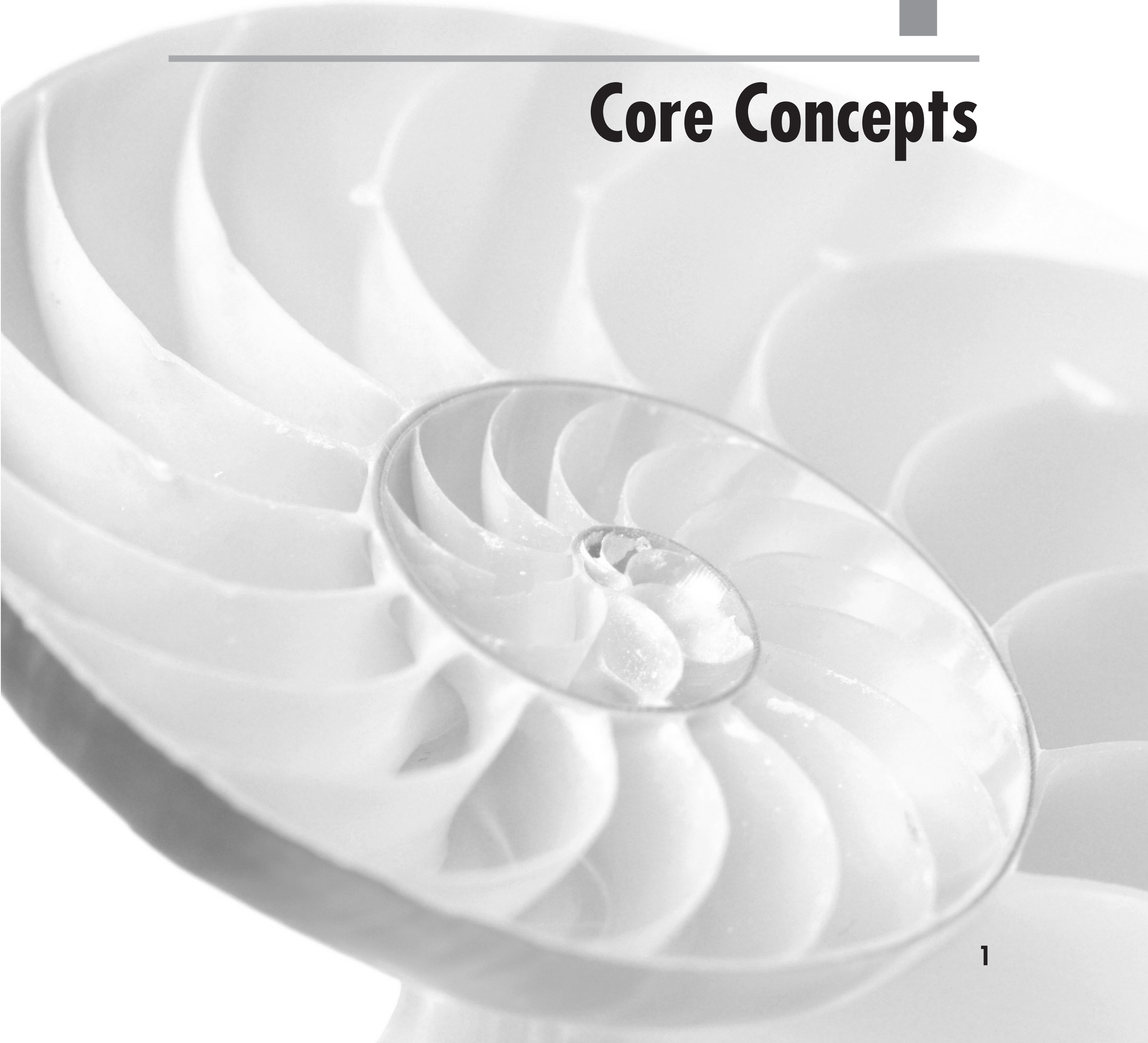
CHAPTER 1

Core Concepts



Startups face extreme amounts of uncertainty. To build a successful startup, you must be as flexible as possible. You also need to be resourceful and adapt quickly to changing conditions. These extreme requirements put on the software teams make scalability even more important and challenging than in slowly changing businesses. Things that can take an entire year in a corporate environment may need to happen in just a matter of weeks in a startup. If you are successful and lucky, you may need to scale your capacity up tenfold in a matter of weeks, just to have to scale back down a few months later.

Scalability is a difficult matter for any engineer, and it presents special challenges in the startup environment. As such, leveraging the work done by major players in this space, including Amazon, Azure, and Google clouds, can reduce the overall scope of your work and allow you to focus on addressing your specific needs. As we discuss scalability concepts in the book, we'll also look at some of the services you can apply to address each challenge. Understanding scalability is best approached gradually, and to that end, I'll keep things simple to begin with by focusing on the core concepts from a high level. Anyone with a basic understanding of web application development should feel comfortable diving into the book. As we move forward, I'll take a deeper dive into details of each concept. For now, it's important to establish three main pillars of scalability: what it is and how it evolves, what it looks like in a large-scale application, and what its application architecture looks like.

To fully grasp the concepts in this chapter, it may be worth revisiting it after you've read the entire book. At first, some concepts may seem quite abstract, but you'll find everything comes together nicely as you more fully understand the big picture. This chapter contains a number of diagrams as well. These diagrams often carry much more information than you may notice at first glance. Getting comfortable with drawing infrastructure and architecture diagrams will not only help you get the most out of this book, but may also help you during your next job interview.

What Is Scalability?

Before we dive into the core concepts, let's make sure we are approaching scalability with a unified definition. You're likely reading this book because you want to enable your web applications to scale—or to scale more efficiently. But what does it mean to scale?

> *Scalability* is an ability to adjust the capacity of the system to cost-efficiently fulfill the demands. Scalability usually means an ability to handle more users, clients, data, transactions, or requests without affecting the user experience. It is important to remember that scalability should allow us to scale down as much as scale up and that scaling should be relatively cheap and quick to do.

The ability to scale is measured in different dimensions, as we may need to scale in different ways. Most scalability issues can be boiled down to just a few measurements:

- **Handling more data** This is one of the most common challenges. As your business grows and becomes more popular, you will be handling more and more data. You will have to efficiently handle more user accounts, more products, more location data, and more pieces of digital content. Processing more data puts pressure on your system, as data needs to be sorted, searched through, read from disks, written to disks, and sent over the network. Especially today, with the growing popularity of big data analytics, companies become greedier and greedier about storing ever-growing amounts of data without ever deleting it.
- **Handling higher concurrency levels** Concurrency measures how many clients your system can serve at the same time. If you are building a web-based application, concurrency means how many users can use your application at the same time without affecting their user experience. Concurrency is difficult, as your servers have a limited amount of central processing units (CPUs) and execution threads. It is even more difficult, as you may need to synchronize parallel execution of your code to ensure consistency of your data. Higher concurrency means more open connections, more active threads, more messages being processed at the same time, and more CPU context switches.
- **Handling higher interaction rates** The third dimension of scalability is the rate of interactions between your system and your clients. It is related to concurrency, but is a slightly different dimension. The rate of interactions measures how often your clients exchange information with your servers. For example, if you are building a website, your clients would navigate from

page to page every 15 to 120 seconds. If you are building a multiplayer mobile game, however, you may need to exchange messages multiple times per second. The rate of interactions can be higher or lower independently of the amount of concurrent users, and it depends more on the type of the application you are building. The main challenge related to the interaction rate is latency. As your interactions rate grows, you need to be able to serve responses quicker, which requires faster reads/writes and often drives requirements for higher concurrency levels.

The scalability of your system will usually be defined by the combination of these three requirements. Scaling down is usually less important than the ability to scale up, but reducing waste and inefficiencies is an important factor nonetheless, especially so for startups, where every investment may become a waste as business requirements change.

As you have probably noticed, scalability is related to performance, but it is not the same thing. Performance measures how long it takes to process a request or to perform a certain task, whereas scalability measures how much we can grow (or shrink).

For example, if you had 100 concurrent users, with each user sending a request, on average, once every 5 seconds, you would end up with a throughput requirement of 20 requests per second. Performance would decide how much time you need to serve these 20 requests per second, and scalability would decide how many more users you can handle and how many more requests they can send without degrading the user experience.

Finally, scalability of a software product may be constrained by how many engineers can be working on the system. As your system grows, you will need to consider organizational scalability as well; otherwise, you will not be able to make changes or adapt quickly enough. Even though organizational scalability may seem unrelated to technology, it actually may be limited by the architecture and design of your system. If your system is very tightly interconnected, you may struggle to scale your engineering team, as everyone will work on the same codebase. Growing a single engineering team above 8 to 15 people becomes inefficient, as the communication overhead grows exponentially as the team size grows.[40]

HINT

To fully appreciate how scalability affects startups, try to assume a more business-oriented perspective. Ask yourself, "What are the constraints that could prevent our business from growing?" It is not just about raw throughput; it involves development processes, teams, and code structure. I will explore these aspects of scalability in more detail in Chapter 9 of this book.

Evolution from a Single Server to a Global Audience

As a young engineer I used to build web applications that were hosted on a single server, and this is probably how most of us get started. During my career I have worked for different companies and I have witnessed applications in different scalability evolution stages. Before we go deeper into scalability, I would like to present some of these evolution stages to better explain how you go from a single server sitting under your desk to thousands of servers spread all over the world.

I will keep it at a very high level here, as I will go into more detail in later chapters. Discussing evolution stages will also allow me to introduce different concepts and gradually move toward more complex topics. Keep in mind that many of the scalability evolution stages presented here can only work if you plan for them from the beginning. In most cases, a real-world system would not evolve exactly in this way, as it would likely need to be rewritten a couple of times. Most of the time, a system is designed and born in a particular evolution stage and remains in it for its lifetime, or manages to move up one or two steps on the ladder before reaching its architectural limits.

HINT

Avoid full application rewrites at all costs,[45] especially if you work in a startup. Rewrites always take much longer than you initially expect and are much more difficult than initially anticipated. Based on my experience, you end up with a similar mess just two years later.

Single-Server Configuration

Let's begin with a single-server setup, as it is the simplest configuration possible and this is how many small projects get started. In this scenario, I assume that your entire application runs on a single machine. Figure 1-1 shows how all the traffic for every user request is handled by the same server. Usually, the Domain Name System (DNS) server is used as a paid service provided by the hosting company and is not running on your own server. In this scenario, users connect to the DNS to obtain the Internet Protocol (IP) address of the server where your website is hosted. Once the IP address is obtained, they send Hypertext Transfer Protocol (HTTP) requests directly to your web server.

Since your setup consists of only one machine, it needs to perform all the duties necessary to make your application run. It may have a database management system running (like MySQL or Postgres), as well as serving images and dynamic content from within your application.

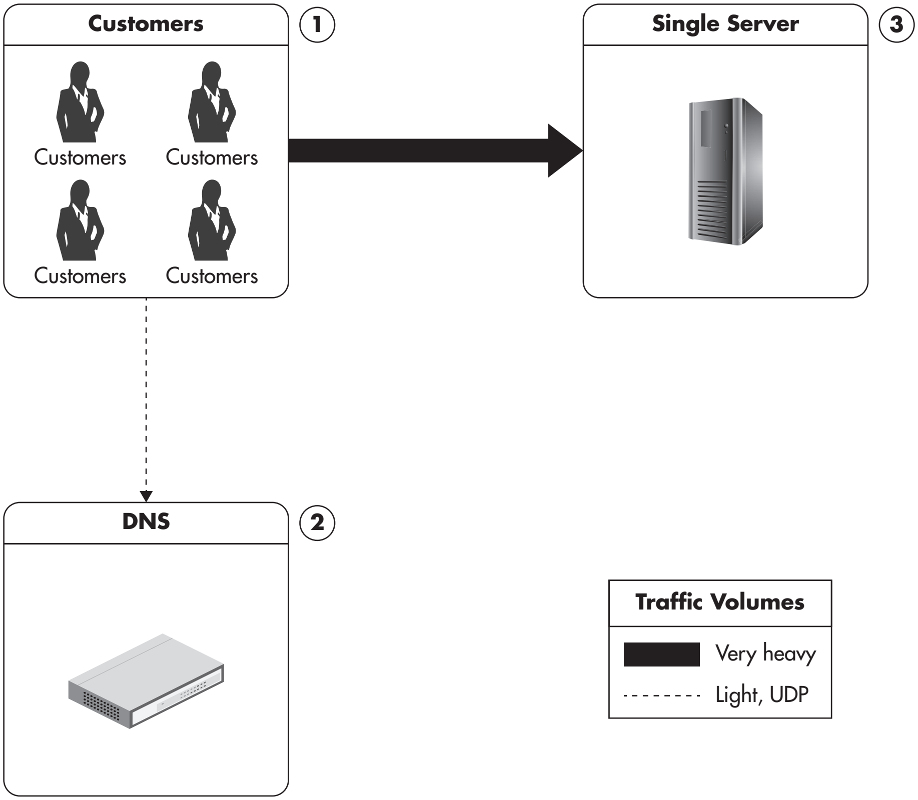


Figure 1-1 *Single-server configuration*

Figure 1-1 shows the distribution of traffic in a single-server configuration. Clients would first connect to the DNS server to resolve the IP address of your domain, and then they would start requesting multiple resources from your web server. Any web pages, images, Cascading Style Sheet (CSS) files, and videos have to be generated or served by your server, and all of the traffic and processing will have to be handled by your single machine. I use different weights of arrows on the diagram to indicate the proportion of traffic coming to each component.

An application like this would be typical of a simple company website with a product catalog, a blog, a forum, or a self-service web application. Small websites may not even need a dedicated server and can often be hosted on a virtual private server (VPS) or on shared hosting.

Virtual private server is a term used by hosting providers to describe a virtual machine for rent. When you purchase a VPS instance, it is hosted together with other VPS instances on a shared host machine. VPS behaves as a regular server—you have your own operating system and full privileges. VPS is cheaper than a dedicated server, as multiple instances can exist at the same time on the same physical machine. VPS is a good starting point, as it is cheap and can usually be upgraded instantly (you can add more random access memory [RAM] and CPU power with a click of a button).

Shared hosting is the cheapest hosting solution, where you purchase a user account without administrative privileges. Your account is installed on a server together with many other customers' accounts. It is a good starting point for the smallest websites or landing pages, but it is too limiting so it is not a recommended option.

For sites with low traffic, a single-server configuration may be enough to handle the requests made by clients. There are many reasons, though, why this configuration is not going to take you far scalability-wise:

- Your user base grows, thereby increasing traffic. Each user creates additional load on the servers, and serving each user consumes more resources, including memory, CPU time, and disk input/output (I/O).
- Your database grows as you continue to add more data. As this happens, your database queries begin to slow down due to the extra CPU, memory, and I/O requirements.
- You extend your system by adding new functionality, which makes user interactions require more system resources.
- You experience any combination of these factors.

Making the Server Stronger: Scaling Vertically

Once your application reaches the limits of your server (due to increase in traffic, amount of data processed, or concurrency levels), you must decide how to scale. There are two different types of scaling: vertical and horizontal. I will be covering both techniques in this book, but since vertical scalability is conceptually simpler and it is more common in this evolution stage, let's look at it first.

> *Vertical scalability* is accomplished by upgrading the hardware and/or network throughput. It is often the simplest solution for short-term scalability, as it does not require architectural changes to your application. If you are running your server with 8GB of memory, it is easy to upgrade to 32GB or even 128GB by just replacing the hardware. You do not have to modify the way your application works or add any abstraction layers to support this way of scaling. If you are hosting your application on virtual servers, scaling vertically may be as easy as a few clicks to order an upgrade of your virtual server instance to a more powerful one.

There are a number of ways to scale vertically:

- Adding more I/O capacity by adding more hard drives in Redundant Array of Independent Disks (RAID) arrays. I/O throughput and disk saturation are the main bottlenecks in database servers. Adding more drives and setting up a RAID array can help to distribute reads and writes across more devices. In recent years, RAID 10 has become especially popular, as it gives both redundancy and increased throughput. From an application perspective, a RAID array looks like a single volume, but underneath it is a collection of drives sharing the reads and writes.
- Improving I/O access times by switching to solid-state drives (SSDs). Solid-state drives are becoming more and more popular as the technology matures and prices continue to fall. Random reads and writes using SSDs are between 10 and 100 times faster, depending on benchmark methodology. By replacing disks you can decrease I/O wait times in your application. Unfortunately, sequential reads and writes are not much faster and you will not see such a massive performance increase in real-world applications. In fact, most open-source databases (like MySQL) optimize data structures and algorithms to allow more sequential disk operations rather than depending on random access I/O. Some data stores, such as Cassandra, go even further, using solely sequential I/O for all writes and most reads, making SSD even less attractive.

- Reducing I/O operations by increasing RAM. (Even 128GB RAM is affordable nowadays if you are hosting your application on your own dedicated hardware.) Adding more memory means more space for the file system cache and more working memory for the applications. Memory size is especially important for efficiency of database servers.
- Improving network throughput by upgrading network interfaces or installing additional ones. If your server is streaming a lot of video/media content, you may need to upgrade your network provider's connection or even upgrade your network adapters to allow greater throughput.
- Switching to servers with more processors or more virtual cores. Servers with 12 and even 24 threads (virtual cores) are affordable enough to be a reasonable scaling option. The more CPUs and virtual cores, the more processes that can be executing at the same time. Your system becomes faster, not only because processes do not have to share the CPU, but also because the operating system will have to perform fewer context switches to execute multiple processes on the same core.

Vertical scalability is a great option, especially for very small applications or if you can afford the hardware upgrades. The practical simplicity of vertical scaling is its main advantage, as you do not have to rearchitect anything. Unfortunately, vertical scaling comes with some serious limitations, the main one being cost. Vertical scalability becomes extremely expensive beyond a certain point.[43]

Figure 1-2 shows the approximate relationship of price per capacity unit and the total capacity needed. It shows that you can scale up relatively cheaply first, but beyond a certain point, adding more capacity becomes extremely expensive. For example, getting 128GB of RAM (as of this writing) could cost you $3,000, but doubling that to 256GB could cost you $18,000, which is much more than double the 128GB price.

The second biggest issue with vertical scalability is that it actually has hard limits. No matter how much money you may be willing to spend, it is not possible to continually add memory. Similar limits apply to CPU speed, number of cores per server, and hard drive speed. Simply put, at a certain point, no hardware is available that could support further growth.

Finally, operating system design or the application itself may prevent you from scaling vertically beyond a certain point. For example, you will not be able to keep adding CPUs to keep scaling MySQL infinitely, due to increasing lock contention (especially if you use an older MySQL storage engine called MyISAM).

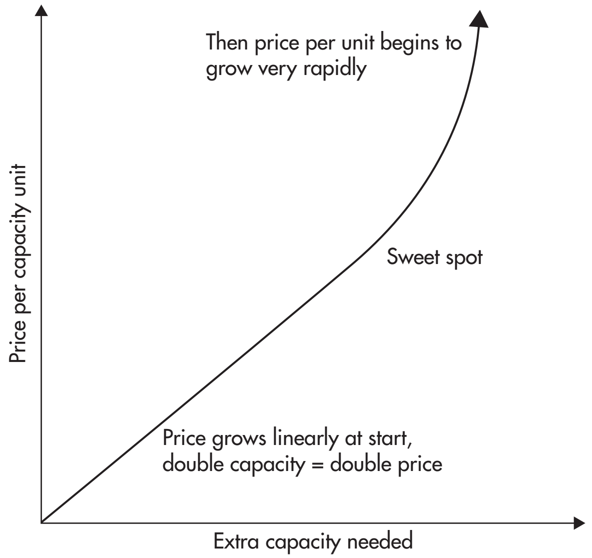


Figure 1-2 *Cost of scalability unit*

> *Locks* are used to synchronize access between execution threads to shared resources like memory or files. *Lock contention* is a performance bottleneck caused by inefficient lock management. Operations performed very often should have fine-grained locks; otherwise, your application may spend most of its time waiting for locks to be released. Once you hit a lock contention bottleneck, adding more CPU cores does not increase the overall throughput.

High-performance open-source and commercial applications should scale onto dozens of cores; however, it is worth checking the limitations of your application before purchasing the hardware. Homegrown applications are usually much more vulnerable to lock contention, as efficient lock management is a complex task requiring a lot of experience and fine-tuning. In extreme cases, adding more cores may yield no benefits at all if the application was not designed with high concurrency in mind.

As you can see in Figure 1-3, vertical scalability does not affect system architecture in any way. You can scale vertically each of our servers, network connections, or

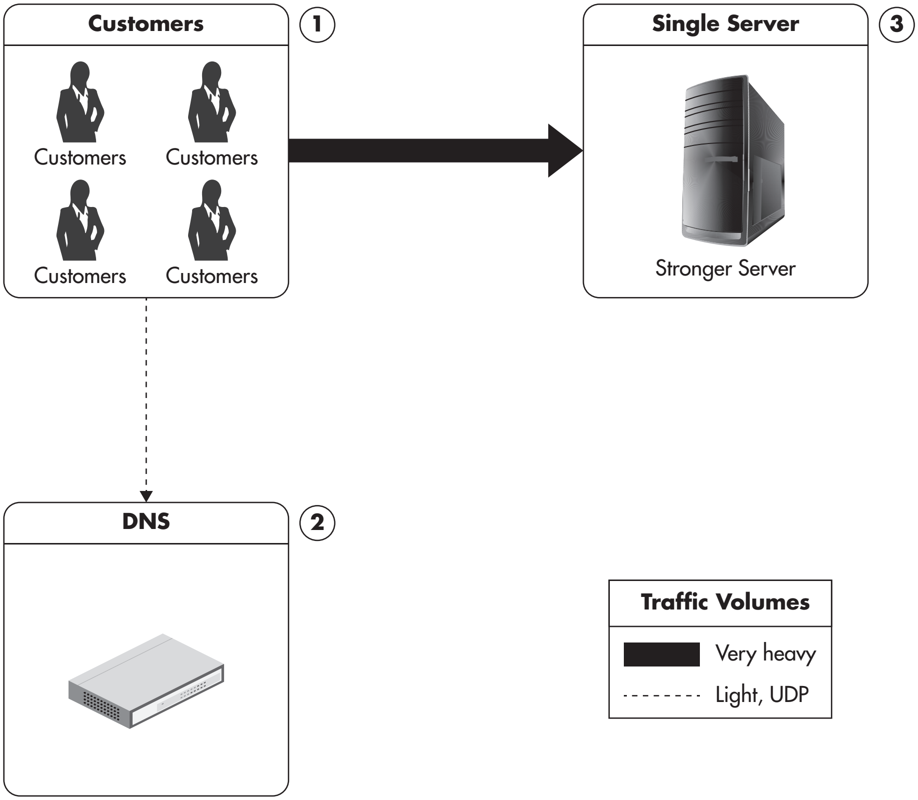


Figure 1-3 *Single server, but stronger*

routers without needing to modify your code or rearchitecting anything. All you need to do is replace a piece of hardware with a stronger or faster piece of hardware.

Isolation of Services

Vertical scalability is not the only option at this early stage of evolution. Another simple solution is moving different parts of the system to separate physical servers by installing each type of service on a separate physical machine. In this context, a service is an application like a web server (for example, Apache) or a database engine (for example, MySQL). This gives your web server and your database a separate, dedicated machine. In the same manner, you can deploy other services like File Transfer Protocol (FTP), DNS, cache, and others, each on a dedicated physical machine. Isolating services to separate servers is just a slight evolution from a single-server setup. It does not take you very far, however, as once you deploy each service type on a separate machine, you have no room to grow.

> *Cache* is a server/service focused on reducing the latency and resources needed to generate the result by serving previously generated content. Caching is a very important technique for scalability. I will discuss caching in detail in Chapter 6.

Figure 1-4 shows a high-level infrastructure view with each service deployed to a separate machine. This still looks similar to a single-server setup, but it slowly

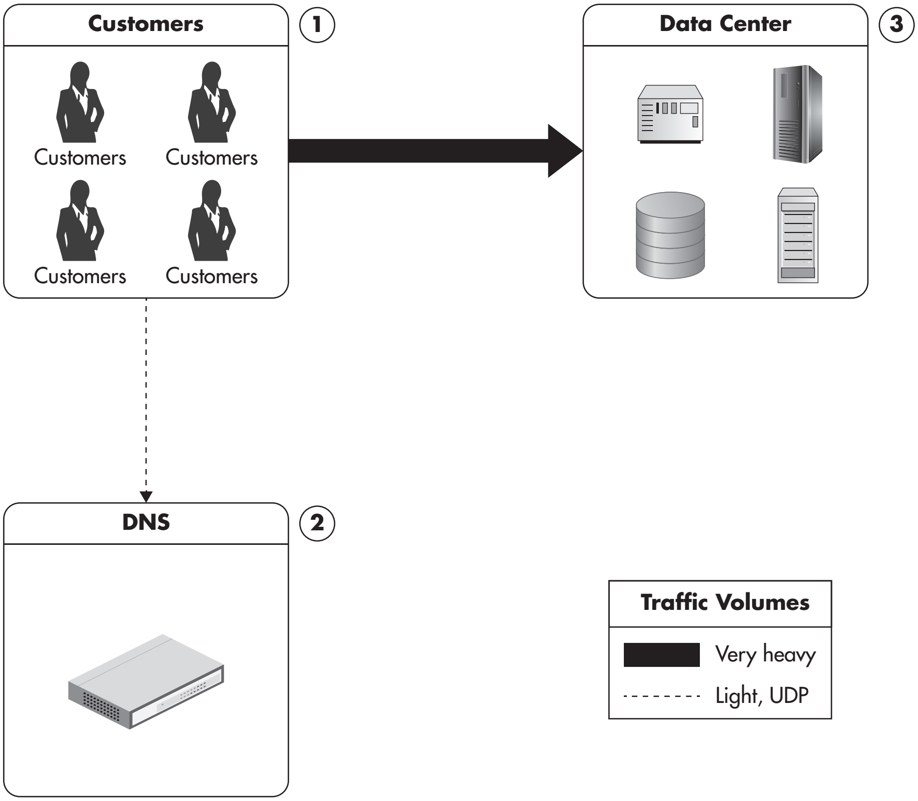


Figure 1-4 *Configuration with separate services residing on different servers*

increases the number of servers that can share the load. Servers are usually hosted in a third-party data center. They are often VPS, rented hardware, or collocated servers. I represent the data center here as a set of servers dedicated to different functions. Each server has a certain role, such as web server, database server, FTP, or cache. I will discuss the details of data center layout later in this chapter.

Isolation of services is a great next step for a single-server setup, as you can distribute the load among more machines than before and scale each of them vertically as needed. This is a common configuration among small websites and web development agencies. Agencies will often host many tiny websites for different clients on shared web servers. A bigger client with a more popular website would move to a separate web server and a separate database. This allows an agency to balance the load between applications of their clients and better utilize resources, keeping each of the web applications simple and fairly monolithic.

In a similar way to agencies hosting customers' websites on separate machines, you can divide your web application into smaller independent pieces and host them on separate machines. For example, if you had an administrative console where customers can manage their accounts, you could isolate it into a separate web application and then host it on a separate machine.

HINT

The core concept behind isolation of services is that you should try to split your monolithic web application into a set of distinct functional parts and host them independently. The process of dividing a system based on functionality to scale it independently is called functional partitioning.

Figure 1-5 shows a scenario in which a web application uses functional partitioning to distribute the load among even more servers. Each part of the application would typically use a different subdomain so that traffic would be directed to it based simply on the IP address of the web server. Note that different partitions may have different servers installed, and they may also have different vertical scalability needs. The more flexibility we have in scaling each part of the system, the better.

Content Delivery Network: Scalability for Static Content

As applications grow and get more customers, it becomes beneficial to offload some of the traffic to a third-party content delivery network (CDN) service.

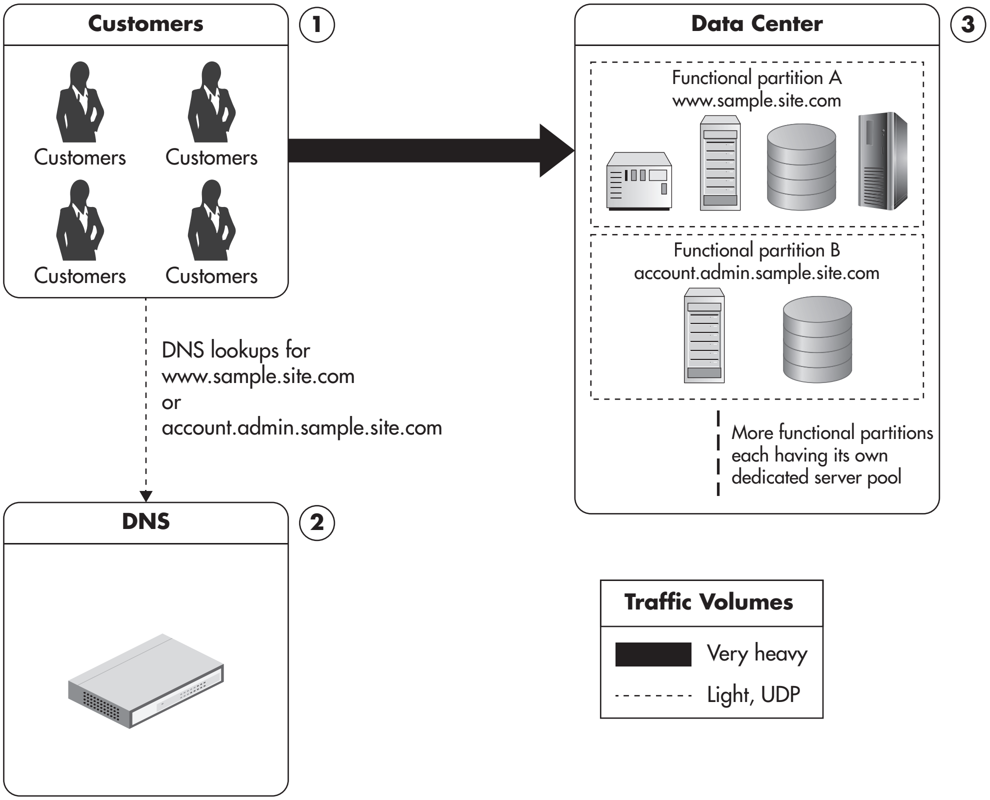

Customers
1
Customers
Customers
Customers
Customers
Data Center
3
Functional partition A
www.sample.site.com
Functional partition B
account.admin.sample.site.com
More functional partitions each having its own dedicated server pool
DNS lookups for www.sample.site.com or account.admin.sample.site.com
DNS
2
Traffic Volumes
Very heavy
Light, UDP

Figure 1-5 *Configuration showing functional partitioning of the application*

> *A content delivery network* is a hosted service that takes care of global distribution of static files like images, JavaScript, CSS, and videos. It works as an HTTP proxy. Clients that need to download images, JavaScript, CSS, or videos connect to one of the servers owned by the CDN provider instead of your servers. If the CDN server does not have the requested content yet, it asks your server for it and caches it from then on. Once the file is cached by the CDN, subsequent clients are served without contacting your servers at all.

By integrating your web application with a CDN provider, you can significantly reduce the amount of bandwidth your servers need. You will also need fewer web servers to serve your web application's static content. Finally, your clients may benefit from better resource locality, as CDN providers are usually global companies with data centers located all around the world. If your data center is located in North America, clients connecting from Europe would experience higher latencies. In such case, using CDN would also speed up page load times for these customers, as CDN would serve static content from the closest data center.

Figure 1-6 shows a web application integrated with a CDN provider. Clients first connect to the DNS server. Then, they request pages from your servers and load additional resources, such as images, CSS, and videos, from your CDN provider. As a result, your servers and networks have to deal with reduced traffic, and since CDNs solve a specific problem, they can optimize the way they serve the content cheaper than you could. I will explain CDN in more detail in Chapter 6.

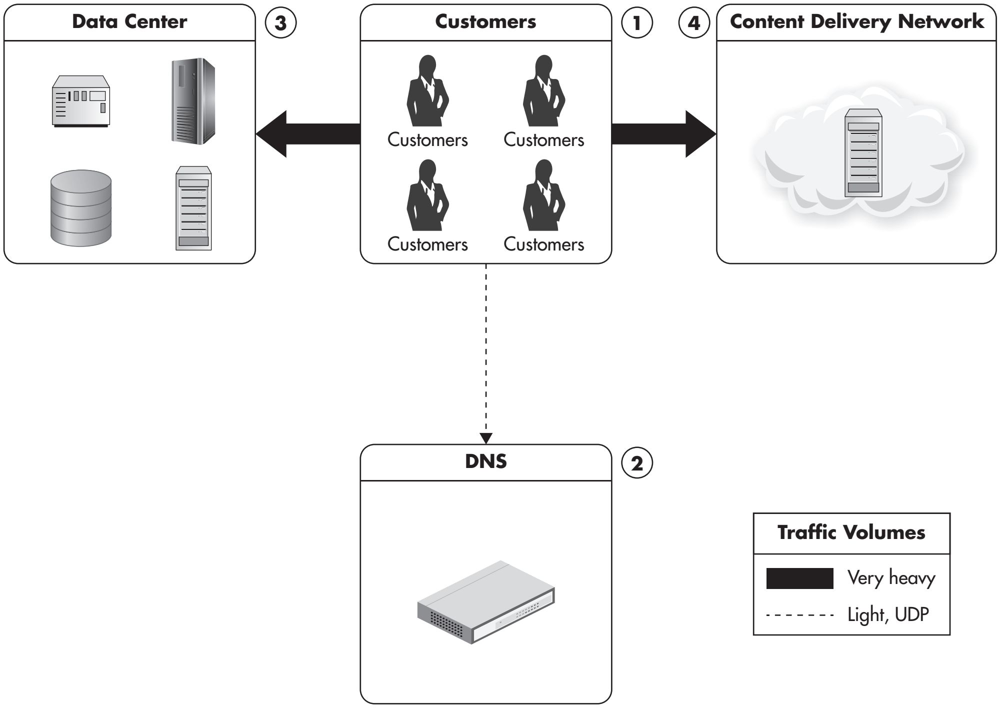


Figure 1-6 *Integration with a content delivery network provider*

The important thing to note here is that this is the first time I mentioned scaling using a third-party service. We did not have to add more servers or learn how to scale HTTP proxies. We simply used the third-party service and relied on its ability to scale. Even though it may seem like "cheating in the scalability game," it is a powerful strategy, especially for startups in their early stages of development, who cannot afford significant time or money investments.

Distributing the Traffic: Horizontal Scalability

All of the evolution stages discussed so far were rather simple modifications to the single-server configuration. Horizontal scalability, on the other hand, is much harder to achieve and in most cases it has to be considered before the application is built. In some rare cases, it can be "added" later on by modifying the architecture of the application, but it usually requires significant development effort. I will describe different horizontal scalability techniques throughout this book, but for now, let's think of it as running each component on multiple servers and being able to add more servers whenever necessary. Systems that are truly horizontally scalable do not need strong servers—quite the opposite; they usually run on lots and lots of cheap "commodity" servers rather than a few powerful machines.

> *Horizontal scalability* is accomplished by a number of methods to allow increased capacity by adding more servers. Horizontal scalability is considered the holy grail of scalability, as it overcomes the increasing cost of capacity unit associated with scaling by buying ever-stronger hardware. In addition, when scaling horizontally you can always add more servers—you never reach a hard limit, as is the case with vertical scalability.

Figure 1-7 shows a simplified comparison of costs related to horizontal and vertical scalability. The dashed line represents costs of vertical scalability, and the solid line represents horizontal scalability.

Horizontal scalability technologies often pay off at the later stage. Initially they tend to cost more because they are more complex and require more work. Sometimes they cost more because you need more servers for the most basic

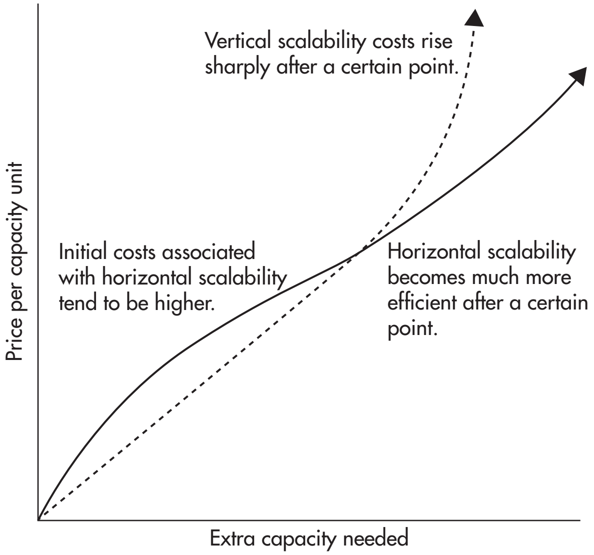


Figure 1-7 *Comparison of vertical and horizontal scaling costs*

setup, and other times it is because you need more experienced engineers to build and operate them. The important thing to note is that once you pass a certain point of necessary capacity, horizontal scalability becomes a better strategy. Using horizontal scalability, you avoid the high prices of top-tier hardware and you also avoid hitting the vertical scalability ceiling (where there is no more powerful hardware).

It is also worth noting that scaling horizontally using third-party services like CDN is not only cost effective, but often pretty much transparent. The more traffic you generate, the more you are charged by the provider, but the cost per capacity unit remains constant. That means that doubling your request rate will just cost you twice as much. It gets even better, as for some services, price per unit decreases as you scale up. For example, Amazon CloudFront charges $0.12 per GB for the first 10TB of transferred data, but then decreases the price to $0.08 per GB.

HINT

Cloud service providers are able to charge lower rates for higher-traffic clients because their overheads of maintenance, integration, and customer care are lower per capacity unit when dealing with high-traffic sites.

Let's quickly review the high-level infrastructure overview of the evolution so far. Once we start deploying different parts of the system onto different servers and adding some horizontal scalability, our high-level diagram may look something like Figure 1-8.

The thing that distinguishes horizontally scalable systems from the previous evolution stages is that each server role in our data center can be scaled by adding more servers. That can usually be implemented in stages of partially horizontal scalability, where some services scale horizontally and others do not. As I mentioned before, achieving true horizontal scalability is usually difficult and expensive. Therefore, systems should start by scaling horizontally in areas where it is the easiest to achieve, like web servers and caches, and then tackle the more difficult areas, like databases or other persistence stores.

At this stage of evolution, some applications would also use a round-robin DNS service to distribute traffic among web servers. Round-robin DNS is not the only way to distribute traffic among multiple web servers; we will consider different alternatives in detail in Chapter 3.

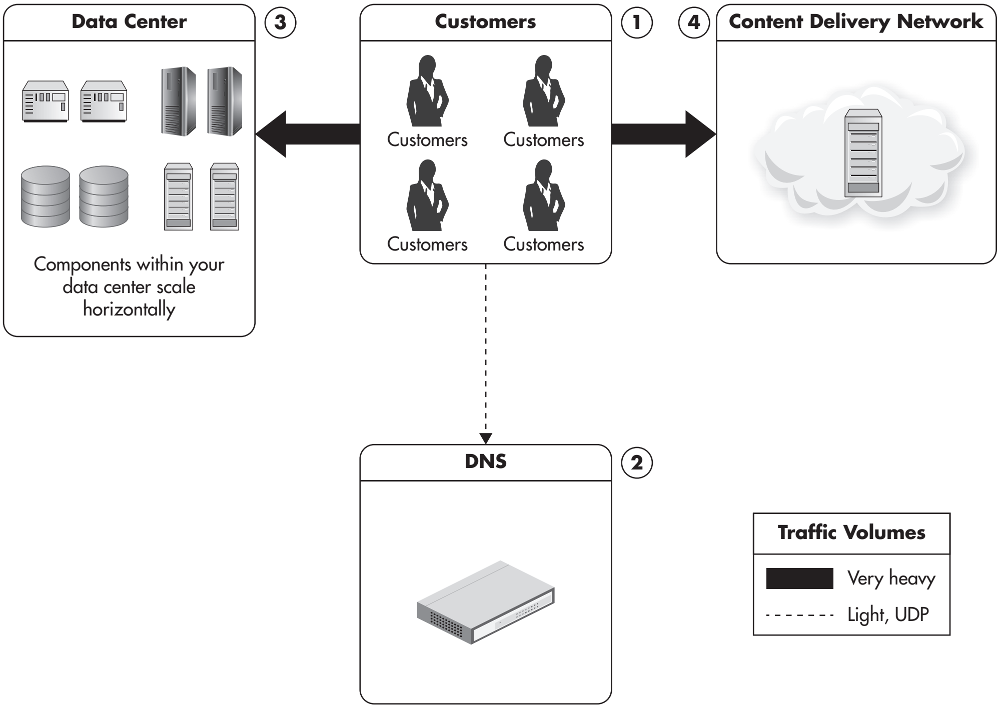


Figure 1-8 *Multiple servers dedicated to each role*

Round-robin DNS is a DNS server feature allowing you to resolve a single domain name to one of many IP addresses. The regular DNS server takes a domain name, like ejsmont.org, and resolves it to a single IP address, like 173.236.152.169. Thus, round-robin DNS allows you to map the domain name to multiple IP addresses, each IP pointing to a different machine. Then, each time a client asks for the name resolution, DNS responds with one of the IP addresses. The goal is to direct traffic from each client to one of the web servers—different clients may be connected to different servers without realizing it. Once a client receives an IP address, it will only communicate with the selected server.

Scalability for a Global Audience

The largest of websites reach the final evolution stage, which is scalability for a global audience. Once you serve millions of users spread across the globe, you will require more than a single data center. A single data center can host plenty of servers, but it causes clients located on other continents to receive a degraded user experience. Having more than one data center will also allow you to plan for rare outage events (for example, caused by a storm, flood, or fire).

Scaling for a global audience requires a few more tricks and poses a few more challenges. One of the additions to our configuration is the use of geoDNS service.

GeoDNS is a DNS service that allows domain names to be resolved to IP addresses based on the location of the customer. Regular DNS servers receive a domain name, like yahoo.com, and resolve it to an IP address, like 206.190.36.45. GeoDNS behaves the same way from the client's perspective. However, it may serve different IP addresses based on the location of the client. A client connecting from Europe may get a different IP address than the client connecting from Australia. As a result, clients from both Europe and Australia could connect to the web servers hosted closer to their location. In short, the goal is to direct the customer to the closest data center to minimize network latency.

Another extension of the infrastructure is to host multiple edge-cache servers located around the world to reduce the network latency even further. The use of edge-cache servers depends on the nature of your application. Edge-cache servers are most efficient when they act as simple reverse proxy servers caching entire pages, but they can be extended to provide other services as well.

> *Edge cache* is a HTTP cache server located near the customer, allowing the customer to partially cache the HTTP traffic. Requests from the customer's browser go to the edge-cache server. The server can then decide to serve the page from the cache, or it can decide to assemble the missing pieces of the page by sending background requests to your web servers. It can also decide that the page is uncacheable and delegate fully to your web servers. Edge-cache servers can serve entire pages or cache fragments of HTTP responses.

Figure 1-9 shows a high-level diagram with multiple data centers serving requests from clients located in different parts of the world. In this scenario, users located in Europe would resolve your domain name to an IP address of one of your European edge servers. They would then be served results from the cache or from one of your application servers. They would also load static files, such as CSS or JavaScript files, using your CDN provider, and since most CDN providers have data centers located in multiple countries, these files would be served from the closest data center as well. In a similar way, users from North America would be directed to American edge-cache servers and their static files would be served from the American CDN data center. As your application grows even further, you may want to divide your main data center into multiple data centers and host each of them closer to your audience. By having your data stores and all of your application components closer to your users, you save on latency and network costs.

Now that we have discussed the wider application ecosystem and the infrastructure at a very high level, let's look at how a single data center might support scalability.

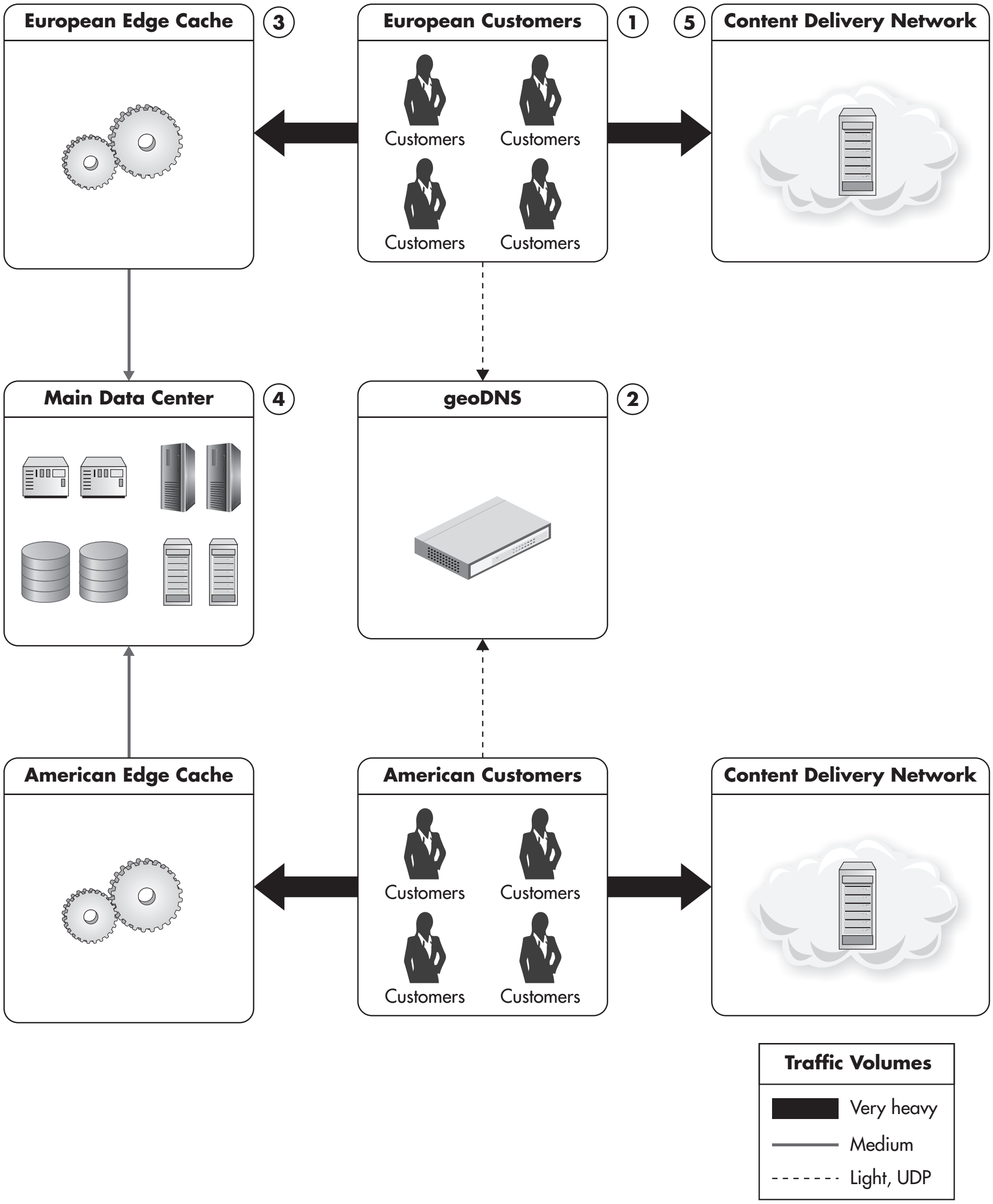


Figure 1-9 *Customers from different locations are served via local edge caches.*

Overview of a Data Center Infrastructure

Let's now turn to the different technologies used in modern web applications. As with the previous section, we'll take a deeper dive into these topics throughout the book, but I first want to lay out the overall communication flow and functions of each technology type.

Figure 1-10 shows a high-level overview of the communication flow starting from the user's machine and continuing all the way throughout different layers of the infrastructure. It is one of the most important diagrams in this book, as it shows you all of the key components that you need to be familiar with to design and implement scalable web applications. You can think of it as a reference diagram, as we will come back to different parts of it in different chapters. In fact, the structure of this book was designed to align closely to the structure of a data center, with each area of responsibility being covered by different chapters of the book.

Many of the components shown serve a specialized function and can be added or removed independently. However, it is common to see all of the components working together in large-scale applications. Let's take a closer look at each component.

The Front Line

The front line is the first part of our web stack. It is a set of components that users' devices interact with directly. Parts of the front line may reside inside of our data center or outside of it, depending on the details of the configuration and third-party services used. These components do not have any business logic, and their main purpose is to increase the capacity and allow scalability.

Going from the top, clients' requests go to the geoDNS server to resolve the domain names. DNS decides which data center is the closest to the client and responds with an IP address of a corresponding load balancer (2).

> *A load balancer* is a software or hardware component that distributes traffic coming to a single IP address over multiple servers, which are hidden behind the load balancer. Load balancers are used to share the load evenly among multiple servers and to allow dynamic addition and removal of machines. Since clients can only see the load balancer, web servers can be added at any time without service disruption.

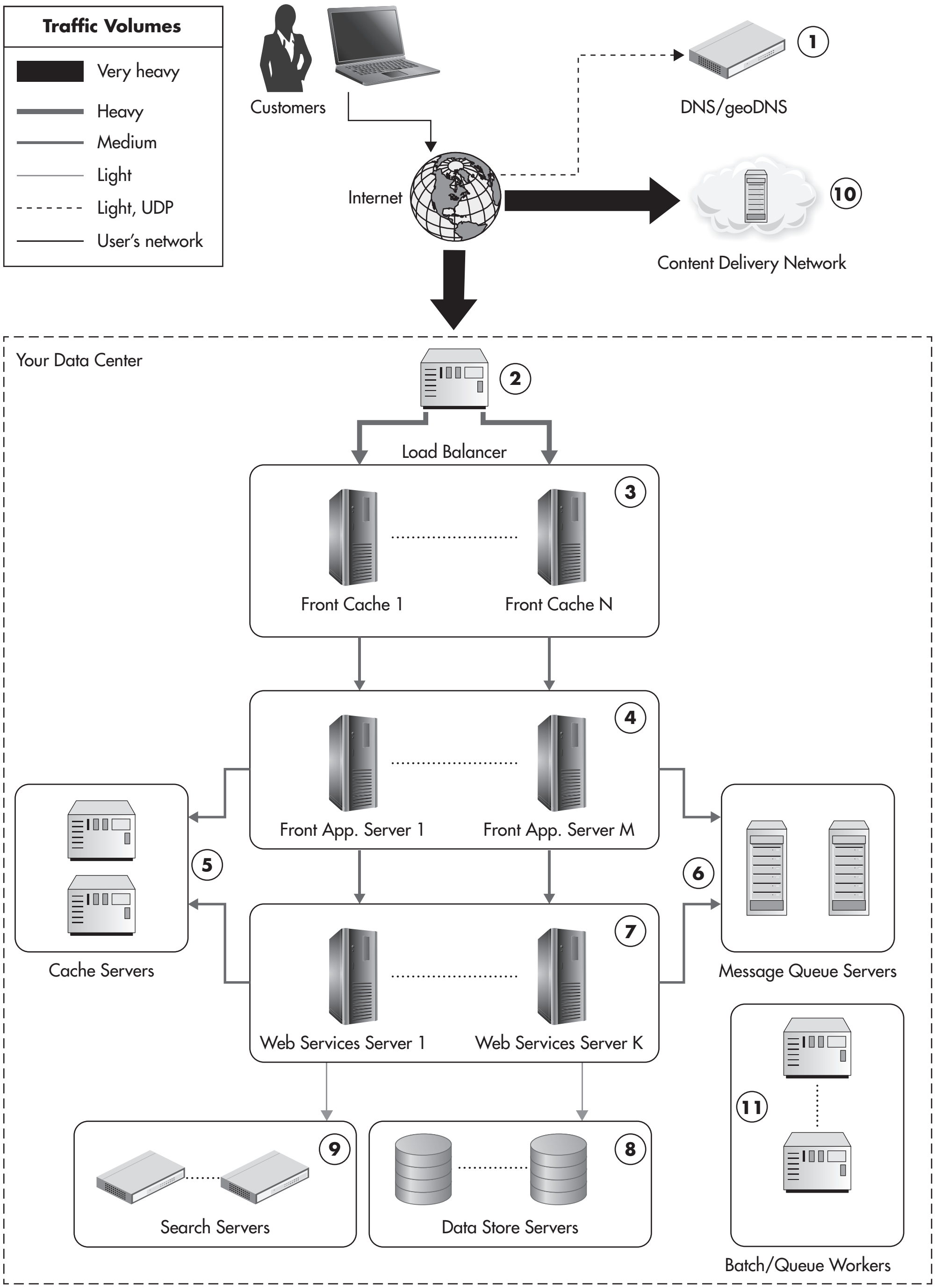


Figure 1-10 *High-level overview of the data center infrastructure*

Web traffic from the Internet is usually directed to a single IP address of a strong hardware load balancer. It then gets distributed evenly over to front cache servers (3) or directly over front-end web application servers (4). Front cache servers are optional; they can be deployed in remote locations outside of the data center or skipped altogether. In some cases it may be beneficial to have a layer of front-end cache servers to reduce the amount of load put on the rest of the infrastructure.

It is common to use third-party services as load balancers, CDN, and reverse proxy servers; in such cases this layer may be hosted entirely by third-party providers. We'll take a closer look at the benefits and drawbacks of scaling them using third parties in Chapter 3.

Web Application Layer

The second layer of our stack is the web application layer. It consists of web application servers (4) responsible for generating the actual HTML of our web application and handling clients' HTTP requests. These machines would often use a lightweight (PHP, Java, Ruby, Groovy, etc.) web framework with a minimal amount of business logic, since the main responsibility of these servers is to render the user interface. All the web application layer is supposed to do is handle the user interactions and translate them to internal web services calls. The simpler and "dumber" the web application layer, the better. By pushing most of your business logic to web services, you allow more reuse and reduce the number of changes needed, since the presentation layer is the one that changes most often.

Web application servers are usually easy to scale since they should be completely stateless. If developed in a stateless manner, adding more capacity is as simple as adding more servers to the load balancer pool. I will discuss the web application layer together with the frontline layer in Chapter 3.

Web Services Layer

The third layer of our stack consists of web services (7). It is a critical layer, as it contains most of our application logic. We keep front-end servers simple and free of business logic since we want to decouple the presentation layer from the business logic. By creating web services, we also make it easier to create functional partitions. We can create web services specializing in certain functionality and scale them independently. For example, in an e-commerce web application, you could have a product catalog service and a user profile service, each providing very different types of functionality and each having very different scalability needs.

The communication protocol used between front-end applications and web services is usually Representational State Transfer (REST) or Simple Object Access Protocol (SOAP) over HTTP. Depending on the implementation, web services should be relatively simple to scale. As long as we keep them stateless, scaling horizontally is as easy as adding more machines to the pool, as it is the deeper data layers that are more challenging to scale.

In recent years, integration between web applications has become much more popular, and it is a common practice to expose web services to third parties and directly to customers. That is why web services are often deployed in parallel to front-end application servers rather than hidden behind them, as shown in Figure 1-10.

I will discuss the web services layer in detail in Chapter 4. For now, let's think of web services as the core of our application and a way to isolate functionality into separate subsystems to allow independent development and scalability.

Additional Components

Since both front-end servers (4) and web services (7) should be stateless, web applications often deploy additional components, such as object caches (5) and message queues (6).

Object cache servers are used by both front-end application servers and web services to reduce the load put on the data stores and speed up responses by storing partially precomputed results. Cache servers will be covered in detail in Chapter 6.

Message queues are used to postpone some of the processing to a later stage and to delegate work to queue worker machines (11). Messages are often sent to message queues from both front-end applications and web service machines, and they are processed by dedicated queue worker machines. Sometimes web applications also have clusters of batch-processing servers or jobs running on schedule (controlled by cron). These machines (11) are not involved in generating responses to users' requests; they are offline job-processing servers providing features like asynchronous notifications, order fulfillment, and other high-latency functions. Message queues and queue workers are covered further in Chapter 7.

Data Persistence Layer

Finally, we come to the data persistence layer (8) and (9). This is usually the most difficult layer to scale horizontally, so we'll spend a lot of time discussing different scaling strategies and horizontal scalability options in that layer. This is also an area of rapid development of new technologies labeled as *big data* and *NoSQL*, as increasing amounts of data need to be stored and processed, regardless of their source and form.

The data layer has become increasingly more exciting in the past ten years, and the days of a single monolithic SQL database are gone. As Martin Fowler says, it is an era of polyglot persistence, where multiple data stores are used by the same company to leverage their unique benefits and to allow better scalability. We'll look further at these technologies in Chapters 5 and 8.

In the last five years, search engines became popular due to their rich feature set and existence of good open-source projects. I present them as a separate type of component, as they have different characteristics than the rest of the persistence stores, and I believe it is important to be familiar with them.

Data Center Infrastructure

By having so many different platforms in our infrastructure, we have increased the complexity multiple times since our single-server setup. What we have achieved is the ability to share the load among multiple servers. Each component in Figure 1-10 has a certain function and should help to scale your application for millions of users.

The layered structure of the components is deliberate and helps to reduce the load on the slower components. You can see that traffic coming to the load balancer is split equally over all front-end cache servers. Since some requests are "cache hits," traffic is reduced and only part of it reaches front-end servers (4). Here, application-level cache (5) and message queues (6) help reduce the traffic even further so that even fewer requests reach back-end web services (7). The web service can use message queues and cache servers as well. Finally, only if necessary, the web services layer contacts search engines and the main data store to read/write the necessary information. By adding easily scalable layers on top of the data layer, we can scale the overall system in a more cost-effective way.

It is very important to remember that it is not necessary to have all of these components present in order to be able to scale. Instead, use as few technologies as possible, because adding each new technology adds complexity and increases maintenance costs. Having more components may be more exciting, but it makes releases, maintenance, and recovery procedures much more difficult. If all your application needs is a simple search functionality page, maybe having front-end servers and a search engine cluster is all you need to scale. If you can scale each layer by adding more servers and you get all of the business features working, then why bother using all of the extra components? We'll continue to look back to Figure 1-10 as we cover the components in further detail.

Overview of the Application Architecture

So far, we've looked at the infrastructure and scalability evolution stages. Let's now take a high-level look at the application itself.

The application architecture should not revolve around a framework or any particular technology. Architecture is not about Java, PHP, PostgreSQL, or even database schema. Architecture should evolve around the business model. There are some great books written on domain-driven design and software architecture[1–3] that can help you get familiar with best practices of software design. To follow these best practices, we put business logic in the center of our architecture. It is the business requirements that drive every other decision. Without the right model and the right business logic, our databases, message queues, and web frameworks are useless.

Moreover, it is irrelevant if the application is a social networking website, a pharmaceutical service, or a gambling app—it will always have some business needs and a domain model. By putting that model in the center of our architecture, we make sure that other components surrounding it serve the business, not the other way around. By placing technology first, we may get a great Rails application, but it may not be a great pharmaceutical application.[t1]

> *A domain model* is created to represent the core functionality of the application in the words of business people, not technical people. The domain model explains key terms, actors, and operations, without caring about technical implementation. The domain model of an automated teller machine (ATM) would mention things like cash, account, debit, credit, authentication, security policies, etc. At the same time, the domain model would be oblivious to hardware and software implementation of the problem. The domain model is a tool to create our mental picture of the business problems that our application is supposed to solve.

Figure 1-11 shows a simplified representation of how application components can be laid out. This already assumes that users use our system as a single application, but internally, our application is broken down into multiple (highly autonomous) web services.

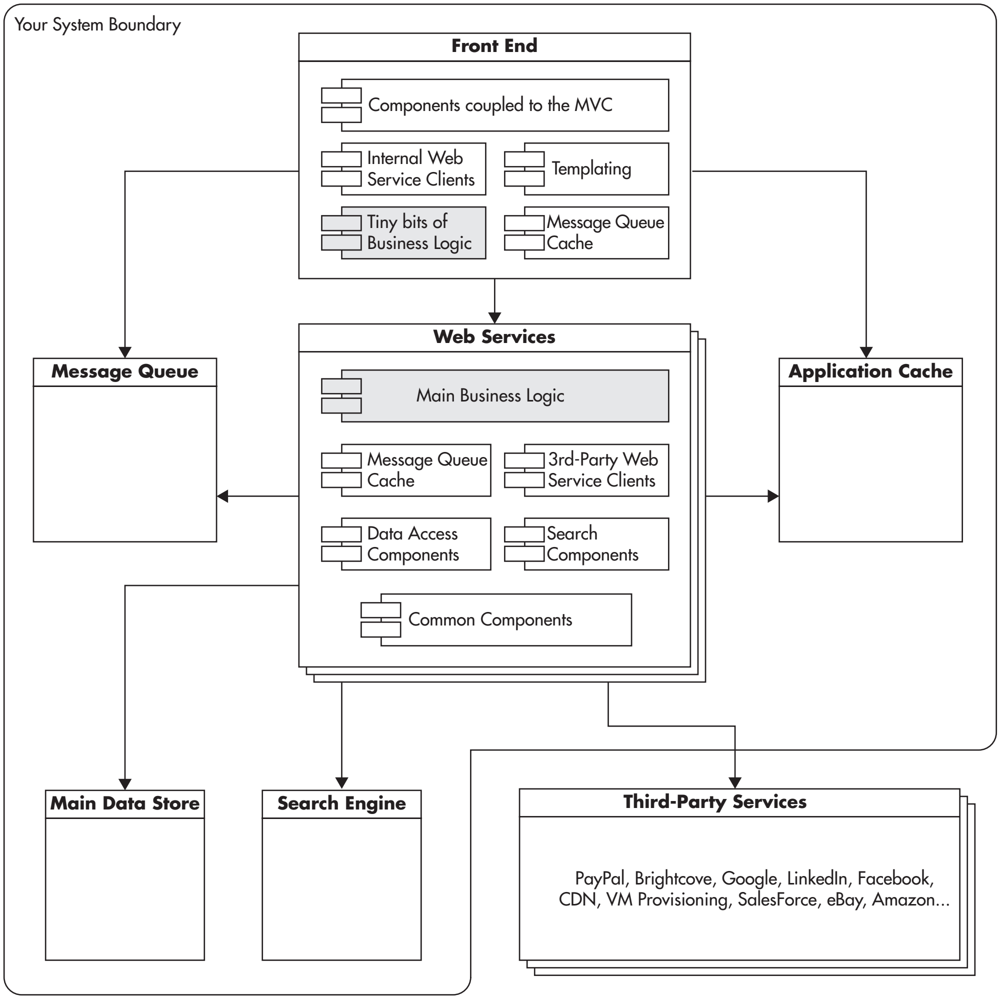


Figure 1-11 *High-level view of an application architecture*

Let's discuss each area of the diagram presented on Figure 1-11 in more detail in the following sections.

Front End

Similar to the way we discussed the infrastructure diagrams, let's take it from the top and look at Figure 1-11 from the point of the client's request. Keep in mind that the center of the architecture lives in the main business logic, but for the sake of simplicity, let's start with the front-end components.

The front end should have a single responsibility of becoming the user interface. The user can be interacting with the application via web pages, mobile applications, or web service calls. No matter what the actual delivery mechanism is, the front-end application should be the layer translating between the public interface and internal service calls. The front end should be considered as "skin," or a plugin of the application, and as something used to present the functionality of the system to customers. It should not be considered a heart or the center of the system. In general, the front end should stay as "dumb" as possible.

By keeping the front end "dumb," we will be able to reuse more of the business logic. Since the logic will live only in the web services layer, we avoid the risk of coupling it with our presentation logic. We will also be able to scale front-end servers independently, as they will not need to perform complex processing or share much state, but may be exposed to high concurrency challenges.

Front-end code will be closely coupled to templates and the web framework of our choice (for example, Spring, Rails, Symfony). It will be constrained by the user interface, user experience requirements, and the web technologies used. Front-end applications will have to be developed in a way that will allow communication over HTTP, including AJAX and web sessions. By hiding that within the front-end layer, we can keep our services layer simpler and focused solely on the business logic, not on the presentation and web-specific technologies.

Templating, web flows, and AJAX are all specific problems. Keeping them separated from your main business logic allows for fast and independent changes. Having the front end developed as a separate application within our system gives us another advantage: we can use a different technology stack to develop it. It is not unreasonable to use one technology to develop web services and a different one to develop the front-end application. As an example, you could develop the front end using Groovy, PHP, or Ruby, and web services could be developed in pure Java.

HINT

You can think of a front-end application as a plugin that can be removed, rewritten in a different programming language, and plugged back in. You should also be able to remove the "HTTP"-based front-end and plug in a "mobile application" front end or a "command line" front end. This attitude allows you to keep more options open and to make sure you decouple the front end from the core of the business logic.

The front end should not be aware of any databases or third-party services. Projects that allow business logic in the front-end code suffer from low code reuse and high complexity.

Finally, allow front-end components to send events to message queues and use cache back ends, as they are both important tools in increasing the speed and scaling out. Whenever we can cache an entire HTML page or an HTML fragment, we save much more processing time than caching just the database query that was used to render this HTML.

Web Services

"SOAs are like snowflakes—no two are alike." –David Linthicum

Web services are where most of the processing has to happen, and also the place where most of the business logic should live. Figure 1-11 shows a stack of web services in a central part of the application architecture. This approach is often called a service-oriented architecture (SOA). Unfortunately, SOA is a fairly overloaded term, so you may get a different definition, depending on who you speak with about it.

Service-oriented architecture (SOA) is architecture centered on loosely coupled and highly autonomous services focused on solving business needs. In SOA, it is preferred that all the services have clearly defined contracts and use the same communication protocols. I don't consider SOAP, REST, JSON, or XML in the definition of SOA, as they are implementation details. It does not matter what technology you use or what protocols are involved as long as your services are loosely coupled and specialized in solving a narrow set of business needs. I will explain coupling and best design principles in the next chapter.

HINT

Watch out for similar acronyms: SOA (service-oriented architecture) and SOAP (which originally was an acronym of Simple Object Access Protocol). Although these two can be seen together, SOA is an architecture style and SOAP is a set of technologies used to define, discover, and use web services. You can have SOA without SOAP, and you can also use SOAP in other architecture styles.

I encourage you to learn more about SOA by reading some of the recommended texts,[31,33,20] but remember that SOA is not an answer to all problems and other architecture styles exist, including layered architecture, hexagonal architecture, and event-driven architecture. You may see these applied in different systems.

A multilayer architecture is a way to divide functionality into a set of layers. Components in the lower layers expose an application programming interface (API) that can be consumed by clients residing in the layers above, but you can never allow lower layers to depend on the functionality provided by the upper layers. A good example of layered architecture is an operating system and its components, as shown in Figure 1-12. Here, you have hardware, device drivers, operating system kernel, operating system libraries, third-party libraries, and third-party applications. Each layer is consuming services provided by the layers below, but never vice versa. Another good example is the TCP/IP programming stack, where each layer adds functionality and depends on the contract provided by the layer below.

Layers enforce structure and reduce coupling as components in the lower layers become simpler and less coupled with the rest of the system. It also allows us to replace components in lower layers as long as they fulfill the same API. An important side effect of layered architecture is increased stability as you go deeper

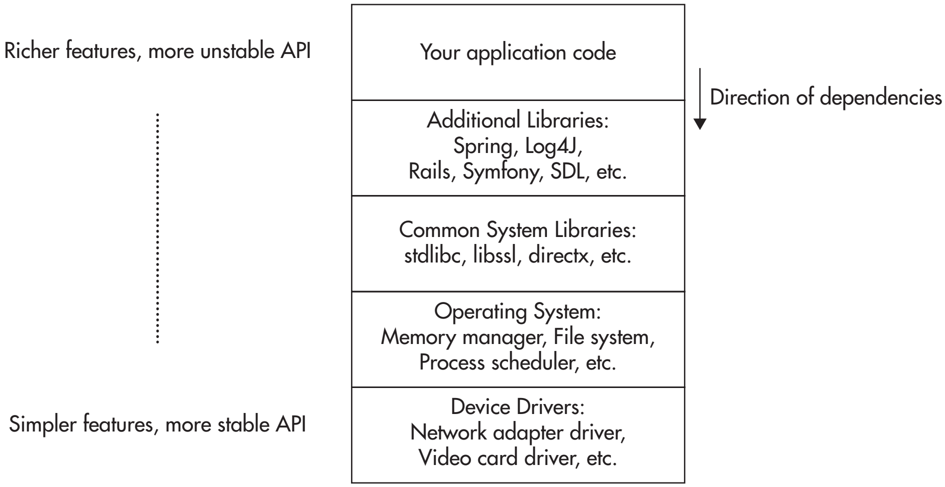

Richer features, more unstable API
Your application code
Direction of dependencies
Additional Libraries: Spring, Log4J, Rails, Symfony, SDL, etc.
Common System Libraries: stdlibc, libssl, directx, etc.
Operating System: Memory manager, File system, Process scheduler, etc.
Simpler features, more stable API
Device Drivers: Network adapter driver, Video card driver, etc.

Figure 1-12 *Example of a multilayered architecture*

into the layers. You can change the API of upper layers freely since few things depend on them. On the other hand, changing the API of lower layers may be expensive because there may be a lot of code that depends on the existing API.

Hexagonal architecture assumes that the business logic is in the center of the architecture and all the interactions with the data stores, clients, and other systems are equal. There is a contract between the business logic and every nonbusiness logic component, but there is no distinction between the layers above and below.

In hexagonal architecture, users interacting with the application are no different from the database system that the application interacts with. They both reside outside of the application business logic and both deserve a strict contract. By defining these boundaries, you can then replace the person with an automated test driver or replace the database with a different storage engine without affecting the core of the system.

Event-driven architecture (EDA) is, simply put, a different way of thinking about actions. Event-driven architecture, as the name implies, is about reacting to events that have already happened. Traditional architecture is about responding to requests and requesting work to be done. In a traditional programming model we think of ourselves as a person requesting something to be done, for example, createUserAccount(). We typically expect this operation to be performed while we are waiting for a result, and once we get the result, we continue our processing. In the event-driven model, we don't wait for things to be done. Whenever we have to interact with other components, we announce things that have already happened and proceed with our own processing. Analogous to the previous example, we could announce an event UserAccountFormSubmitted. This mental shift leads to many interesting implications. Figure 1-13 shows the difference in interaction models. We'll look more closely at EDA in more detail in Chapter 7.

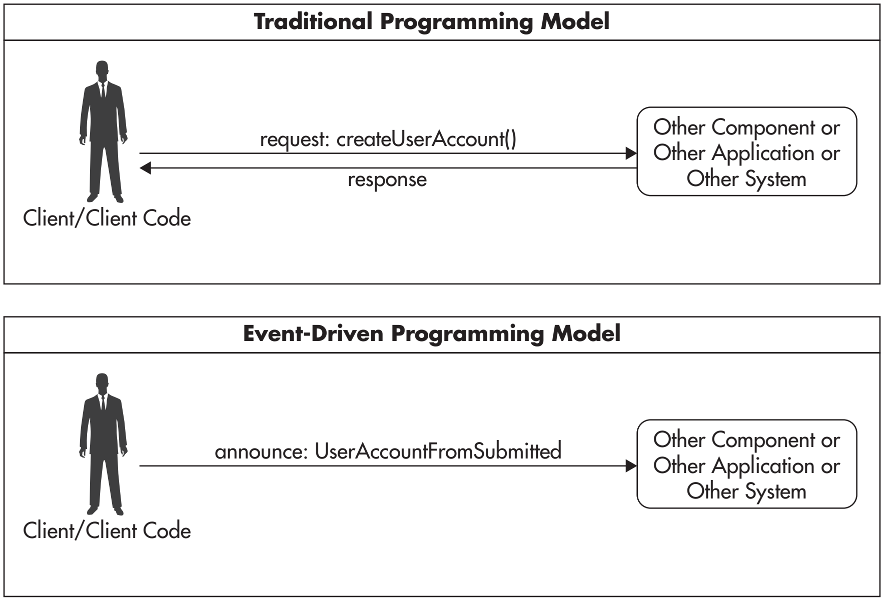

Traditional Programming Model
request: createUserAccount()
response
Other Component or Other Application or Other System
Client/Client Code
Event-Driven Programming Model
announce: UserAccountFromSubmitted
Other Component or Other Application or Other System
Client/Client Code

Figure 1-13 *Comparison of traditional and event-driven interactions*

No matter the actual architecture style of the system, all architectures will provide a benefit from being divided into smaller independent functional units. The purpose is to build higher abstractions that hide complexity, limit dependencies, allow you to scale each part independently, and make parallel development of each part practical.

HINT

Think of the web services layer as a set of highly autonomous applications, where each web service becomes an application itself. Web services may depend on each other, but the less they depend on each other, the better. A higher level of abstraction provided by services allows you to see the entire system and still understand it. Each service hides the details of its implementation and presents a simplified, high-level API.

Ideally, each web service would be fully independent. Figure 1-14 shows a hypothetical portfolio of web services belonging to an e-commerce platform. In this example, the text analysis service could be an independent service able to detect the meaning of articles based solely on their content. Such a service would not require user data or assistance from any other services; it would be fully independent.

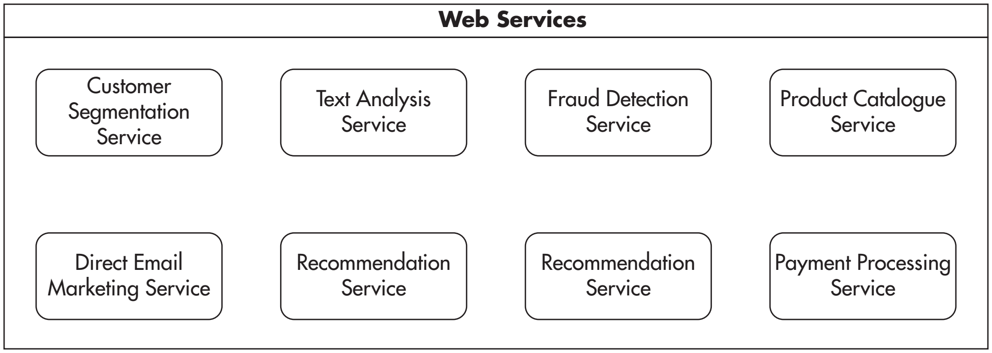

Web Services
Customer Segmentation Service
Text Analysis Service
Fraud Detection Service
Product Catalogue Service
Direct Email Marketing Service
Recommendation Service
Recommendation Service
Payment Processing Service

Figure 1-14 *Conceptual view of services in the web services layer*

Unfortunately, it is usually impossible to isolate all services like this. Most times, there will be some dependencies between different services. For example, a customer segmentation service could be a service based on user activity, and social network data produces a customer profile. To assign users to different customer segments, we may need to integrate this service with main user data, activity history, and third-party services. The customer segmentation service would most likely be coupled to services other than the text analysis service.

No matter what the implementation of your web services, don't forget their main purpose: to solve business needs.

Supporting Technologies

Figure 1-11 shows web services surrounded by a few smaller boxes labeled message queue, application cache, main data store, and search engine. These are isolated since they are usually implemented in different technologies, and most often they are third-party software products configured to work with our system. Because they are third-party technologies, they can be treated as black boxes in the context of architecture.

Notice that the database (main data store) is simply a little box in the corner of the diagram. This is because the data store is just a piece of technology; it is an implementation detail. From the application architecture point of view, the data store is something that lets us write and read data. We do not care how many servers it needs; how it deals with scalability, replication, or fault tolerance; or even how it persists data.

HINT

Think of the data store as you think of caches, search engines, and message queues—as plug-and-play extensions. If you decide to switch to a different persistence store or to exchange your caching back ends, you should be able to do it by replacing the connectivity components, leaving the overall architecture intact.

By abstracting the data store, you also free your mind from using MySQL or another database engine. If the application logic has different requirements, consider a NoSQL data store or an in-memory solution. Remember, the data store is not the central piece of the architecture, and it should not dictate the way your system evolves.

Finally, I decided to include third-party services in the architecture diagram to highlight their importance. Nowadays computer systems do not operate in a vacuum; large systems often have integrations with literally dozens of external systems and often critically depend on their functionality. Third-party services are outside of our control, so they are put outside of our system boundary. Since we do not have control over them, we cannot expect them to function well, not have bugs, or scale as fast as we would wish. Isolating third-party services by providing a layer of indirection is a good way to minimize the risk and our dependency on their availability.

Summary

Architecture is the perspective of the software designer; infrastructure is the perspective of the system engineer. Each perspective shows a different view of the same problem—building scalable software. After reading this chapter, you should be able to draw a high-level picture of how the architecture and the infrastructure come together to support the scalability of a web application. This high-level view will be important as we begin to drill into the details of each component.

As you can see, scalability is not an easy topic. It touches on many aspects of software design and architecture, and it requires broad knowledge of many different technologies. Scalability can only be tamed once you understand how all the pieces come together, what their roles are, and what their strong points and weak points are. To design scalable web applications is to understand the impact of the architecture, infrastructure, technologies, algorithms, and true business needs. Let's now move forward to principles of good software design, as this is a prerequisite to building scalable web applications.