Designing Systems

It's rare for a web application nowadays to be built in isolation. As a full stack developer, you're likely to have to work on entire systems, not just a single component within a system, and doing this effectively means being able to think at a different level of abstraction when problem solving.

In a modern organization, the system will be constantly growing and changing, although individual components may remain stable. Your job is to design those components in a way that maximizes agility—that is, when something changes in the future, it shouldn't be painful to transition. These components can exist at any level of abstraction, from very high levels of an organization's architecture to individual classes and modules in a codebase, and these principles can be applied at any level as well.

The type of systems a full stack developer will come across in their career will range from the simple (such as a brochureware site) to the massively complex (e.g., an account management site for a utility, which has many functions and has to integrate with the central billing and CRM components of a huge enterprise architecture), and it's important to know how to work within all these types of environments. Even what might at first seem like a simple, standalone component can quickly become entangled in other systems without careful thought, or can become a silo that could duplicate functionality or isolate data that already exists inside the organization.

A well-designed system can quickly become greater than the sum of its parts through the network effect. Some organizations have an "architect" role (or several) that is solely responsible for designing these systems and the interactions between them, usually at an application level. Even when there is a dedicated architect, it is important to ensure that you're thinking about the system as a whole too, and not just the single components within it.

System Architectures

Two common types of system architectures you may come across are "monolithic" and "microservices," but in reality, most systems will lie somewhere on the spectrum between those two extremes. These generally refer to organizing individual applications within the system, and how they interact, rather than at a smaller level. Both styles have their pros and cons: monoliths can be easier to build initially, but have the downside of becoming large and difficult to change if multiple teams work on the same codebase. Microservices require a strong platform to grow from, but can be easier to adapt and change in the future.

WHAT IS A MICROSERVICE?

A microservice is a small service that takes responsibility for one part of a system. It can be deployed and scaled independently of any other part of the system, and communicates through well-defined APIs. Microservices can be thought of as applications of the single responsibility principle from SOLID (discussed in more detail later in this chapter) at a system rather than class level. A microservice architecture is therefore composed of a number of these microservices that are deployed and orchestrated as a larger system of individual components.

By contrast, a monolith is a single codebase with a single application that performs all the functions of that system. It may be scalable, but the scale is achieved by duplicating the whole system on other machines, rather than individual components of it. Although it may expose an API for external systems to integrate with it, internal components of the system communicate via method or function calls.

Figure 4-1 shows the contrast between these two architectures. Although the microservices system seems more complex, this is only because it forces you to make interactions between services explicit, rather than the potentially complex hidden interaction model within a single app.

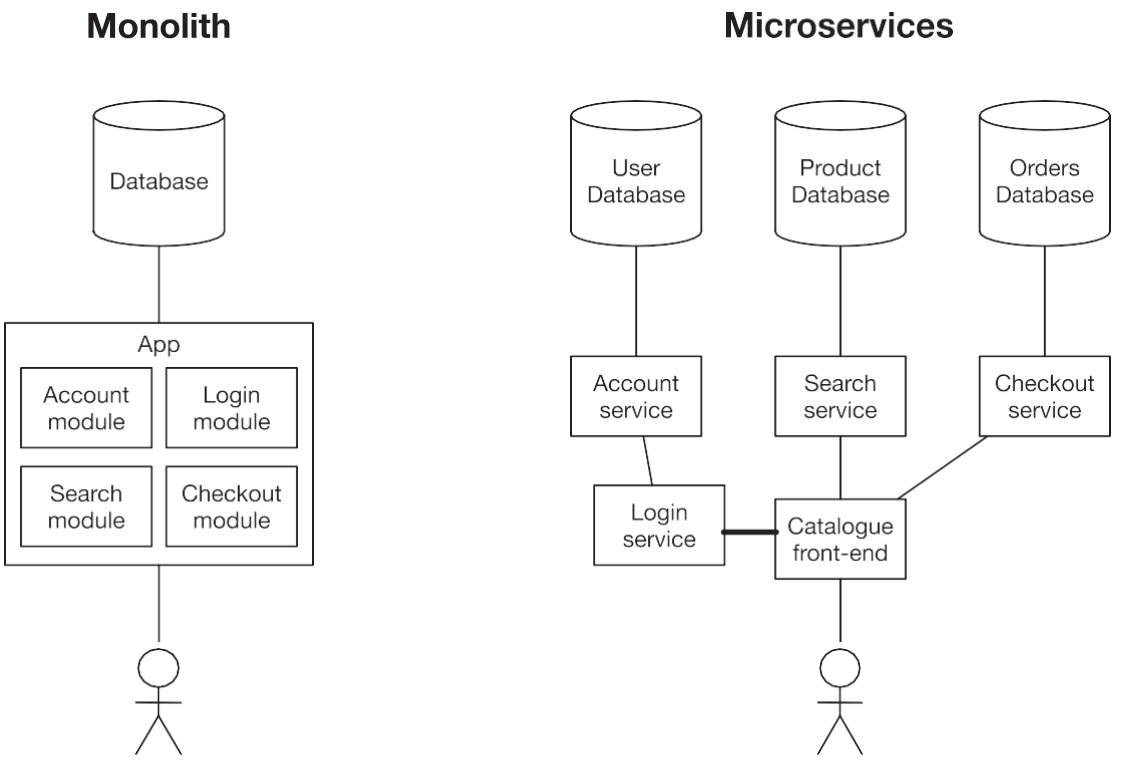

Figure 4-1. *A monolith systems diagram versus a microservice equivalent*

When working with large systems, a well-designed microservice architecture will allow you to limit those changes and growths to individual parts of the system, meaning it can be quicker to move since there are fewer moving parts involved. But this can be hard to design. For small systems, the overhead of managing microservices can be too big to be worth it, and can slow down initial development. Microservices are slow to get started, but often allow you to maintain a good pace of development that can be tricky with a monolithic design.

Even when designing systems with growth in mind, it can be especially helpful to start with a single application and codebase, and then break that out into microservices when that single application starts to become too big.

The trick to designing a system at the application level is to make each component as small as it needs to be, but no smaller. This is easier said than done, and there is no hard-and-fast science or rules to apply. System design can be more of an art, and very context

dependent, so it's useful for a team to design collaboratively. Following patterns for software architectures that might already exist inside your organization is a good start, as it can make integrating against dependencies easier.

What a software architecture is *not* is a list of a technologies that are used to build your system. At the point the architecture is designed, you should only care about capturing the concepts and their interactions, and then determine where they sit within the system. Once you have identified these, you can use your non-functional requirements to determine the properties each component in your system must have, and then find the best technology fit for each one of those. Trying to force technology choices into an architecture before the concepts, interactions, and systems are identified can lead to decisions that compromise the goals of making a good software architecture.

Identifying Concepts

The tiered application hierarchy is a common approach to designing systems, whether it's the ubiquitous Model-View-Controller architecture (shown in Figure 4-2) of most web applications (or a variation on this, such as Model-View-View Model) or the three-tier model of an enterprise architecture. In order to fit your system into these models, you first need to identify two things: the domain concepts that you care about (for example, this could be customers, stock inventory, or news articles), and the ways the users interact with them.

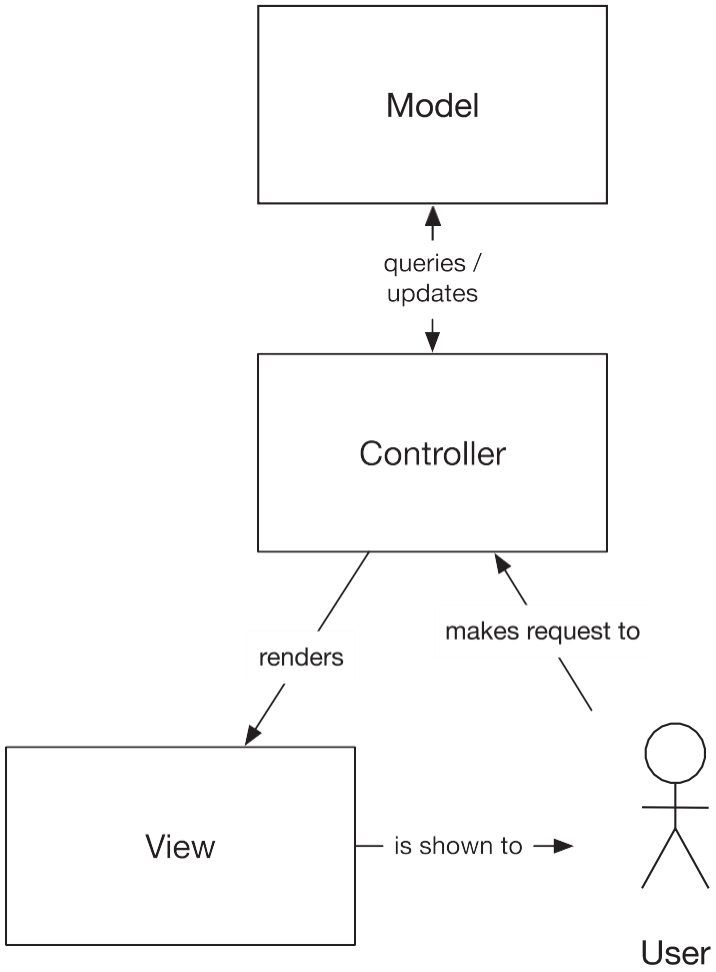

Figure 4-2. *Model-View-Controller architecture*

The domain concepts will form your model layer in a Model-View-Controller (MVC) app, or your persistence/business logic components in a three-tier architecture. The interaction methods will form the view and controller (or view model) of an MVC app, or the presentation layer in a three-tier architecture.

In MVC, the view deals with actually presenting the detail to a user (in a web framework, this is often HTML templates), whereas the controller deals with getting all the data and values that are needed to create a template, and performing any actions that a request may have asked for.

It can be tempting to add logic relating to models into a controller in an MVC app (for example, building a model up), but a controller is too heavily coupled to a particular view, and most logic actually belongs to a concept itself, making the model a more appropriate place for it. The “fat model, thin controller” approach can make refactoring an architecture much easier at a later date, as it can allow you to extract concepts into their own APIs, which are shared across multiple components.

In an object-oriented world, the downside of the fat model approach is that it can bloat your classes significantly. Rather than adding lots of methods to a class, you can instead implement other useful classes that help you manipulate models. If you’re using a language that mixes different paradigms, such as JavaScript or Python, then implementing these as functions can also be helpful, leaving the only state in the model. There are many different types of helpers you can separate from a model. For example, builders are useful to help build a model (out of a form, for instance), and validation logic could also be extracted when it makes sense to do so.

The Model-View-View Model (MVVM) pattern differs from MVC in that the controller disappears and a View Model is bound specifically to a view. In a typical MVC controller, you may have a controller with a method for dealing with a GET and one for dealing with a POST. For the GET case, the controller would fetch an appropriate model and render the view by passing through values. In the POST case, the controller would read the request, perform any appropriate validation, and then apply those changes to a model. In MVVM, a view model is instead "bound" to a view, so that when the view is rendered, getters on the view model make the values available in the template appropriately. To deal with the POST case, setters are used to make the changes back.

Beyond variations on MVC like MVVM, MVC has evolved beyond its initial function. Although many server-side web frameworks, such as PHP's Laravel, Ruby's Rails or Python's Django use MVC (although Django confusingly uses the terms Model, View, and Template for the Model, Controller, and View, respectively), it is not uncommon to see slight adaptations to the MVC architecture, especially when it comes to things like kicking off long-running actions.

The lines of MVC tend to become blurred in client-side code. As JavaScript has become more common on the server too, it's not uncommon to find NodeJS applications that do not strictly subscribe to MVC patterns. This is often because of the tendency of JS applications to be made by composing libraries together, rather than choosing frameworks. However, many apps (sometimes by design, other times through a natural tendency, as it is an effective way of organizing concerns) using NodeJS will have an MVC-like structure, even if not explicitly.

In the popular React+Redux combination, it is common to structure your app such that the Redux store is your model, and then have the React components as the view. The Redux *connect()* function then maps the state in the store and dispatchable actions to properties, and this mapping provides the role of the controller. However, views can also nest other components, which themselves might be wrapped in a *connect()*, so a view can include another controller. This hierarchical method might look like MVC close up, but from further away it becomes clear it is not, though the separation of concerns remains useful.

MVC often falls down when there is rich UI interaction. For example, when manipulating the UI, not every action necessarily needs to persist in the model, but there is some temporary state that is stored somewhere. If you wanted to implement a drag-and-drop interface, then in pure MVC, you would have to persist every movement to the store going through a controller. Instead, it is common for the "View" to instead store its own state, only going through the controller when the item is dropped, so that the model is correctly updated. MVC is a good starting point, but you should not artificially constrain yourself strictly to the pattern-it has limitations when it's okay to break the pattern too.

Similar to the way a React+Redux app is composed of many layers of things that are loosely analogous to layers of views and controllers, you will often find yourself doing the same in your application too. Your server-side code will often be structured in MVC, along with any rich UI code, although the two may not necessarily be linked. Although some models may be shared, it's likely your server-side and client-side code will differ somewhat, and the concerns of the controllers will be different. The server-side code is often more concerned with validation and security, while the client-side code may be less concerned with these factors.

When identifying the concepts that make up your models, it's also important to understand the context in which you're making those identifications. Domain-driven design (DDD) is a methodology in software engineering that helps manage these problems. At the core of DDD is the concept of the domain: the things that your organization knows and does that are relevant to a project. The domain is based in the reality of your organization, and it is through reality that we can identify the models. By starting at the level of what people in your organization currently do and how they refer to things, and using this as a basis for building your models, it becomes easier to build software that matches the actions people actually want to do.

Sometimes, this collection of models is known as an ontology—that is, the structure that defines your domain. It is usually helpful to use consistent terminology throughout a system. For example, an e-commerce store might refer to "items of stock" in the logistics area, but "products" in the front end, but these may mean the same thing, so using a single term throughout your system can simplify things.

Although it can be very tempting to rely on a single model definition that can be used throughout a system, it is also important to recognize that those models are also used in different contexts. As I touched on before, the context of an item in client-side code may be different to that of the data store, so using an appropriate variation of the model in that context makes sense. These models may be different implementations of the same concepts, but they need to translate to the same concept in another context. For example, moving from a client-side to server-side context can involve POSTing a form to a server, and it can be tempting to just re-use the POST data directly. However, what you should instead do is translate between the two contexts, using a method that accepts the form data of that model, and returns a new object that makes sense in the context of the server-side code. This translation can often be fairly simple, but is also a good place to do things like authorization checks or validation. Performing this translation as things move between your contexts can reduce bugs, as each context comes with its own set of assumptions, and moving data between contexts without translating it appropriately can mean different assumptions on that data that are no longer true.

A common pitfall here is to conflate similar concepts in your domain, or work at a level of abstraction that's too high. One example may be in a large enterprise system, which has to deal with the concepts of customers and employees. It can be tempting to try to unify these into some sort of "people" concept, but in most systems, the concepts of employees and customers are different enough to benefit from remaining separate, and any concerns about "duplicating" data (for example, if an employee also appears in the customer database) are never realized.

Many of the identified core concepts will have state that needs to be managed, which means that there needs to be a data store and the business logic for querying and manipulating the store. The exception here is where components and concepts for user interaction are used, which either have no persistence, or only persistence relating to temporary or session data, rather than core concepts. When the architecture is realized, this persistence and business logic can be split into separate components—one for your database, and then an API that interacts with that database and implements your business logic. When modelling your system architecture, it's useful to think of them as one concept that's bound together.

When it comes to realizing the architecture, it's important to choose the right data store for these concepts to enable persistence. NoSQL datastores are popular, but there's still a place for relational data, especially when integrity is very important, or the model is very well-defined and stable. This is discussed further in the Storing Data chapter.

Dealing with the business logic of the core domain concept with the datastore as a single high-level component provides a good level of abstraction for the other components to deal with those concepts. All manipulation of these concepts in your persistence layer then becomes an implementation detail of an API, rather than the persistence layer becoming an integration point itself. Multiple applications accessing the same database directly can cause problems for data integrity and deployments, especially if any schema changes are involved, so the API that implements the business logic wraps the database and becomes the single point of interaction for other components. Remember that the goal here is for each component to be able to grow and move independently of each other, and this encapsulation enables that.

Identifying User Interactions

User journeys and stories (or epics) can help identify user interactions. Once you identify which journeys a user wants to take, they can come together into a single front-end component. It's very tempting to group your user interactions by concept, but this will often result in a lot of unrelated code in the same component, and ideally you want each set of related things to move at its own speed. Separating by user can often go far. For example, on an e-catalog site, staff may be responsible for updating and maintaining the catalog that users then browse, but the user journeys for browsing the catalog are very distinct from maintaining it, so these should be different components.

For those who have worked with a framework that's good at making CRUD (Create-Read-Update-Delete) apps, this may at first seem counterintuitive—it's typical in these frameworks to designate one controller per concept, with actions for things that occur on those controllers. Often, the code you need to execute this is auto-generated for you! However, one of the key goals we want to achieve is the ability to grow each part of the system independently, without introducing risks to other parts of the system, so we can maintain our pace of change. The user stories and interactions in a management system will often grow in a very different direction to one relating to the browsing component, and by linking the two together, you may find yourself being dragged down by having to change more than if you'd kept them separate.

Handling Commonalities

Once you have identified the individual components in your system, you are then likely to have a good understanding of the boundaries of those components, but it's also likely you have identified some common dependencies in your system (these are likely to be core elements of your organization, such as content, customers, or inventory). If these common dependencies are domain concepts, they can be abstracted together behind an API, and if they're interactions, a shared library or component between your front-end applications is the best place for it.

Working with Legacy and External Dependencies

In many organizations, there are likely to be systems outside the scope of what you're responsible for. This could be an off-the-shelf or SaaS system bought by your organization or a project managed by another team. You need to capture these dependencies so you know how your system interacts with them. It's important to remember that although you're likely to have less influence over these external dependencies, they are not set in stone, and you can request that the people that own these make changes. If you're having to ask for lots of changes, perhaps this component should actually belong to your team, in order to reduce the risk of introducing blockers. If these systems are legacy, or are a dependency for a number of components within your system, it can be helpful to insert a facade or an abstraction layer between it and you. Especially if a system is legacy, introducing a new interface can assist in decommissioning it, as this interface can be changed to abstract away a replacement service without having to update all the clients that depend on the legacy component. This is known as the strangler pattern.

When designing your system, it's important to understand any non-digital components that might exist in it, and how any interactions they have—for example, printing out dispatch labels, or shipping a physical product. You may not be responsible for building these, but it's important to ensure that the design of any physical processes fits in with your architecture in order to avoid painful integrations later. Ultimately, you should be able to trace every interaction with a user, data, and response to an action that is generated through the entire system, including its real-world components. Treating these non-digital dependencies like external dependencies will often help.

Component Interactions

It's important to understand the relationships between these components, and the way they need to communicate. The way to do this is to reflect on the different scenarios through which a user may come to use the system, and the paths and workflows a user takes to accomplish their goals. These are called user journeys, and it's useful to identify which front-end component each journey belongs to, consider which concepts it needs to interact with, and determine the nature of that interaction.

In a typical web architecture, there are three ways for components to interact: asking for data; taking an action synchronously, such as when the result of the action, or knowing when it was completed, matters; or taking an action asynchronously, when either the result doesn't matter, or the time taken to perform this action is so long that it makes no sense to wait for it to finish before showing the user. The latter case includes actions that are completely digital and take a long time, like video transcoding, or involve activity in the real world, such as sending out a membership card or shipping a product. For these types of asynchronous actions, all the user needs to know is whether or not the request to take the action has succeeded, so the asynchronicity can be hidden behind a synchronous service.

There are two common communication types between components in a system: requests using HTTP and message queues. For requesting data or making synchronous changes, HTTP is a natural fit, and a particular type of HTTP interaction known as REST is a popular choice—I will cover this later. HTTP GET can be used for requesting data, and HTTP POST or PUT should be used for making changes. For asynchronous actions, where it doesn't matter to the user whether it succeeds or not (this could be something like recording analytics), message queues are a good fit. However, if the code is front-end code, it is often easy to make a fire-and-forget XHR call (XMLHttpRequest, a browser API for making HTTP requests, and an important part of the asynchronous JavaScript and XML-AJAX technique). With this kind of fire-and-forget call, the result of processing may not need to be surfaced to the user— they simply need to know that the request has been made—although for others, tracking progress may be needed, where RESTful queries can be used to interact with jobs on a message queue.

With all of these things identified, it's then important to reassess the boundaries of these components, and also at what level these components live. If there are two components that seem to communicate with each other a lot, this could indicate that they actually belong in the same component. On the other hand, if there's a central component that everything seems to talk to, this may suggest that this component does too much, and there could be smaller elements that should be extracted, or that perhaps it's capturing a concept at too high a level of abstraction.

When defining the interactions above, we also determine the actions that a concept needs to be able to perform. Some of these actions, especially if they're large or asynchronous, may belong in their own components. With this in mind, it's important to identify these new components and their communication style. When multiple events are triggered by a single action, or when an event happens in response to multiple actions, it can be helpful to introduce topic-based message queues. In this approach, the thing that causes an event does not necessarily need to know all the actions that event will cause. It simply sends a message to a topic queue, and everything that listens to that queue will perform appropriately.

An enterprise service bus (ESB) is a popular tool that, on the surface, appears similar to a message queue, but offers a much deeper set of features. In an ESB, you place a message onto the bus, and then the ESB distributes that message to all the components that need it. The ESB becomes a single point of integration for all components, as shown in Figure 4-3, which is both its major selling point and its biggest weakness. It is often the ESB that controls how messages are routed (rather than an application determining which topics to subscribe to), and ESBs can perform additional actions on messages in flight, such as changing the transport mechanism or transforming the message itself, as well as auditing and security. They essentially become a single point of failure and a single integration point at which to scale. The most common use case for ESBs is integrating many distinct components together, as the ESB can hide the complexity of integration, but you should instead consider having well-defined APIs on these services, as this can allow for more independent scaling of systems.

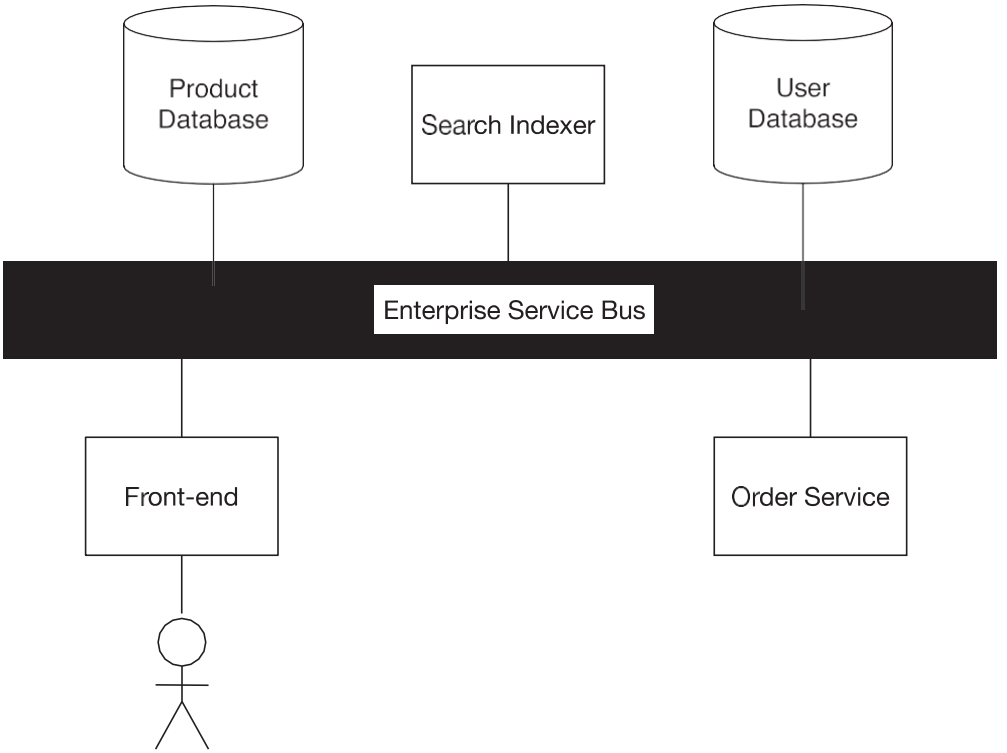

Figure 4-3. *Modules connected to an enterprise service bus*

It can be very tempting to introduce an ESB to handle the interactions between backend components, especially if you're moving from a legacy system that uses an ESB already. But you should be aware that many ESBs are designed to be plugged into existing components (often sold by the same vendor as the ESB) and serve as a place to configure those components. This is where the need for many of the features of an ESB come from. By making use of these features, you run the risk of subtle logic ending up in your ESB, so changes will be spread across multiple applications, and managing changes to this single shared component become tightly controlled to avoid risk. A microservice approach where each components maintains a higher degree of independence aims to avoid this.

Using standard approaches for communications (like message queues and REST) will generally give you more flexibility than an ESB can, and usually at a much lower cost. Avoiding this kind of vendor lock-in can give your organization more agility too; there's no need to get locked in to expensive contracts.

One of the primary reasons to use an ESB is that it offers abstraction towards many different legacy components. To get around this, a common pattern known as the facade pattern can be used. Where a component has an odd API, or an action is actually spread across many different backend services, a facade can be implemented. An application that uses the facade pattern doesn't actually do much by itself; it simply exists to present a common API on top of another service and translate between that new API and the original one.

Though there are desirable features to an ESB, such as logging, these benefits can be implemented in alternate ways. For example, a common HTTP library could be used across your organization with defaults for logging, or tools such as API gateways can provide these (but beware, as many API gateways provide the same kind of functionality as ESBs, and should be avoided). Otherwise, providing easy-to-use libraries or other functionalities to teams means they can be integrated in a way that makes sense for that component.

A warning for using message queues: make sure you understand the characteristics of the message queue you use. For example, guarantees about delivery vary between implementations. You should also keep the actual communication logic of the message queue at the edge of the application, in case you need to move to a different message queue implementation (for example, if you move from AWS cloud to another one). This mitigates the risk of vendor lock-in, where some technologies may no longer make business sense for an organization to use, but technology coupling prevents the organization from making that choice.

Applications vs. Modules

Some of these components could just be modules inside a larger application, or they could be applications by themselves. A good rule of thumb to follow is the “single responsibility principle,” in which each application only does one thing, but does it well. This is sometimes known as “the UNIX way,” after a design principle used to build the original UNIX command line tools.

However, the real world involves many more trade-offs, and for whichever platform you’re using to deploy, there will always be an overhead for each application, and it’s important to balance the overhead for each application with the benefits of keeping each component small. An ideal platform would make that overhead as small as possible: being self-service, with high levels of automation, while providing common

functions. This is discussed further in the Deployment chapter. There is also often a cost implication of having multiple applications. It's very common in the cloud to deploy one application per (virtual) machine, so more applications require more machines, although a technique known as containerization can address this at the risk of introducing additional complexity and a lower level of isolation to your platform.

Cross-Functional Requirements

The cross-functional requirements of your system will also have an effect on the architecture, and how you decide whether or not different components belong as distinct applications or just as modules inside a larger service. Cross-functional requirements (often called non-functional requirements) are those that cross every feature of your system, rather than a feature built once in isolation. The most important cross-functional requirements are those that specify the desired performance and scale of a system, as well as attributes such as security.

Satisfying performance requirements can be hard in a distributed system. Generally, the more connections exist between your components in order to complete a single journey, the slower they will run unless you optimize (caching is one common way to do so).

Making a RESTful call across an HTTP interface will always be much slower than calling a function in another module in the same application. Instead of introducing caching, you may be tempted to make a shared library that is distributed with each application in your system, but this can be an anti-pattern. If your business logic changes, you now need to update that library in every component that uses it (which means you have to keep track of everywhere it's used) and redeploy that component. If it's a big change, different parts of your system might have different versions of the business logic as all these deployments happen. It's much better to have one place where your business logic is specified, that can be managed without having an impact on a large number of systems when it changes. Be wary of even small shared libraries, if having different versions of that library in production will cause problems.

One example of a successful case of using a shared library was an HTTP library. This library wrapped an HTTP request library, but added additional logging to help diagnose issues and transparently handled our HTTP authorization scheme. Having multiple copies in production caused no issues—for example, fixing a bug didn't call for a new version of that library to be immediately rolled out, as the bug only affected one service that used a particular feature. The shared library was decoupled from the logic of the backends it communicated with to enable this.

In terms of security concerns, you can identify which actions in your system are "trusted" and which are not to ensure that the endpoints are appropriately protected. It can be very effective to separate applications along security lines. For example, a component that can register a user (an action anyone can perform) may want to live in a separate application from the one that can search the user database (an action that may be limited to call center staff). This allows "security-in-depth," assuming your platform and API calls have a good security system, allowing you to put the searching application behind platform-level security rather than relying on application-level security. Platform-level security is usually more heavily audited and reviewed, and this reduces the risk of an application-level bug.

Regardless of the architecture you decide on, it's often very effective to build each application to be stateless; for example, not storing sessions server-side, or in a database that lives outside your application. This allows you to make deployments by replacing an application at the level of the machine and scale horizontally with multiple boxes behind a load balancer, which can go a long way to addressing scale. More detailed techniques for achieving this are discussed later on in the book.

Caching

When considering performance or load requirements, one powerful aspect of HTTP is its native support for caching. It can be tempting to put caches everywhere, especially around your HTTP GET calls, but this has an effect on the user experience of your application, as the information the user is seeing may be out of date. This is especially problematic when the user has updated something themselves (such as saving a new address and then still seeing the old one). Caches are also cumulative, so if there are multiple layers of caching between the user, it can take an unintentionally long time for the caches to expire.

Caches are also effectively an additional data-store, as they keep multiple copies of the data, and this can become out of date with the truth. In some contexts, this does not present a problem; for example, a new product not appearing in search results for five minutes may not be problematic, but a news site publishing breaking news that has to wait 30 minutes for a front-page index to update is. It's important to understand the effects of these delays on your user experience. When caches are used, it can be useful to combine them with event-based communications so that caches are invalidated in response to certain events. It's also possible to go one step beyond this and use events alone to build up a view of the data that only a specific application relies on, rather than using a core API for a single point of truth. This is an advanced technique!

Designing for Failure

The final thing to consider are the failure modes of this system, such as what happens when a single component fails or otherwise becomes unavailable. For message-queue-based approaches (whether as an asynchronous action, or in an event-based system), it's important to understand what happens if a message is received more than once, in the wrong order, or not at all, and to choose the underlying platform based on these requirements. For example, when adding a new component to a system that responds to events, consider whether it needs to have received all previous events that have occurred in the system to be up-to-date, or whether it will respond appropriately only to new ones.

Most responses to these failure modes can be handled in the individual applications themselves, and there are good techniques for doing this. For RESTful communications, the circuit breaker pattern is useful for protecting against failure of the underlying services, alongside caching layers (especially making use of functionality like stale-on-error). If possible, making synchronous communications asynchronous is also effective, as the messages remain on the queue until processed.

THE CIRCUIT BREAKER PATTERN

In real life, a circuit breaker is something that trips when it detects failure. The software circuit breaker is fairly similar. When the circuit breaker is "closed," everything operates normally, but when failure is detected, the circuit breaker "opens," stopping any requests to that backend service. After a period of time, the circuit breaker lets through a single request to see if the back end has recovered. If that request is successful, then the circuit breaker closes and things return to normal—otherwise, it waits again. The waiting is often some sort of exponential backoff with a random delay, to avoid accidentally flooding the back end the moment it recovers.

The method of determining failure can vary. A common method is incrementing a counter every time there is an error, and resetting it when there is success, with failure being detected when the counter passes a certain number. This has the downside of letting a partially broken service go through unprotected, so an alternative is to instead measure the percentage of failed to successful requests over a period of time, and determining failure based on a percentage of failed requests over a certain period of time. Remember to differentiate between expected and unexpected failures though! A 404 could be a valid response from a back-end service if you're requesting a resource you're not sure exists, and shouldn't count as a failure.

This is especially useful if the failure of the back-end service is related to load, and it needs some time to recover. Also, if the back-end service is failing by timing out, then not bothering to do the request can speed up the render of the page, especially if it's a non-critical part of the application that's failed.

Finally, it's also important to understand the effect of these failures on your user stories, and when designing the user experience. For example, if your site is an e-commerce site, but the component that determines whether an item is in stock or not is down, what should be presented to the user? There are a number of options, such as allowing orders and then refunding if the item turns out not to be in stock, or simply refusing to accept orders. The correct answer depends on the context of your organization.

It might seem that the need to handle many of these requirements would disappear if a monolith was built, rather than microservices, but in reality that system would be more brittle, as a failure could propagate to unrelated systems in the same monolith, so failure handling could not be avoided, although the cases to handle are different. The benefits of having a flexible system architecture, consisting of small, single-responsibility modules with well-defined interfaces, vastly outweighs the downside of having to manage failure modes between those modules, and you will end up with a more robust system as a result.

Designing Modules

In addition to designing the architecture of your entire system, you should also consider the structure of the individual components and modules within that system. The main difference between the design of an individual component and that of a system stems from different overheads relating to each of their constituent pieces. In a system, making a component too small introduces overhead for coordinating that component, but making it too big increases flexibility for scaling or building resilience into your system. Inside a single component, however, breaking it down into many smaller pieces introduces very little in the way of overhead, giving you the flexibility and simplicity of having many parts that do one thing well, without the coordination downside.

Another aspect to consider is that, inside a component, it becomes trivial to refactor across module boundaries (as long as they do not stray into other parts of the system), which means that you can minimize up-front architectural planning inside an individual component, as the cost of correcting mistakes is smaller. The cost is not zero though, so putting some thought into application structure is beneficial.

One major upfront decision about an individual component is which technologies to use: programming languages, frameworks, and libraries. In a perfect world, choosing the best technology and language to implement the requirements of a particular component would be the obvious answer, but a component always exists as part of a larger system. Using languages and frameworks that are already in use in other components can maximize the efficiency of a team because they do not have to context switch. In a greenfield build, this can be a risky choice. In web development, the two dominant approaches are to use a framework such as Angular, Rails, or Django, or to instead build something by bolting libraries together (perhaps with a relatively small framework facilitating this). Frameworks can be extremely powerful, but inflexible. For example, Rails lets you create monolithic CRUD applications very easily, but only if you work in the way that it expects you to work. If you don't, you may encounter resistance, and it may actually be slower than not using the framework at all. On the other hand, using libraries can be slower to initiate, and involve writing more code to glue the libraries together, but if you need to build a very domain-specific type of application, this gives you the flexibility to configure your application as you wish. Frameworks are also harder to move away from, as almost all code must be built in the style of that framework, whereas libraries are easier to replace in an application because only the parts that interact with that library are dependent on it.

When choosing a framework or deciding which libraries to use, make sure that you fully understand the capability and use case of the framework or libraries you pick, and that they actually meet your needs.

Designing individual components is much better understood than designing the kind of complex, distributed systems that many organizations now use. The concept of microservices didn't come around until 2011, whereas design patterns for applications were first discussed in the 1980s, and the famous *Design Patterns* book published in 1994. Of course, since then, the field of developing applications has continued to evolve and be further refined. A design pattern is a recommended (and tried-and-tested) approach for solving a particular type of problem in application design. When using frameworks, you are often forced to use particular patterns, but when using libraries, you will often need to implement the patterns yourself.

There are many design patterns, but there are subtleties to the context in which you should apply them. Many of the design patterns in the eponymous book were developed in the context of the C++ language, and others in early versions of Java. In other languages, features that do not exist in C++ or older versions of Java can make a design pattern superfluous.

Design Patterns is also mostly concerned with patterns for object-oriented programming, which has been the dominant programming paradigm in recent decades—however, it is not the only one. Languages like JavaScript and Python allow you to mix procedural and functional styles alongside OOP, but understanding object-oriented principles of design allows you to apply the same concepts to these paradigms too.

In addition to design patterns, many principles have evolved that help drive good design and engineering. Many of these have adopted catchy acronyms like KISS, DRY, and SOLID. "Keep it simple stupid" (KISS) encourages developers to write code that is approachable for an average developer to help with future maintainability, rather than adding in additional layers of abstraction that may be elegant but are unnecessary to solve the problem at hand. "Don't repeat yourself" (DRY) encourages you to only implement needed functionality or concepts once, and then use them across multiple modules where appropriate. Correct application of DRY is discussed later in this chapter. SOLID refers to five different concepts:

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

The SOLID principles scale up or down depending on the scale you're working at. The larger components of a well-architected system could be said to be SOLID, as well as the individual classes or functions within each component.

The *single responsibility principle*, for example, is one of the driving factors behind microservices. The principle states that each individual class in your system should only take responsibility for a single thing. A corollary to this is that for each responsibility your system takes on, there should be a single part of your system responsible for handling it. For example, if a part of your system calculates VAT costs, then if the VAT rate changes, then you should only need to update a single part of your system, and that same class should also not be concerned with other tasks, such as calculating delivery costs.

According to the *open/closed principle*, a particular bit of code should be open for extension but closed for modification. This is mostly applied to reusable bits of functionality. It dictates that each other class that uses your class should be able to

extend its functionality beyond what is provided to satisfy its use case, without having to modify the module it is inheriting from. In object-oriented programming, this is often taken to mean inheritance, but it can also mean that any dependencies specified are done so using interfaces, so alternative implementations can be used.

Barbara Liskov originally described the *Liskov substitution principle* in her 1994 paper as "Let φ(□) be a property provable about objects □ of type T. Then φ(□) should be true for objects □ of type S where S is a subtype of T." (Barbara Liskov and Jeannette Wing, "A behavioral notion of subtyping," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 16, #6, 1994). Although this mathematical description can seem intimidating, it simply means that if you extend a module, or provide modules with the same interface and intend for them to be used interchangeably, then the assumptions that the user of that class has about how to interact and use it must hold between everything that implements or extends that interface. For example, if you have different modules for calculating delivery costs that implement the same interface, but one of those requires you to enter the total weight of the shipment, and the others do not, this can be said to violate the principle. All modules must also require the total weight to be known, even if nothing is done with it, in order to keep the interface and the assumptions the same. If you do not use this, then any time these classes are used in your codebase, you will have to be aware of an edge case, rather than it being a truly reusable component.

Some programming languages allow you to formally specify these cases in the form of preconditions or postconditions, and to test them. A precondition is an assertion that must be true for you to be able to correctly use an interface, and a postcondition is something you can assert about the output of a function or method, if the preconditions were satisfied. For example, in our VAT calculator case, we could express a precondition that the input must be a list of objects that consist of a price, which is a non-negative, and whether or not it's VAT exempt, which is a boolean. The postcondition is that it will give you a non-negative integer. Some computer science theory goes further than this to fully specify pre- and postconditions in mathematical form in order to make systems that are mathematically provable as correct, but this is something you will be unlikely to come across on a day-to-day basis.

The 'I' in SOLID is the *interface-segregation principle*, which says that a user of an interface should not be required to depend on more methods than it actually needs in order to complete its task. This is related to the single responsibility principle of keeping classes small and focused on a single thing. If a class gets too big, or an interface does too much, then it's best to break it into smaller classes with more tightly scoped interfaces.

For example, a shopping cart can be added to or read from, but during the checkout process, it only needs to be read from. If the interface for your shopping cart gets too large, you could split it into two—one that deals with adding and another with reading—and then have a class that implements both interfaces.

Dependency inversion is one of the more famous components of SOLID and, especially in frameworks for strong object-oriented languages like Java, one you are forced to tackle head-on, especially if you want to do any unit testing. The approach to building a running system out of components that implement dependency inversion is called dependency injection. Dependency injection can also be a headache for new developers, those who are not used to enterprise environments, and those who are coming from weaker OO languages, but this is more due to overly large and onerous dependency injection frameworks than the concept itself. With dependency inversion, you are not responsible for finding the classes or modules that you need to do your tasks; instead, you are given them. Sometimes this is done using dependency injection frameworks like Spring, but you can also write code that injects the appropriate methods directly (known as wiring code), without using a framework.

As a result of dependency inversion, you do not depend on concrete instances of your dependencies, but instead on the interfaces. For example, a class that talks to a backend service, rather than instantiating its own HTTP client, can be given a client that implements the same interface. In development mode, this could be a client that implements the need to work through corporate proxies, but in production mode, which may be deployed to AWS, it could be one that does not use proxies, but perhaps implements a caching layer instead.

Although the SOLID principles are formulated around object-oriented programming, they can be repurposed for other paradigms too. For example, in dependency injection, dependencies are often passed to the constructor, but in functional programming style, the dependency inversion principle can be implementedusing closures.