git is essential. You need to be able to use pull requests to implement code reviews.

resource consumption, and security, need to be measured statistically: take samples of behavior, and use them to predict the risk of failure.

Finally, even in production, you must continue to measure the risk of failure. Even before going into production, you can establish manual gates—formal code reviews, penetration testing, user acceptance, and so forth. These may be legally unavoidable (due to laws affecting your industry), but they can still be integrated into the continuous delivery mindset.

Running services can be monitored and sampled. You can use key metrics, especially those relating to message flow rates, to determine service and system health. Chapter 6 has a great deal more to say about this aspect of the microservice architecture.

5.7 Running a microservice system

The tooling to support microservices is developing quickly, and new tools are emerging at a high rate. It isn't useful to examine in detail something that will soon be out of date, so this chapter focuses on general principles; that way, you can compare and assess tools and select those most suitable for your context. You should expect and prepare to build some tooling yourself. This isn't a book on deployment in general, so it doesn't discuss best practices for deploying system elements, such as database clusters, that aren't microservices. It's still recommended that these be subject to automation and, if possible, controlled by the same tooling. The focus of this chapter is on the deployment of your own microservices. Your own microservices implement the business logic of the system and are thus subject to a higher degree of change compared to other elements.

5.7.1 Immutability

It's a core principle of the approach described here that microservice artifacts are immutable. This preserves their ability to act as primitive operations. A microservice artifact can be a container, a virtual machine image, or some other abstraction.[33] The essential characteristics of the artifact are that it can't be changed internally and that it has only two states: active and inactive.

The power of immutability is that it excludes side effects from the system. The behavior of the system and microservice instances is more predictable, because you can be sure they aren't affected by changes you aren't aware of. An immutable artifact contains everything the microservice needs to run, at fixed versions. You can be absolutely sure that your language-platform version, libraries, and other dependencies are exactly as you expect. Nobody can manually log in to the instance and make unaudited changes. This predictability allows you to calibrate risk estimations more accurately.

Running immutable instances also forces you to treat microservices as disposable. An instance that develops a problem or contains a bug can't be "fixed"; it can only be deactivated and replaced by a new instance. Matching capacity to load isn't about building new installations on bigger machines, it's about running more instances of

[33] For very large systems, you might even consider an AWS autoscaling group to be your base unit.

the artifact. No individual instance is in any way special. This approach is a basic building block for building reliable systems on unreliable infrastructure.

The following subsections provide a reference overview of microservice deployment patterns. You'll need to compare these patterns against the capabilities of the automation tooling you're using. Unfortunately, you should expect to be disappointed, and you'll need to augment your tooling to fully achieve the desired benefits of the patterns.

Feel free to treat these sections as a recipe book for cooking up your own patterns rather than a prescription. You can skim the deployment patterns without guilt.[34] To kick things off, figure 5.9 is a reminder of the diagramming conventions. In particular, the number of live instances is shown in braces ({1}) above the name of the microservice, and the version is shown below (1.0.0).

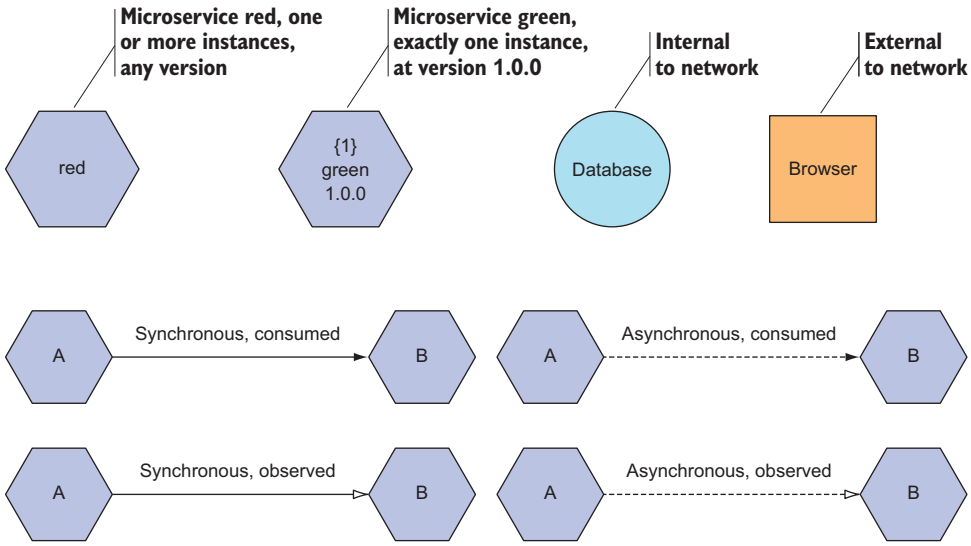


Figure 5.9 Microservice diagram key

Rollback pattern

This deployment pattern is used to recover from a deployment that has caused one or more failure metrics to exceed their thresholds. It enables you to deploy new service versions where the estimated probability of failure is higher than your thresholds, while maintaining overall operation within thresholds.

To use the Rollback pattern (figure 5.10), apply to a system the *activate* primitive for a given microservice artifact. Observe failure alerts, and then *deactivate* the same artifact. Deactivation can be manual or automatic; logs should be preserved. Deactivation should return the system to health. Recovery is expected but may not occur in cases where the offending instance has injected defective messages into the system (for this, see the Kill Switch pattern).

[34] In production, I'm fondest of the Progressive Canary pattern.

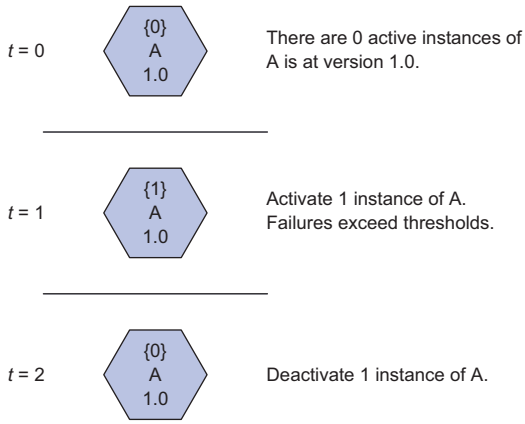


Figure 5.10 Rollback pattern sequence

HOMEOSTASIS PATTERN

This pattern (figure 5.11) lets you maintain desired architectural structure and capacity levels. You implement a declarative definition of your architecture, including rules for increasing capacity under load, by applying activation and deactivation primitives to the system. Simultaneous application of primitives is permitted, although you must take care to implement and record this correctly. Homeostasis can also be implemented by allowing services to issue primitive operations, and defining local rules for doing so (see the Mitosis and Apoptosis patterns).

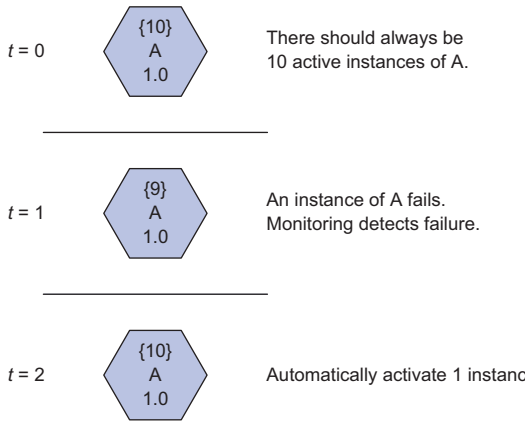


Figure 5.11 Homeostasis pattern sequence

HISTORY PATTERN

The History pattern (figure 5.12) provides diagnostic data to aid your understanding of failing and healthy system behavior. Using this pattern, you maintain an audit trail of the temporal order of primitive operation application—a log of what was activated/deactivated, and when. A complication is that you may allow simultaneous application of sets of primitives in your system.

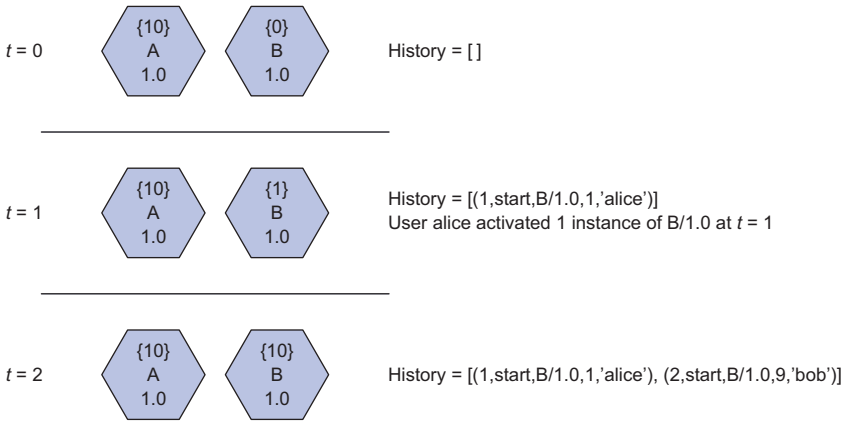


Figure 5.12 History pattern sequence

The audit history lets you diagnose problems by inspecting the behavior of previous versions of the system—these can be resurrected by applying the primitives to simulated systems. You can also deal with defects that are introduced but not detected immediately by moving backward over the history.

5.7.2 Automation

Microservice systems in production have too many moving parts to be managed manually. This is part of the trade-off of the architecture. You must commit to using tooling to automate your system—and this is a never-ending task. Automation doesn't cover all activities from day one, nor should it, because you need to allocate most of your development effort to creating business value. Over time, you'll need to automate more and more.

To determine which activity to automate next, divide your operational tasks into two categories. In the first category, Toil,[35] place those tasks where effort grows at least linearly with the size of the system. To put it another way, from a computational complexity perspective, these are tasks where human effort is at least $O(n)$, where n is the number of microservice types (not instances). For example, configuring log capture for a new microservice type might require manual configuration of the log-collection subsystem. In the second category, Win, place tasks that are less than $O(n)$ in the number of microservice types: for example, adding a new database secondary reader instance to handle projected increased data volumes.

The next task to automate is the most annoying task from the Toil pile, where *annoying* means "most negatively impacting the business goals." Don't forget to include failure risk in your calculation of negative impact.

Automation is also necessary to execute the microservice deployment patterns. Most of the patterns require the application of large numbers of primitive operations

[35] This usage of the term originates with the Google Site Reliability Engineering team.

over a scheduled period of time, under observation for failure. These are tasks that can't be performed manually at scale.

Automation tooling is relatively mature and dovetails with the requirements of modern, large-scale enterprise applications. A wide range of tooling options are available, and your decision should be driven by your comfort levels with custom modification or scripting; you'll need to do some customization to fully execute the microservice deployment patterns described next.[36]

CANARY PATTERN

New microservices and new versions of existing microservices introduce considerable risk to a production system. It's unwise to deploy the new instances and immediately allow them to take large amounts of load. Instead, run multiple instances of known-good services, and slowly replace these with new ones.

The first step in the Canary pattern (figure 5.13) is to validate that the new microservice both functions correctly and isn't destructive. To do so, you activate a single new instance and direct a small amount of message traffic to this instance. Then, watch your metrics to make sure the system behaves as expected. If it doesn't, apply the Rollback pattern.

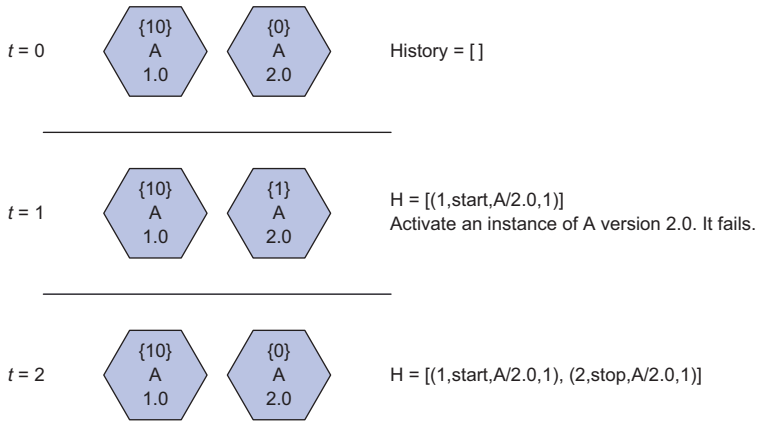


Figure 5.13 Canary pattern sequence

PROGRESSIVE CANARY PATTERN

This pattern (figure 5.14) lets you reduce the risk of a full update by applying changes progressively in larger and larger tranches. Although Canary can validate the safety of a single new instance, it doesn't guarantee good behavior at scale, particularly with respect to unintended destructive behavior. With the Progressive Canary pattern, you deploy a progressively larger number of new instances to take progressively more traffic, continuing to validate during the process. This balances the need for full

[36] Plan to evaluate tools such as Puppet (https://puppet.com), Chef (www.chef.io/chef), Ansible (www.ansible.com), Terraform (www.terraform.io), and AWS CodeDeploy (https://aws.amazon.com/codedeploy).

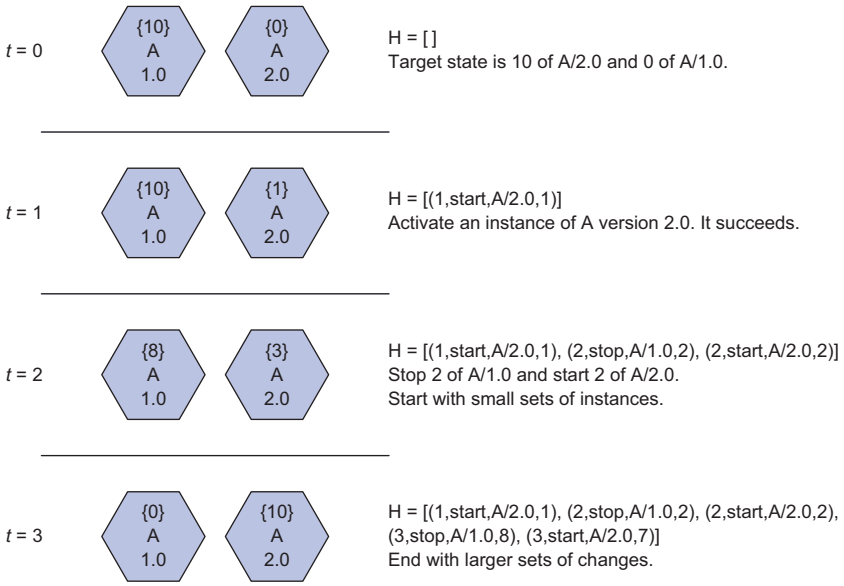


Figure 5.14 Progressive Canary pattern sequence

deployment of new instance versions to occur at reasonable speed with the need to manage the risk of the change.

Primitives are applied concurrently in this pattern. The Rollback pattern can be extended to handle decommissioning of multiple instances if a problem does arise.

BAKE PATTERN

The Bake pattern (figure 5.15) reduces the risk of failures that have severe downsides. It's a variation of Progressive Canary that maintains a full complement of existing instances but also sends a copy of inbound message traffic to the new instances. The output from the new instances is compared with the old to ensure that deviations are below the desired threshold. Output from the new instances is discarded until this criterion is met. The system can continue in this configuration, validating against production traffic, until sufficient time has passed to reach the desired risk level.

This pattern is most useful when the output must meet a strict failure threshold and where failure places the business at risk. Consider using Bake when you're dealing with access to sensitive data, financial operations, and resource-intensive activities that are difficult to reverse.[37] The pattern does require intelligent load balancing and additional monitoring to implement.

[37] The canonical description of this technique is given by GitHub's Zach Holman in his talk "Move Fast & Break Nothing," October 2014 (https://zachholman.com/talk/move-fast-break-nothing). It isn't necessary to fully replicate the entire production stack; you only need to duplicate a sample of production traffic to measure correctness within acceptable levels of risk.

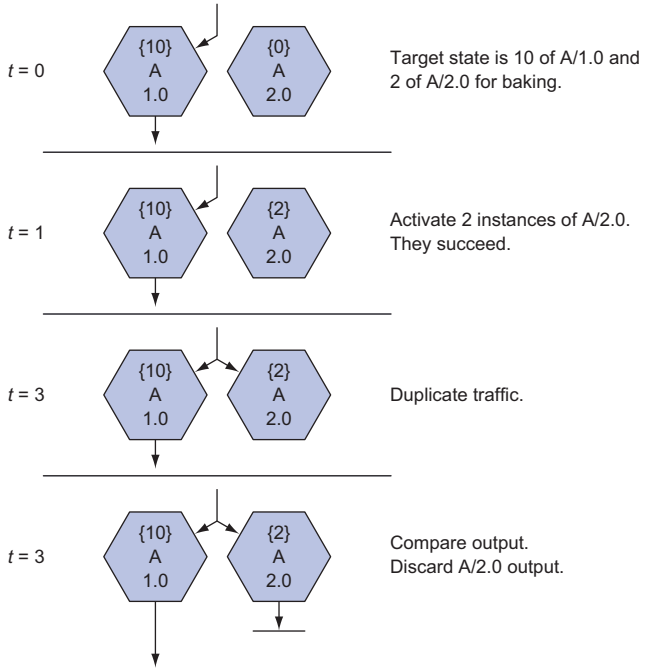


Figure 5.15 Bake pattern sequence

MERGE PATTERN

Performance is impacted by network latency. As the system grows and load increases, certain message pathways will become bottlenecks. In particular, latency caused by the need to send messages over the network between microservices may become unacceptable. Also, security concerns may arise that require encryption of the message stream, causing further latency.

To counteract this issue, you can trade some of the flexibility of microservices for performance by merging microservices in the critical message path. By using a message abstraction layer and pattern matching, as discussed in earlier chapters, you can do so with minimal code changes. Don't merge microservices wholesale; try to isolate the message patterns you're concerned with into a combined microservice. By executing a message pathway within a single process, you remove the network from the equation.

The Merge pattern (figure 5.16) is a good example of the benefit of the microservices-first approach. In the earlier part of an application's lifecycle, more flexibility is needed, because understanding of the business logic is less solid. Later, you may require optimizations to meet performance goals.

SPLIT PATTERN

Microservices grow over time, as more business logic is added, so you need to add new kinds of services to avoid building technical debt. In the early lifecycle of a system, microservices are small and handle general cases. As time goes by, more special cases are added to the business logic. Rather than handling these cases with more complex

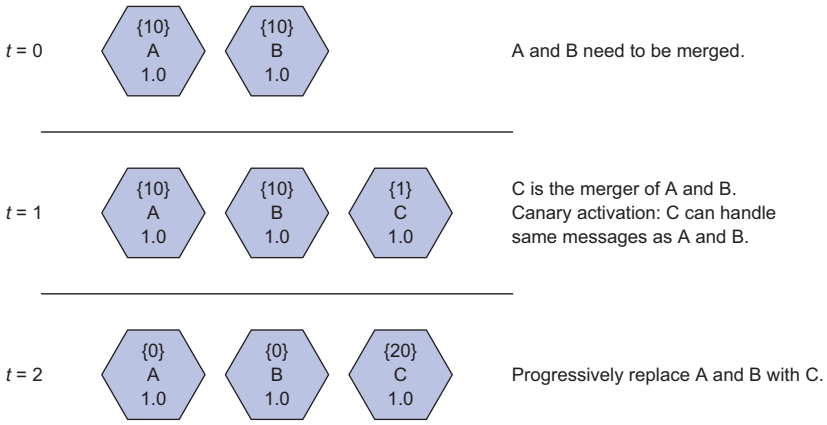


Figure 5.16 Merge pattern sequence

internal code and data structures, it's better to split out special cases into focused microservices. Pattern matching on messages makes this practical and is one of the core benefits of the pattern-matching approach.

The Split pattern (figure 5.17) captures one of the core benefits of the microservice architecture: the ability to handle frequently changing, underspecified requirements. Always look for opportunities to split, and avoid the temptation to use more-familiar language constructs (such as object-oriented design patterns), because they build technical debt over time.

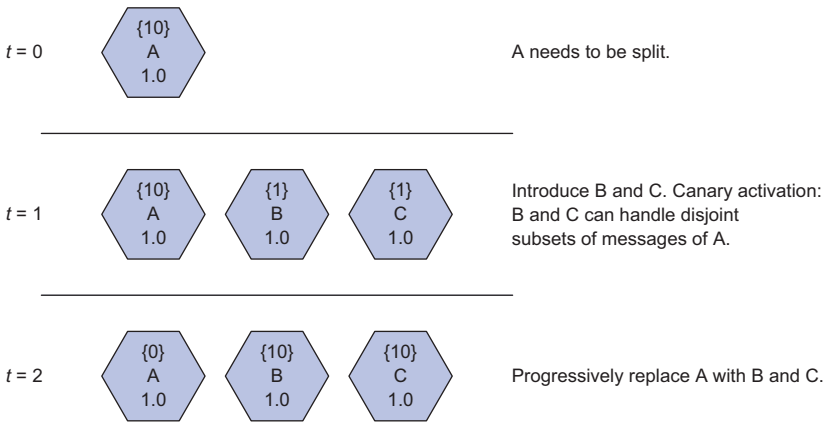


Figure 5.17 Split pattern sequence

5.7.3 Resilience

Chapter 3 discussed some of the common failure modes of microservice systems. A production deployment of microservices needs to be resilient to these failure modes. Although the system can never be entirely safe, you should put mitigations in place. As

always, the extent and cost of the mitigation should correspond to the desired level of risk.

In monolithic systems, failure is often dealt with by restarting the instances of that fail. This approach is heavy-handed and often not very effective. The microservice architecture offers a finer-grained menu of techniques for handling failure. The abstraction of a messaging layer is helpful, because this layer can be extended to provide automatic safety devices (ASDs). Bear in mind that ASDs aren't silver bullets, and may themselves cause failure, but they're still useful for many modes of failure.

SLOW DOWNSTREAM

In this failure mode, taking the perspective of a given client microservice instance, responses to outbound messages have latency or throughput levels that are outside of acceptable levels. The client microservice can use the following dynamic tactics, roughly in order of increasing sophistication:

- *Timeouts*—Consider messages failed if a response isn't delivered within a fixed timeout period. This prevents resource consumption on the client microservice.
- *Adaptive timeouts*—Use timeouts, but don't set them as fixed configuration parameters. Instead, dynamically adjust the timeouts based on observed behavior. As a simplistic example, time out if the response delay is more than three standard deviations from the observed mean response time. Adaptive timeouts reduce the occurrence of false positives when the overall system is slow and avoid delays in failure detection when the system is fast.
- *Circuit breaker*—Persistently slow downstream services should be considered effectively dead. Implementation requires the messaging layer to maintain metadata about downstream services. This tactic avoids unnecessary resource consumption and unnecessary degradation of overall performance. It does increase the risk of overloading healthy machines by redirecting too much traffic to them, causing a cascading failure similar to the unintended effects of the ASDs at Three Mile Island.
- *Retries*—If failure to execute a task has a cost, and there's tolerance for delay, it may make more sense to retry a failed message by sending it again. This is an ASD that has great potential to go wrong. Large volumes of retries are a self-inflicted *denial of service* (DoS) attack. Use a retry budget to avoid this by retrying only a limited number of times and, if the metadata is available, doing so only for a limited number of times per downstream. Retries should also use a randomized exponential backoff delay before being sent, because this gives the downstream a better chance of recovery by spreading load over time.
- *Intelligent round-robin*—If the messaging layer is using point-to-point transmission to send messages, then it necessarily has sufficient metadata to implement round-robin load balancing among the downstreams. *Simple round-robin* keeps a list of downstreams and cycles through them. This ignores differences in load between

messages and can lead to individual downstreams becoming overloaded. *Random round-robin* is found empirically to be little better, probably because the same clustering of load is possible. If the downstream microservices provide backpressure metadata, then the round-robin algorithm can make more-informed choices: it can choose the least loaded downstream, weight downstreams based on known capacity, and restrict downstreams to a subset of the total to avoid a domino effect from a circuit breaker that trips too aggressively.

Upstream overload

This is the other end of the overload scenario: the downstream microservice is getting too many inbound messages. Some of the tactics to apply are as follows:

- *Adaptive throttles*—Don't attempt to complete all work as it comes in. Instead, queue the work to a maximum rate that can be safely handled. This prevents the service from *thrashing*. Services that are severely resource constrained will spend almost all of their time swapping between tasks rather than working on tasks. On thread-based language platforms, this consumes memory; and on event-driven platforms, this manifests as a single task hogging the CPU and stalling all other tasks. As with timeouts, it's worth making throttles adaptive to optimize resource consumption.
- *Backpressure*—Provide client microservices with metadata describing current load levels. This metadata can be embedded in message responses. The downstream service doesn't actively handle load but relies on the kindness of its clients. The metadata makes client tactics for slow downstreams more effective.
- *Load shedding*—Refuse to execute tasks once a dangerous level of load has been reached. This is a deliberate decision to fail a certain percentage of messages. This tactic gives most messages reasonable latency, and some total failure, rather than allowing many messages to have high latency with sporadic failure. Appropriate metadata should be returned to client services so they don't interpret load shedding incorrectly and trigger a circuit breaker. The selection of tasks to drop, add to the queue, or execute immediately can be determined algorithmically and is context dependent. Nonetheless, even a simple load shedder will prevent many kinds of catastrophic collapse.

In addition to these dynamic tactics, upstream overload can be reduced on a longer time frame by applying the Merge deployment pattern.

Lost actions

To address this failure mode, apply the Progressive Canary deployment pattern, measuring message-flow rates to ensure correctness. Chapter 6 discusses measurement in more detail.

Poison messages

In this failure mode, a microservice generates a poisonous message that triggers a defect in other microservices, causing some level of failure. If the message is continuously

retried against different downstream services, they all suffer failures. You can respond in one of these ways:

- *Drop duplicates*—Downstream microservices should track message-correlation identifiers and keep a short-term record of recently seen inbound messages. Duplicates should be ignored.
- *Validation*—Trade the flexibility of schema-free messages for stricter validation of inbound message data. This has a less detrimental effect later in the project when the pace of requirement change has slowed.

Consider building a *dead-letter service*. Problematic messages are forwarded to this service for storage and later diagnosis. This also allows you to monitor message health across the system.

GUARANTEED DELIVERY

Message delivery may fail in many ways. Messages may not arrive or may arrive multiple times. Dropping duplicates will help within a service. Duplicated messages sent to multiple services are more difficult to mitigate. If the risk associated with such events is too high, allocate extra development effort to implement idempotent message interactions.[38]

EMERGENT BEHAVIOR

A microservice system has many moving parts. Message behavior may have unintended consequences, such as triggering additional workflows. You can use correlation identifiers for after-the-fact diagnosis, but not to actively prevent side effects:

- *Time to live*—Use a decrementing counter that's reduced each time an inbound message triggers the generation of outbound messages. This prevents unbounded side effects from proceeding without any checks. In particular, it stops infinite message loops. It won't fully prevent all side effects but will limit their effects. You'll need to determine the appropriate value of the counter in the context of your own system, but you should prefer low values. Microservice systems should be shallow, not deep.

CATASTROPHIC COLLAPSE

Some emergent behavior can be exceptionally pathological, placing the system into an unrecoverable state, even though the original messages are no longer present. In this failure mode, even with the Homeostasis pattern in place, service restarts can't bring the system back to health.

For example, a defect may crash a large number of services in rapid succession. New services are started as replacements, but they have empty caches and are thus unable to handle current load levels. These new services crash and are themselves

[38] Be careful not to overthink your system in the early days of a project. It's often better to accept the risk of data corruption in order to get to market sooner. Be ethical, and make this decision openly with your stakeholders. Chapter 8 has more about making such decisions.

replaced. The system can't establish enough capacity to return to normal. This is known as the *thundering herd* problem. Here are some ways to address it:

- *Static responses*—Use low-resource emergency microservices that return hard-coded responses to temporarily take load.
- *Kill switch*—Establish a mechanism to selectively stop large subsets of services. This gives you the ability to quarantine the problem. You can then restart into a known-good state.

In addition to using these dynamic tactics, you can prepare for disaster by deliberately testing individual services with high load to determine their failure points. Software systems tend to fail quickly rather than gradually, so you need to establish safe limits in advance.

The following sections describe microservice deployment patterns that provide resilience.

Apoptosis pattern

The Apoptosis[39] pattern removes malfunctioning services quickly, thus reducing capacity organically. Microservices can perform self-diagnosis and shut themselves down if their health is unsatisfactory. For example, all message tasks may be failing because local storage is full; the service can maintain internal statistics to calculate health. This approach also enables a graceful shutdown by responding to messages with metadata indicating a failure during the shutdown, rather than triggering timeouts.

Apoptosis (figure 5.18) is also useful for matching capacity with load. It's costly to maintain active resources far in excess of levels necessary to meet current load. Services can choose to self-terminate, using a probabilistic algorithm to avoid mass shutdowns. Load is redistributed over the remaining services.

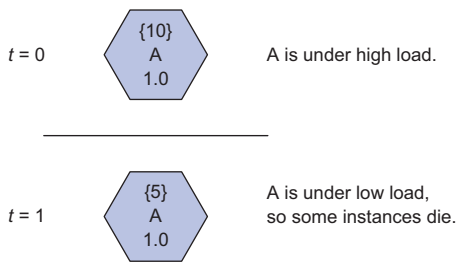


Figure 5.18 Apoptosis pattern sequence

Mitosis pattern

The Mitosis[40] pattern responds to increased load organically, without centralized control. Individual microservices have the most accurate measure of their own load levels. You can trigger the launching of new instances if local load levels are too high; this should be done using a probabilistic approach, to avoid large numbers of simultaneous launches. The newly launched service will take some of the load, bringing levels back to acceptable ranges.

[39] Living cells commit suicide if they become too damaged: apoptosis. This prevents cell damage from accumulating and causing cancers.

[40] Living cells replicate by splitting in two. *Mitosis* is the name of this process.

Mitosis (figure 5.19) and Apoptosis should be used with care and with built-in limits. You don't want unbounded growth or a complete shutdown. Launch and shutdown should occur via primitive operations executed by the infrastructure tooling, not by the microservices.

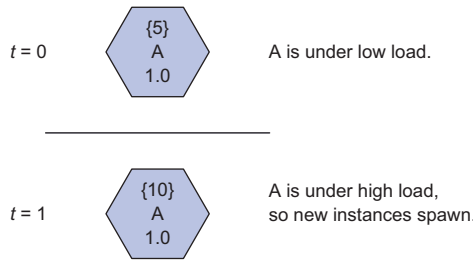


Figure 5.19 Mitosis pattern sequence

KILL SWITCH PATTERN

With this pattern (figure 5.20), you disable large parts of the system to limit damage. Microservices systems are complex, just like the Three Mile Island reactor. Failure events at all scales are to be expected. Empirically, these events follow a power law in terms of occurrence. Eventually, an event with potential for significant damage will occur.

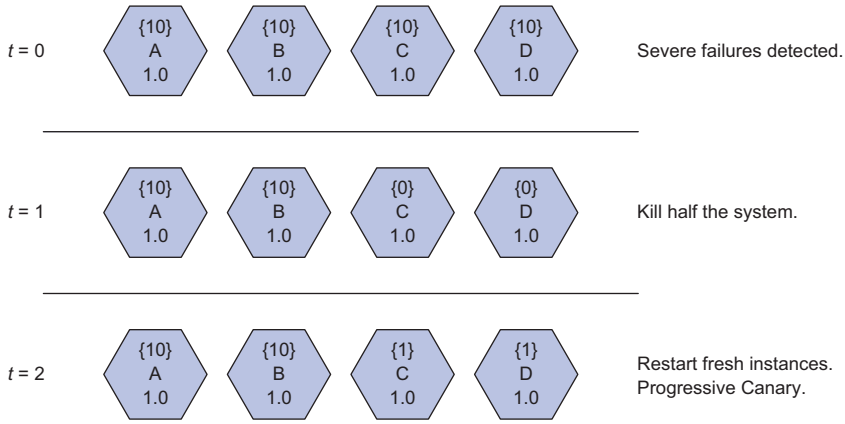


Figure 5.20 Kill Switch pattern sequence

To limit the damage, rapid action is required. It's impossible to understand the event during its occurrence, so the safest course of action is to scram the system. You should be able to shut down large parts of the system using secondary communication links to each microservice. As the event progresses, you may need to progressively shut down more and more of the system in order to eventually contain the damage.

5.7.4 Validation

Continuous validation of the production system is the key practice that makes microservices successful. No other activity provides as much risk reduction and value. It's the only way to run a CD pipeline responsibly.

What do you measure in production? CPU load levels? Memory usage? These are useful but not essential. Far more important is validation that the system is behaving as desired. Messages correspond to business activities or parts of business activities, so

you should focus on the behavior of messages. Message-flow rates tell you a great deal about the health of the system. On their own, they're less useful than you may think, because they fluctuate with the time of day and with seasonality; it's more useful to compare message-flow rates with each other.

A given business process is encoded by an expanding set of messages generated from an initial triggering message. Thus, message-flow rates are related to each other by ratios. For example, for every message of a certain kind, you expect to see two messages of another kind. These ratios don't change, no matter what the load level of the system or the number of services present. They're *invariant*.

Invariants are the primary indicator of health. When you deploy a new version of a microservice using the Canary pattern, you check that the invariants are within expected bounds. If the new version contains a defect, the message-flow ratios will change, because some messages won't be generated. This is an immediate indicator of failure. Invariants, after all, can't vary. We'll come back to this topic in chapter 6 and examine an example in chapter 9.

The following sections present some applicable microservice deployment patterns.

VERSION UPDATE PATTERN

This pattern (figure 5.21) lets you safely update a set of communicating microservices. Suppose that microservices A and B communicate using messages of kind *x*. New business requirements introduce the need for messages of kind *y* between the services. It's unwise to attempt a simultaneous update of both; it's preferable to use the Progressive Canary deployment pattern to make the change safely.

First, you update listening service B so that it can recognize the new message, *y*. No other services generate this message in production yet, but you can validate that the new version of B doesn't cause damage. Once the new B is in place, you update A, which emits *y* messages.

This multistage update (composed of Progressive Canaries for each stage) can be used for many scenarios where message interactions need to change. You can use it

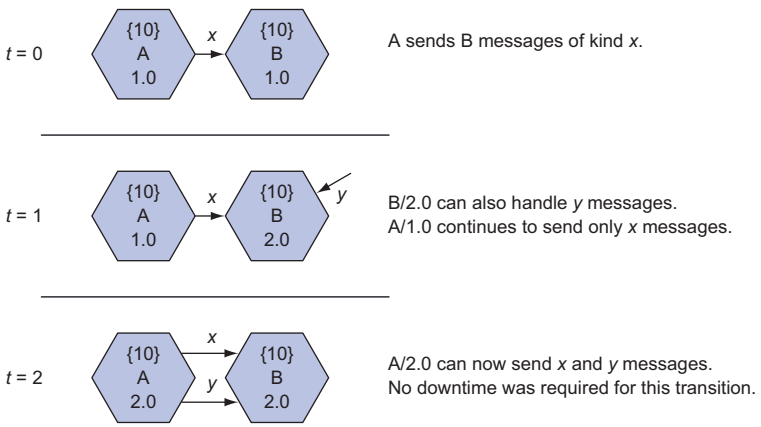


Figure 5.21 Version Update pattern sequence

when the internal data of the messages changes. (B, in this case, must retain the ability to handle old messages until the change is complete.) You can also use it to inject a third service between two existing services, by applying the pattern first to one side of the interaction and then to the other. This is a common way to introduce a cache, such as the one you saw in chapter 1.

CHAOS PATTERN

You can ensure that a system is resistant to failure by constantly failing at a low rate. Services can develop fragile dependencies on other services, despite your best intentions. When dependencies fail, even when that failure is below the acceptable threshold, the cumulative effect can cause threshold failures in client services.

To prevent creeping fragility, deliberately fail your services on a continuous basis, in production. Calibrate the failure to be well below the failure threshold for the business, so it doesn't have a significant impact on business outcomes. This is effectively a form of insurance: you take small, frequent, losses to avoid large, infrequent, losses that are fatal.

The most famous example of the Chaos pattern is the Netflix Chaos Monkey, which randomly shuts down services in the Netflix infrastructure. Another example is Google Game Days, where large-scale production failures are deliberately triggered to test failover capabilities.

5.7.5 Discovery

Pattern matching and transport independence give you decoupled services. When microservice A knows that microservice B will receive its messages, then A is *coupled* to B. Unfortunately, message transport requires knowledge of the location of B; otherwise, messages can't be delivered. Transport independence hides the mechanism of transportation from A, and pattern matching hides the identity of B. Identity is coupling.

The messaging abstraction layer needs to know the location of B, even as it hides this information from A. A (or, at least, the message layer in A) needs to discover the location of B (in reality, the set of locations of all the instances of B). This is a primary infrastructural challenge in microservice systems. Let's examine the common solutions:

- *Embedded configuration*—Hardcode service locations as part of the immutable artifact.
- *Intelligent load balancing*—Direct all message traffic through load balancers that know where to find the services.
- *Service registries*—Services register their location with a central registry, and other services look them up in the registry.
- *DNS*—Use the DNS protocol to resolve the location of a service.
- *Message bus*—Use a message bus to separate publishers from subscribers.
- *Gossip*—Use a peer-to-peer membership gossip protocol to share service locations.

No solution is perfect, and they all involve trade-offs, as listed in table 5.1.

Table 5.1 Service-discovery methods

Discovery	Advantages	Disadvantages
Embedded configuration	Easy implementation. Works (mostly) for small static systems.	Doesn't scale, because large systems are under continuous change. Very strong identity concept: raw network location.
Intelligent load balancing	Scalable with proven production-quality tooling. Examples: NGINX and Netflix's Hystrix.	Non-microservice network element that requires separate management. Load balancers force limited transport options and must still discover service locations themselves using one of the other discovery mechanisms. Retains identity concept: request URL.
Registry	Scalable with proven production-quality tooling. Examples: Consul, ZooKeeper, and etcd.	Non-microservice network element. High dependency on the chosen solution, because there are no common standards. Strong identity concept: service name key.
DNS	Unlimited scale and proven production-quality tooling. Well understood. Can be used by other mechanisms to weaken identity by replacing raw network locations.	Non-microservice network element. Management overhead. Weak identity concept: hostname.
Message bus	Scalable with proven production-quality tooling. Examples: RabbitMQ, Kafka, and NServiceBus.	Non-microservice network element. Management overhead. Weak identity concept: topic name.
Gossip	No concept of identity! Doesn't require additional network elements. Early adopter stage, but shown to work at scale.[a]	Message layer must have additional intelligence to handle load balancing. Rapidly evolving implementations.

a) The SWIM algorithm has found success at Uber. See "How Ringpop from Uber Engineering Helps Distribute Your Application" by Lucie Lozinski, February 4, 2016, https://eng.uber.com/intro-to-ringpop.

5.7.6 Configuration

How do you configure your microservices? Configuration is one of the primary causes of deployment failure. Does configuration live with the service, immutably packaged into the artifact? Or does configuration live on the network, able to adapt dynamically to live conditions, and providing an additional way to control services?

If configuration is packaged with the service, then configuration changes aren't different from code changes and must be pushed through the CD pipeline in the same manner. Although this does provide better risk management, it also means you may suffer unacceptable delays when you need to make changes to configuration. You also add additional entries to the artifact store and audit logs for every configuration change, which can clutter these databases and make them less useful. Finally, some network components, such as intelligent load balancers, still need dynamic configuration if they're to be useful, so you can't place all configuration in artifacts.

On the other hand, network configuration removes your ability to reproduce the system deterministically or to benefit fully from the safety of immutability. The same

artifact deployed tomorrow could fail even though it worked today. You need to define separate change-control processes and controls for configuration, because you can't reuse the artifact pipeline for this purpose. You'll need to deploy network services and infrastructure to store and serve configuration. Even if most configuration is dynamic, you still have to bake in at least some configuration—in particular, the network location of the configuration store! When you look closely, you find that many services have large amounts of potential configuration arising from third-party libraries included in them. You need to decide to what extent you'll expose this via the dynamic-configuration store. You're unlikely to find much value in exposing all of it. The most practical option is to embed low-level configuration into your deployment artifacts.

You'll end up with a hybrid solution, because neither approach provides a total solution. The immutable-packaging approach has the advantage of reusing the delivery pipeline as a control mechanism and offering more predictable state. Placing most of your configuration into immutable artifacts is a reasonable trade-off. Nonetheless, you should plan for the provision and management of dynamic configuration.

There are two dangerous anti-patterns to avoid when it comes to configuration. Using microservices doesn't protect you from them, so remain vigilant:

- *Automation workarounds*—Configuration can be used to code around limitations of your automation tooling: for example, using feature flags rather than generating new artifacts. If you do too much of this, you'll create an uncontrolled secondary command structure that damages the properties of the system that make immutability so powerful.
- *Turing's revenge*—Configuration formats tend to be extended with programming constructs over time, mostly as conveniences to avoid repetition.[41] Now, you have a new, unasked-for programming language in your system that has no formal grammar, undefined behavior, and no debugging tools. Good luck!

5.7.7 Security

The microservice architecture doesn't offer any inherent security benefits and can introduce new attack vectors if you aren't careful. In particular, there's a common temptation to share microservice messages with the outside world. This is dangerous, because it exposes every microservice as an attack surface.

There must be an absolute separation: you need a *demilitarized zone* (DMZ) between the internal world of microservice messages and the outside world of third-party clients. The DMZ must translate between the two. In practice, this means a microservice system should expose traditional integration points, such as REST APIs, and then convert requests to these APIs into messages. This allows for strict sanitization of input.

Internally, you can't ignore that fact that microservices communicate over a network, and networks represent an opportunity for attack. Your microservices should live in their own private networks with well-defined ingress and egress routes. The rest

[41] Initially declarative domain-specific languages, such as configuration formats, tend to accumulate programmatic features over time. It's surprisingly easy to achieve Turing completeness with a limited set of operations.

of the system uses these routes to interact with the microservice system as a whole. The specific microservices to which messages are routed aren't exposed.

These precautions may still be insufficient, and you need to consider the case where an attacker has some level of access to the microservice network. You can apply the security principle of *defense in depth* to strengthen your security in layers. There's always a trade-off between stronger security and operational impact.

Let's build up a few layers. Microservices can be given a right of refusal, and they can be made more pedantic in the messages they accept. Highly paranoid services will lead to higher error rates but can delay attackers and make attacks more expensive. For example, you can limit the amount of data a service will return for any one request. This approach means custom work for each service.

Communication between services can require shared secrets and, as a further layer, signed messages. This protects against messages injected into the network by an attacker. The distribution and cycling of the secrets introduces operational complexity. The signing of messages requires key distribution and introduces latency.

If your data is sensitive, you can encrypt all communication between microservices. This also introduces latency and management overhead and isn't to be undertaken lightly. Consider using the Merge pattern for extremely sensitive data flows, avoiding the network as much as possible.

You need secure storage and management of secrets and encryption keys in order for these layers to be effective. There's no point in encrypting messages if the keys are easily accessible within the network—but your microservices must be able to access the secrets and keys. To solve this problem, you need to introduce another network element: a key-management service that provides secure storage, access control, and audit capabilities.[42]

5.7.8 Staging

The staging system is the control mechanism for the CD pipeline. It encompasses traditional elements, such as a build server for CI. It can also consist of multiple systems that test various aspects of the system, such as performance.

The staging system can also be used to provide manual gates to the delivery pipeline. These are often unavoidable, either politically or legally. Over time, the effectiveness of CD in managing risk and delivering business value quickly can be used to create sufficient organizational confidence to relax overly ceremonial manual sign-offs.

The staging system provides a self-service mechanism for development teams to push updates all the way to production. Empowering teams to do this is a critical component in the success of CD. Chapter 7 discusses this human factor.

Staging should collect statistics to measure the velocity and quality of code delivery over time. It's important to know how long it takes, on average, to take code from concept to production, for a given risk level, because this tells you how efficient your CD pipeline is.

[42] Some examples are HashiCorp Vault (www.vaultproject.io), AWS key-value stores (KVS), and, if money is no object, hardware security modules (HSMs).

The staging system has the most variance between projects and organizations. The level of testing, the number of staging steps, and the mechanism of artifact generation are all highly context-specific. As you increase the use of microservices and CD in your organization, avoid being too prescriptive in your definition of the staging function; you must allow teams to adapt to their own circumstances.

5.7.9 Development

The development environment needed for microservices should enable developers to focus on a small set of services at a time—often, a single service. The message-abstraction layer comes into its own here, because it makes it easy to mock the behavior of other services.[43] Instead of having to implement a complex object hierarchy, microservice mocking only requires implementing sample message flows. This makes it possible to unit-test microservices in complete isolation from the rest of the system.

Microservices can be specified as a relation between inbound and outbound messages. This allows you to focus on a small part of the system. It also enables more-efficient parallel work, because messages from other microservices (which may not yet exist) can easily be mocked.

Isolation isn't always possible or appropriate. Developers often need to run small subsets of the system locally, and tooling is needed to make this practical. It isn't advisable for development to become dependent on running a full replica of the production system. As production grows to hundreds of different services and beyond, it becomes extremely resource intensive to run services locally, and, ultimately, doing so becomes impossible.

If you're running only a subset of the system, how do you ensure that appropriate messages are provided for the other parts of the system? One common anti-pattern is to use the build or staging system to do this. You end up working against shared resources that have extremely nondeterministic state. This is the same anti-pattern as having a shared development database.

Each developer should provide a set of mock messages for their service. Where do these mock messages live? At one extreme, you can place all mock-message flows in a common mocking service. All developers commit code to this service, but conflicts are rare because work isn't likely to overlap. At the other extreme, you can provide a mock service along with each service implementation. The mock service is an extremely simple service that returns hardcoded responses.

The practical solution for most teams is somewhere in the middle. Start with a single universal mocking service, and apply the Split pattern whenever it becomes too unwieldy. Sets of services with a common focus will tend to get their own mocking service. The development environment is typically a small set of actual services, along with one or two mocking services. This minimizes the number of service processes needed on a developer machine.

[43] For a practical example, see the code in chapter 9.

The mock messages are defined by the developers building a given microservice. This has an unfortunate side effect: Some developers will focus on expected behavior. Others will use their service in unexpected ways, so the mocking will be incomplete. If you allow other developers to add mock messages to services they don't own, then the mocks will quickly diverge from reality. The solution is to add captured messages to the list of sample messages. Capture sample message flows from the production or staging logs, and add them to the mock service. This can be done manually for even medium-sized systems.

Beware the distributed monolith!

How do you know you're building a distributed monolith? If you need to run all, or most, of your services to get any development work done. If you can't write a microservice without needing all the other microservices running, then you have a problem.

It's easy to end up needing to run a large cloud server instance for every developer—in which case development will slow to a crawl. This is why you must invest in a messaging abstraction layer and avoid the mini-web-servers anti-pattern.

You'll need to think carefully about the mocking strategy you'll use in your project. It must allow your developers to build with chosen subsets of the system.

5.8 Summary

- Failure is inevitable and must be accepted. Starting from that perspective, you can work to distribute failure more evenly over time and avoid high-impact catastrophes.
- Traditional approaches to software quality are predicated on a false belief in perfectionism. Enterprise software systems aren't desktop calculators and won't give the correct answer 100% of the time. The closer you want to get to 100%, the more expensive the system becomes.
- The risk of failure is much higher than generally believed. Simple mathematical modeling of risk probabilities in component-based systems (such as enterprise software) brings this reality starkly into focus.
- Microservices provide an opportunity to measure risk more accurately. This enables you to define acceptable error rates that your system must meet.
- By packaging microservices into immutable units of deployment, you can define a set of deployment patterns that mitigate risk and can be automated for efficient management of your production systems.
- The accurate measurement of risk enables the construction of a continuous delivery pipeline that enables developers to push changes to microservices to production at high velocity and high frequency, while maintaining acceptable risk levels.

Writing tests for microservices

Deploying applications is the only way we can bring feature to life and evolve our systems.

Any change introduced in an application has the potential to disrupt normal operation. By having the proper test coverage, the potential risk is highly minimised. In microservice applications each service must have its own test coverage, but it is also important to test interactions between services to ensure features involving multiple components will operate, or continue operating, according to expectation. This chapter covers the different types of tests and shows how to write good automated tests.

Chapter 7 from *Microservices in .NET Core*
by Christian Horsdal Gammelgaard

Writing tests for microservices

This chapter covers

- Writing good automated tests
- Understanding the test pyramid and how it applies to microservices
- Testing microservices from the outside
- Writing fast, in-process tests for endpoints
- Using Nancy.Testing for integration and unit tests

Up to this point, you've written a few microservices and set up collaborations between some of them. The implementations are fine, but you haven't written any tests for them. As you write more and more microservices, developing systems without good automated tests becomes unmanageable. In the first half of this chapter, I'll discuss what you need to test for each individual microservice. Then we'll dive into code, looking first at testing endpoints using the Nancy.Testing library, and then at testing a complete microservice as if you were sending it requests from another microservice.

7.1 What and how to test

In chapter 1, you saw three characteristics of a microservice that make it good for continuous delivery:

- *Individually deployable*—As soon as any small, safe change has been made to a microservice, the microservice can be deployed to production. But how do you know a change is safe? This is where testing and, particularly, test automation come into the picture. Several other activities, like code reviews, static code analysis, and designing public APIs for backward compatibility, also play into determining that a change is safe, but testing is where much of your confidence will come from.
- *Replaceable*—You should strive to be able to replace the implementation of a microservice with another functionally equivalent implementation within the normal pace of work. Again, tests play an important role, because a good set of tests lets you assess whether the new implementation really is equivalent to the old one.
- *Maintainable by a small team*—Microservices are sufficiently small and focused that a team can maintain several of them. This has the advantage that you can write tests that cover all parts of your microservices.

If you want to become confident about changes quickly and be able to replace a badly implemented microservice, testing has to be fast and repeatable. To make testing fast and repeatable, you must automate a significant part of it—and that's the focus of this chapter.

7.1.1 The test pyramid: what to test in a microservices system

The *test pyramid* shown in figure 7.1 is a tool you can use to guide which kinds of tests you should write and how many you should have of each kind. You can find variations of the test pyramid in different writings; all of them put tests on different levels, where the levels at the top of the pyramid are broad in scope and the tests at the bottom are narrow. The test pyramid illustrates that you should aim for having many narrowly focused tests (the ones at the wide bottom of the pyramid) and only a few broadly scoped tests (the ones at the narrow top).

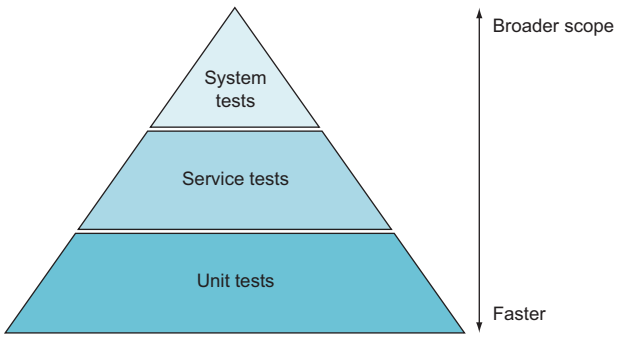


Figure 7.1 The test pyramid illustrates that you should have a few system-level tests, many service-level tests, and even more unit-level tests.

The version of the test pyramid that I use here has three levels:

- *System tests (top level)*—Tests that span the complete system of microservices and are usually implemented through the GUI.
- *Service tests (middle level)*—Tests that work against one, but only one, complete microservice.
- *Unit tests (bottom level)*—Tests that test one small piece of functionality in a microservice. Unit tests call code in the microservice under test in-process and usually involve only part of a microservice.

Note that when I use the term *unit test*, the word *unit* refers to a small piece of functionality. I define the scope of a unit test not in terms of any particular code construct, like a class or a method, but rather in terms of functionality. When we look at implementations of unit tests later, you'll see that unit tests can easily span all layers of a microservice: for example, from a Nancy module, through a domain object, down to a data access class.

Although the test pyramid tells you to have more tests as you move down the levels, exactly how many tests you should have on each level is situational. It depends on such factors as the size of the system, the complexity of the system, and the cost of failure.

7.1.2 System-level tests: testing a complete microservice system end-to-end

The tests at the top of the pyramid have a very broad scope and therefore cover a lot of code with just a few tests. Because they have such a broad scope, they're also imprecise. When a system-level test breaks, it isn't immediately clear where the problem lies. The test can potentially use the entire system, so the issue could be anywhere.

An example of a system-level test is one that uses the web UI of the point-of-sale system we talked about in earlier chapters to add a number of items to an invoice, apply a discount code, and pay using a test credit card. If that test passes, it gives you confidence that invoices are created, that discounts can be applied, and that you can receive credit card payments. During such a system test, you might assert that the amount due on the invoice is as expected. If that assertion fails, any number of things could have caused the problem: you might be using the wrong price for one or more items, you might have applied the discount incorrectly, or you might have misinterpreted the invoice data. In other words, such a failure could be caused by at least a handful of different microservices. To figure out which one is the culprit, you need to investigate.

The specific way a system-level test fails can give some hints as to where the problem lies, but there's usually a lot of code that could be at fault. From the system test alone, it won't even be clear which microservice caused the failure. On the other hand, when system-level tests pass, they give you a good deal of confidence.

The second downside to system-level tests is that they tend to be slow. This again is the flip side of them involving the complete system: real HTTP requests are made, things are written to real data stores, and real event feeds are polled.

Considering that system-level tests, when successful, can give you good confidence, but that they're both slow and imprecise, my advice is to *write system-level tests for the success path of the most important use cases*. This should give you coverage for the success paths of all the most important parts of the system. You can, optionally, supplement this with some tests for the most common and important failure scenarios. Exactly how many system tests this amounts to is, as mentioned earlier, entirely situational. This advice applies equally to microservices, traditional SOA, and monoliths. There's nothing microservice-specific about system-level tests. For this reason, I won't show implementations of any system-level tests in this chapter.

7.1.3 Service-level tests: testing a microservice from outside its process

The tests in the middle level of the test pyramid interact with one microservice as a whole and in isolation—the collaborators of the microservice under test are replaced with *microservice mocks*. Like system tests, these tests interact with the microservice under test from the outside. But unlike system-level tests, they interact directly with the public API of the microservice and make assertions about responses to the microservice as well as the interactions the microservice has with other microservices: for instance, about the commands the microservice under test sends to other microservices.

> **A microservice mock simulates a real microservice and records interactions**
>
> A *microservice mock* can be used in place of a real microservice in service-level tests. It implements the same endpoints as the real microservice, but instead of using real business logic to implement the endpoints, the mock has dumbed-down endpoint implementations; usually endpoints in a mock return hardcoded responses. Furthermore, a mock often records the requests made to the endpoints, so the test code can inspect the requests made during the test.
>
> This is similar to the mock objects widely used in tests for object-oriented code. But where mock objects replace a real object, a microservice mock replaces a real microservice.

Like system-level tests, service-level tests test scenarios rather than single requests. That is, they make a sequence of requests that together form a meaningful scenario. The requests made from the microservice under test to its mocked collaborators are real HTTP requests, and the responses are real HTTP responses.

For examples, recall the Loyalty Program microservice from the example point-of-sale system. In chapter 4, you saw that it collaborated with a number of other microservices, as shown in figure 7.2, using all three collaboration styles: events, queries, and commands.

To test Loyalty Program in isolation, you can create mock versions of its collaborators. As shown in figure 7.3, when Loyalty Program interacts with a mocked collaborator, it gets back a hardcoded response.

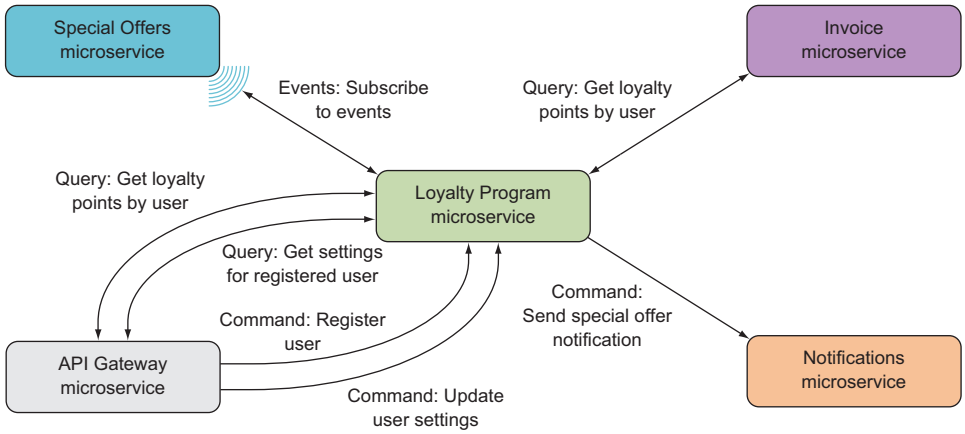


Figure 7.2 The Loyalty Program microservice collaborates with a number of other microservices through all three types of collaboration: events, queries, and commands.

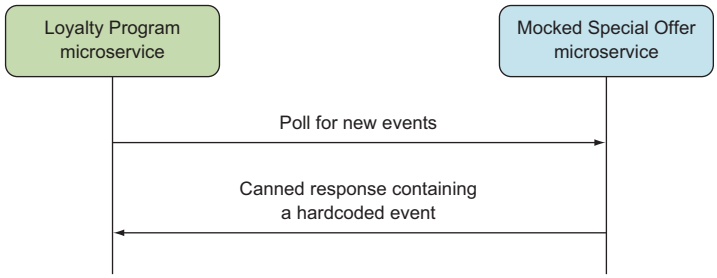


Figure 7.3 For service-level testing, the Loyalty Program microservice interacts with mocked versions of its collaborators. The mocked microservices respond to requests with hardcoded responses.

A service-level test for the Loyalty Program microservice could do the following:

- Send a command to create a user
- Wait for the Loyalty Program microservice to query a mock Special Offer microservice for events, and get back a hardcoded event about a new special offer
- Record any commands sent to the Notifications microservice, and assert that a command for a notification to the new user about the new special offer was sent

When a test like this passes, you can have confidence that important aspects of the Loyalty Program microservice work. When it fails, you know that the problem is within Loyalty Program itself.

Service-level tests are much more precise than system-level tests, because they cover only a single microservice: if such a test fails, the problem should lie within the microservice under test, assuming the test setup itself isn't buggy. Because microservices are small—they're replaceable, after all—knowing that a problem lies within a certain microservice is a lot more precise than what you get from system-level tests.

On the other hand, service-level tests are still slow, because they interact with the microservice under test over HTTP, because the microservice uses a real database, and because it interacts with its mocked collaborators over HTTP.

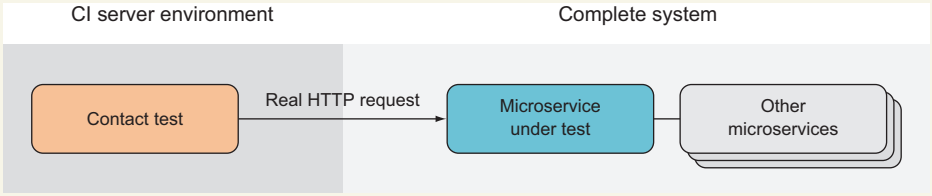
Contract tests

As you know by now, there's a lot of collaboration between microservices in a microservices system. You implement the collaborations as requests from one microservice to another. If you aren't careful, changes in an endpoint can break the microservices that call that endpoint. This is where contract tests come into the picture.

When any two microservices in the system collaborate, the one making requests to the other has some expectations about how the other microservice will behave. That is, given a collaboration, the calling microservice expects the called microservice to implement a certain contract. A *contract test* is a test with the purpose of determining whether the called microservice implements the contract expected by the calling microservice.

Contract tests are written from the point of view of the caller and are there for the sake of the calling microservice: as long as the contract test passes, the assumptions the caller makes about the contract are still valid. Consequently, the contract tests are part of the caller's code base. They aren't part of the same code base as the endpoints they test. Contract tests shouldn't have any knowledge of how the microservices they test are implemented. This is where contract tests differ from service-level tests. With service-level tests, you isolate the microservice under test by providing it with mocked microservices in place of its collaborators. You don't want to do that for contract tests, because the contract tests shouldn't know about the other collaborators of the microservice they test. In other words, contract tests run against the complete system.

Because contact tests are part of the code base of one microservice but test things in other microservices, and because they run against the complete system, it can be a good idea to run them against a QA or staging environment. Moreover, it's a good idea to have them run automatically every time the microservice under test is deployed. When a contract breaks, it's a strong indication that the collaboration between the microservice the contract test belongs to and the microservice under test is broken, too.



A contract test runs against the complete system. It may, for instance, run against a staging or QA environment, where the complete microservices system is deployed.

In terms of implementation, contract tests look a bit like the service-level tests you'll write later in this chapter. The difference is that contract tests are a slightly higher level in the test pyramid, between system-level tests and service-level tests. Contract tests don't set up mocked collaborators, whereas service-level tests do; but just like service-level tests, they work by making real HTTP requests to the microservice under test.

My recommendation regarding service-level tests is that you should write such tests for the success versions of all functionality the microservice under test offers. Such tests will naturally use all endpoints of the microservice as well as rely on any event subscriptions in the microservice. In other words, they will cover all success paths in the microservice. In general, I recommend writing service-level tests only for the most important failure scenarios. Again, the number of service-level tests needed and how many failure scenarios they should cover depends on the system and the cost of failure in that particular system.

7.1.4 Unit-level tests: testing endpoints from within the process

The tests at the bottom of the test pyramid also deal with a single microservice, but these tests don't work over HTTP and don't deal with the entire microservice. These unit tests interact with the parts of the microservice under test directly and in memory. To call the endpoints implemented in your Nancy modules, you'll use the Nancy.Testing library that comes as a companion library alongside Nancy. Nancy.Testing lets you write tests that make calls to Nancy endpoints in memory. The calls go through Nancy in exactly the same way HTTP requests would, but without going through the network stack. To the code in your Nancy modules, calls made with Nancy.Testing look exactly like real HTTP requests.

At the unit-test level, I'll show you two kinds of tests (see figure 7.4): one that uses a database and one that uses a mock in place of the database. I consider both to be unit tests, even though the first type uses a database. Two things make a test a unit test: its scope is a small piece of functionality, and the test code and the production code in the microservice run in the same process.

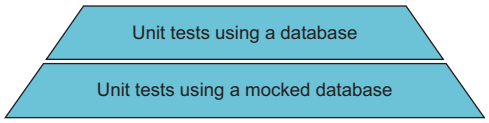


Figure 7.4 At the unit-test level, there are two kinds of tests: those that use a database and those that don't.

The narrow scope of a unit test makes it precise: when it fails, the problem lies in a small amount of code. A narrow scope also enables you to write tests that cover failure scenarios properly. Both types of unit tests are faster than service-level tests, but of course the tests that mock out databases are faster than those that use a database. Therefore, you can have both and will probably have more tests that mock the database than tests that don't.

Sometimes you may also have even narrower unit tests that test the business logic in the microservices directly by instantiating domain objects and testing them directly. I take a pragmatic approach to deciding how narrow the narrowest unit tests should be: I use a test-first workflow that starts from the outside, with tests that use Nancy.Testing making calls to endpoint handlers in Nancy modules. I start with tests that cover the broad strokes of what the endpoint should do, an then I progressively add tests for more details. Only when it becomes awkward to test a particular detail through the endpoint handler do I begin to write narrower unit tests. For instance, covering a particular case in the business logic with tests that call through the endpoint handler might require a

lot of setup code. That's a signal to switch down to a test that has a narrower scope: just those cases in the business logic. I'll write tests for those cases that work directly on the classes that should implement that particular part of the business logic.

For the Loyalty Program microservice, you need unit tests that test the endpoint that lets you create users with a number of different inputs covering both possible valid inputs and invalid inputs. Likewise, you need tests that try to read both existing and nonexistent users from the query endpoint that lets you read users. You need similar tests for the other endpoints in the microservice. Loyalty Program is sufficiently simple that you don't need to switch down to tests that are narrower than the microservice's endpoints. So, the units tests I'll show you later all work by calling endpoint handlers through Nancy.Testing.

7.2 Testing libraries: Nancy.Testing and xUnit

In this chapter, you'll use two new libraries:

- Nancy.Testing (https://github.com/NancyFx/Nancy/wiki/Testing-your-application)
- xUnit (https://xunit.github.io/)

I'll give you a brief introduction to each, and then you'll implement tests for some of the microservices you wrote in earlier chapters.

7.2.1 Meet Nancy.Testing

The Nancy.Testing library is a companion to Nancy that makes it easy to test endpoints implemented in Nancy modules. The main entry point into Nancy.Testing is the `Browser` type, which accepts method calls like `Get("/")`, `Post("/user")`, `Put("/user/42")`, and `Delete("/user/42")` that let tests call `GET`, `POST`, `PUT`, and `DELETE` endpoints in Nancy modules, respectively. When a test calls an endpoint through the `Browser` type, the call goes through the real Nancy pipeline. This means routes are resolved the same way as for real HTTP requests, the dependency injection container is set up and used as usual, and serialization and deserialization run as they normally do. In short, to the endpoint, the call looks exactly like a real HTTP request. The cool thing is that it's all done in process, so it's much faster than a real HTTP request would be. The return value of each method is a `NancyResponse` object and contains everything a real HTTP response would, including headers, status codes, and a body.

In addition to the `Browser` type, the Nancy.Testing library provides `ConfigurableBootstrapper`, which offers a nice API for creating ad hoc bootstrappers used in tests. Among other things, `ConfigurableBootstrapper` allows you to do the following:

- Create `Browser` objects that see only one Nancy module instead of all modules in the application
- Override registrations in the dependency injection container: for instance, to provide mock objects in place of real ones
- Add hooks to the Nancy pipeline, such as an error handler

Finally, Nancy.Testing comes with a bunch of convenience methods that make writing assertions against `NancyResponse` objects easy.

Nancy.Testing offers a wealth of functionality that makes it easier to write tests. Going through all of it is beyond the scope of this chapter, but you'll see some of its power. I find the APIs in the library to be quite discoverable, so I'm sure once you get going, you'll discover more of what Nancy.Testing has to offer.

You can find further information on Nancy.Testing in the Nancy documentation (https://github.com/NancyFx/Nancy/wiki/Testing-your-application), or you can jump right in and start using it. I think you'll find that the APIs are quite discoverable through IntelliSense.

7.2.2 Meet xUnit

xUnit (http://xunit.github.io) is a unit-test tool for .NET. It has a library part that allows you to write automated tests and a runner part that can run those tests. To write a test with xUnit, you create a method with a `Fact` attribute over it and put the code to perform the test there. The xUnit runner scans for methods with a `Fact` attribute and executes all of them. In addition, xUnit has an API for making assertions in tests. If an assertion fails, the xUnit runner picks up the failure and reports it back when it's finished running tests. The xUnit test runner can be run by `dotnet` and is therefore well suited for the projects you're building in this book.

Other .NET test tools similar to xUnit—NUnit, for instance—are available that you can also use. This book sticks with xUnit because it's used for the test projects that Yeoman and Visual Studio create. If you prefer another tool, feel free to use it, as long as it works with `dotnet`.

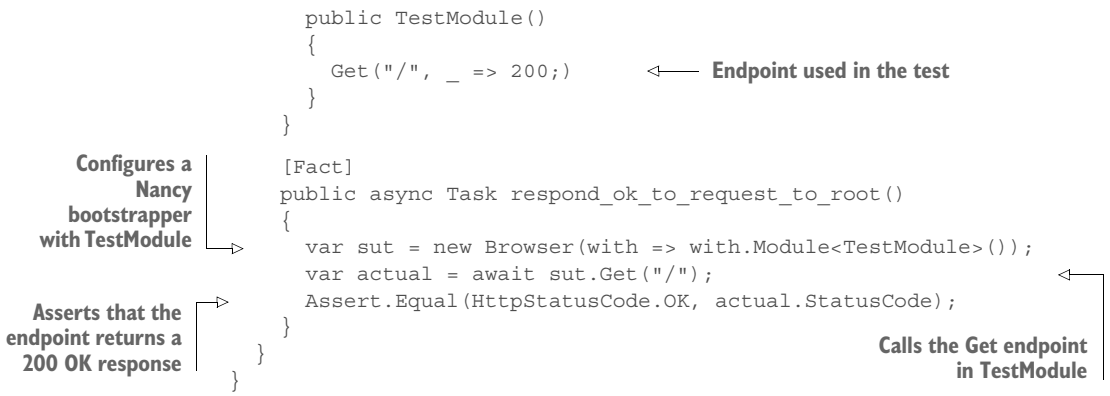
7.2.3 xUnit and Nancy.Testing working together

Putting Nancy.Testing and xUnit together, you can write succinct tests for endpoints implemented in Nancy modules. In section 7.3.1, you'll set up a project for these unit tests and run them with `dotnet`; but for now, I just want to give you a quick peek at how the tests will look. The following test calls the `Get` endpoint in `TestModule` and makes the assertion that the response status code is 200 OK.

Listing 7.1 Simple test using xUnit and Nancy.Testing

```
namespace LoyaltyProgramUnitTests
{
  using Nancy;
  using Nancy.Testing;
  using Xunit;

  public class TestModule_should
  {
    public class TestModule : NancyModule
    {
```



Naming conventions

My tests follow these naming conventions:

- My tests work on an object called `sut` for *system under test*. In the previous test, `sut` is a `Browser` object that I use to make a call to an endpoint.
- I name my test classes after the thing they test—`TestModule` in this example test—followed by `_should`.
- I name the `Fact` method after the scenario being tested and the expected result. I separate the words in `Fact` method names with underscores and try to make sure they form a sentence when combined with the name of the surrounding class. For instance, in this test, concatenating the class name and the `Fact` method name and replacing underscores with spaces, you get "TestModule should respond ok to requests to root."

Whether you like these conventions is a matter of taste. I happen to like them, but they're in no way essential to writing good tests.

You can run the previous test with `dotnet`; it will execute in-memory and give you good coverage because the call to `sut.Get("/")` executes the real Nancy pipeline, including the implementation of the endpoint in `TestModule`. The string argument `"/"` is the relative URL to which the fake request is made. In section 7.3.1, we'll look at setting up a project for these unit tests and how to run them with `dotnet`.

For the rest of this chapter, we'll work at the code level and implement unit tests and service-level tests for the Loyalty Program microservice. When you implemented Loyalty Program in chapter 4, it didn't have an event feed; but for these examples you'll add an event feed that other microservices can subscribe to.

7.3 Writing unit tests using Nancy.Testing

In this section, you'll implement some unit tests for the endpoints in the Loyalty Program microservice. In chapter 4, you saw that Loyalty Program has three command and query endpoints:

- An HTTP `GET` endpoint at URLs of the form /users/{userId} that responds with a representation of the user
- An HTTP `POST` endpoint to /users/ that expects a representation of a user in the body of the request and then registers that user in the loyalty program
- An HTTP `PUT` endpoint at URLs of the form /users/{userId} that expects a representation of a user in the body of the request and then updates an already-registered user

Let's write tests for these endpoints. The Loyalty Program microservice has an event feed for which you'll also write a test. You won't write comprehensive tests for the endpoints and event feed in Loyalty Program—only enough to see how tests against Nancy endpoints are written.

In the following subsections, you'll do the following:

- Set up a test project to house unit tests for the Loyalty Program microservice.
- Write tests that use `Browser` from Nancy.Testing to test endpoints in Loyalty Program and that let the code in the microservice use the real database. You'll write three such tests, one for each of these pieces of functionality:
  - A test that tries to read a user that doesn't exist
  - A test that creates a user and reads it back out
  - A test that modifies a user and reads it back out
- Write tests that also use `Browser` to test an endpoint but are limited in scope by a mocked database injected in the endpoint under test. These tests test the event feed in the microservice.

When you're finished, you'll have learned to write unit tests for Nancy endpoints both with and without a real database.

7.3.1 Setting up a unit-test project

Before you can start writing tests, you need a project to house them. For that, create a new project next to the `LoyaltyProgram` project, and call it `LoyaltyProgramUnitTests`. If you create the project with Visual Studio, choose the Class Library (.NET Core) template from the dialog; and if you use you Yeoman, choose Unit Test Project (xUnit.net) from the menu.

Your solution should look similar to this:

```
C:.
├──LoyaltyProgram
|       Bootstrapper.cs
|       project.json
|       README.md
|       Startup.cs
|       UsersModule.cs
|       YamlSerializerDeserializer.cs
|
├──LoyaltyProgram
```

```
|   └─────EventFeed
|             Event.cs
|             EventsFeedModule.cs
|             EventStore.cs
|             IEventStore.cs
|
├──LoyaltyProgramEventConsumer
|       Program.cs
|       project.json
|
└──LoyaltyProgramUnitTests
        project.json
        Class1.cs
```

If you used Yeoman to create the new `LoyaltyProgramUnitTests` project, you're ready to run your first tests. But if you used the Visual Studio template, you need to edit the Class1.cs and project.json files a bit. The following listing shows how Class1.cs should look.

Listing 7.2 Class1.cs file

```
using Xunit;

namespace UnitTest
{
    // see example explanation on xUnit.net website:
    // https://xunit.github.io/docs/getting-started-dotnet-core.html
    public class Class1
    {
        [Fact]
        public void PassingTest()
        {
            Assert.Equal(4, Add(2, 2));
        }

        [Fact]
        public void FailingTest()
        {
            Assert.Equal(5, Add(2, 2));
        }

        int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

In the project.json file, add the following to set up a test command that refers to the `xunit` test runner:

```
"testRunner": "xunit",
```

The xunit test runner is added to the project via the NuGet package dotnet-test-xunit, and the xUnit package is installed. Here are all the dependencies:

```
"dependencies": {
    "dotnet-test-xunit": "2.2.0-preview2-build1029",
    "Microsoft.NETCore.App": {
      "version": "1.0.0",
      "type": "platform"
    },
    "xunit": "2.1.0"
  },
```

You can now go to the LoyaltyProgramUnitTests folder in PowerShell and restore the NuGet packages as usual, using dotnet:

```
PS> dotnet restore
```

The Class1.cs file now contains two small xUnit tests: one that passes and one that fails. You run them with dotnet like this:

```
PS> dotnet test
```

Once you have the initial tests running, add a dependency on Nancy.Testing so you can use Browser and later ConfigurableBootstrapper. Also add a dependency on LoyaltyProgram so you can begin testing it. The dependencies now look like this:

```
"dependencies": {
    "dotnet-test-xunit": "2.2.0-preview2-build1029",
    "Microsoft.NETCore.App": {
      "version": "1.0.0",
      "type": "platform"
    },
    "xunit": "2.1.0",
    "Nancy.Testing": "2.0.0--barneyrubble",
    "LoyaltyProgram": {"target": "project"}        <-- Project reference
  },
```

The last line is the reference to the LoyaltyProgram project. As you can see, the project references in project.json look almost like NuGet references. You don't specify a version for LoyaltyProgram because you want the test to run against the version of the LoyaltyProgram code that you have next to the LoyaltyProgramUnitTests project.

7.3.2 Using the Browser object to unit-test endpoints

Now that you have a test project set up, you can begin adding tests to it. The first test you'll add is very simple: given that there are no registered users in the Loyalty Program microservice, the test queries for a user and expects to get back a response with a 404 Not Found status code. Add a file called userModule_should.cs to the LoyaltyProgramUnitTests project, and put the following code in it.

Listing 7.3 First test for the users endpoint

```
namespace LoyaltyProgramUnitTests
{
  using LoyaltyProgram;
  using Nancy;
  using Nancy.Testing;
  using Xunit;

  public class UserModule_should
  {
    private Browser sut;

    public UserModule_should()
    {
      this.sut = new Browser(
        new Bootstrapper(),
        defaultsTo => defaultsTo.Accept("application/json"));
    }

    [Fact]
    public void respond_not_found_when_queried_for_unregistered_user()
    {
      var actual = await sut.Get("/users/1000");
      Assert.Equal(HttpStatusCode.NotFound, actual.StatusCode);
    }
  }
}
```

Remember that sut stands for "system under test."

Real LoyaltyProgram bootstrapper

All "requests" accept JSON

Requests a user that doesn't exist

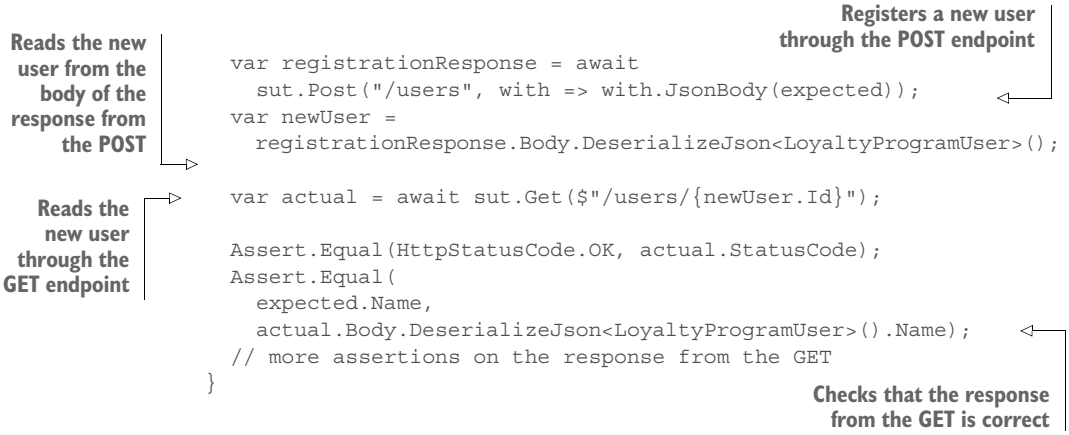
The most interesting part of this test class is in the constructor, where you create a `Browser` object. When xUnit runs, it creates an instance of `UserModule_should` and then calls a method with the `Fact` attribute on that instance. Unlike most other .NET test frameworks, xUnit create a new, clean instance for each `Fact` method.

The `Browser` object in listing 7.3 is initialized with the real bootstrapper from `LoyaltyProgram`. This means the `LoyaltyProgram` application that the `Browser` calls into is wired up exactly the same way it is when it runs on top of a real web server and receives real HTTP requests. Furthermore, for convenience, you set a default `Accept` header on `Browser`. This header will be added to all requests made through the `Browser` object unless explicitly overridden. For instance, `sut.Get("/users/1000")` has the `Accept` header set.

Let's move on to a test that registers a new user and then queries it to check that it was registered as it should be. Add the following test to the `UserModule_should` class.

Listing 7.4 Test for registering a user through the users endpoint

```
[Fact]
public void allow_to_register_new_user()
{
  var expected =
    new LoyaltyProgramUser() { Name = "Chr" };
```




```
    var registrationResponse = await
      sut.Post("/users", with => with.JsonBody(expected));
    var newUser =
      registrationResponse.Body.DeserializeJson<LoyaltyProgramUser>();

    var actual = await sut.Get($"/users/{newUser.Id}");

    Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
    Assert.Equal(
      expected.Name,
      actual.Body.DeserializeJson<LoyaltyProgramUser>().Name);
    // more assertions on the response from the GET
  }
```

Here, you see another use of the `Browser` object. For instance, you add a body to the `Post` via the lambda in the second argument. In that lambda, you can do a variety of things to the request, such as adding headers, cookies, form values, a host name, or an identity, or choosing between HTTP and HTTPS. Here, you add a body to the request.

The last test you'll add registers a user and then modifies it via the `PUT` endpoint in the Loyalty Program microservice. Add it to UserModule_should.cs.

Listing 7.5 Test for modifying users through the users endpoint

```
[Fact]
public void allow_modifying_users()
{
  var expected = "jane";
  var user = new LoyaltyProgramUser() { Name = "Chr" };
  var registrationResponse = await
    sut.Post("/users", with => with.JsonBody(user));
  var newUser =
    registrationResponse.Body.DeserializeJson<LoyaltyProgramUser>();

  newUser.Name = expected;
  var actual = await
    sut.Put($"/users/{newUser.Id}", with => with.JsonBody(newUser));

  Assert.Equal(
    expected,
    actual.Body.DeserializeJson<LoyaltyProgramUser>().Name);
}
```


There's nothing new in this code compared to what you've seen in the two previous tests. But I wanted to include it because it's a good illustration of the kind of unit tests I think you should write for the endpoints in your microservices: unit tests that focus on the behavior the endpoints provide rather than on testing just one endpoint in isolation.

7.3.3 Using a configurable bootstrapper to inject mocks into endpoints

Now that you've tested the endpoints in `UserModule`, let's turn to testing the `LoyaltyProgram` event feed. The event feed is a Nancy module that depends on an `IEventStore` to store and read events. Here's the `IEventStore` interface.

Listing 7.6 `IEventStore` interface

```
using System.Collections.Generic;

namespace LoyaltyProgram.EventFeed
{
  public interface IEventStore
  {
    IEnumerable<Event> GetEvents(
      long firstEventSequenceNumber,
      long lastEventSequenceNumber);
    void Raise(string eventName, object content);
  }
}
```

Reads events from the event store

Stores events to the event store

You saw an event feed in chapter 4, but I'll repeat it here, to remind you how it works.

Listing 7.7 Event feed

```
namespace LoyaltyProgram.EventFeed
{
  using Nancy;

  public class EventsFeedModule : NancyModule
  {
    public EventsFeedModule(IEventStore eventStore) : base("/events")
    {
      Get("/", _ =>
      {
        long firstEventSequenceNumber, lastEventSequenceNumber;
        if (!long.TryParse(this.Request.Query.start.Value,
          out firstEventSequenceNumber))
          firstEventSequenceNumber = 0;
        if (!long.TryParse(this.Request.Query.end.Value,
          out lastEventSequenceNumber))
          lastEventSequenceNumber = 50;

        return
          eventStore.GetEvents(
            firstEventSequenceNumber,
            lastEventSequenceNumber);
      });
    }
  }
}
```

Gets the start value from the query string

Gets the end value from the query string

Reads events "start" through "end" from the event store

As you can see, the event feed is a Nancy module that responds to requests to /events with the events it reads from `IEventStore`. You want to write a test to check whether the event feed returns exactly the event from the `IEventFeed`. Toward that end, you want to control which events `IEventStore` returns. So, you'll create a fake implementation of `IEventStore` and use that in the test.

Listing 7.8 Fake `IEventStore` to use in tests

```
public class FakeEventStore : IEventStore
{
  public IEnumerable<Event> GetEvents(
    long firstEventSequenceNumber,
    long lastEventSequenceNumber)
  {
    if (firstEventSequenceNumber > 100)
      return Enumerable.Empty<Event>();
    else
      return
        Enumerable
        .Range((int) firstEventSequenceNumber,
               (int) (lastEventSequenceNumber - firstEventSequenceNumber))
        .Select(i =>
          new Event(
            i,
            DateTimeOffset.Now,
            "some event",
            new Object()));
  }

  public void Raise(string eventName, object content) {}
}
```

Returns a list of fake events when firstEventSequenceNumber is less than 100

With this fake implementation of an event store, you know the event store will return a list of events only if the `firstEventSequenceNumber` argument is less than 100. Otherwise, `FakeEventStore` will return an empty list of events. If you inject this `IEventStore` implementation into `EventsFeedModule`, you'll know which events `EventsFeedModule` will get from the event store and therefore which events it should return.

You can use another feature of Nancy.Testing to inject the fake `IEventStore` implementation into `EventsFeedModule`: `ConfigurableBootstrapper`, which allows you to modify how the Nancy application under test is configured. Here, you'll use `ConfigurableBootstrapper` to set up `FakeEventStore` as the implementation of `IEventStore` when creating the `Browser` object. That is done with the following piece of code.

Listing 7.9 Using the fake event store while testing

```
this.sut = new Browser(
  with => with
    .Module<EventsFeedModule>()
    .Dependency<IEventStore>(typeof(FakeEventStore)),
  withDefault => withDefault.Accept("application/json"));
```

with has the type ConfigurableBootstrapper

Limits Browser to using EventsFeedModule only

Registers FakeEventStore as the implementation of IEventStore

Adds a JSON Accept header to all requests

With this code in the tests, constructor instances of `EventsFeedModule` will have `FakeEventStore` injected. You can use that to write two tests:

- A test that asserts that events are returned from the feed when the start number in the request is less than 100
- A test that asserts that no events are returned when the start number is greater than 100

Listing 7.10 Tests for the event feed, using the fake event store

```
using System;
using System.Collections.Generic;
using System.Linq;
using LoyaltyProgram.EventFeed;
using Nancy;
using Nancy.Testing;
using Xunit;

public class EventFeed_should
{
  private Browser sut;

  public EventFeed_should()
  {
    this.sut = new Browser(
      with => with
        .Module<EventsFeedModule>()
        .Dependency<IEventStore>(typeof(FakeEventStore)),
      withDefault => withDefault.Accept("application/json"));
  }

  [Fact]
  public void return_events_when_from_event_store()
  {
    var actual = await sut.Get("/events/", with =>
    {
      with.Query("start", "0");
      with.Query("end", "100");
    });
```

Creates Browser configured to use FakeEventStore

Makes a request to /events with the query string "start=0&end=100"

```
    Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
    Assert.StartsWith("application/json", actual.ContentType);
    Assert.Equal(100,
      actual.Body.DeserializeJson<IEnumerable<Event>>().Count());
  }

  [Fact]
  public void return_empty_response_when_there_are_no_more_events()
  {
    var actual = wait sut.Get("/events/", with =>
    {
      with.Query("start", "200");
      with.Query("end", "300");
    });

    Assert.Empty(actual.Body.DeserializeJson<IEnumerable<Event>>());
  }
}
```

Makes a request to /events with the query string "start=200&end=300"

Now that you have some unit tests in place, you can run them with `dotnet`, as you saw earlier. When you do, xUnit will scan for classes with `Fact` methods and then execute each `Fact` method. The output from the tests shows a summary of how many tests ran, how many errors there were, how many tests failed, and how many were skipped:

```
PS > dotnet test
xUnit.net .NET CLI test runner (64-bit .NET Core win10-x64)
  Discovering: LoyaltyProgramUnitTests
  Discovered:  LoyaltyProgramUnitTests
  Starting:    LoyaltyProgramUnitTests
  Finished:    LoyaltyProgramUnitTests
=== TEST EXECUTION SUMMARY ===
   LoyaltyProgramUnitTests  Total: 6, Errors: 0, Failed: 0, Skipped: 0, Time:
      2.375s
SUMMARY: Total: 1 targets, Passed: 1, Failed: 0.
```

As you can see, six tests were run, and none of them failed. In other words, all tests passed.

Now that you have tests for `EventsFeedModule` and `UsersModule`, you're off to a good start writing unit tests for endpoints in your microservices. In real life, these tests aren't sufficient; I'd write more tests for edge cases and error scenarios. But now you know how to write those tests using Nancy.Testing.

7.4 Writing service-level tests

Let's move on to writing service-level tests for the entire Loyalty Program microservice. Service-level tests interact with a microservice from the outside and provide the microservice with mocked versions of its collaborators.

Loyalty Program makes requests to two collaborators: the event feed in the Special Offers microservice and the API of the Notifications microservice. The service-level tests for Loyalty Program go through these steps:

1. Set up two endpoints in the same process as the test:
   - One that works as a mocked special-offer event feed
   - One that works as a mocked notification endpoint
2. Start the Loyalty Program microservice in separate processes, and configure it to use the mocked endpoints in place of the real collaborators. This means whenever Loyalty Program needs to call one of its collaborators, it will call one of the mocked endpoints.
3. Execute a scenario against Loyalty Program as a sequence of HTTP requests.
4. Record any calls to the mocked endpoints.
5. Make assertions on the responses from Loyalty Program and on the requests made to the mocked endpoints.

Figure 7.5 shows the runtime setup for the service-level tests for the Loyalty Program microservice.

You'll follow these steps to create the test setup from figure 7.5:

1. Create a test project for the service-level tests.
2. Create the mocked endpoints for the special-offers event feed and the notification endpoint.
3. Start both processes of the Loyalty Program microservice: the Nancy application containing the HTTP API and the event consumer.
4. Write test code that executes a test scenario against Loyalty Program.

When that setup is in place, you'll write a test that uses it.

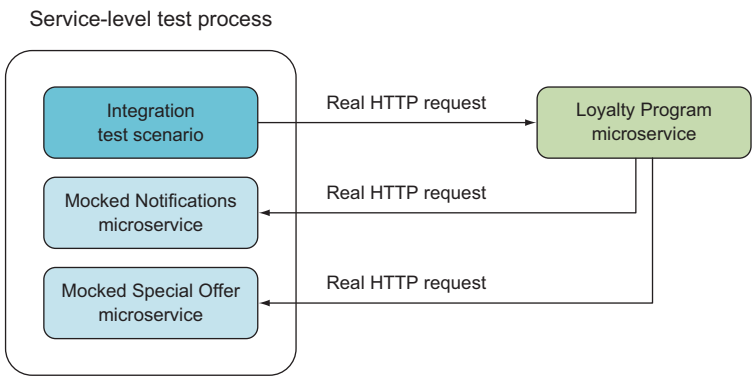


Figure 7.5 A service-level test executes a scenario against the API of the microservice under test but configures the microservice to use mocked endpoints running in the same process as the test, in place of real collaborators. When a service-level test runs, it makes real HTTP requests to the microservice under test, which makes real HTTP requests back to mocked endpoints as needed. The test can inspect the responses from the microservice under test as well as the calls it makes to the mocked endpoints.

7.4.1 Creating a service-level test project

For the service-level tests, you'll create a new test project exactly like the unit-test project you create earlier. That is, create a project based on either the ASP.NET Test Project Template in Visual Studio or the Unit Test project template in Yeoman, and call it `LoyaltyProgramIntegrationTest`. Just like the unit-test project, place this new project side by side with `LoyaltyProgram`. You now have four projects:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         4/6/2016   8:53 PM                LoyaltyProgram
d-----         4/6/2016   8:53 PM                LoyaltyProgramEventConsumer
d-----         4/6/2016   8:53 PM                LoyaltyProgramIntegrationTest
d-----         8/6/2016  10:59 PM                LoyaltyProgramUnitTests
```

These are the two projects that make up the Loyalty Program microservice—the Nancy application and the event consumer—and the test projects that go along with the microservice.

7.4.2 Creating mocked endpoints

As shown in figure 7.5, you need to create mocked versions of the endpoints in the Special Offers microservice and the Notifications microservice that the Loyalty Program microservice uses. You'll do so by writing two simple Nancy modules, each of which implements an endpoint that returns a hardcoded response. Listing 7.11 shows the mocked special-offers event feed endpoint, and listing 7.12 shows the mocked notifications endpoint.

Listing 7.11 Mock event feed returning hardcoded events

```
public class MockEventFeed : NancyModule
{
  public static AutoResetEvent polled =
    new AutoResetEvent(initialState: false);

  public MockEventFeed()
  {
    this.Get("/events", _ =>
    {
      polled.Set();
      return new []
      {
        new
        {
          SequenceNumber = 1,
          Name= "baz",
          Content = new
          {
```

Signals to the test that Loyalty Program has been polled for events

Returns a hardcoded response

```
                OfferName = "foo",
                Description = "bar",
                item = new { ProductName = "name" }
              }
            }
          };
        });
    }
}
```

Listing 7.12 Mock endpoint that records when it was called

```
public class MockNotifications : NancyModule
{
  public static AutoResetEvent notificationWasSent =
    new AutoResetEvent(initialState: false);

  public MockNotifications()
  {
    this.Get("/notify", _ =>
    {
      notificationWasSent.Set();
      return 200;
    });
  }
}
```

Used later in the test to make assertions on

Returns a hardcoded response

The plan is to run these two modules in the test process. To do that, you'll use Nancy on top of ASP.NET Core like you usually do. You need to add the Microsoft.AspNetCore.Owin NuGet packages and add Nancy and `LoyaltyProgram` as dependencies. The dependencies section in the project.json file now looks like this.

Listing 7.13 Integration project dependencies, including Nancy

```
"dependencies": {
    "dotnet-test-xunit": "2.2.0-preview2-build1029",
    "Microsoft.NETCore.App": {
      "version": "1.0.0",
      "type": "platform"
    },
    "xunit": "2.1.0",
    "Microsoft.AspNetCore.Owin": "1.0.0",
    "Nancy": "2.0.0-barneyrubble",
    "LoyaltyProgram": { "target": "project" }
  },
```

Next, add a file called RegisterUserAndGetNotification.cs containing the following code, which uses `Nancy.Hosting.Self` to start a Nancy application in the test process.

Listing 7.14 Starting up Nancy inside the test process

```
public class RegisterUserAndGetNotification : IDisposable
{
  private readonly NancyHost hostForMockEndpoints;

  public RegisterUserAndGetNotification()
  {
    StartFakeEndpoints();
  }

  private void StartFakeEndpoints()
  {
    this.hostForFakeEndpoints = new WebHostBuilder()
      .UseKestrel()
      .UseContentRoot(Directory.GetCurrentDirectory())
      .UseStartup<FakeStartup>()
      .UseUrls("http://localhost:5001")
      .Build();

    new Thread(() => this.hostForFakeEndpoints.Run()).Start();
  }
}

public class FakeStartup
{
  public void Configure(IApplicationBuilder app)
  {
    app.UseOwin(buildFunc => buildFunc.UseNancy());
  }
}
```

Creates an ASP.NET Core application

Uses FakeStartup to bootstrap the ASP.NET Core application

Lets the ASP.NET Core application listen on port 5001

Adds Nancy to the ASP.NET Core application

Later, you'll add a `Fact` method to this class: then, when you run xUnit, it will find this class and instantiate it to execute `Fact`. The constructor starts up Nancy, which will automatically discover the `MockEventsFeed` and `MockUsersModule` modules and expose the endpoints defined in them. This is all you need to create mocked endpoints in the service-level test process.

7.4.3 Starting all the processes of the microservice under test

With the mocked endpoints running, you're ready to start up Loyalty Program. The microservice consists of two processes: a Nancy application and the event consumer. You add the code to start those to the setup in RegisterUserAndGetNotification. The following listing shows only new code—leave the existing code to start and stop Nancy.

Listing 7.15 Starting the microservice in a separate process

```
public class RegisterUserAndGetNotification : IDisposable
{
  ...
  private Process eventConsumer;
```

```
  private Process web;

  public RegisterUserAndGetNotification()
  {
    StartLoyaltyProgram();
    ...
  }

  private void StartLoyaltyProgram()
  {
    StartEventConsumer();
    StartLoyaltyProgramApi();
  }

  private void StartLoyaltyProgramApi()
  {
    var apiInfo = new ProcessStartInfo("dotnet.exe")
    {
      Arguments = "run",
      WorkingDirectory = "../LoyaltyProgram"
    };
    this.api = Process.Start(apiInfo);
  }

  private void StartEventConsumer()
  {
    var eventConsumerInfo = new ProcessStartInfo("dotnet.exe")
    {
      Arguments = "run localhost:5001",
      WorkingDirectory = "../LoyaltyProgramEventConsumer"
    };
    this.eventConsumer = Process.Start(eventConsumerInfo);
  }

  public void Dispose()
  {
    this.eventConsumer.Dispose();
    this.api.Dispose();
  }
}
```

Setup for running the command "dotnet run" in the LoyaltyProgram folder

Starts the LoyaltyProgram process

Setup for running the event consumer

Starts the event-consumer process

Closes the processes, and releases resources

This code spawns two `dotnet` processes, one for each process in the Loyalty Program microservice. This is like running `dotnet` from the command line, so running the Nancy application is the same as usual. Running the event consumer is different, and you need to solve these two problems:

- The event consumer expects to run as a Windows service. Now it also needs to be able to run like a simple process.
- In the following line from listing 7.15, the event consumer doesn't understand the command-line argument `localhost:5001`, which is the host name for the mocked endpoints you want the event consumer to use in place of the real collaborators:

```
Arguments = "run localhost:5001",
```

Both of these issues are easy to solve. You just change the `Main` method in the event consumer to the following.

Listing 7.16 Letting the consumer run as a Windows or normal process

```
public static void Main(string[] args) => new Program().Entry(args);

public void Entry(string[] args)
{
  this.subscriber = new EventSubscriber(args[0]);
  if (args.Length >= 2 && args[1].Equals("--service"))
    Run(this);
  else
  {
    OnStart(null);
    Console.ReadLine();
  }
}
```

Reads the host name from the command-line argument

Runs as a service if there's a --service in the command-line arguments

Runs the start method by hand

Now both processes of the Loyalty Program microservice are started from the test startup code. A nice side effect of the changes to the event consumer is that it's also easier to run by hand for testing reasons.

7.4.4 Executing the test scenario against the microservice under test

Finally, you're ready to write the test. It has three steps:

1. Make an HTTP request to register a user.
2. Wait for the Loyalty Program microservice to poll for events.
3. Assert that a request to the notifications endpoint was made.

In code, the test goes in the RegisterUserAndGetNotification file and is as follows.

Listing 7.17 Service-level test using an outside loyalty program

```
[Fact]
public void Scenario()
{
  RegisterNewUser();
  WaitForConsumerToReadSpecialOffersEvents();
  AssertNotificationWassent();
}

private async Task RegisterNewUser()
{
  using (var httpClient = new HttpClient())
  {
    httpClient.BaseAddress = new Uri("http://localhost:5000");
    var response = await
```

```
      httpClient.PostAsync(
        "/users/",
        new StringContent(
          JsonConvert.SerializeObject(new LoyaltyProgramUser()),
          Encoding.UTF8,
          "application/json")).ConfigureAwait(false);
    Assert.Equal(HttpStatusCode.Created, response.StatusCode);
    Console.WriteLine("registered users");
  }
}

private static void WaitForConsumerToReadSpecialOffersEvents()
{
  Assert.True(MockEventFeed.polled.WaitOne(30000));
  Thread.Sleep(100);
}

private static void AssertNotificationWassent()
{
  Assert.True(MockNotifications.NotificationWasSent);
}
```

Sends a request to register a user

Puts a user into the request

Waits for the microservice to poll the event feed, and fails if it doesn't poll

Waits to give the microservice time to handle the event from the feed

You can run the test in PowerShell with `dotnet`:

```
PS> dotnet test
```

This will open two command windows: one with each of the processes in the Loyalty Program microservice. The test runs, and, when it finishes, the two windows are closed. The output from xUnit is as follows:

```
Discovering: LoyaltyProgramIntegrationTest
  Discovered:  LoyaltyProgramIntegrationTest
  Starting:    LoyaltyProgramIntegrationTest
    LoyaltyProgramIntegrationTests.RegisterUserAndGetNotification.Scenario
  Finished:    LoyaltyProgramIntegrationTest
=== TEST EXECUTION SUMMARY ===
   LoyaltyProgramIntegrationTest  Total: 1, Errors: 0, Failed: 0, Skipped:
      ➥ 0, Time: 12.563s
```

This test is slow, and you had to do some setup before you were ready to write it. This is why such tests are higher on the test pyramid than the unit tests you wrote earlier. You should have only a few of this kind of test, whereas you can have many unit tests.

7.5 Summary

- The test pyramid tells you to have few system-level tests that test the complete system, several service-level tests for each microservice, and many unit tests for each microservice.
- System-level tests are likely to be slow and are very imprecise.

- You should write system-level tests for important success scenarios, to provide some test coverage for most of the system.
- Service-level tests are likely to be slow, but they're faster and more precise than system-level tests.
- You should write service-level tests for success scenarios and important failure scenarios for each microservice. This adds more test coverage to each microservice than just the system-level tests.
- You can use the process for writing service-level tests as the basis for writing contract tests that verify the assumption one microservice makes about the API and behavior of another microservice. In terms of the test pyramid, contract tests are between system-level tests and service-level tests.
- Unit tests are fast and should be kept fast. They're also precise, because they target a specific, narrow piece of functionality.
- You should write unit tests for success and failure scenarios alike. Use them to cover edge cases that are harder to cover with higher-level tests.
- I recommend working in an outside-in fashion with each microservice: write service-level tests first, and then begin writing unit tests when the service-level tests become awkward to work with.
- The Nancy.Testing library is a powerful companion to Nancy that makes it easy to test endpoints in Nancy modules.
- You use the `Browser` type in Nancy.Testing to test endpoints through a nice API that lets you simulate HTTP requests. Calls through the `Browser` object look exactly like real HTTP requests to the endpoint handlers in Nancy modules.
- You test endpoints through `Browser` both with real data stores and with mocked data stores.
- You can write service-level tests where you do the following:
  - Write mocked endpoints for the collaborators of the microservice under test, and use Nancy to host these in the test process.
  - Start up all the processes of the microservice under test, passing in the configuration through command-line arguments.
  - Write scenarios that interact with the microservice under test via HTTP requests.
  - Make assertions both on the response from the microservice under test and on the requests it makes to its collaborators.
- You can use the xUnit test framework to write and run your automated tests.
- xUnit can be run with `dotnet`.

index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y


manning