





|  |
| --- |
| Course Overview  Course Overview  Hi, everyone. My name is Michael Hoffman. Welcome to my course, Spring Cloud 2022: The Big Picture. Along with being an author at Pluralsight, I'm also an architect at NVISIA, based out of Chicago, Illinois. I've been implementing microservices with Spring Cloud for several years, and I'm very excited to share my experiences with you. While microservices take advantage of being cloud native and distributed, developing them can be very challenging. Over time, patterns have emerged to address these challenges. This includes service discovery, registration, routing, inter‑service communication, and event‑driven communication. The Spring team took on these challenges when they engineered Spring Cloud. Spring Cloud provides you with a set of tools that address these patterns and greatly accelerate your microservice development. Some of the major topics that we will cover include service discovery using Spring Cloud Netflix, API Gateway development using Spring Cloud Gateway, and event‑driven microservices using Spring Cloud Stream. By the end of the course, you'll have a high‑level overview of the most popular tools in Spring Cloud version 2022. Before beginning the course, you should be experienced with developing applications using the Spring Boot framework. I hope you'll join me on this journey to learn Spring Cloud with the Spring Cloud 2022: The Big Picture course at Pluralsight.  Introducing Spring Cloud  Introduction  Hi, I'm Michael Hoffman. Welcome to this course on Spring Cloud 2022. In most organizations, a common goal is for systems to take advantage of being distributed and cloud native. This can have a number of benefits, including elasticity, reliability, resiliency, agility, speed, and cost reduction. Microservice architecture is often synonymous with cloud native. Microservice architecture is a faster way to ship product features to market when compared to a traditional monolithic architecture. Let's look at an example for context. I'm going to cover a use case for an organization's e‑commerce website. The website supports hundreds of educational bookstores across the United States. The bookstores sell textbooks and merchandise. The bookstore website is based on a monolithic architecture model. The application is hosted on‑prem and data centers that are owned by the bookstore organization. The application is structured in the modules representing different bounded contexts that include customer and order. While modularity is a key principle for the architecture, developers have broken the principle over time due to challenges, such as tight deadlines for feature development. The modules then communicate with different integrations, including databases, message brokers, and APIs. Every year, hundreds of features are added to the website. It's become so complex that it takes developers 3‑6 months to complete a release. If the bookstore wanted to take advantage of a microservice architecture and become cloud native, what factors would they need to consider? Let's start with deployment considerations. With a single deployment, any change that goes into production includes all of the modules of the website. Can you think of some of the challenges that the bookstore faces with this deployment model? Since the bookstore deploys to production every 3‑6 months, each release contains a large number of features. Without isolation, the number of changes going into production at the same time increases the complexity of the deployment, as well as the risk for issues. In fact, the last two production releases failed and required emergency fixes. To address this, the bookstore increased their testing coverage and the number of testing cycles. Due to the number of features that were being pushed in a release, the number of bugs found also increased, as well as the length of testing cycles. This impacted agility by increasing the cost of releases and lengthening the time that it requires for features to be shipped. Despite more rigorous testing, not all test cases could be covered and significant bugs were still released to production. Finally, when failures occur during a deployment, the entire website gets rolled back. Along with the website, downstream dependencies also require rollback, such as the database schema. As a result of the rollback, no features are released into production until the next deployment window. If the bookstore website was converted into a microservice architecture, it might look something like this. A web client would interact with microservice APIs, like order service and customer service. How could this microservice architecture address the deployment challenges of the monolith? The bookstore could isolate changes to the web client or to the individual microservice. This should reduce the risk of impact during deployment. For example, if a feature adds two new attributes to the customer microservice, it's unlikely that other services will be impacted. In fact, that customer microservice feature could be deployed to production before it's used by a web client or another microservice. With changes isolated, the bookstore should also see a reduction in testing cycles. While some regression may still be required, it's likely that testing will have a more focused approach for each microservice. Finally, deployment failures could be isolated enough that only a partial rollback is required. If the customer microservice change for two new attributes succeeds, but a separate and unrelated change fails, then the customer microservice change doesn't need to be rolled back. That covers deployment considerations. Next, let's look at considerations for scaling. The bookstore has an interesting business model. At certain times, a rush of orders will come in. This rush coincides with the start of school or the start of a semester. As most U.S. schools begin their school year in August, the company sees their biggest spike of orders during July and August. These 2 months account for a whopping 75% of yearly revenue. During these two months, the company's IT organization freezes all deployments and converts into busy season mode. This means all hands on deck to make sure orders are being processed. During their busy period, the bookstore needs to scale their infrastructure to handle the load. Think about some of the challenges that they may face with scaling. Each year, the operations team provisions more servers to keep up with the increase in load. These new servers are hosted on‑prem and will be maintained even past the busy period. Scaling servers in this manner has resulted in significant increases to operational costs. While the new servers are required to support the load of their busy period, they quickly lose value once this period is over. For example, CPU usage on the servers is often less than 1%. More operational staff needs to be hired to manage the servers, as they require upgrades and patches to be applied regularly. When operations provisions these new servers, they're making an educated guess about the expected load during the busy period. Unfortunately, this guess is usually wrong. To compensate, operations will convert servers from lower‑level environments into production servers whenever the volume spikes. As a result of converting these servers, development and testing of emergency fixes happens on local workstations, increasing the risk that a fix will have negative impacts when deployed to production. If the bookstore website was converted into a microservice architecture, how would it address the scaling challenges of this monolith? Here is where the bookstore can take advantage of the benefits of a cloud‑native application. Rather than provisioning servers, operational staff can use the cloud to assure proper scaling of individual services to handle increased load. For example, orders and customer service will likely require additional resources as new students will be placing new orders. Item service would likely not need to be scaled, as the number of items stays fairly static during the busy period. Most importantly, both during the busy period and after, these services can automatically be resized to reduce the cost of operation. If there are unplanned spikes during the busy period, there's no restriction on the number of services that can be brought up to handle the load. This infinite scaling would assure the bookstore meets the demands of customers and reduce any potential downtime or performance concerns with placing orders. These two considerations would have a definite positive impact if the bookstore moved to a cloud‑native microservice approach for the website. But there is a third consideration that needs to be made. Realizing the benefits of cloud‑native systems and microservice architecture can be very challenging. Engineering teams need to tackle the many new and longstanding problems with distributed systems. How can these problems be addressed? By applying best practices and patterns. One such pattern is the automatic registration and discovery of services. Why would the bookstore need service discovery? Service discovery could support automatically scaling resources up or down to meet the increased demand of their busy period. This automated scaling can assure that performance requirements are met for a system, increasing customer confidence and assuring the success of all transactions. When microservices are provisioned, clients need to interact with them consistently and successfully. The bookstore web client shouldn't need to know that the number of order service instances has changed, or even if the order service has been physically relocated to a new region. To address service routing, an API gateway pattern is implemented. An API gateway can have a number of benefits, including centralized API management and reliable communication with services. Finally, services need to communicate with other services. This communication may be between microservices in the same system or with managed services from cloud providers. For example, the bookstore's order service will need to retrieve customer data from the customer service to complete an order. This communication can happen with a number of protocols. Having approaches that can reduce the complexity of API interactions helps developers be more productive and speeds up the delivery process. In 2015, the Spring team released the first version of Spring Cloud. Spring Cloud is a set of tooling that helps you consistently and successfully build Spring Boot applications to be distributed and cloud native. The tooling implements many of the common patterns and best practices in distributed models, including the ones that I highlighted. This court will focus on the most recent 2022 release. The course will help you answer several questions about Spring Cloud. You'll discover key libraries that are part of Spring Cloud. Examples include Spring Cloud Netflix, Spring Cloud Gateway, and Spring Cloud Stream. You'll learn about the problem that each Spring Cloud library solves. Finally, you'll hear the trade‑offs for each Spring Cloud library to help guide you in choosing whether or not to use it in your solution. Now, let's see who this course is targeted to. As Spring Cloud is addressing distributed patterns for Spring Boot applications, it's expected that you either have or will be using Spring Boot in your solutions. The primary audience is practitioners, such as developers, engineers, and architects. As the content is presented at a high level, other personas, like product owners or technical leaders, may also obtain value from the information in the course. I expect that you are evaluating Spring Cloud and want to understand concepts, as well as leading opinions so that you can be better informed in making decisions. It's also important that you know what to expect from this course. You shouldn't expect deep dives, coding exercises, or demonstrations. As I cover concepts, I'll be referring you to external resources where you can perform a deeper dive or find more technical solutions.  Version Check  Spring Cloud uses the concept of a release train for versioning. In addition, each Spring Cloud library includes its own version. Given this, it's important to do a version check so that you understand which versions were targeted by the course. The course was created with version Spring Cloud 2022.0.3. This version of the release train is also known as Kilburn and version 3.0. Let's cover the Spring Cloud and Spring Cloud libraries this course is applicable to. First, this course applies to any minor version of Spring Cloud 2022.x. In addition, there are several Spring Cloud library versions to be aware of. This course covers the 4.x version of Spring Cloud Netflix, Spring Cloud OpenFeign, Spring Cloud Gateway, Spring Cloud Stream, and Spring Cloud Function. This course does not apply to versions of Spring Cloud that are version 2021 or older. These older versions included libraries that may no longer be supported by spring  What Is Spring Cloud 2022?  In this section, I'm going to start unpacking Spring Cloud 2022. First, let's set some context by looking at a typical Spring Boot microservice application. I'll cover why Spring Boot was developed and how Spring Cloud fits in with it. As the Spring framework evolved, the number of libraries and transitive dependencies increased greatly. Spring Boot was developed as an opinionated way of adding and managing these dependencies for your application. The most common way for using Spring Boot is as a backend service that supports APIs access by web clients, mobile apps, and IoT devices. Let's look at an example of the customer microservice created to support the bookstore website. Communication inbound to and outbound from the service is through HTTP and a REST controller. This REST controller provides endpoints for working with resources using HTTP methods like GET, POST, PUT, and DELETE. The REST controller then executes business service logic. Oftentimes this logic is performing some task like merging a customer account. The service logic leverages repositories to read and write data. Repositories interact with other service APIs, data stores, and message brokers. For example, after merging a customer, the customer service may publish a message to a customer message broker for other domains to subscribe to. These domains will receive the message and handle merging customer data that's related to their domain. And finally, properties allow you to configure the runtime for your Spring Boot service. Spring provides you a rich feature for properties with multiple ways that a property value can be overwritten. The Spring framework and Spring Boot support dependencies, either at one of these layers or across them. Spring Web brings in a server, such as Apache Tomcat, and leverages Spring MVC to provide a framework for developing REST controllers. It also supports mapping requests and responses from HTTP endpoints. Spring Security adds support for authentication and authorization of requests. Spring Data provides a framework for SQL and noSQL data access. Spring Context provides support for externalized configuration properties. This is just a small number of the dependencies available, but it demonstrates how Spring Boot supports these dependencies. So how does Spring Cloud fit into this view? Spring Cloud provides libraries that are managed by Spring Boot either as part of the microservice or as part of a separate component. Let's take a more holistic view of the customer microservice. The microservice will be accessed by devices like mobile phones, IoT, or a browser. Let's cover a real example where you could take advantage of distributed services. At the same time these devices are reading and writing customer data, a business user decides to merge two customers. The customer with the merge has 700 accounts, which means that all accounts will have their credit rechecked. Rechecking credit scores is an intensive process, and it can greatly impact primary business operations. If the customer service could be scaled up to add an additional instance, it would alleviate the load for credit checks and greatly reduce the impact for users. This new instance needs to be registered and discovered to start participating in the credit rescoring process. It will have its own host and port so that clients can successfully call it. The Spring team developed a Spring Cloud library that supports this. Spring Cloud Netflix integrates with Netflix Eureka. Eureka provides a registry for a service instance. The Spring Cloud Netflix library also provides a client API to make it easy for services to register themselves with the Eureka service so that they can be discovered and participate in request processing. With services being provisioned this way, it will be helpful to have a mechanism in place for routing to them. The Spring team developed Spring Cloud Gateway for this purpose. Spring Cloud Gateway is an API gateway that handles client request routing, in addition to other features for rate limiting, path rewriting, filters, and security. Spring Cloud Gateway also integrates with service discovery. With these libraries in place, my microservice can easily scale up to meet the demands from this credit recheck load. I'm going to remove some of this view to focus on service communication. A key consideration for having a cloud‑native microservice is taking advantage of cloud services provided to you or services you've provisioned in the cloud. These services could include a database, a message broker, a serverless function, or another Spring Boot microservice. Communicating with these different endpoints can increase complexity for your microservice. To address this, the Spring team has created libraries in Spring Cloud to help you with implementing this communication. Spring Cloud OpenFeign provides a declarative HTTP client for communicating with external services. This removes a lot of the boilerplate code required for implementing your own web service client in a microservice. Instead of setting headers, binding request attributes, and call a REST template, OpenFeign provides an interface to abstract this logic away from you. Spring Cloud Stream provides an abstraction for calling popular messaging systems, like RabbitMQ, Kafka, and SQS. Similar to OpenFeign, Stream is removing some of the boilerplate code required for binding the message and calling the underlying messaging APIs. Finally, Spring Cloud Function provides a uniform model for implementing serverless functions across serverless providers. In total, Spring Cloud 2022 includes over 30 libraries as part of the release. As this course is providing a big picture overview of Spring Cloud, I don't plan to cover every library available. Now that you know what Spring Cloud is, I feel it's important for you to know how Spring Cloud evolved into its current 2022 release. When Spring Cloud was first released about 2015, Netflix had become the gold standard for microservice architecture. Given this, Spring Cloud libraries focused on integrating with these Netflix offerings. Netflix open‑sourced many of the libraries that helped them achieve success with the microservice architecture. This is a list of 10 libraries Spring Cloud integrated with in early releases. Around 2018, Netflix shifted its focus for its open source offerings as it evolved its architecture. As a result, Netflix either placed libraries into maintenance mode or stopped supporting them all together. In the Spring Cloud 2019 release version 2.1, any Netflix library that was put in maintenance mode saw the corresponding Spring Cloud integration put in maintenance mode as well. Eureka is the only remaining Netflix open source integration that's left in the Spring Cloud Netflix library. It is still actively maintained with a new version having recently been released. Feign was a Netflix library, but it's since been transferred to a developer community. It is now called OpenFeign. In the releases leading up to 2022, the Spring team has continued to enhance the Spring Cloud library. This has led to some turbulence with developers who are using the libraries. Version 2022 saw several libraries enter maintenance mode, including Spring Cloud CLI, Spring Cloud Cloud Foundry, and Spring Cloud Sleuth. It also saw significant investment in other libraries for enhancements and fixes. Now that you have context on what Spring Cloud is, you may be asking, how should you use it in your application?  Using Spring Cloud 2022 in Your Solution  In this section, I'll provide you with options for leveraging Spring Cloud 2022. First, it's important to understand that most Spring Cloud libraries are independent of one another. By bringing Spring Cloud into your architecture, you can choose to use anywhere from 1 to all 30 libraries, depending on your requirements. Adding Spring Cloud will not bloat your deployment artifact with unwanted dependencies. A key factor in choosing Spring Cloud libraries is your microservice platform. Kubernetes is a container orchestration platform that directly implements some of the same patterns as Spring Cloud. This doesn't mean Kubernetes completely replaces Spring Cloud, but it's likely to reduce the list of Spring Cloud candidates that you want to leverage. Let's look at an option where a microservice leverages Spring Cloud and is deployed to Kubernetes. I'm going to use the Spring Boot customer domain service again for this example. The microservice will be deployed out to a Kubernetes cluster. Here's the customer service. When I showed this diagram earlier, I included Spring Cloud Netflix as a dependency. In this diagram, you may be wondering why I left it out. Kubernetes provides native support for server‑side discovery and registration, along with support for tooling, like a service mesh. Given this, it's likely that you'll prefer to use the native tooling. Kubernetes also provides support for configuration through config maps and secrets. While it's not covered in this course, these features are preferred over Spring Cloud config for externalized microservice properties. Another consideration to make is the API gateway. Kubernetes with a service mesh can support some API Gateway features. However, Spring Cloud Gateway could also be used as an external API Gateway. Focusing again on the service, you could see that libraries like Spring Cloud OpenFeign, Spring Cloud Stream, and Spring Cloud Function are independent of this deployment platform. If you're not deploying to Kubernetes, then the view I presented earlier, it's a viable option. Again, the key difference is the inclusion of Spring Cloud Netflix for service discovery. Now that you have a better idea of what Spring Cloud is and how it can be used, let's unpack each of the key libraries.  Service Discovery with Spring Cloud  Concepts of Service Discovery  Hi, I'm Michael Hoffman. In this module, I plan to unpack the pattern for service discovery. Then I'll explain how Spring Cloud Netflix and Netflix Eureka help you to implement this pattern. Let's start with the definition of service discovery. It's a pattern for dynamically locating and making requests to a service on a network. This pattern helps support a key principle in microservice architecture. Microservice instances are intended to be ephemeral. This means that on demand, service instances can be quickly and easily created or destroyed with little to no impact on service clients. To better understand this pattern, I'll cover a few key concepts. First, let's look at why the automatic configuration of new service instances is important to service discovery. If we revisit the bookstore organization from earlier in the course, I explained how the website had migrated to a microservice architecture. Services are included for domains, such as order and customer. During their busy season, the bookstore wants to scale up instances of the customer service to handle increased load. To understand the importance of automatic provisioning, let's look at the alternative. What if the bookstore manually provisioned a new instance of the customer service? Can you think of a problem with this approach? The new customer service will have a different URI to access it, so the order service won't be able to automatically locate it. The order service would require a mechanism to recognize that the instance exists. Likely, this means a configuration change. Applying these changes manually is error prone. You'll need to enter the right address in the right port. It's like the other services call the customer service, so this change needs to be manually applied across all of them. And what happens if an instance goes down? You'll need to manually remove all the references to the service instance. Automatic configuration reduces the time it takes to provision new instances. It also removes the manual step of provisioning, reducing the potential for errors. To support automatic provisioning, a service registry can be used. The best way to think about a service registry is as a phone book. When a service instance comes online, it broadcasts itself to the registry. The new service instance can now be looked up and called by any other service. In turn, these services are also retrieving the list of available services from the registry. This communication to and from the registry happens automatically. There's no configuration change that needs to be manually applied. A service registry provides a number of benefits. Most importantly, it adds dynamic registration and look‑up of instances. It can also provide you with observability to the list of instances available. In a distributed service environment, just registering a service is not enough. You need to think about what could happen if something goes wrong with an instance. Given this, another foundational concept with service discovery is the use of health checks. Health checks provide resiliency to your microservices. How could health checks be used as part of service discovery? Here's the order service. It's currently aware of two customer service instances from the service registry. All three services have been registered with the registry. The service registry is frequently pinging the health check endpoint of each instance. One instance of the customer service gets a request to recheck the credit score for a very large customer. The service crashes due to an out of memory error. The registry knows that one instance has crashed and will evict the service instance. The order service is frequently receiving a list of customer service instances from the registry. Because the registry took this instance out of the list, the order service knows about it. As a result, the order service will stop calling the crashed instance, rather than continuing to call it and erroring out. If the service comes back online, it can re‑register and be called again by the order service. Health checks provide a key benefit to resiliency in microservices architecture. Services can automatically adapt when failures occur, reducing the impact to customers and to data processing. Discovering a new service instance is not enough. A last concept to consider is load balancing the requests to it. Once a new customer service instance has been registered, the order service knows the total number of instances has gone from 1 to 2, but requests will continue to be sent to the first instance without some load balancing mechanism. As part of service discovery, a strategy, such as round robin, needs to be put in place to assure requests are evenly distributed among services. With this strategy in place, requests can be staggered across instances. Load balancing requests across instances assures that service discovery won't inadvertently overload a single instance, potentially bringing it down. That covers the main concepts. Now, let's look at how Spring Cloud implements these concepts.  Understanding Spring Cloud Netflix  Spring Cloud provides several options for implementing the service discovery pattern. Spring Cloud Netflix is one of the most widely used options. Spring Cloud Netflix enables service discovery through Netflix Eureka. Netflix Eureka is an open source service that implements a client server model for service discovery. It supports a service registry, load balancing, and failover. Netflix shared Eureka as open source in 2012. Given the popularity of Netflix components for implementing microservice architecture, the Spring Cloud team chose it as a tool in the library. Spring Cloud Netflix supports service discovery with Netflix Eureka. The library allows you to declare Spring Boot applications as either a Eureka client or a Eureka server. The server acts as the service registry. The server can be clustered for high availability. Servers can also be secured using Spring Security. Clients can then register with the server in order to discover dependent services. Both the client and server can be highly customized through configuration properties. Let's walk through a basic setup of Spring Cloud Netflix. The first step for setting up Spring Cloud Netflix is the Eureka server. The Eureka server is simply a Spring Boot application with a Eureka server dependency. While I'm showing a single node, it's likely you'll have at least two nodes of the server running for high availability. Once the server is deployed and running, it will start supporting registration by other service clients. Let's look at the code for declaring a server. I'm displaying a standard Spring Boot application class. What's interesting about this is that I've decorated the class with an annotation, @EnableEurekaServer. This annotation tells Spring Boot that I want to configure the application to run as a Netflix Eureka server. From a programming perspective, that's all the application logic I need to define the server. In addition to the annotation, I need to add configuration properties. Let's look at that now. Here's the application.properties file I'm using to configure the Eureka server. I'm using the minimal properties for server configuration. However, there are more to explore depending on your requirements. The first property defines the name of the Spring Boot application as Eureka server. Next, I'm setting the server port to 8761. Finally, I'm setting Eureka client properties. You may be wondering why. The Eureka server is also a Eureka client. This disables the server from trying to register to itself. This is all that's required to have a full functioning Eureka server. Now let's look at the Eureka client. With the server started, clients can now start registering to it. These clients are also Spring Boot applications. Similar to the server, they will leave\_\_\_\_ the Spring Cloud Eureka dependency and require configuration to take part in service discovery. Once you've configured the client to locate the Eureka server, it will let the server know that it wants to be registered. The Eureka server will add the service location using the service application name as the location path. The Eureka server will also send back the list of services available in the registry to the client. This registry cache will be sent on a frequent basis to assure that the client knows if a new service instance was provisioned or if it was evicted. The Eureka client can now call other service clients using the application name instead of the IP address in port. Let's look at the configuration for one of the service clients. Once we've added the Spring Cloud Netflix Eureka client dependency, configuration is fairly simple. Here, I'm showing two properties in the order service. The first property is defining the Spring Boot application name that will be used for service location. Then I'm setting the URL of the Eureka server that I want this service to register to. This is all you need to do to get a basic Eureka client and server started. Lets look at two additional features provided by Spring Cloud Netflix. First, let's look at health checks. As health checks are integral to automatically provisioning services, it should come as a surprise that Eureka includes support for them. By default, Eureka server uses the Eureka client's health check as a heartbeat. Once the service is registered, the status will be updated as available. If the service becomes unavailable as a result of the health check, the Eureka server will take the service out of the registry for you. Next, let's look at load balancing. While Eureka does not provide load balancing out of the box, Spring Cloud Netflix integrates nicely with Spring Cloud Gateway to provide it for you. You just need to configure Spring Cloud Gateway in order to find the services in the registry. Spring Cloud Gateway can then handle the load balancing strategy that you require. Now that you have an understanding of Spring Cloud Netflix, let's look at why you would use it in your solution.  Using Spring Cloud Netflix  It's important to understand the benefits and risks of using Spring Cloud Netflix. I plan to cover several key considerations you should make when choosing Spring Cloud Netflix for service discovery in your solution. First, Netflix Eureka is a mature solution that has seen multiple feature releases. Given this, it should be applicable to most service discovery use cases. While Eureka is mature, the library has had limited maintenance from Netflix. This is a pattern, as Netflix no longer supports or has reduced maintenance of many of its open source offerings. Between October of 2021 and November of 2022, there has been only one release of version 1. The release fixed a single bug. Version 2 was released in December of 2022, but it has also seen limited development. An argument could be made that the limited need for support is due to the robustness of Eureka. Support and maintenance of Eureka is a potential risk that you need to weigh when deciding on Spring Cloud Netflix. Finally, if you're using Kubernetes, it's unlikely that you'll choose Spring Cloud Netflix. Kubernetes has built‑in support for service discovery as part of its platform. That covers Spring Cloud Netflix and Netflix Eureka for service discovery. If you would like to understand more about Spring Cloud Netflix, I provided you with the website for the library's page. You'll find examples, source code, and the full documentation.  API Gateway with Spring Cloud  Concepts of an API Gateway  Hi, I'm Michael Hoffman. In this module, I'll cover the pattern for an API Gateway. Then I'll show you how Spring Cloud Gateway implements this pattern. Let's start by defining what an API Gateway is. It's a pattern for routing client application requests to microservices. Clients could be any type of requester, such as an internal web application, an external application, or a mobile application. It centralizes management and communication both to and from microservices via a proxy. Why would you need an API Gateway in your architecture? Microservices are distributed, which means the communication to and the management of these microservices is also distributed. Do you think of some of the challenges that this could present? Security could be a concern. What if I want to restrict certain clients from making requests to a microservice API? How do we implement authentication and authorization consistently across these microservices? Version management is another concern. What if I want to deploy a new version of the customer service? How could I support routing to both customer service versions for a period of time? A final concern is observability. How do I consistently collect metrics across microservice requests and responses for analytics and monitoring. An API Gateway addresses these concerns, along with several others, by providing a single entry point for your clients to communicate with your microservices. The goal is to reduce the complexity of communication between the clients and microservices. As the complexity of client needs and microservice implementations increase, you can abstract most of this complexity through the API Gateway. I'm going to cover four key concepts of an API Gateway. Transforming the client request is a common practice for an API Gateway. As requests come to the gateway, a variety of rules can be applied. These rules can result in normalizing the client request, sanitizing request data to protect against security attacks, and transforming the request to support multiple versions of a microservice. Let's look at an example of this. I'm going to revisit the bookstore example from earlier in the course. The bookstore is experimenting with new features during order intake. Their current order intake is through an order microservice behind an API gateway. This is version 1 of the service. When they deploy out experimental features, version 1 of the service has to maintain reliability. A second version of the order service can be deployed with the experimental features. It's likely that feature flags will also be used to control the functionality of the new service version. The API Gateway rules can be configured to route requests to both versions of the service in parallel. The API Gateway can also modify the incoming request to support the API contract of order service version 2. Finally, the API Gateway can monitor requests to track if the experiment was a success. Along with versioning, the bookstore is also using the API Gateway to manipulate request headers. For example, the bookstore can have rules in place to validate, scrub, inject, or update security headers. These rules could prevent a request from even reaching the microservice if the request is not properly authorized. These are just a few examples of how an API Gateway can transform requests. Hopefully these examples demonstrate to you how request transformation can improve the communication between clients and microservices. Let's look at another key concept. When you expose a microservice to a client, it's important to consider what could go wrong. For example, clients may intentionally or unintentionally make requests that can have a negative impact on the quality of your microservice. One approach for protecting your microservices is by using rate limiting. Rate limiting could have a variety of benefits, including the prevention of denial‑of‑service attacks, cost control for resource usage, and monetization of service calls. Let's look at an example of how rate limiting could be applied. The bookstore has a third‑party vendor integration. The third party can perform searches on the items provided by the bookstore. Performing a search through the item service could be resource‑intensive, since the bookstore has thousands of items for sale. How can the bookstore assure that customers aren't impacted if the vendor has a burst of requests? Rate limiting can be added to the API Gateway to limit the number of requests to the item service. In this case, the bookstore decided on 10 requests per second. With this limit in place, the chances of the item service being negatively impacted by the vendor is greatly reduced. If this limit needs to increase, the change is centralized to the API Gateway, making it easier to manage. This limit could even be applied to individual vendors based on usage. Rate limiting is a valuable concept in an API Gateway. It provides a centralized and standardized approach for protecting your microservices. A third concept of an API Gateway is load balancing. I detailed the benefit of load balancing in the previous module on service discovery. Load balancing can be used when scaling up microservices to assure the load is equally distributed. Finally, an API Gateway provides you with a way to centrally monitor all incoming requests. Monitoring at the gateway provides you visibility into potential security attacks, error detection, service health, and performance. Let's return to the bookstore example. Once the bookstore allows an external vendor to access their services, it's going to be critical to have monitoring in place. The bookstore can log requests to the API Gateway. With logging in place, think of some ways that the bookstore is going to see benefits. If there's an SLA in place with the vendor, reports can be generated that either prove or disprove that the SLA is being met. API Gateway logs can be used for detecting and resolving errors with vendor requests. The bookstore can also plan around service usage trends as the vendor usage increases. Monitoring is an important concept for an API Gateway, as it provides observability into the performance, reliability, and security of your microservices. That covers the main concepts. Now, let's look at how Spring Cloud Gateway implements these concepts.  Understanding Spring Cloud Gateway  Spring Cloud provides a single library for implementing an API Gateway. The library is called Spring Cloud Gateway. Spring Cloud Gateway is a powerful and highly customizable API Gateway implementation. It handles the routing of client requests to a microservice, providing filters for modifying the request to and response from the proxy microservice. Spring Cloud Gateway is built on project reactor to support non‑blocking APIs to improve scaling. Spring Cloud Gateway also has support for non‑HTTP protocols, like WebSockets.  To get a better understanding of Spring Cloud Gateway, let's see how it handles a request. The box I'm showing for Spring Cloud Gateway is just a Spring Boot application with the gateway library as a dependency.  When clients make a request to Spring Cloud Gateway, **a route mapper will match the request to the route.** The mapper uses a predicate to match anything from the request, including headers, parameters, the host, and the path.  Once the request is matched to a route, the request is processed through a chain of filters.  **These filters are the rules that are applied to the request** before the proxy call is made to the customer service.  Filters are how Spring Cloud Gateway supports request transformation, such as adding a request header. After the request processing is complete, a proxy call is made to the customer service. The response is returned to Spring Cloud Gateway. At this point, the request and response are processed through another chain of filters. Once this final processing is complete, the response is returned to the client. Part of the power of Spring Cloud Gateway is that route mapping and filters can be defined through either configuration or through code. The filter chain is extensible enough to meet almost any requirement you have for routing. In addition, Spring Cloud Gateway provides a number of filters for you out of the box. Let's look at an example of defining a Spring Cloud Gateway route.  I'm showing the **application.yaml** for a gateway application.  The route **has an ID, a URI, a predicate, and a filter.**  The predicate uses the path to match all client requests that will be mapped to this route configuration. A filter will add a request header to the request. Finally, the route will call the customer service URI using the updated request. Note that I'm using the text lb: in the URI. When Spring Cloud Gateway encounters the schema, it will load balance requests across instances for you.  Routes can also be defined through code. I'm showing you an example of a route configuration class that uses the Spring Cloud Gateway RouteLocator and RouteLocatorBuilder.  This is the same route definition as the previous one that I defined in an application configuration. You'll need to decide which approach works best for defining your routes. With an understanding of how Spring Cloud Gateway works, let's look at some key filters that support API Gateway concepts. The first filter is for implementing a rate limiter on requests, appropriately named request rate limiter gateway filter. To use this filter, you'll need to define a strategy for rate limiting, such as the number of requests per second. Spring Cloud Gateway will then return an HTTP status of 429 when the number of requests goes over the threshold. The reactive load balancer client filter supports load balancing requests across instances. This filter will be applied when the URI uses the schema lb:, as I showed in the earlier example. Finally, the gateway metrics filter leverages actuator to gather metrics for Spring Cloud Gateway. These metrics can then be used for monitoring and observability of your routes. Now that you have an understanding of Spring Cloud Gateway, let's look at why you would use it in your solution.  Using Spring Cloud Gateway  It's important to understand the tradeoffs before deciding to use Spring Cloud Gateway in your system. Let's look at some considerations in making this decision. First, Spring Cloud Gateway is a mature library. Point versions are released monthly, commonly to address minor bug fixes. While Spring Cloud Gateway is sufficient for implementations of any size, its extensibility and customizability make it one of the best API Gateway candidates for larger solutions. This is especially true with multitenant solutions where hundreds of clients may require their own routing logic and filters. The final consideration to make is that Spring Cloud Gateway is self‑managed. In some scenarios, it might be less complex and more cost‑effective to use a managed service, such as Amazon's API Gateway. That covers Spring Cloud Gateway. If you would like to understand more about the library, I've provided you with the website link. You'll find examples, source code, and the full documentation.  Event-driven Microservices with Spring Cloud  Concepts of Event-Driven Architecture  Hi, I'm Michael Hoffman. In this module, I'll cover the concepts of microservice communication driven by events. Then I'll show you how Spring Cloud Stream supports and simplifies developing microservices for an event‑driven architecture using event stream processing. Let's start with a simple definition of event‑driven architecture. It's an architecture pattern where event messages are used for communicating between systems. A key benefit of this pattern is that it leads to highly decoupled systems. Within an event‑driven architecture, a popular approach is event stream processing. This approach is intended to analyze and take action on a series of data points or events. These events are being continuously generated often by multiple sources over a period of time. Given the popularity of event stream processing, an interesting challenge has developed. The market for messaging services is diverse. An organization may implement one or more of the messaging services that I've listed. Building microservices to interact with these messaging services requires a different programming model per service. What's worse is that different microservices using the same messaging service could end up with disparate approaches. What if there was a way to abstract this disparity with a common programming model. Let's look at a candidate for a common messaging programming model. On the left side is the microservice integrating with a message broker. In this case, it's the bookstore's customer service. On the right are message brokers the bookstore uses in its architecture, including Apache Kafka, RabbitMQ, and Amazon Web Services SQS. Communicating with each message broker includes both input and output destinations for event messages. Rather than these inputs and outputs communicating directly with the broker, an abstraction layer could be put in place for binding to the broker. Each binder is specific to the message broker implementation. The input and output destinations can then be configured appropriately, such as setting an input destination to be a RabbitMQ topic. Next, let's look at a common pattern for stream processing. There are three phases for this pattern. On the left is the source, which is originating the event data. The data is then run through one or more processors. The processors are where the actual business logic happens. Finally, the data is stored or sent to another system in the sink. With an understanding of this model and pattern, let's dive into how Spring Cloud Stream helps you implement them.  Understanding Spring Cloud Stream  Spring Cloud provides the Spring Cloud Stream library to help you implement event stream processing and event‑driven microservice patterns. Spring Cloud Stream is a powerful tool for helping you quickly and easily implement message‑driven Spring Boot microservices. Spring Cloud Stream implements a common programming model for messaging. This abstracts you away from the details of interacting with message brokers. You can then focus on the important task of implementing business logic for your event‑driven solutions. Spring Cloud Stream implements the common programming model that I detailed earlier. Spring Cloud Stream ships with a number of bindings out of the box, including the ones I've displayed. The binder interface is pluggable, so you could create your own binder if one doesn't exist. Binders are very powerful. They abstract you away from the boilerplate code required for communicating with these messaging products. Binders are added as a dependent library to your Spring Boot application. Let's revisit the bookstore example. Anytime the customer service deletes a customer entity, other domain services may be entrusted. For example, the order service could delete all related orders for the customer. The bookstore wants to leverage RabbitMQ as the broker for these events. This requires the binder RabbitMQ dependency be added to the customer service. With this dependency in place, the customer service can be coded to create the message and send it to the binder. To support the stream processing pattern of source processor sink, Spring Cloud Stream leverages Java's functional interfaces. To take advantage of this, you will need to include the Spring Cloud Function library as a dependency. With Spring Cloud Function, you can use a supplier for your source, functions for your processors, and a consumer for your sink. Let's look at some code examples. I'm showing you a slice of the configuration source code for a supplier. When the application is executed, the customerDeleteSupplier method will return the ID for a deleted customer. Notice that there are no references to RabbitMQ. However, the RabbitMQ binder is a dependency. As a result, at startup, Spring Cloud Stream will bind a topic exchange for you to support producing messages from calls to the supplier bean. Here's the implementation of a consumer. Similar to the supplier, Spring Cloud Stream binds a topic exchange for you. Any messages produced to the exchange will be consumed by this customerDeleteConsumer bean. Finally, this is an example of implementing a function. This function could be applied to the input and output of an exchange. In this case, the customer bean deleted is checked to see if it exists before attempting to delete it. There are several key features of Spring Cloud Stream that I'd like to highlight for you. As you saw in the code examples, no logic was required to serialize or deserialize the message going to and coming from RabbitMQ. Spring Cloud Stream automatically handles this conversion for you by supporting many common formats, like JSON. Spring Cloud Stream supports partitioning data across multiple instances of an application, allowing you to process messages in parallel across consumers. Finally, Spring Cloud Stream integrates with Spring Boot actuator to support monitoring and metrics collection for your messaging application. Now that you have an understanding of Spring Cloud Stream, let's look at why you would use it in your solution.  Using Spring Cloud Stream  It's important to understand the tradeoffs before deciding to use Spring Cloud Stream for your event‑driven microservice application. Let's cover some considerations for making this decision. Similar to Spring Cloud Gateway, Spring Cloud Stream has matured and evolved over the span of releases. The introduction of functional interfaces should result in code that's easier to read and maintain. Spring Cloud Stream supports binding the most popular messaging brokers, reducing the chance for a gap with supporting your solution. As is common with abstractions, there could be overheads use a Spring Cloud Stream if the underlying messaging infrastructure is not configured properly. It's important to still have an understanding of the choice in binding, as well as the configuration and scaling required to meet your needs. Also, if your use case is simple, adding Spring Cloud Stream may overcomplicate directly calling the messaging system's APIs. That covers Spring Cloud Stream for event‑driven microservices. If you would like to understand more about the library, I provided you with a website link. You'll find examples, source code, and the full documentation  Additional Spring Cloud Capabilities  Understanding Spring Cloud Capabilities  Hi, I'm Michael Hoffman. In this module, I plan to cover additional capabilities in Spring Cloud 2022. Up until now, I've highlighted several key libraries in Spring Cloud, including Spring Cloud Netflix, Spring Cloud Gateway, and Spring Cloud Stream. In total, Spring Cloud 2022 covers over 28 libraries. These libraries are in various stages of evolution. Also, some libraries have been deprecated in this version and will be removed in a future release. If you plan to use a library in your solution, I highly recommend reading the library's page on the Spring Cloud website to determine if it's still active.  Understanding Spring Cloud OpenFeign  In this section, I'm going to provide you with an overview of Spring Cloud OpenFeign. Spring Cloud OpenFeign is a library that reduces the boilerplate client code for a microservice to call another microservice. OpenFeign supports this client communication through Java interfaces. OpenFeign uses the interface definition to generate the client call. This abstracts you from the complexities and repetitiveness of implementing a client on your own. As part of this library, Spring Cloud is using the open source library OpenFeign, which was originally part of a Netflix offering before being handed over to the community. This handover was due to Netflix no longer using it internally. Let's look at how this works with a bookstore example. The order service needs customer information to complete an order. Traditionally, the service would implement REST client code to call the customer service and get a customer. With OpenFeign, the order service will implement a FeignClient instead. The FeignClient will then handle the underlying logic for calling the customer service. The order service only needs to implement an interface describing the call. Here's an example of what the code will look like for this customer service client. The interface is annotated with a client name and URL provided. Then an interface method is defined and annotated with GetMapping to describe a call for getting a customer by ID. In just a few lines of code, OpenFeign simplifies the repetitive logic needed for calling a REST template, binding the path variable, and converting the incoming response. Spring Cloud OpenFeign supports several additional features. FeignClients are customizable. As part of configuration, you could change the behavior of encoders and decoders, configure the underlying HTTP clients, and add support for security access tokens. FeignClients can use placeholders for URLs so that they are dynamically replaced at runtime. FeignClients also support customizable error handling to respond appropriately when a client call fails. Finally, FeignClients support observability and monitoring through micrometer. Here are some considerations to make when choosing OpenFeign for your solution. Spring Cloud OpenFeign has continued to mature since its creation by Netflix. It's supported by the open source community and continues to see regular releases. The library supports most features required for client interaction, so you can be confident that Spring Cloud OpenFeign will meet your integration requirements. As Spring Cloud OpenFeign is abstracting you from the underlying HTTP mechanisms, it has the potential for adding performance overhead to processing. It's important to understand how your FeignClient has been configured to ensure it's making the most optimal calls to your microservices. That covers Spring Cloud OpenFeign for simplifying microservice‑to‑microservice communication. If you would like to understand more about the library, I provided you with a website link. You'll find examples, source code, and the full documentation. |