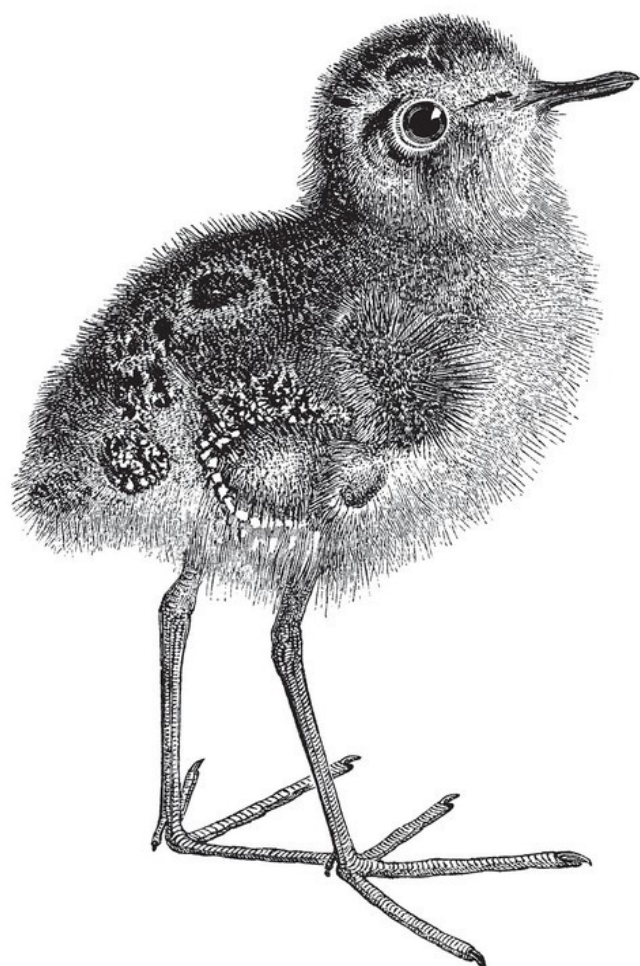


O'REILLY®

Building Multi-Tenant SaaS Architectures

Principles, Practices and Patterns using AWS



**Early
Release**

**RAW &
UNEDITED**

Tod Golding

Building Multi-Tenant SaaS Architectures

Principles, Practices and Patterns Using AWS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Tod Golding



Beijing • Boston • Farnham • Sebastopol • Tokyo

Building Multi-Tenant SaaS Architectures

by Tod Golding

Copyright © 2024 Tod Golding. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisition Editor: Louise Corrigan
- Development Editor: Melissa Potter
- Production Editor: Gregory Hyman
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- March 2024: First Edition

Revision History for the Early Release

- 2023-01-24: First Release
- 2023-03-15: Second Release
- 2023-04-27: Third Release
- 2023-06-05: Fourth Release
- 2023-07-17: Fifth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098140649> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Building Multi-Tenant SaaS Architectures*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of

or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-14064-9

Chapter 1. The SaaS Mindset

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

I’ve worked with a number of teams that were building software-as-a-service (SaaS) solutions. When I sit down with them to map out their path to SaaS, they tend to start out with what seems like a reasonable, high-level view of what it means to be SaaS. However, as I go a layer deeper and get into the details of their solution, I often discover significant variations in their vision. Imagine, for example, someone telling you they want to construct a building. While we all have some notion of a building having walls, windows, and doors, the actual na-

ture of these structures could vary wildly. Some teams might be envisioning a skyscraper and others might be building a house.

It's kind of natural for there to be confusion around SaaS. As is the case in all technology realms, the SaaS universe has been continually evolving. The emergence of the cloud, shifting customer needs, and the economics of the software domain are in constant motion. How we defined SaaS yesterday may not be the way we'll define it today. The other part of the challenge here is that the scope of SaaS goes well beyond the technical. It is, in many respects, a mindset that spans all the dimensions of a SaaS provider's organization.

With that in mind, I thought the natural place to start this journey was by bringing more clarity to how I define SaaS and how I think this definition shapes your approach to architecting, designing, and building a SaaS solution. The goal in this chapter is to build a foundational mental model that will, ideally, reduce some of the confusion about what it means to be SaaS. We'll move beyond some of the vague notions of SaaS and, at least for the scope of this book, attach more concrete guiding principles to the definition of SaaS that will shape the strategies that we'll explore in the coming chapters.

To get there, we'll need to look at the forces that motivated the move to SaaS and see how these forces directly influenced the resulting architecture models. Following this evolution will provide a more con-

crete view into the foundational principles that are used to create a SaaS solution that realizes the full value proposition of SaaS, blending the technical and business parameters that are at the core of developing modern SaaS environments.

While you may feel comfortable with what SaaS means to you, it's possible that the foundational concepts we'll explore here might challenge your view of SaaS and the terminology we use to describe SaaS environments. So, while it may be tempting to treat this chapter as optional, I would say that it may be one of the most important chapters in the book. It's not just an introduction, it's about creating a common vocabulary and mental model that will be woven into the architecture, coding, and implementation strategies that we'll be covering throughout this book.

The Classic Software Model

Before we can dig into the defining SaaS, we need to first understand where this journey started and the factors that have driven the momentum of the SaaS delivery model. Let's start by looking at how software was traditionally built, operated, and managed. These pre-SaaS systems were typically delivered in an "installed software" model where companies were hyper-focused on the features and functions of their offerings.

In this model, software was sold to a customer and that customer often assumed responsibility for installing the system. They might install it in some vendor-provided environment or they might install it on self-hosted infrastructure. The acquisition of these offerings would, in some cases, be packaged with professional services teams that could oversee the installation, customization, and configuration of the customer's environment.

Figure 1-1 provides a conceptual view of the footprint of the traditional software delivery model.

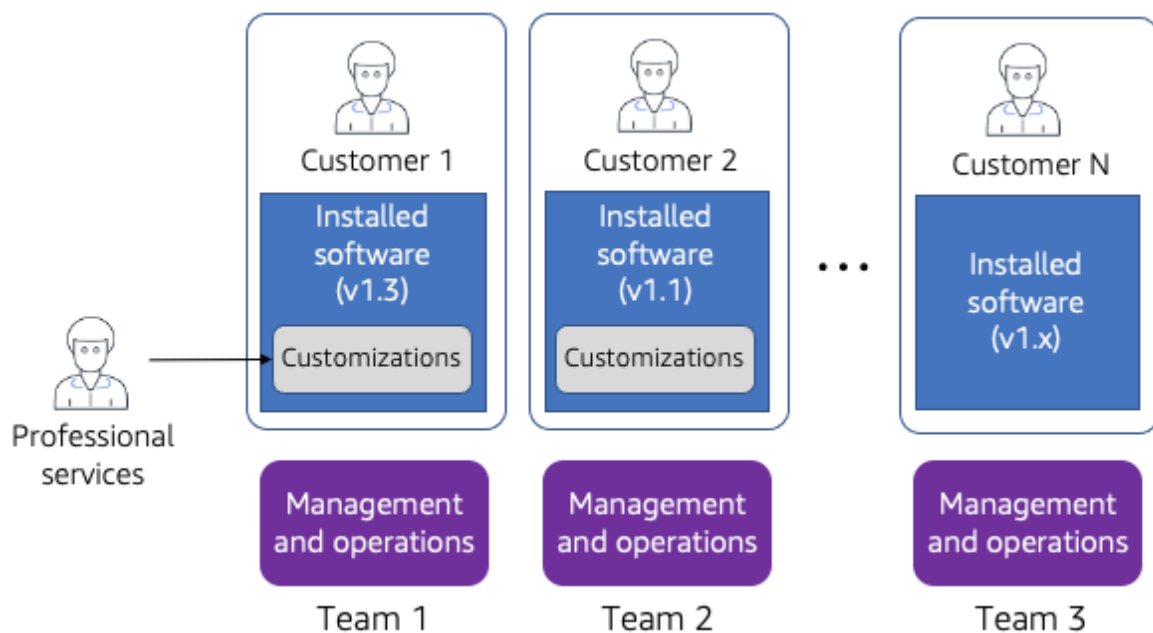


Figure 1-1. The installed software model

Here, you'll see a representation of multiple customer environments. We have Customers 1 and 2 that have installed and are running spe-

cific versions of the software provider's product. As part of their onboarding, they also required one-off customizations to the product that were addressed by the provider's professional services team. We also have other customers that may be running different versions of our product that may or may not have any customizations.

As each new customer is onboarded here, the provider's operations organization may need to create focused teams that can support the day-to-day needs of these customer installed and customized environments. These teams might be dedicated to an individual customer or support a cross-section of customers.

This classic mode of software delivery is a much more sales driven model where the business focuses on acquiring customers and hands them off to technology teams to address the specific needs of each incoming customer. Here, landing the deal often takes precedence over the need for agility, scale, and operational efficiency. These solutions are also frequently sold with long-term contracts that limit a customer's ability to easily move to any other vendor's offering.

The distributed and varying nature of these customer environments often slowed the release and adoption of new features. Customers tend to have control in these settings, often dictating how and when they might upgrade to a new version. The complexity of testing and

deploying these environments could also become unwieldy, pushing vendors toward quarterly or semi-annual releases.

The Natural Challenges of the Classic Model

To be completely fair, building and delivering software in the model described above is and will continue to be a perfectly valid approach for some businesses. The legacy, compliance, and business realities of any given domain might align well to this model.

However, for many, this mode of software delivery introduced a number of challenges. At its core, this approach focused more on being able to sell customers whatever they needed in exchange for trade offs around scale, agility, and cost/operational efficiency.

On the surface, these tradeoffs may not seem all that significant. If you have a limited number of customers and you're only landing a few a year, this model could be adequate. You would still have inefficiencies, but they would be far less prominent. Consider, however, a scenario where you have a significant installed base and are looking to grow your business rapidly. In that mode, the pain points of this approach begin to represent a real problem for many software vendors.

Operational and cost efficiencies are often amongst the first areas where companies using this model start to feel the pain. The incremental overhead of supporting each new customer here begins to

have real impacts on the business, eroding margins and continually adding complexity to the operational profile of the business. Each new customer could require more support teams, more infrastructure, and more effort to manage the one-off variations that accompany each customer installation. In some cases, companies actually reach a point where they'll intentionally slow their growth because of the operational burdens of this model.

The bigger issue here, though, is how this model impacts agility, competition, growth, and innovation. By its very nature, this model is anything but nimble. Allowing customers to manage their own environments, supporting separate versions for each customer, enabling one-off customization—these are all areas that undermine speed and agility. Imagine what it would mean to roll out a new feature in these environments. The time between having the idea for a feature, iterating on its development, and getting it in front of all your customers is often a slow and deliberate process. By the time a new feature arrives, the customer and market needs may have already shifted. This also can impact the competitive footprint of these companies, limiting their ability to rapidly react to emerging solutions that are built around a lower friction model.

While the operational and development footprint were becoming harder to scale, the needs and expectations of customers were also shifting. Customers were less worried about their ability to

manage/control the environment where their software was running and more interested in maximizing the value they were extracting from these solutions. They demanded lower friction experiences that would be continually innovating to meet their needs, giving them more freedom to move between solutions based on the evolving needs of their business.

Customers were also more drawn to pricing models that better aligned with their value and consumption profile. In some cases, they were looking for the flexibility of subscription and/or pay-as-you-go pricing models.

You can see the natural tension that's at play here. For many, the classic delivery model simply didn't align well with the shifting market and customer demands. The emergence of the cloud also played a key role here. The cloud model fundamentally altered the way companies looked at hosting, managing, and operating their software. The pay-as-you-go nature and operational model of the cloud had companies looking for ways to take advantage of the economies of scale that were baked into the cloud experience. Together, these forces were motivating software providers to consider new business and technology models.

The Move to Shared Infrastructure

By now, the basic challenges of the traditional model should be clear. While some organizations were struggling with this model, others already understood this approach would simply not scale economically or operationally. Larger business-to-consumer (B2C) software companies, for example, knew that supporting thousands or even millions of customers in a one-off model simply wouldn't work.

These B2C organizations really represented the early days of SaaS, laying the groundwork for future SaaS evolution. Achieving scale for these organizations was, from the outset, about building systems from the ground up that could support the massive scale of the B2C universe. They thrived based on their ability to operate in a model where all customers were presented with a single, unified experience.

This shift to sharing infrastructure amongst customers opened all new opportunities for software providers. To better understand this, [Figure 1-2](#) provides a conceptual view of how applications can share infrastructure in a SaaS model.

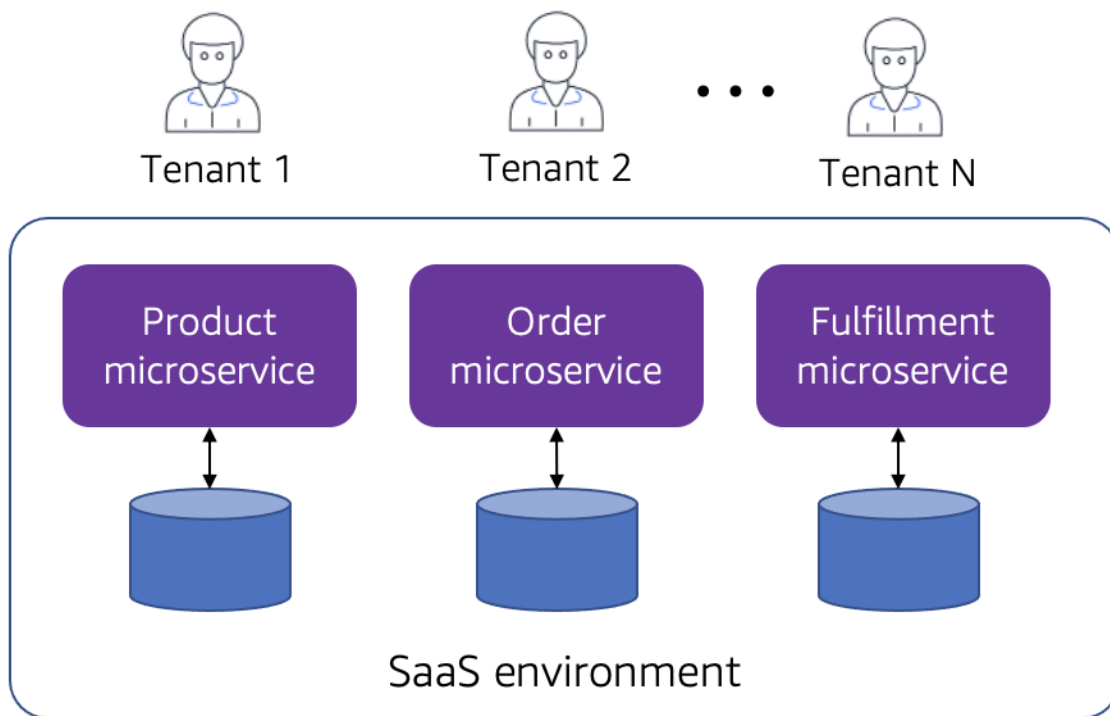


Figure 1-2. A shared infrastructure model

In [Figure 1-2](#) you'll see a simplified view of the traditional notion of SaaS. You'll notice that we've completely moved away from the distributed, one-off, custom nature of the classic model we saw in [Figure 1-1](#). In this approach, you'll see that we have a single SaaS environment with a collection of infrastructure. In this example, I happened to show microservices and their corresponding storage. A more complete example would show all the elements of your system's application architecture.

If we were to take a peek inside one of these microservices at runtime, we could potentially see any number of tenants (aka customers)

consuming the system's shared infrastructure. For example, if we took three snapshots of the Product microservice at three different time intervals, we might see something resembling the image in [Figure 1-3](#).

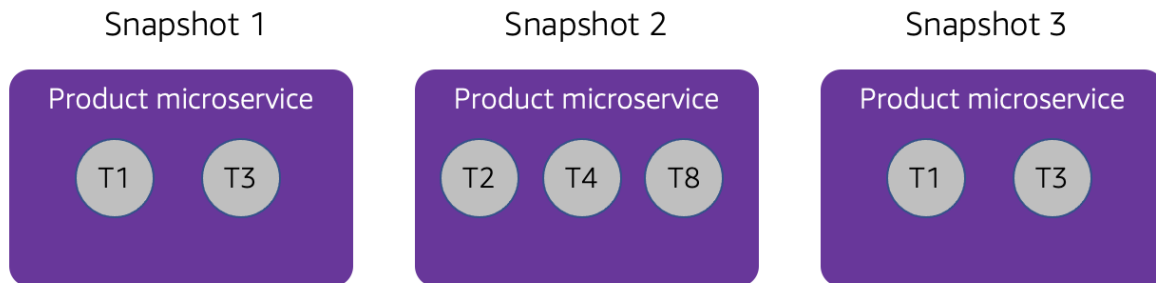


Figure 1-3. Shared microservices

In snapshot 1, our product microservice has two tenants consuming our service (tenant1 and tenant2). Another snapshot could have another collection of tenants that are consuming our service. The point here is that the resource no longer belongs to any one consumer, it is a shared resource that is consumed by any tenant of our system.

This shift to using shared infrastructure also meant we needed a new way to describe the consumers of our software. Before, when every consumer had its own dedicated infrastructure, it was easy to continue to use the term “customer”. However, in the shared infrastructure model of SaaS, you’ll see that we describe the consumers of our environment as “tenants”.

It's essential that you have a solid understanding of this concept since it will span almost every topic we cover in this book. The notion of tenancy maps very well to the idea of an apartment complex where you own a building and you rent it out to different tenants. In this model, the building correlates to the shared infrastructure of your solution and the tenants represent the different occupants of your apartments. These tenants of your building consume shared building resources (power, water, and so on). As the building owner, you manage and operate the overall building and different tenants will come and go.

You can see how this term better fits the SaaS model where we are building a service that runs on shared infrastructure that can accommodate any number of tenants. Yes, tenants are still customers, but the term “tenant” lets us better characterize how they land in a SaaS environment.

The Advantage of Shared Infrastructure

So, we can see how SaaS moves us more toward a unified infrastructure footprint. If we contrast this with the one-off, installed software model we covered above, you can see how this approach enables us to overcome a number of challenges.

Now that we have a single environment for *all* customers, we can manage, operate, and deploy all of our customers through a single

pane of glass. Imagine, for example, what it would look like to deploy an update in the shared infrastructure model. We would simply deploy our new version to our unified SaaS environment and all of our tenants would immediately have access to our new features. Gone is the idea of separately managed and operated versions. With SaaS, every customer is running the same version of your application. Yes, there may be customizations within that experience that are enabled/disabled for different personas, but they are all part of a single application experience.

NOTE

This notion of having all tenants running the same version of your offering represents a common litmus test for SaaS environments. It is foundational to enabling many of the business benefits that are at the core of adopting a SaaS delivery model.

You can imagine the operational benefits that come with this model as well. With all tenants in one environment, we can manage, operate, and support our tenants through a common experience. Our tools can give us insights into how tenants are consuming our system and we can create policies and strategies to manage them collectively. This brings all new levels of efficiency to our operational model, reducing the complexity and overall footprint of the operational team. SaaS organizations take great pride in their ability to manage and op-

erate a large collection of tenants with modestly sized operational teams.

This focus on operational efficiency also directly feeds the broader agility story. Freed from the burden of one-off, custom versions, SaaS teams will often embrace their agility and use it as the engine of constant innovation. These teams are continually releasing new features, gathering more immediate customer feedback, and evolving their systems in real-time. You can imagine how this model will directly impact customer loyalty and adoption.

The responsiveness and agility of this SaaS model often translates into competitive advantages. Teams will use this agility to reach new market segments, pivoting in real-time based on competitive and general market dynamics.

The shared infrastructure model of SaaS also has natural cost benefits. When you have shared infrastructure and you can scale that infrastructure based on the actual consumption patterns of your customers, this can have a significant impact on the margins of your business. In an ideal SaaS infrastructure model, your system would essentially only consume the infrastructure that is needed to support the current load of your tenants. This can represent a real game-changer for some organizations, allowing them to take on new tenants at any pace knowing that each tenant's infrastructure costs will

only expand based on their actual consumption activities. The elastic, pay-as-you-go nature of cloud infrastructure aligns nicely with this model, supporting the pricing and scaling models that fit naturally with the varying workloads and consumption profiles of SaaS environments.

Thinking Beyond Infrastructure

While looking at SaaS through the lens of shared infrastructure makes it easier to understand the value of SaaS, the reality is that SaaS is much more than shared infrastructure. In fact, as we move forward, we'll see that shared infrastructure is just one dimension of the SaaS story. There are economies of scale and agility can be achieved with SaaS—with or without shared infrastructure.

In reality, we'll eventually see that there are actually many ways to deploy and implement your SaaS application architecture. The efficiencies that are attributed to SaaS can certainly be maximized by sharing infrastructure. However, efficiency starts with surrounding your application with constructs that can streamline the customer experience and the management/operational experience of your SaaS environment. It's these constructs that—in concert with your SaaS application architecture—enable your SaaS business to realize its fundamental operational, growth, and agility goals.

To better understand this concept, let's look at these additional SaaS constructs. The diagram in [Figure 1-4](#) provides a highly simplified conceptual view of these common SaaS services.

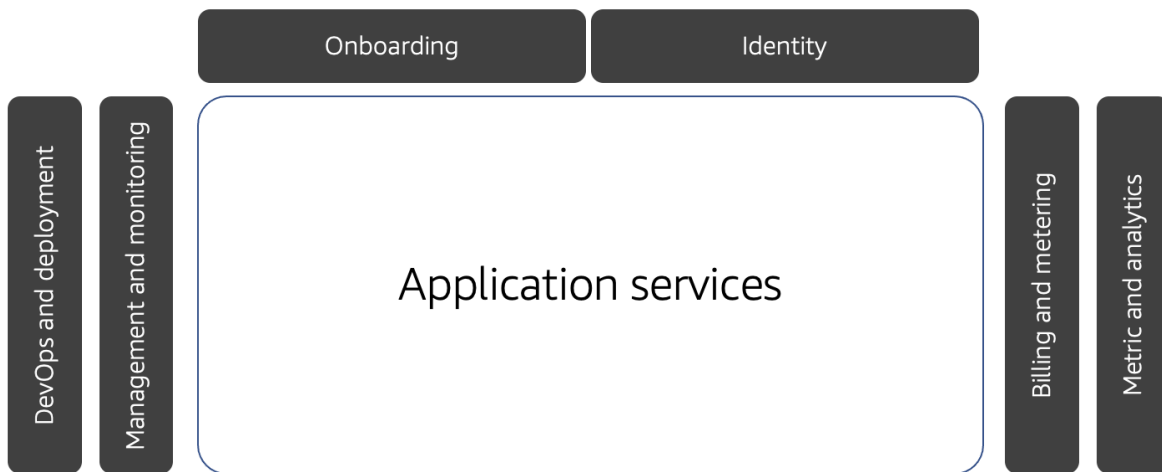


Figure 1-4. Surrounding your application with shared services

At the center of this diagram you'll see a placeholder for application services. This is where the various components of your SaaS application are deployed. Around these services, though, are a set of services that are needed to support the needs of our overall SaaS environment. At the top, I've highlighted the onboarding and identity services, which provide all the functionality to introduce a new tenant into your system. On the left, you'll see the placeholders for the common SaaS deployment and management functionality. And, on the right, you'll see fundamental concepts like billing, metering, metrics, and analytics.

Now, for many SaaS builders, it's tempting to view these additional services as secondary components that are needed by your application. In fact, I have seen teams that will defer the introduction of these services, putting all their initial energy into supporting tenancy in their application services.

In reality, while getting the application services right is certainly an important part of your SaaS model, the success of your SaaS business will be heavily influenced by the capabilities of these surrounding services. These services are at the core of enabling much of the operational efficiency, growth, innovation, and agility goals that are motivating companies to adopt a SaaS model. So, these components—which are common to *all* SaaS environments—must be put front and center when you are building your SaaS solution. This is why I have always encouraged SaaS teams to start their SaaS development at this outer edge, defining how they will automate the introduction of tenants, how they'll connect tenants to users, how they'll manage your tenant infrastructure, and a host of other considerations that we'll be covering throughout this book. It's these building blocks—which have nothing to do with the functionality of your application—that are going to have a significant influence on the SaaS footprint of your architecture, design, code, and business.

So, if we turn our attention back to the diagram in [Figure 1-4](#), we can see this big hole in the middle that represents where we'll ultimately

place our application services. The key takeaway here is that, no matter how we design and build what lands in that space, you'll still need some flavor of these core shared services as part of every SaaS architecture you build.

Re-Defining Multi-Tenancy

Up to this point, I've avoided introducing the idea of multi-tenancy. It's a word that is used heavily in the SaaS space and will appear all throughout the remainder of this book. However, it's a term that we have to wander into gracefully. The idea of multi-tenancy comes with lots of attached baggage and, before sorting it out, I wanted to create some foundation for the fundamentals that have driven companies toward the adoption of the SaaS delivery model. The other part of the challenge here is that the notion of multi-tenancy—as we'll define it in this book—will move beyond some of the traditional definitions that are typically attached to this term.

For years, in many circles, the term multi-tenant was used to convey the idea that some resource was being shared by multiple tenants. This could apply in many contexts. We could say that some piece of cloud infrastructure, for example, could be deemed multi-tenant because it was allowing tenants to share some resource under the hood. In reality, many services running in the cloud may be running in

a multi-tenant model to achieve their economies of scale. As a cloud consumer, this may be happening entirely outside of your view. Even outside the cloud, teams could build solutions where compute, databases, and other resources could be shared amongst customers. This created a very tight connection between multi-tenancy and the idea of a shared resource. In fact, in this context, this is a perfectly valid notion of multi-tenancy.

Now, as we start thinking about SaaS environments, it's entirely natural for us to bring the mapping of multi-tenancy with us. After all, SaaS environments do share infrastructure and that sharing of infrastructure is certainly valid to label as being multi-tenant.

To better illustrate this point, let's look at a sample SaaS model that brings together the concepts that we've been discussing in this chapter. The image in [Figure 1-5](#) provides a view of a sample multi-tenant SaaS environment.

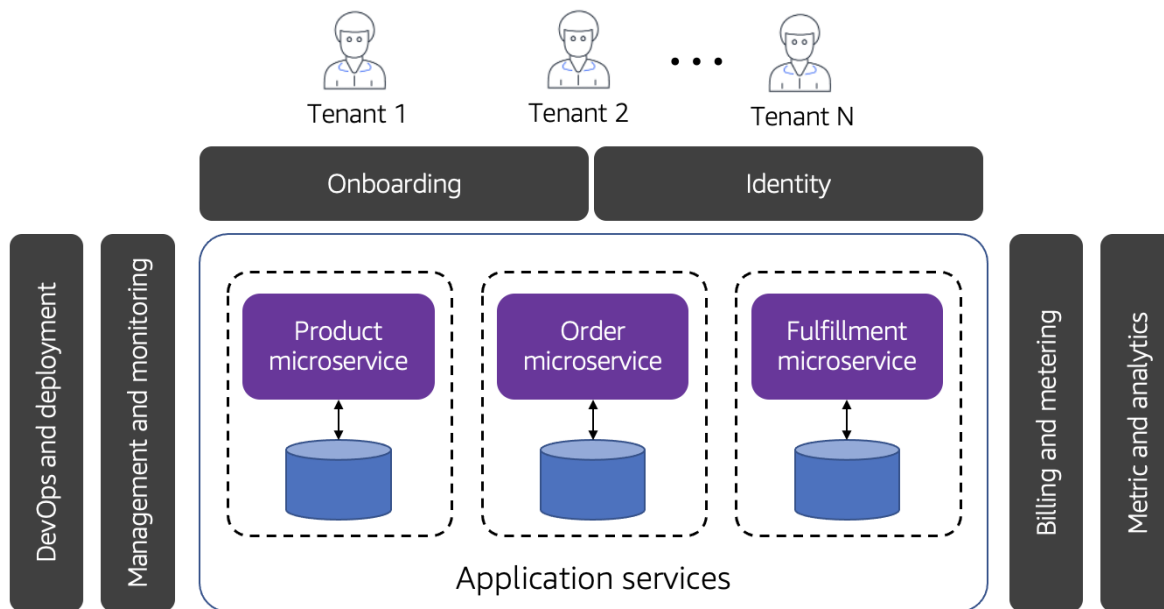


Figure 1-5. A sample multi-tenant environment

Here we have landed the shared infrastructure of our application services inside surrounding services that are used to introduce tenancy, manage, and operate our SaaS environment. Assuming that all of our tenants are sharing their infrastructure (compute, storage, and so on), then this would fit with the classic definition of multi-tenancy. And, to be fair, it would not be uncommon for SaaS providers to define and deliver their solution following this pattern.

The challenge here is that SaaS environments don't exclusively conform to this model. Suppose, for example, I create a SaaS environment that looks like the drawing in [Figure 1-6](#).

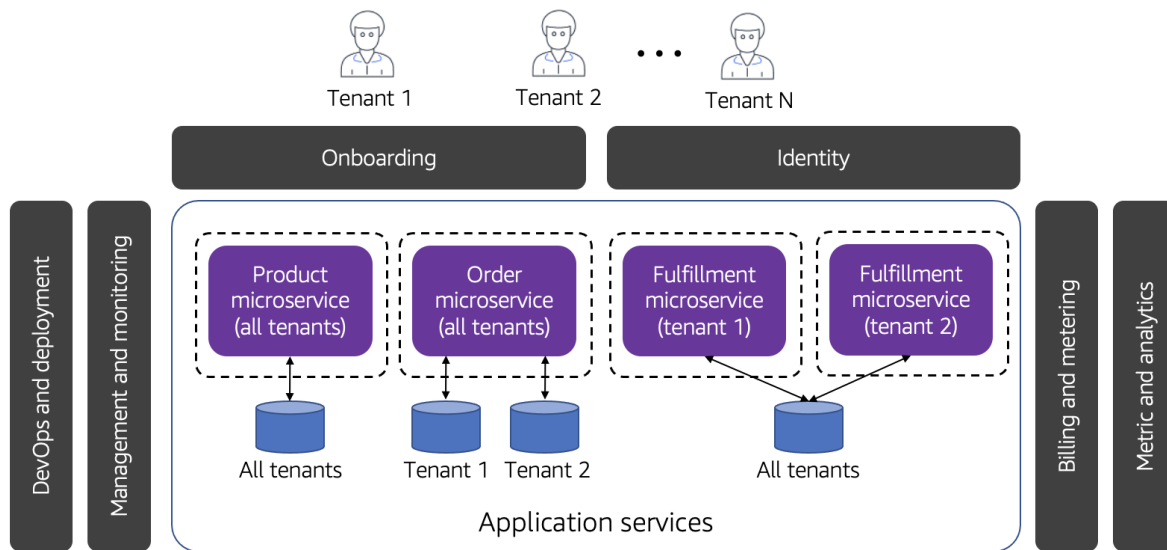


Figure 1-6. Multi-tenancy with shared and dedicated resources

Here you'll see that we've morphed the footprint of some of our application microservices. The Product microservice is unchanged. Its compute and storage infrastructure is still shared by all tenants. However, as we move to the Order microservice, you'll see that we've mixed things up a bit. Our domain, performance, and/or our security requirements may have required that we separate out the storage for each tenant. So, here the compute of our Order microservice is still shared, but we have separate databases for each tenant.

Finally, our Fulfillment microservice has also shifted. Here, our requirements pushed us toward a model where each tenant is running dedicated compute resources. In this case, though, the database is still shared by all tenants.

This architecture has certainly added a new wrinkle to our notion of multi-tenancy. If we're sticking to the purest definition of multi-tenancy, we wouldn't really be able to say everything running here conforms to the original definition of multi-tenancy. The storage of the Order service, for example, is not sharing any infrastructure between tenants. The compute of our Fulfillment microservices is also not shared here, but the database for this service is shared by all tenants.

Blurring these multi-tenant lines is common in the SaaS universe. When you're composing your SaaS environment, you're not sticking to any one absolute definition of multi-tenancy. You're picking the combinations of shared and dedicated resources that best align with the business and technical requirements of your system. This is all part of optimizing the footprint of your SaaS architecture around the needs of the business.

Even though the resources here are not shared by all tenants, the fundamentals of the SaaS principles we outlined earlier are still valid. For example, this environment would not change our application deployment approach. All tenants in this environment would still be running the same version of the product. Also, the environment is still being onboarded, operated, and managed by the same set of shared services we relied on in our prior example. This means that we're still extracting much of the operational efficiency and agility from this en-

vironment that would have been achieved in a fully shared infrastructure (with some caveats).

To drive this point home, let's look at a more extreme example. Suppose we have a SaaS architecture that resembles the model shown in [Figure 1-7](#). In this example, the domain, market, and/or legacy requirements have required us to have all compute and storage running in a dedicated model where each tenant has a completely separate set of infrastructure resources.

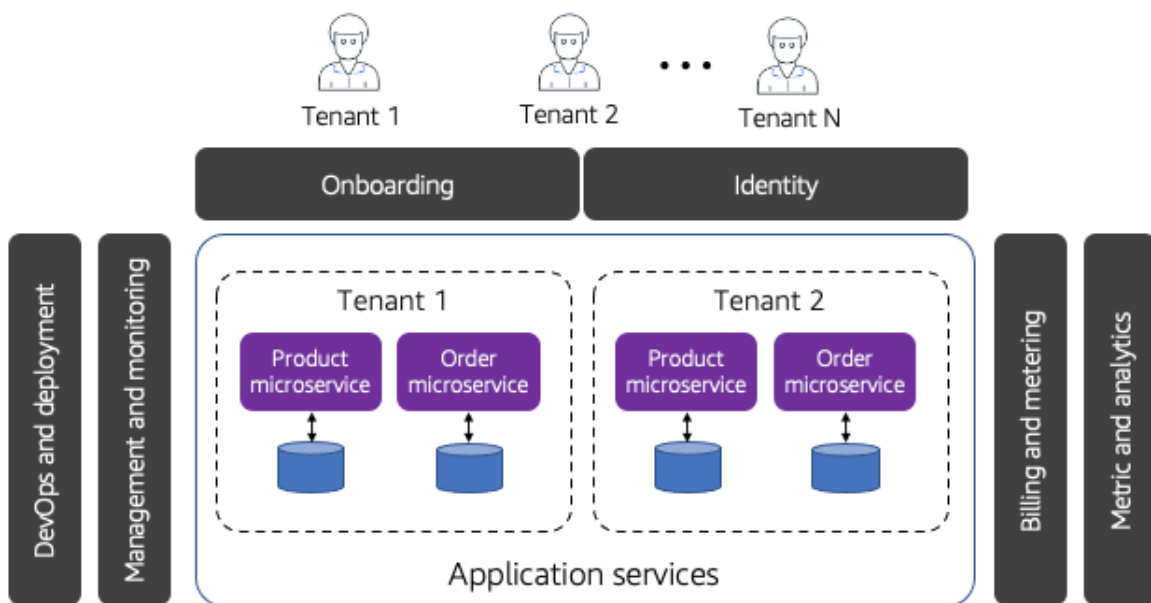


Figure 1-7. A multi-tenant environment with fully dedicated resources

While our tenants aren't sharing infrastructure in this model, you'll see that they continue to be onboarded, managed, and operated through the same set of shared services that have spanned all of the

examples we've outlined here. That means that all tenants are still running the same version of the software and they are still being managed and operated collectively.

This may seem like an unlikely scenario. However, in the wild, SaaS providers may have any number of different factors that might require them to operate in this model. Migrating SaaS providers often employ this model as a first stepping stone to SaaS. Other industries may have such extreme isolation requirements that they're not allowed to share infrastructure. There's a long list of factors that could legitimately land a SaaS provider in this model.

So, given this backdrop, it seems fair to ask ourselves how we want to define multi-tenancy in the context of a SaaS environment. Using the literal shared infrastructure definition of multi-tenancy doesn't seem to map well to the various models that can be used to deploy tenant infrastructure. Instead, these variations in SaaS models seem to demand that we evolve our definition of what it means to be multi-tenant.

For the scope of this book, at least, the term multi-tenant will definitely be extended to accommodate the realities I've outlined here. As we move forward, multi-tenant will refer to any environment that on-boards, deploys, manages, and operates tenants through a single

pane of glass. The sharedness of any infrastructure will have no correlation to the term multi-tenancy.

In the ensuing chapters, we'll introduce new terminology that will help us overcome some of the ambiguity that is attached to multi-tenancy.

AVOIDING THE SINGLE-TENANT TERM

Generally, whenever we refer to something as multi-tenant, there's a natural tendency to assume there must be some corresponding notion of what it means to be single-tenant. The idea of single tenancy seems to get mapped to those environments where no infrastructure is shared by tenants.

While I follow the logic of this approach, this term doesn't really seem to fit anywhere in the model of SaaS that I have outlined here. If you look back to [Figure 1-7](#) where our solution had no shared infrastructure, I also noted that we would still label this a multi-tenant environment since all tenants were still running the same version and being managed/operated collectively. Labeling this a single-tenant would undermine the idea that we aren't somehow realizing the benefits of the SaaS model.

With this in mind, you'll find that the term single-tenant will not be used at any point beyond this chapter. Every design and architecture we discuss will still be deemed a multi-tenant architecture. Instead, we'll attach new terms to describe the various deployment models that will still allow us to convey how/if infrastructure is being shared within a given SaaS environment. The general goal here is to disconnect the concept of multi-tenancy from the sharing of infrastructure and use it as a broader term to characterize any environment that is built, deployed, managed, and operated in a SaaS model.

This is less about what SaaS is or is not and more about establishing a vocabulary that aligns better with the concepts we'll be exploring throughout this book.

Where are the boundaries of SaaS?

We've laid a foundation here for what it means to be SaaS, but there are lots of nuances that we haven't really talked about. For example, suppose your SaaS application requires portions of the system to be deployed in some external location. Or, imagine scenarios where your application has dependencies on other vendor's solutions.

Maybe you are using a third-party billing system or your data must reside in another environment. There are any number of different reasons why you may need to have parts of your overall SaaS environment hosted somewhere that may not be entirely under your control.

So, how would this more distributed footprint fit with the idea of having a single, unified experience for all of your tenants? Afterall, having full control over all the moving parts of your system certainly maximizes your ability to innovate and move quickly. At the same time, it's impractical to think that some SaaS providers won't face domain and technology realities that require them to support externally hosted components/tools/technologies.

This is where we don't want to be too extreme with our definition of SaaS. To me, the boundary is more around how these external dependencies are configured, managed, and operated. If their presence is entirely hidden from your tenants and they are still managed and operated through your centralized experience, this is still SaaS to me. It may introduce new complexities, but it doesn't change the spirit of the SaaS model we're trying to build.

Where this gets more interesting is when SaaS providers have reliance on external resources that are in the direct view of their tenants. If, for example, my SaaS solution stores data in some tenant-hosted database, that's where things get more dicey. Now, you may have a dependency on infrastructure that is not entirely under your control. Updating this database, changing its schema, managing its health—these get more complicated in this model. This is where we start to ask questions about whether this external resource is breaking the third-wall of SaaS, exposing tenants to infrastructure and creating expectations/dependencies that undermine the agility, operations, and innovation of your SaaS environment.

My general rule of thumb here (with some exceptions) is that we're providing a service experience. In a service model, our tenant's view is limited to the surface of our service. The tools, technologies, and resources that are used to bring that service to life should be entirely hidden from our tenants. In many respects, this is the hard barrier that

prevents our system from falling back into patterns that might lead to one-off dependencies and variations.

The Managed Service Provider Model

There's one last wrinkle that we need to address as we try to refine our view of what it means to be a multi-tenant SaaS environment.

Some organizations have opted into what's referred to as a Managed Service Provider (MSP) model. In some cases, they'll categorize MSP as a variant of SaaS. This certainly has created some confusion in the SaaS domain. To better understand the challenges here, let's start by looking at an MSP environment and see how/where it fits in this discussion. [Figure 1-8](#) provides a conceptual view of an MSP environment.

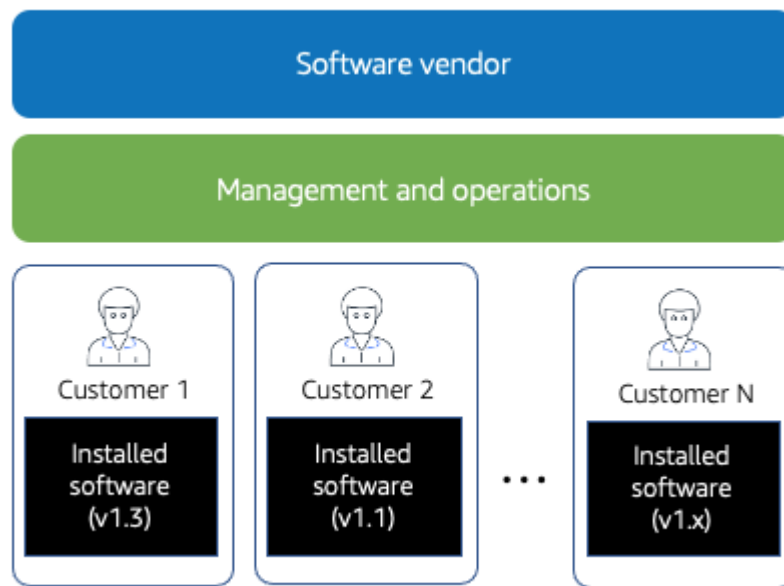


Figure 1-8. A Managed Service Provider (MSP) model

This model definitely resembles the classic installed software model that we outlined earlier. At the bottom of this diagram, you'll see a collection of customers that are running various versions of a software vendor's product. Each one of these customers will be running in its own infrastructure/environment.

With MSP, though, we'll try to get efficiencies and economies of scale out of moving the operations to a centralized team/entity. This is the service that these MSPs provide. They often own responsibility for installing, managing, and supporting each of these customers, attempting to extract some scale and efficiency out of tooling and mechanisms that they use to operate these customer environments.

I've also represented the software vendor at the top of the diagram. This is here to convey the idea that the software provider may have third-party relationships with one or more MSPs that are managing their customer environments.

You can see how some might equate the MSP model to SaaS. After all, it does seem to be trying to provide a unified managed and operations experience for all customers. However, if you look back at the principles that we used to describe SaaS, you can see where there are substantial gaps between MSP and SaaS. One of the biggest differences here is that customers are being allowed to run separate versions. So, while there may be some attempts to centralize management and operations, the MSP is going to have to have one-off variations in their operational experience to support the different footprints of each customer environment. This may require dedicated teams or, at a minimum, it will mean having teams that can deal with the complexities of supporting the unique needs of each customer. Again, MSP still adds lots of value here and certainly creates efficiencies, but it's definitely different than having a single pane of glass that gets its efficiencies from having customers run a single version of a product and, in many cases, getting efficiencies from sharing some or all of their infrastructure. At some level in the MSP model, you're likely to still inherit aspects of the pain that comes with one-off customer variations. MSPs can introduce some measures to offset some of the challenges here, but they'll still face the operational and agility com-

plexities that come with supporting unique, one-off needs of separate customer environments.

The other difference here relates more to how SaaS teams are structured and operated. Generally, in a SaaS organization, we're attempting to avoid drawing hard lines between operations teams and the rest of the organization. We want operations, architects, product owners, and the various roles of our team working closely together to continually evaluate and refine the service experience of their offering.

This typically means that these teams are tightly connected. They're equally invested in understanding how their tenants are consuming their systems, how their imposing load, how they're onboarding, and a host of other key insights. SaaS businesses want and need to have their fingers on the pulse of their systems. This is core to driving the success of the business and being connected more directly to the overall tenant experience. So, while this is a less concrete boundary, it still represents an important difference between SaaS and MSP.

Now, it's important to note that MSP is an entirely valid model. It often represents a good fit for some software providers. MSP can even be a stepping stone for some SaaS providers, providing access to some efficiencies while the team continues to push forward toward its SaaS delivery model. The key here is that we have a clear understanding of

the boundaries between SaaS and MSP and avoid viewing SaaS and MSP as somehow being synonymous.

At its Core, SaaS is a Business Model

By now you should have a better sense of how we characterize what it means to be SaaS. It should be clear that SaaS is very much about creating a technology, business, and operational culture that is focused squarely on driving a distinct set of business outcomes. So, while it's tempting to think about SaaS through the lens of technology patterns and strategies, you should really be viewing SaaS more as a business model.

To better understand this mindset, think about how adopting SaaS impacts the business of a SaaS provider. It directly influences and shapes how teams build, manage, operate, market, support, and sell their offerings. The principles of SaaS are ultimately woven into the culture of SaaS companies, blurring the line between the business and technology domains. With SaaS, the business strategy is focused on creating a service that can enable the business to react to current and emerging market needs without losing momentum or compromising growth.

Yes, features and functions are still important to SaaS companies. However, in a SaaS company, the features and functions are never introduced at the expense of agility and operational efficiency. When you're offering a multi-tenant SaaS solution, the needs of the many should always outweigh the needs of the few. Gone are the days of chasing one-off opportunities that require dedicated, one-off support at the expense of long-term success of the service.

This shift in mindset influences almost every role in a SaaS company. The role of a product owner, for example, changes significantly. Product owners must expand their view and consider operational attributes as part of constructing their backlog. Onboarding experience, time to value, agility—these are all examples of items that must be on the radar of the product owner. They must prioritize and value these operational attributes that are essential to creating a successful SaaS business. Architects, engineers, and QA members are equally influenced by this shift. They must now think more about how the solution they're designing, building, and testing will achieve the more dynamic needs of their service experience. How your SaaS offering is marketed, priced, sold, and supported also changes. This theme of new and overlapping responsibilities is common to most SaaS organizations.

So, the question is: what are the core principles that shape and guide the business model of SaaS companies? While there might be some debate about the answer to the question, there are some key themes

that seem to drive SaaS business strategies. The following outlines these key SaaS business objectives:

Agility

This term is often overloaded in the software domain. At the same time, in the SaaS universe, it is often viewed as one of the core pillars and motivating factors of the SaaS business. So many organizations that are moving to SaaS are doing so because they've become operationally crippled by their current model. Adopting SaaS is about moving to a culture and mindset that puts emphasis on speed and efficiency. Releasing new versions, reacting to market dynamics, targeting new customer segments, changing pricing models—these are amongst a long list of benefits that companies expect to extract from adopting a SaaS model. How your service is designed, how it's operated, and how it's sold are all shaped by a desire to maximize agility. A multi-tenant offering that reduced costs without realizing agility would certainly miss the broader value proposition of what it means to be a SaaS company.

Operational Efficiency

SaaS, in many respects, is about scale. In a multi-tenant environment, we're highly focused on continually growing our base of customers without requiring any specialized resources or

teams to support the addition of these new customers. With SaaS, you're essentially building an operational and technology footprint that can support continual and, ideally, rapid growth. Supporting this growth means investing in building an efficient operational footprint for your entire organization. I'll often ask SaaS companies what would happen if 1,000 new customers signed up for their service tomorrow. Some would welcome this and others cringe. This question often surfaces key questions about the operational efficiency of a SaaS company. It's important to note that operational efficiency is also about reacting and responding to customer needs. How quickly new features are released, how fast customers onboard, how quickly issues are addressed—these are all part of the operational efficiency story. Every part of the organization may play a part in building out an operationally efficient offering.

Innovation

With classic software models, teams can feel somewhat handcuffed by the realities of their environment. Customers may have one-off customizations, they may have a distributed operational model—there could be any number of factors that make it difficult for them to consider making any significant shifts in their approach. In a SaaS environment, where there's more emphasis on agility and putting customers in a unified environ-

ment, teams are freed up to consider exploring new, out-of-the-box ideas that could directly influence the growth and success of the business. In many respects, this represents the flywheel of SaaS. You invest in agility and operational efficiency and this promotes greater innovation. This is all part of the broader value proposition of the SaaS model.

Frictionless Onboarding

SaaS businesses must give careful consideration to how customers get introduced into their environments. If you are trying to remain as agile and operationally efficient as possible, you must also think about how customer onboarding can be streamlined. For some SaaS businesses, this will be achieved through a classic sign-up page where customers can complete the on-boarding process in an entirely self-service manner. In other environments, organizations may rely on an internal process that drives the onboarding process. The key here is that every SaaS business must be focused on creating an onboarding experience that removes friction and enables agility and operational efficiency. For some, this will be straightforward. For others, it may take more effort to re-think how the team builds, operates, and automates its onboarding experience.

Growth

Every organization is about growth. However, SaaS organizations typically have a different notion of growth. They are investing in a model and an organizational footprint that is built to thrive on growth. Imagine building this highly efficient car factory that optimized and automated every step in the construction process. Then, imagine only asking it to produce two cars a day. Sort of pointless. With SaaS, we're building out a business footprint that streamlines the entire process of acquiring, onboarding, supporting, and managing customers. A SaaS company makes this investment with the expectation that it will help support and fuel the growth machine that ultimately influences the margins and broader success of the business. So, when we talk about growth here, we're talking about achieving a level of acceleration that couldn't be achieved without the agility, operational efficiency, and innovation that 's part of SaaS. How much growth you're talking here is relative. For some, growth may be adding 100 new customers and for others it could mean adding 50,000 new customers. While this nature of your scale may vary, the goal of being growth-focused is equally essential to all SaaS businesses.

The items outlined here represent some of the core SaaS business principles. These are concepts that should be driven from the top down in a SaaS company where the leadership places clear emphasis on driving a business strategy that is focused on creating growth

through investment in these agility, operational efficiency, and growth goals.

Certainly, technology will end up playing a key role in this business strategy. The difference here is that SaaS is not a technology first mindset. A SaaS architect doesn't design a multi-tenant architecture first then figure out how the business strategy layers on top of that. Instead, the business and technology work together to find the best intersection of business goals and multi-tenant strategies that will realize those strategies.

As we get further into architecture details, you'll see this theme is laced into every dimension of our architecture. As we look at topics like tenant isolation, data partitioning, and identity, for example, you'll see how each of these areas are directly influenced by a range of business model considerations.

Building a Service—Not a Product

Many software providers would view themselves as being in the business of creating products. And, in many respects, this aligns well with their business model. The mindset here is focused on a pattern where we build something, the customer acquires it, and it's, for the most part, theirs to use. There are plenty of permutations and nu-

ances within this product-centric model, but they all gravitate toward a model that is focused on creating something more static and having customers buy it.

In this product-focused mindset, the emphasis is generally on defining the features and functions that will allow a software provider to close gaps and land new opportunities. Now, with SaaS, we shift from creating a product to creating a service. So, is this just terminology or does it have a meaningful impact on how we approach building a SaaS offering? It turns out, this is certainly more than a terminology shift.

When you offer software as a service, you think differently about what success looks like. Yes, your product needs to meet the functional needs of your customers. That dimension of the problem doesn't go away. As a service, though, you are much more focused on the broader customer experience across all dimensions of your business.

Let's look at an example that better highlights the differences between a service and a product. A restaurant provides a good backdrop for exploring these differences. When you go out to dinner, you're certainly looking forward to the food (the product in this example). However, the service is also a part of your experience. How fast you're greeted at the door, how soon the waiter comes to your table, how soon you get water, and how quickly your food arrives are all

measures of your service experience. No matter how good the food is, your quality of service will have a lot to do with your overall impression of the restaurant.

Now, think about this through the lens of a SaaS offering. Your SaaS tenants will have similar service expectations. How easily they can onboard your solution, how long it takes to realize value, how quickly new features are released, how easily they can provide feedback, how frequently the system is down—they are all dimensions of a service that must be front-and-center for SaaS teams. Having a great product won't matter if the overall experience for customers does not meet their expectations.

This takes on extra meaning when software is delivered in a SaaS model, where the tenant's only view of your system is the surface of your SaaS solution. SaaS tenants have no visibility into the underlying elements of your system. They don't think about patches, updates, and infrastructure configuration. They only care that the service is providing the experience that lets them maximize the value of your solution.

In this service model, we also often see SaaS companies leveraging their operational agility to drive greater customer loyalty. These SaaS providers will get into a mode where they release new capabilities, respond to feedback, and morph their systems at a rapid pace. See-

ing this constant and rapid innovation gives customers confidence that they will be benefactors of this constant evolution. In fact, this is often the tool that allows emerging SaaS companies to take business away from traditional non-SaaS market leaders. While some massive, established market leaders may have a much deeper feature set, their inability to rapidly react to market and customer needs can steer customers to more nimble SaaS-based offerings.

So, while this product vs. service concept may seem like I'm being a bit pedantic, to me it's an important distinction. It connects directly to this idea that SaaS is very much a mindset that shapes how entire SaaS organizations approach their jobs and their customers. In fact, many SaaS organizations will adopt a series of metrics that measure their ability to meet their service centric goals. It may be tempting to view this as something that can be bolted onto your service at some future date. However, many successful SaaS organizations rely on these metrics as a key pillar of their SaaS business.

THE B2B AND B2C SAAS STORY

The value of SaaS has a natural B2C mapping. Many of the highly visible examples of SaaS show up in the B2C model. The problem here is that some builders and organizations will presume that SaaS somehow only fits in the B2C space. I've seen technology teams and service providers suggest that their business-to-business (B2B) products can't or shouldn't be delivered in a SaaS model purely based on the fact that they are B2B.

For this book, I'll not be assuming anything about whether you're B2C or B2B. The practices, strategies, and patterns here are assumed to apply equally to these domains. Yes, there are areas where I might suggest that a given practice might work slightly differently for B2B or B2C. To me, these are mostly exceptions. The reality is, the goal and mindset of these domains is mostly universal. There are just nuances that might influence how these models will address the needs of their customers.

It is important, however, to note that the architecture patterns that are used in some B2C offerings are required to address unique scaling and availability challenges. If you're supporting millions of tenants and onboarding thousands of new tenants each day, the design and strategies you employ are going to be highly specialized. In fact, these B2C SaaS providers are often required to invent new breakthrough technologies that can address their massive scaling and op-

erational challenges. While there's lessons to be learned from these models, the exotic and inventive strategies that are used in these environments are often well outside the needs of the average SaaS builder. Our focus will be more on those B2C and B2B environments that are leveraging the scale and availability of existing cloud technologies to support the scaling and availability needs of their environment. Where it makes sense, I'll highlight areas where you may need to explore more targeted scaling models to support loads that might exceed the natural scaling boundaries of existing tooling, service, and so on.

Defining SaaS

I've devoted the bulk of this chapter to bringing more clarity to the boundaries, scope, and nature of what it means to be SaaS. It only seems fair to take all the information we discussed here and attempt to provide an explicit definition of SaaS that, ideally, incorporates the concepts and principles that we have covered here. Here's the definition I think best summarizes the view of SaaS I'll be using across the rest of this book:

SaaS is a business and software delivery model that enables organizations to offer their solutions in a low-friction, service-centric model

that maximizes value for customers and providers. It relies on agility and operational efficiency as pillars of a business strategy that promotes growth, reach, and innovation.

You'll see here that this definition sticks to the theme of SaaS being a business model. There's no mention here of any technologies or architecture considerations. It's your job as a SaaS architect and builder to create the underlying patterns and strategies that enable the business to realize its objectives. While that may seem like the job of any architect, it should be clear that the unique blend of business and technology demands for SaaS environments will be infused directly into the design, architecture, and implementation of your SaaS solution.

Conclusion

This chapter was all about establishing the foundational elements of the SaaS mindset, providing you with a clearer view of what I mean when I talk about multi-tenancy and the core terms I use to describe a SaaS model. It should also make it clear that your job as a SaaS architect and builder goes well beyond the technology domain. Before you can choose any architecture for your SaaS system, you'll need to have a firm grasp on the nature of key insights from your business to know which strategies and patterns are going to best align with the

realities of your business. This reality will become clearer as we get into the details of how you architect and design SaaS systems. The challenges and needs of SaaS architecture will require you to add all new dimensions to your toolbox. In some cases, you may actually need to be the evangelist of these concepts, driving the teams around you to think differently about how they approach their jobs.

While having a clear view of the SaaS mindset is essential, our goal in this book is to dig into the technical dimensions of SaaS. In the next chapter, we'll start to look at the various architectural constructs and concepts that are used when designing SaaS environments. Having a firm grip on these constructs will provide you with a foundation of terminology and concepts that will be essential as we move into more detailed architecture models and implementation strategies. It will also expose you to the range of fundamental considerations that are part of every SaaS architecture.

Chapter 2. Multi-tenant Architecture Fundamentals

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the second chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

As you progress through this book, you’ll realize that SaaS architecture comes in many shapes and sizes. There are countless permutations of architecture patterns and strategies that are composed to create the SaaS architecture that best aligns with the domain, compliance, and business realities of a SaaS company.

There are, however, some core themes that span all SaaS architectures. In this chapter, we’ll start our architecture journey by building a

foundation of architecture terminology and constructs that will be used as we explore specific architecture models. In many respects, the concepts we'll cover here will represent your playbook for building SaaS systems, providing you with a core set of topics and questions you'll need to address as part of any SaaS environment.

Getting a firm grasp on these core architecture principles is key to developing a solid understanding of the elements of SaaS environments, equipping you with a sense of the landscape of SaaS architecture challenges. The goal here is to bring clarity to these concepts and illustrate how multi-tenancy extends or alters your approach to your existing architecture strategies. You already likely have strong notions of what it means to scale, secure, design, and operate robust architecture. With SaaS, we have to look at how multi-tenancy influences and overlays these key concepts, often introducing new principles that introduce an entirely new set of constraints and considerations that will shape how you approach architecting a SaaS offering.

Adding Tenancy to your Architecture

Let's start our exploration of SaaS architecture concepts by looking at a traditional non-SaaS application. In classic applications, the environment is constructed from the ground up with the assumption that it will be installed and run by individual customers. Each customer es-

entially has its own dedicated footprint. [Figure 2-1](#) provides a conceptual view of how one of these applications might be designed and built.

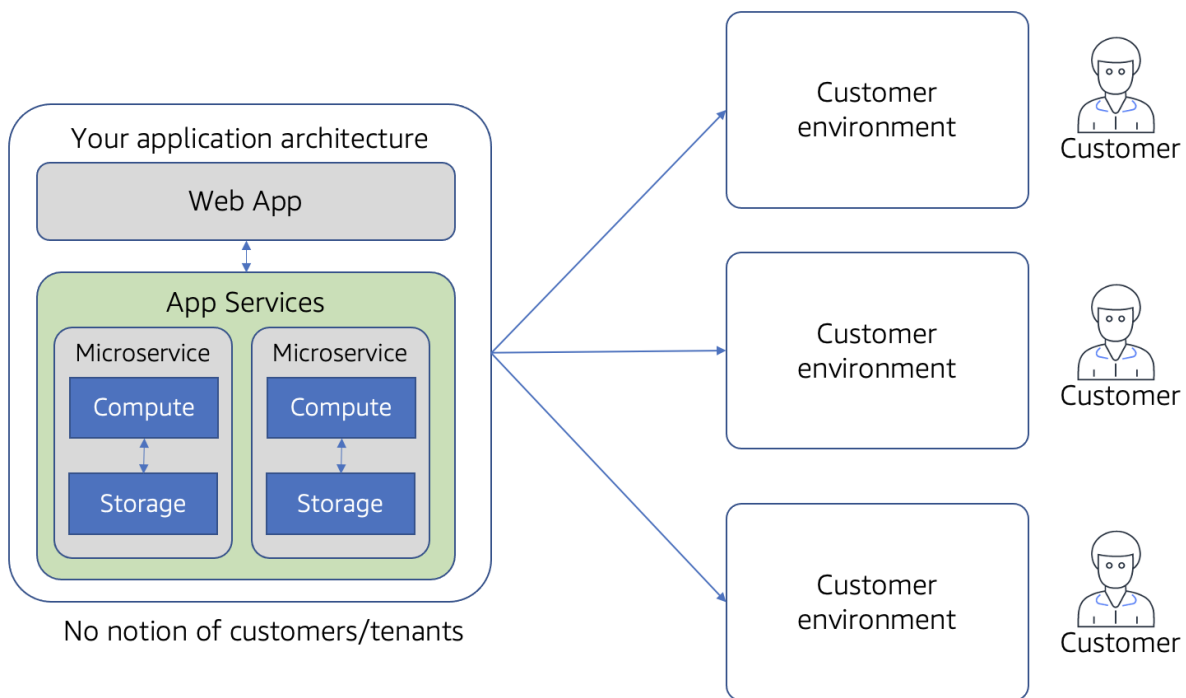


Figure 2-1. Traditional non-SaaS environments

On the left here, you'll see that we have a simplified view of a class application. Here, this application is built and then sold to individual customers. These customers might install the software in their own environment or it might run in the cloud. This approach simplifies the entire architectural model of this environment. The choices about how customers enter the environment, how they access our resources, and how they consume the services of our environment are much

simpler when we know that they will be running in an environment that is dedicated to each customer. The general mindset here is that you have a piece of software and you're just stamping out copies of it for each new customer.

Now, let's think about what it means to deliver this same application in a multi-tenant SaaS environment. The diagram in [Figure 2-2](#) provides a conceptual view of what this might look like. You see here that our customers, which are now tenants, are all consuming the same application.

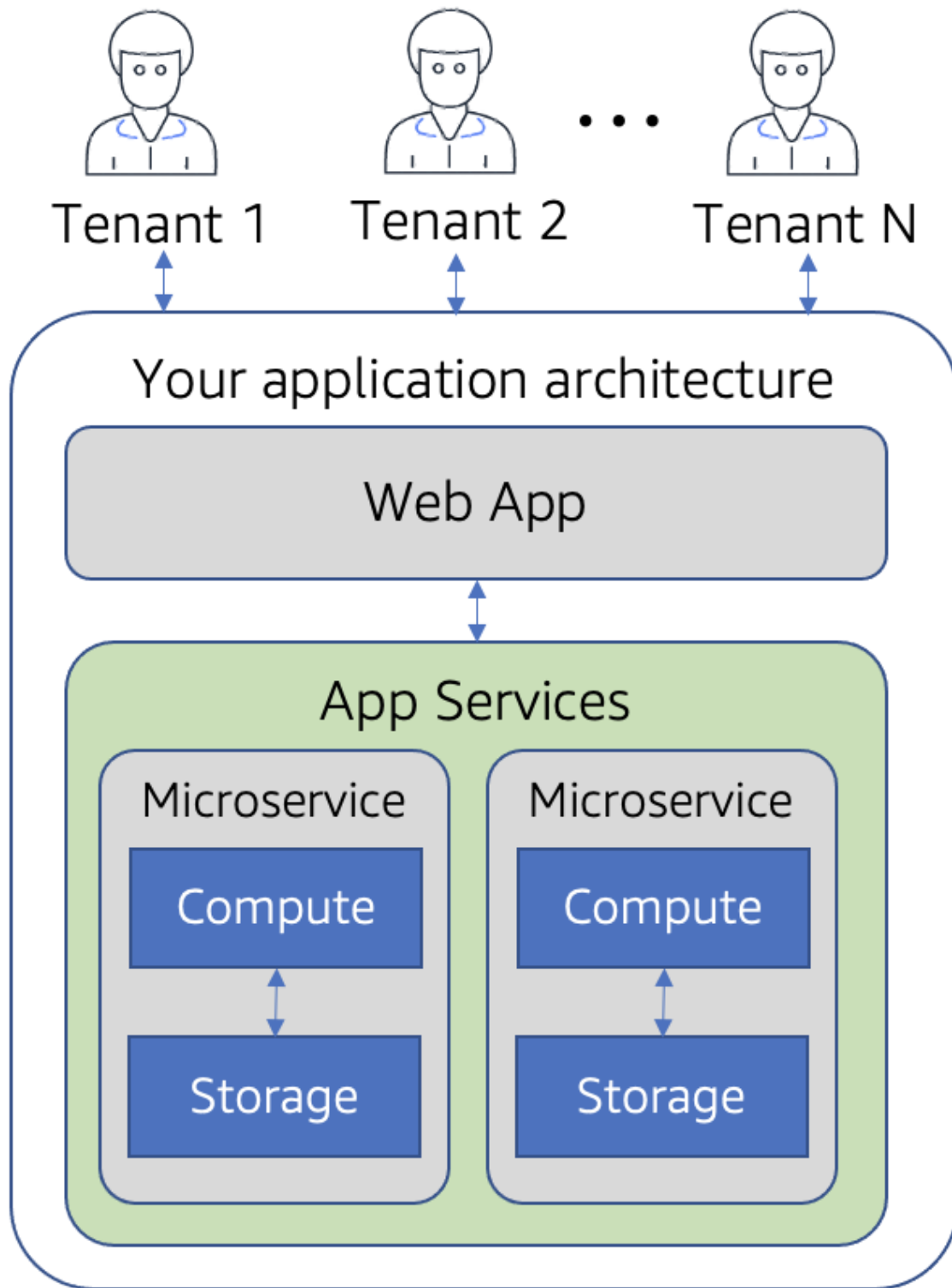


Figure 2-2. The shift to a tenant-centric experience

This shift may seem fairly simple in the diagram. However, it has a profound impact on how we design, build, secure, and manage this environment. We've essentially made the transition from a per customer dedicated model to a multi-tenant architecture. Supporting this model reaches into every dimension of the underlying implementation of your system. It affects how you implement authentication, routing, scaling, performance, storage, and, in targeted areas, how you code the application logic of your system.

As a SaaS architect and builder, it becomes your job to determine how your system will support this notion of tenancy. In some cases, the influence here may be introduced as an extension of existing patterns and strategies. In other cases, it may require you to build and design all new constructs that must be added to support the needs of a multi-tenant setting.

Part of the challenge here is that there's a wide range of parameters that will influence how this notion of tenancy ends up landing in your environment. The key, at this stage, is to understand the fundamental architecture concepts and then look at how the different dimensions of your environment might influence how these concepts are expressed in a given environment. So, for the moment at least, let's stay up a level from these environmental considerations and focus squarely on building an SaaS architecture foundation that will give us the terminology and taxonomy of constructs that we can use to char-

acterize the different moving parts and mechanisms that should be in the vocabulary of every SaaS architect and builder.

The Two Halves of Every SaaS Architecture

If we step back from the details of SaaS, we typically find that every SaaS environment—independent of its domain or design—can be broken down into two very distinct halves. In fact, across our entire discussion of SaaS across this book, you'll find that we'll use these two halves as the lens through which we'll look at how a multi-tenant system is built, deployed, and operated.

[Figure 2-3](#) provides a conceptual representation of the two halves of SaaS. On the right-hand side of the diagram, you'll see what is labeled as the control plane. The control plane is where we'll place all of the cross-cutting constructs, services, and capabilities that support the foundational needs of a multi-tenant SaaS environment.

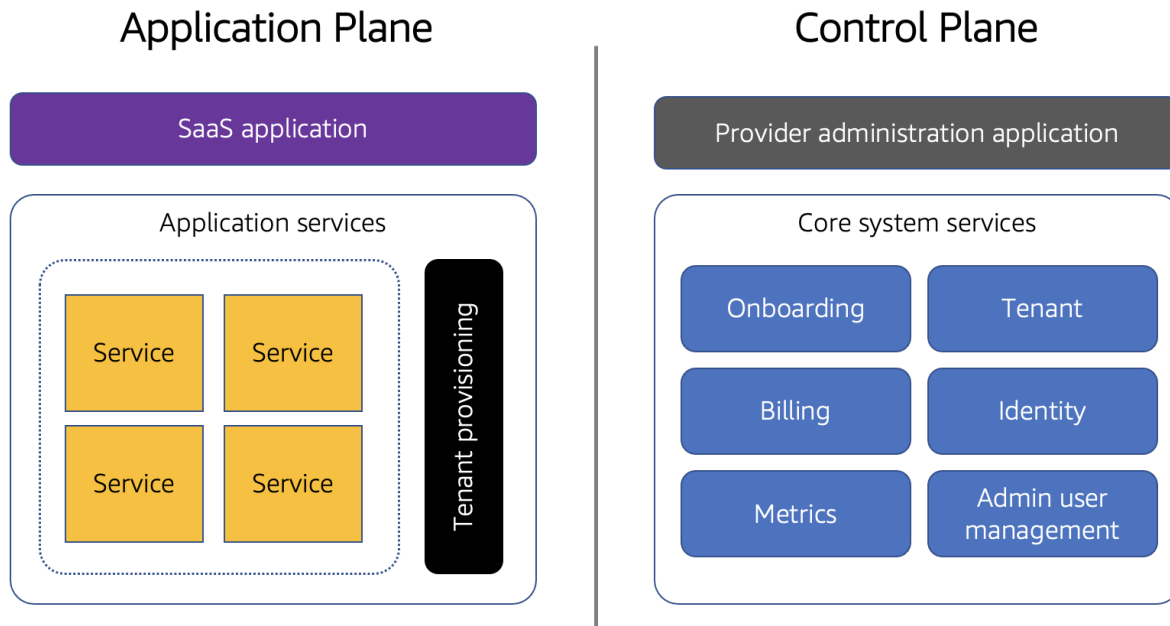


Figure 2-3. SaaS application and control planes

We often describe the control plane as the single pane of glass that is used to orchestrate and operate all the moving parts of your SaaS solution. It is at the core of enabling many of the principles that are essential to the success of your SaaS business. Concepts like tenant onboarding, billing, metrics, and a host of other services live in this control plane. You'll also see that our control plane includes an administration application. This represents the console or administration experience that is used by a SaaS provider to configure, manage, and operate their SaaS environment.

One interesting caveat here is that the services running in the control plane are not built or designed as multi-tenant services. If you think about it, there's actually nothing multi-tenant about the capabilities of

the control plane. It doesn't have functionality that supports the needs of individual tenants. Instead, it provides the services and functionality that spans all tenants.

While architects and builders are often tempted to start the SaaS discussion with the multi-tenant aspects of their application, the foundations of your SaaS architecture often start with the control plane. In many respects, the control plane provides a forcing function, requiring engineers to inject and support the nuances of tenancy from the outset of their development.

In contrast, the application plane is where the features and functionality of your SaaS service are brought to life. This is where we see the manifestation of all the multi-tenant principles that are classically associated with SaaS environments. It's here that we focus more of our attention on how multi-tenancy will shape the design, functionality, security, and performance of our service and its underlying resources. Our time and energy in the application plane is focused squarely on identifying and choosing the technologies, application services, and architecture patterns that best align with the parameters of your environment, timelines, and business. This is where you pour your energy into building out an application footprint that embraces agility and enables the business to support a range of personas and consumption models.

It's important to note that there is no single design, architecture, or blueprint for the application plane. I tend to view the application plane as a blank canvas that gets painted based on the unique composition of services and capabilities that my SaaS service requires. Yes, there are themes and patterns that we'll see that span SaaS application architectures. In fact, by the end of this book, you'll certainly have a healthy respect for the degree to which SaaS applications can vary from one solution to the next. Business, domain, and legacy realities are amongst a long list of factors that can influence the shape of your multi-tenant application strategy.

This view of the two halves of SaaS aligns with the mental model of multi-tenancy that we discussed in Chapter 1. Our application plane could share all tenant infrastructure or it could have completely dedicated infrastructure and it wouldn't matter. As long as we have a control plane that manages and operates these tenant environments through a unified experience, then we're considering this a multi-tenant environment.

This separation of concerns also influences our mental model for how the elements of our SaaS environment are updated and evolved. The services and capabilities of the control plane are versioned, updated, and deployed based on the needs of the SaaS provider, providing a range of services that reduce friction, enable centralized operations, and provide system-wide insights that are used to analyze the activi-

ty, scaling, and consumption patterns of all of your tenants. Meanwhile, our application plane is being driven more by the needs and experience of the system's tenants. Here, updates and deployments are introduced to provide new features, enhance tenant performance, support new tiering strategies, and so on.

Together, these two halves of SaaS represent the most fundamental building blocks of any SaaS environment. Understanding the roles of these planes will have a significant influence on how you'll approach the architecture, design, and decomposition of your SaaS offering.

Inside the Control Plane

Now that we have a better sense of the roles of the control and application planes, let's take a high-level pass at exploring the core concepts that commonly live within the scope of the control plane. We'll dig into each of these topics in much greater detail later in this book, exploring real-world implementation and architecture strategies. At this stage, though, we need to start a level up and develop an understanding of the different components that are part of any control plane you might build. Having a higher level grasp of these components, the roles they play, and how they are related will allow us to explore these building blocks of multi-tenancy without getting distracted by the different nuances that show up when we pivot to the specific influ-

ences of technologies, languages, and domain considerations. Having this foundational view will allow you to see the landscape of options and begin to see the different components that span all SaaS architecture models.

The following is a breakdown of the different services and capabilities that are likely to show up in the control plane of your SaaS architecture, which covers onboarding, identity, metrics, billing, and tenant management.

Onboarding

The control plane is responsible for managing and orchestrating all the steps needed to get a new tenant introduced into your SaaS environment. On the surface, this may seem like a simple concept. However, as you'll see in Chapter 4, there are lots of moving parts to the onboarding experience. The choices you make here, in many respects, are at the core of enabling many of the multi-tenant business and design elements of your SaaS environment.

At this stage, let's stick with a high-level view of the key elements of the onboarding experience. In [Figure 2-4](#) you'll see a conceptualized representation of the components that play a role in the onboarding experience. Here we show a tenant signing up for our SaaS service and triggering the onboarding process via the control plane. After this

initial request, the control plane owns the rest of the onboarding flow, creating and configuring our tenant and its corresponding identity footprint. This includes assigning a unique identifier to our tenant that will be leveraged across most of the moving parts of our multi-tenant architecture.

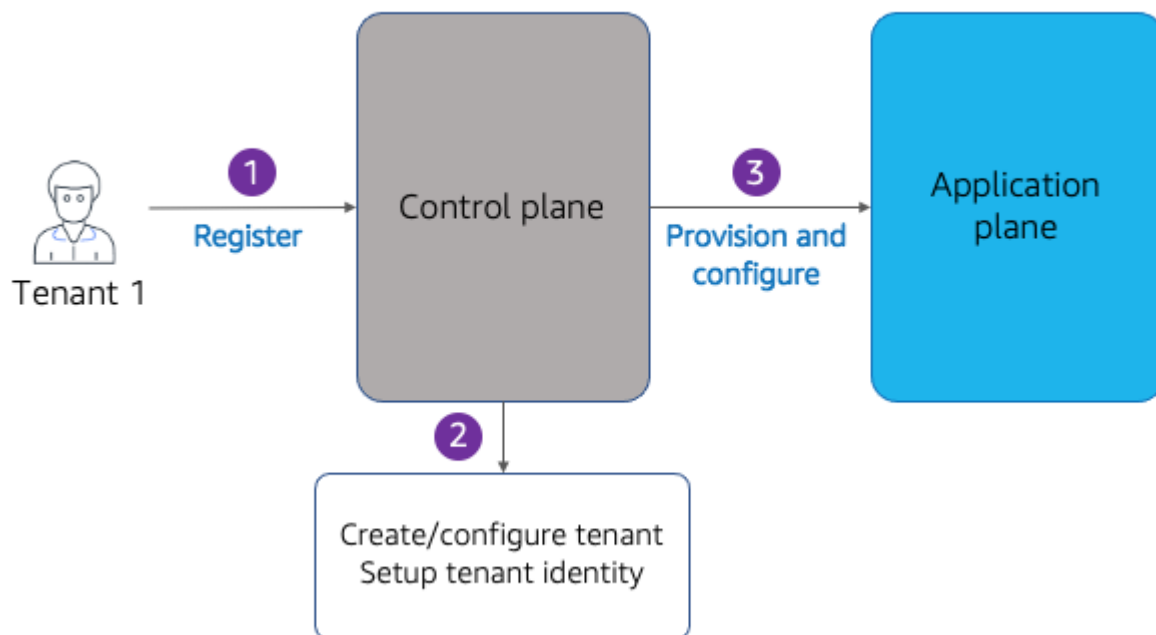


Figure 2-4. Onboarding tenants

You'll also notice that we show the control plane interacting with the application plane, provisioning and configuring any application specific resources that may be needed for each tenant. When we get into more detailed views of sample onboarding principles, we'll see how this part of the onboarding experience can get quite involved.

While there are common themes in the onboarding experience, the actual implementation of onboarding can vary significantly based on the domain you're in, the business goals of your solution, and the footprint of your application architecture. The key here, though, is that onboarding represents a foundational concept that sits at the front door of your SaaS experience. Business teams can and should take great interest in shaping and influencing how you approach building out this aspect of your system.

The higher level takeaway here is that onboarding is at the center of creating and connecting the most basic elements of a multi-tenant environment: tenants, users, identity, and tenant application resources. Onboarding weaves these concepts together and establishes the foundation for introducing tenancy to all the moving parts of your SaaS environment.

Identity

At first glance, you might wonder why identity belongs in the SaaS story. It's true that there are any number of different identity solutions that you can use to construct your SaaS solution. You could even suggest that your identity provider belongs somehow outside the scope of our control plane discussion. However, it turns out that multi-tenancy and the control plane often have a pretty tight binding to your

SaaS architecture. The diagram in [Figure 2-5](#) provides a simplified view of how identity is applied in multi-tenant environments.

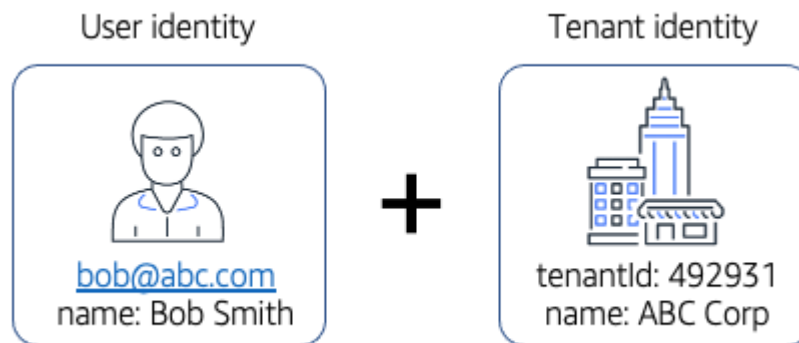


Figure 2-5. Binding users to a tenant identity

On the left you'll see the classic notion of user identity that is typically associated with authentication and authorization. It's true that our SaaS user will authenticate against our SaaS system. However, in a multi-tenant environment, being able to authenticate a user is not enough. A SaaS system must know who you are as a user *and* it must also be able to bind that user to a tenant. In fact, every user that is logged into our system must be attached in some way to a tenant.

This user/tenant binding ends up adding a wrinkle to our system's overall identity experience, requiring architects and builders to develop strategies for binding these two concepts in a way that still conforms to the requirements of your overall authentication model. This gets even more complicated when we start thinking about how we might support federated identity models in multi-tenant environments.

We'll see that, the more the identity experience moves outside of our control, the more complex and challenging it becomes to support this binding between users and tenants. In some cases, you may find yourself introducing constructs to stitch these two concepts together.

When we dig into onboarding and identity in Chapter 4, you'll get a better sense of the key role identity plays in the broader multi-tenant story. Getting identity right is essential to building out a crisp and efficient strategy for introducing tenants into your SaaS architecture. The policies and patterns you apply here will have a cascading impact across many of the moving parts of your design and implementation.

Metrics

When your application is running in a multi-tenant model, it becomes more difficult to create a clear picture of how your tenants are using your system. If you're sharing infrastructure, for example, it's very hard to know which tenants are currently consuming that infrastructure and how the activity of individual tenants might be impacting the scale, performance, and availability of your solution. The population of tenants that are using your system may also be constantly changing. New tenants may be added. Existing tenants might be leaving. This can make operating and supporting multi-tenant environments particularly challenging.

These factors make it especially important for SaaS companies to invest in building out a rich metrics and analytics experience as part of their control plane. The goal here is to create a centralized hub for capturing and aggregating tenant activity that allows teams to monitor and analyze the usage and consumption profile of individual tenants.

The role of metrics here is very wide. The data collected will be used in an operational context, allowing teams to measure and troubleshoot the health of the system. Product owners might use this data to assess the consumption of specific features. Customer success teams might use this data to measure a new customer's time to value. The idea here is that successful SaaS teams will use this data to drive the business, operational, and technology success of their SaaS offering.

You can imagine how metrics will impact the architecture and implementation of many of the moving parts of your multi-tenant system. Microservice developers will need to think about how and where they'll add metrics instrumentation. Infrastructure teams will need to decide how and where they'll surface infrastructure activity. The business will need to weigh in and help capture the metrics that can measure the customer experience. These are just a few examples from a long list of areas where metrics might influence your implementation.

The tenant must be at the center of this metrics strategy. Having data on consumption and activity has significantly less value if it cannot be filtered, analyzed, and viewed through the lens of individual tenants.

Billing

Most SaaS systems have some dependency on a billing system. This could be a home grown billing system or it could be any one of the commercial SaaS billing systems that are available from different billing providers. Regardless of the approach, you can see how billing is a core concept that has a natural home within the control plane.

Billing has a couple touch points within the control plane. It's typically connected to the onboarding experience where each new tenant must be created as a "customer" within your billing system. This might include configuring the tenant's billing plan and setting up other attributes of the tenant's billing profile.

Many SaaS solutions have billing strategies that meter and measure tenant activity as part of generating a bill. This could be bandwidth consumption, number of requests, storage consumption, or any other activity-related events that are associated with a given tenant. In these models, the control plane and your billing system must provide a way for this activity data to be ingested, processed, and submitted to your billing system. This could be a direct integration with the

billing system or you could introduce your own services that process this data and send it to the billing system.

We'll get more into the details of billing integration in Chapter 16. The key here is to realize that billing will likely be part of your control plane services and that you'll likely be introducing dedicated services to orchestrate this integration.

Tenant Management

Every tenant in our SaaS system needs to be centrally managed and configured. In our control plane, this is represented by our Tenant Management service. Typically, this is a pretty basic service that provides all the operations needed to create and manage the state of tenants. This includes tracking key attributes that associate tenants with a unique identifier, billing plans, security policies, identity configuration, and an active/inactive status.

In some cases, teams may overlook this service or combine it with other concepts (identity, for example). It's important for multi-tenant environments to have a centralized service that manages all of this tenant state. This provides a single point of tenant configuration and allows tenants to easily manage them through a single experience.

We'll explore the elements and permutations of implementing tenant management more in Chapter 5.

Inside the Application Plane

Now that we have a better sense of the core concepts with the control plane, let's start looking at the common areas where multi-tenancy shows up in the application plane. While the control plane typically has a consistent set of common services, the application plane is a bit more abstract. How and where multi-tenancy is applied within the application plane can vary significantly based on a wide range of factors. That being said, there are still a range of themes that will surface, albeit in different forms, within your application plane. So, even though there is variation here, every SaaS architect will need to consider how/where they will introduce these themes into the application plane of their solution.

As you dig into the application pane, you'll find that your technology stack and deployment footprint will have a significant influence on how these concepts are applied. In some cases, there may be ready-made solutions that fit your use case precisely. In other cases, you may find yourself inventing solutions to fill gaps in your technology stack. While building out something to fill these gaps may add complexity and overhead to the build of your solution, in most cases you'll

want to take on this added work to ensure that your SaaS solution is not compromising on important elements of your multi-tenant architecture.

In subsequent chapters we'll look at real-world working examples that provide a more concrete view of how these constructs are realized within your application plane. For now, though, let's come up a level and establish a core set of application plane principles that should span every SaaS architecture.

Tenant Context

One of the most fundamental concepts in our application plane is the notion of tenant context. Tenant context does not map to any one specific strategy or mechanism. Instead, it's a broader concept that is meant to convey the idea that our application plane is always functioning in the context of specific tenants. This context is often represented as a token or some other construct that packages all the attributes of your tenant. A common example that you'll use here is a JSON Web Token (JWT) which combines your user and tenant information in one construct that is shared across all the moving parts of your multi-tenant architecture. This JWT becomes our passport for sharing tenant information (context) with any service or code that relies on this context. It's this token that is referred to as your tenant context.

Now, you'll see that this tenant context has a direct influence on how your application architecture processes tenant requests. This may affect routing, logging, metrics, data access, and a host of other constructs live within the application plane. [Figure 2-6](#) provides a conceptual view of tenant context in action.

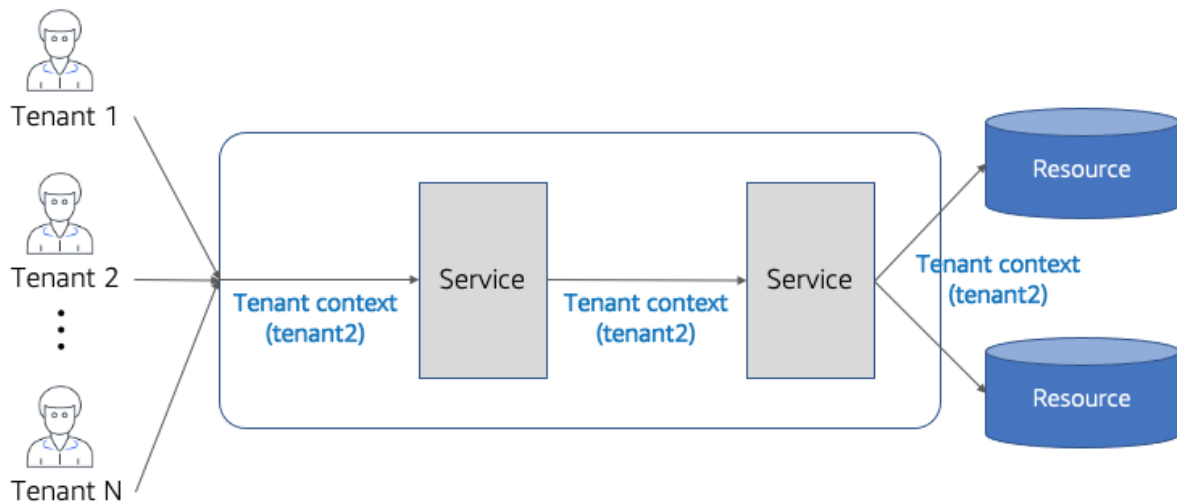


Figure 2-6. Applying tenant context

The flow in [Figure 2-6](#) shows tenant context being applied across the different services and resources that are part of a multi-tenant environment. This starts on the left-hand side of the diagram where my tenants authenticate against the identity that was created during onboarding and acquire their tenant context. This context is then injected into a service of my application. This same context flows into each downstream interaction of my system, enabling you to acquire and apply that context across a range of different use cases.

This represents one of the most fundamental differences of a SaaS environment. Our services don't just work with users—they must incorporate tenant context as part of the implementation of all the moving parts of our SaaS application. Every microservice you write will use this tenant context. It will become your job to figure out how to apply this context effectively without adding too much complexity to the implementation of your system. This, in fact, is a key theme that we'll address when we dig into SaaS microservice in Chapter 8.

As a SaaS architect, this means that you must be always thinking about how tenant context will be conveyed across your system. You'll also have to be thinking about the specific technology strategies that will be used to package and apply this tenant context in ways that limit complexity and promote agility. This is a continual balancing act for SaaS architects and builders.

Tenant Isolation

Multi-tenancy, by its very nature, focuses squarely on placing our customers and their resources into environments where resources may be shared or at least reside side-by-side in common infrastructure environments. This reality means that multi-tenant solutions are often required to apply and implement creative measures to ensure that tenant resources are protected against any potential cross-tenant access.

To better understand the fundamentals of this concept, let's look at a simple conceptual view of a solution running in our application plane (shown in [Figure 2-7](#)).

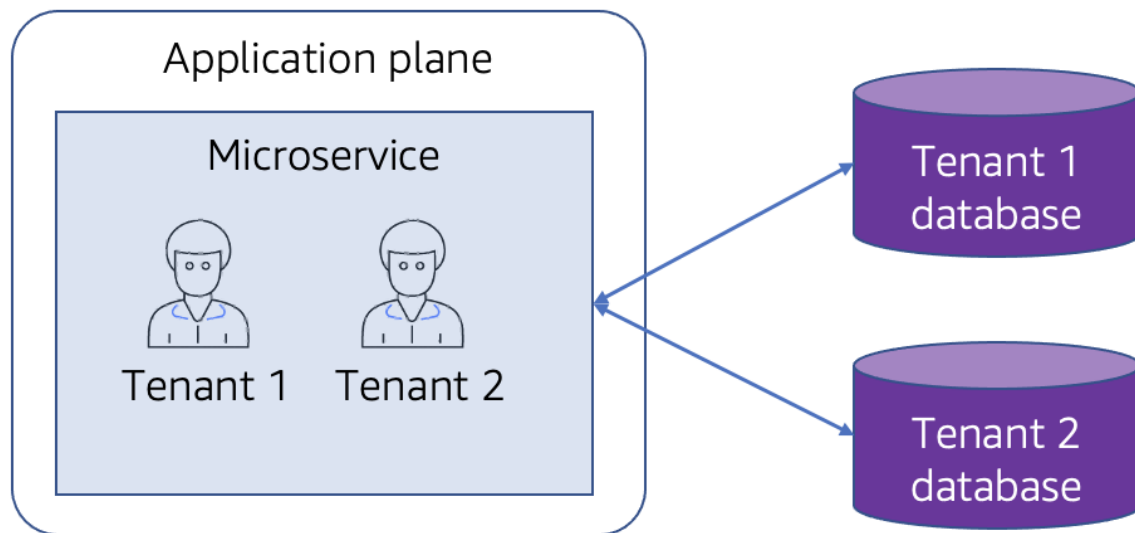


Figure 2-7. Implementing tenant isolation

Here you'll see we have the simplest of application planes running a single microservice. For this example, our SaaS solution has chosen to create separate databases for each tenant. At the same time, our microservice is sharing its compute with all tenants. This means that our microservice can be processing requests from tenants 1 and 2 simultaneously.

While the data for our tenants are stored in separate databases, there is nothing in our solution that ensures that tenant 1 can't access the

database of tenant 2. In fact, this would be the case even if our tenants weren't running in separate, dedicated microservices.

To prevent any access to another tenant's resources, our application plane must introduce a construct to prevent this cross-tenant access. The mechanisms to implement this will vary wildly based on a number of different considerations. However, the basic concept—which is labeled Tenant Isolation—spans all possible solutions. The idea here is that every application plane must introduce targeted constructs that strictly enforce the isolation of individual tenant resources—even when they may be running in a shared construct.

We'll dig into this concept in great detail in Chapter 10. It goes without saying that tenant isolation represents one of the most fundamental building blocks of SaaS architecture. As you build out your application plane you'll need to find the flavor and approach that allows you to enforce isolation at the various levels of your SaaS architecture.

Data Partitioning

The services and capabilities within our application plane often need to store data for tenants. Of course, how and where you choose to store that data can vary significantly based on the multi-tenant profile of your SaaS application. Any number of factors might influence your approach to storing data. The type of data, your compliance require-

ments, your usage patterns, the size of the data, the technology you're using—these are all pieces of the multi-tenant storage puzzle.

In the world of multi-tenant storage, we refer to the design of these different storage models as data partitioning. The key idea here is that you are picking a storage strategy that partitions tenant data based on the multi-tenant profile of that data. This could mean the data is stored in some dedicated construct or it could mean it lands in some shared construct. These partitioning strategies are influenced by a wide range of variables. The storage technology you're using (object, relational, nosql, etc.) obviously has a significant impact on the options you'll have for representing and storing tenant data. The business and use cases of your application can also influence the strategy you select. The list of variables and options here are extensive.

As a SaaS architect, it will be your job to look at the range of different data that's stored by your system and figure out which partitioning strategy best aligns with your needs. You'll also want to consider how/if these strategies might impact the agility of your solution. How data impacts the deployment of new features, the up-time of your solution, and the complexity of your operational footprint are all factors that require careful consideration when selecting a data partitioning strategy. It's also important to note that, when picking a strategy, this

is often a fine-grained decision. How you partition data can vary across the different services within your application plane.

This is a much deeper topic that we'll cover more extensively in Chapter 9. By the end of that chapter, you'll have a much better sense of what it means to bring a range of different strategies to life using a variety of different storage technologies.

Tenant Routing

In this simplest of SaaS architecture models, you may find that all tenants are sharing their resources. However, in most cases, your architecture is going to have variations where some or all of your tenant's infrastructure may be dedicated. In fact, it would not be uncommon to have microservices that are deployed on a per tenant basis.

The main point here is that SaaS application architectures are often required to support a distributed footprint that has any number of resources running in a combination of shared and dedicated models. The image in [Figure 2-8](#) provides a simplified sample of a SaaS architecture that supports a mix of shared and dedicated tenant resources.

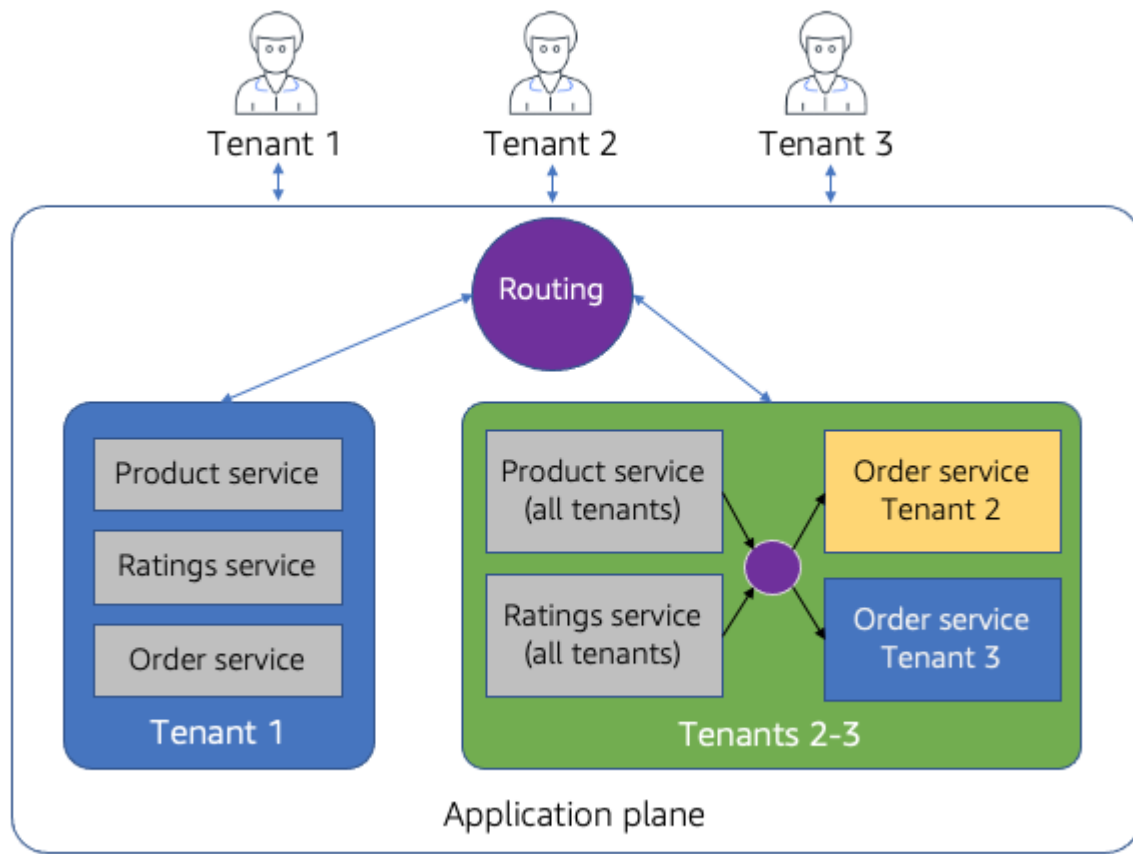


Figure 2-8. Routing on tenant context

In this example, we have three tenants that will be making requests to invoke operations on our application services. In this particular example, we have some resources that are shared and some that are dedicated. On the left, tenant 1 has an entirely dedicated set of services. Meanwhile, on the right-hand side, you'll see that we have the services that are being used by tenants 2 and 3. Here, note that we have the product and rating services that are being shared by both of these tenants. However, these tenants each have dedicated instances of the order service.

Now, as you step back and look at the overall configuration of these services, you can see where our multi-tenant architecture would need to include strategies and constructs that would correctly route tenant requests to the appropriate services. This happens at two levels within this example. If you start at the top of our application plane where our application plane is receiving requests from three separate tenants, you'll notice that there's a conceptual placeholder for a router here. This router must accept requests from all tenants and use the injected tenant context (that we discussed earlier) to determine how and where to route each request. Also, within the tenant 2/3 box on the right, you'll see that there is another placeholder for routing that will determine which instance of the order service will receive requests (based on tenant context).

Let's look at a couple concrete examples to sort this out. Suppose we get a request from tenant 1 to look up a product. When the router receives this request, it will examine the tenant context and route the traffic to the product service on the left (for tenant 1). Now, let's say we get a request from tenant 3 to update a product that must also update an order. In this scenario, the top-level router would send the request to the shared product service on the right (based on the tenant 2 context). Then, the product service would send a request to the order service via the service-to-service router. This router would look at the tenant context, resolve it to tenant 2, and send a request to the order service that's dedicated to tenant 2.

This example is meant to highlight the need for multi-tenant aware routing constructs that can handle the various deployments footprint we might have in a SaaS environment. Naturally, the technology and strategy that you apply here will vary based on a number of parameters. There are also a rich collection of routing tools and technologies, each of which might approach this differently. Often, this comes down to finding a tool that provides flexible and efficient ways to acquire and dynamically route traffic based on tenant context.

We'll see these routing constructs applied in specific solutions later in this book. At this stage, it's just important to understand that routing in multi-tenant environments often adds a new wrinkle to our infrastructure routing model.

Multi-Tenant Application Deployment

Deployment is a pretty well understood topic. Every application you build will require some DevOps technology and tooling that can deploy the initial version of your application and any subsequent updates. While these same concepts apply to the application plane of our multi-tenant environment, you'll also discover that different flavors of tenant application models will add new considerations to your application deployment model.

We've already noted here that tenants may have a mix of dedicated and shared resources. Some may have fully dedicated resources, some may have fully shared, and others may have some mix of dedicated and shared. Knowing this, we have to now consider how this will influence the DevOps implementation of our application deployment.

Imagine deploying an application that had two dedicated microservices and three shared microservices. In this model, our deployment automation code will have to have some visibility into the multi-tenant configuration of our SaaS application. It won't just deploy updated services like you would in a classic environment. It will need to consult the tenant deployment profile and determine which tenants might need a separate deployment of a microservice for each dedicated microservice. So, microservices within our application plane might be deployed multiple times. That, and our infrastructure automation code may need to apply tenant context to the configuration and security profile of each of these microservices.

Technically, this is not directly part of the application plane. However, it has a tight connection to the design and strategies we apply within the application plane. In general, you'll find that the application plane and the provisioning of tenant environments will end up being very inter-connected.

The Gray Area

While the control and application planes cover most of the fundamental multi-tenant architecture constructs, there are still some concepts that don't fit so cleanly into either of these planes. At the same time, these areas still belong in the discussion of foundational SaaS topics. While there are arguments that could be made for landing these in specific planes, to steer clear of the debate, I'm going to handle these few items separately and address the factors that might push them into one plane or the other.

Tiering

Tiering is a strategy most architects have encountered as part of consuming various third-party offerings. The basic idea here is that SaaS companies use tiers to create different variations of an offering with separate price points. As an example, a SaaS provider could offer their customers Basic, Advanced, and Premium tiers where each tier progressively adds additional value. Basic tier tenants might have constraints on performance, number of users, features, and so on. Premium tier tenants might have better SLAs, a higher number of users, and access to additional features.

The mistake some SaaS architects and builders make is that they assume that these tiers are mostly pricing and packaging strategies. In

reality, tiering can have a significant impact on many of the dimensions of your multi-tenant architecture. Tiering is enabled by building a more pliable SaaS architecture that offers the business more opportunities to create value boundaries that they may not have otherwise been able to offer.

Tiering naturally layers onto our discussion of tenant context, since the context that gets shared across our architecture often includes a reference to a given tenant's tier. This tier is applied across the architecture and can influence routing, security, and a host of other aspects of the underlying implementation of your system.

In some implementations of tiering, we'll see teams place this within their control plane as a first class concept. It's true that onboarding often includes some need to map a tenant's profile to a given tier. Tiers are also often correlated to a billing plan, which would seem natural to maintain within the scope of the control plane. At the same time, tiers are also used heavily within the application plane. They can be used to configure routing strategies or they could also be referenced as part of the configuration of throttling policies. The real answer here is that tiering has a home in both planes. However, I would probably lean toward placing it in the control plane since the tier can be managed and returned by interactions with the control plane (authentication, for example). The returned tier can be attached to the

tenant context and applied through that mechanism within the application plane.

Tenant, Tenant Admin, and System Admin Users

The term “user” can easily get overloaded when we’re talking about SaaS architecture. In a multi-tenant environment, we have multiple notions of what it means to be a user—each of which plays a distinct role. [Figure 2-9](#) provides a conceptual view of the different flavors of users that you will need to support in your multi-tenant solution.

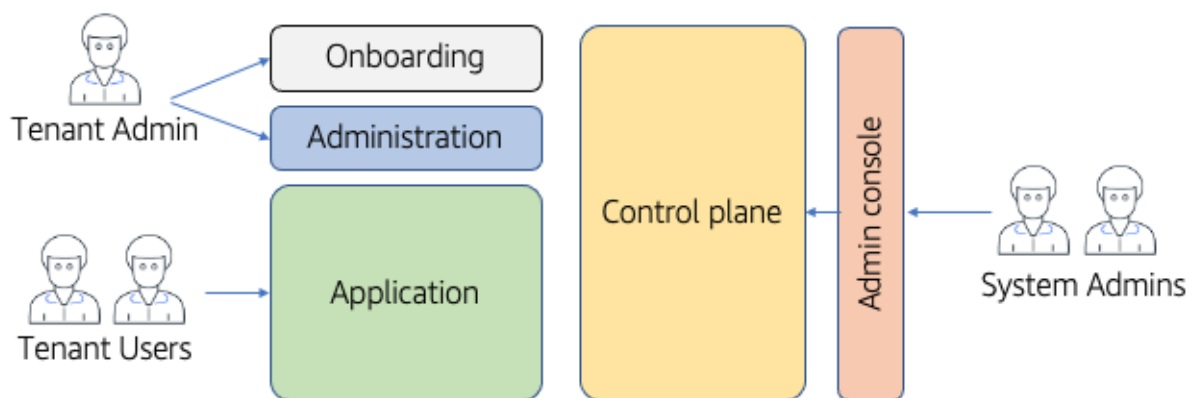


Figure 2-9. Multi-tenant user roles

On the left-hand side of the diagram, you’ll see that we have the typical tenant-related roles. There are two distinct types of roles here: tenant administrators and tenant users. A tenant administrator represents the initial user from your tenant that is onboarded to the system. This user is typically given admin privileges. This allows them to

access the unique application administration functionality that is used to configure, manage, and maintain application level constructs. This includes being able to create new tenant users. A tenant user represents users that are using the application without any administrative capabilities. These users may also be assigned different application-based roles that influence their application experience.

On the right-hand side of the diagram, you'll see that we also have system administrators. These users are connected to the SaaS provider and have access to the control plane of your environment to manage, operate, and analyze the health and activity of a SaaS environment. These admins may also have varying roles that are used to characterize their administrative privileges. Some may have full access, others may have limits on their ability to access or configure different views and settings.

You'll notice that I've also shown an administration console as part of the control plane. This represents an often overlooked part of the system admin role. It's here to highlight the need for a targeted SaaS administration console that is used to manage, configure, and operate your tenants. It is typically something your team needs to build to support the unique needs of your SaaS environment (separate from other tooling that might be used to manage the health of your system). Your system admin users will need an authentication experience to be able to access this SaaS admin console.

SaaS architects need to consider each of these roles when building out a multi-tenant environment. While the tenant roles are typically better understood, many teams invest less energy in the system admin roles. The process for introducing and managing the lifecycle of these users should be addressed as part of your overall design and implementation. You'll want to have a repeatable, secure mechanism for managing these users.

The control plane vs. application plane debate is particularly sticky when it comes to managing users. There's little doubt that the system admin users should be managed via the control plane. In fact, the initial diagram of the two planes shown at the outset of this chapter ([Figure 2-3](#)) actually includes an admin user management service as part of its control plane. It's when you start discussing the placement of tenant users that things can get more fuzzy. Some would argue that the application should own the tenant user management experience and, therefore, management of these users should happen within the scope of the application plane. At the same time, our tenant onboarding process needs to be able to create the identities for these users during the onboarding process, which suggests this should remain in the control plane. You can see how this can get circular in a hurry.

My general preference here, with caveats, is that identity belongs in the control plane—especially since this is where tenant context gets

connected to the user identities. This aspect of the identity would never be managed in the scope of the application plane.

A compromise can be had here by having the control plane manage the identity and authentication experience while still allowing the application to manage the non-identity attributes of the tenant outside of the identity experience. The other option here would be to have a tenant user management service in your control plane that supports any additional user management functionality that may be needed by your application.

Tenant Provisioning

So far, we've highlighted the role of onboarding within the control plane. We also looked at how the onboarding process may need to provision and configure application infrastructure as part of the onboarding experience. This raises an important question: should tenant provisioning live within the control plane or the application plane?

[Figure 2-10](#) provides a conceptual view of the two options. On the left, you'll see the model where tenant provisioning runs within the application plane. In this scenario, all the elements of onboarding (tenant creation, billing configuration, and identity setup) still happen within the scope of the control plane. The provisioning step is triggered and

orchestrated by the onboarding service, but runs within the application plane.

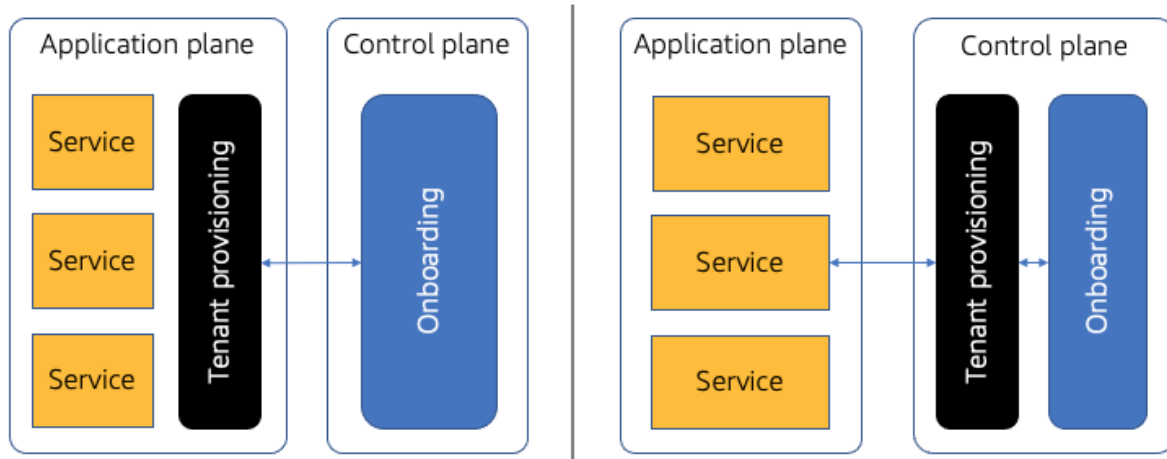


Figure 2-10. Placing the tenant provisioning process

The alternate approach is shown on the right side of this diagram. Here, tenant provisioning is executed from within the control plane. This means that the tenant provisioning would execute infrastructure configuration scripts that are applied within the application plane. This puts all the moving parts of onboarding within the control plane.

The tradeoffs, to me, center around the encapsulation and abstraction of the application plane. If you believe the structure and footprint of application infrastructure should be unknown to the control plane, then you'll favor the model on the left. If you feel strongly that onboarding is already owned by the control plane, you could argue that it's natural for it to also own the application provisioning process.

My bias here leans toward keeping provisioning closest to the resources that are being described and configured. I'd prefer not to make updates to the control plane based on changes in the architecture of the application plane. The tradeoff here is that the control plane must support a more distributed onboarding experience and rely on messaging between the control and application planes to track the provisioning progress/status. Both models have their merits. The key here is that provisioning should be a standalone part of the onboarding experience. So, if at some point you choose to move it, it would be somewhat encapsulated and could move without significant re-think.

Integrating Control and Application Planes

Some organizations will create very specific boundaries between the control and application planes. This might be a network boundary or some other architectural construct that separates these two planes. This has advantages for some organizations in that it allows these planes to be configured, managed, and operated based on the unique needs of each plane. It also introduces opportunities to design more secure interactions between these planes.

With this in mind, we can then start to consider different approaches to integrating the control and application planes. The integration strategy you choose here will be heavily influenced by the nature of the interactions between the planes, the geographic footprint of your solution, and the security profile of your environment.

Some teams may opt for a more loosely-coupled model that is more event/message driven while others may require a more native integration that enables more direct control of the application plane resources. There are few absolutes here and there are a wide range of technologies that bring different possibilities to this discussion. The key here is to be thoughtful in picking an integration model that enables the level of control that fits the needs of your particular domain, application, and environment.

TIP

Much of the discussion here leans toward a model where the application and control planes are deployed and managed in separate infrastructure. While the merits of separating these planes is compelling, it's important to note that there is no rule that suggests that these planes must be divided along some hard boundary. There are valid scenarios where a SaaS provider may choose to deploy the control and application planes into a shared environment. The needs of your environment, the nature of your technology, and a range of other considerations will determine how/if you deploy these planes with more concrete architectural boundaries. The key here to ensure that you divide your system into these distinct planes—regardless of how/where they are deployed.

As we dig into the specifics of the control plane, we can look at the common touch points between these two planes and get into the specific integration use cases and their potential solutions. For now, though, just know that integration is a key piece of the overall control/application plane model.

Picking Technologies for Your Planes

SaaS teams pick the technologies for implementing their SaaS solutions based on any number of different variables. Skill sets, cloud providers, domain needs, legacy considerations—these are just a few of the many parameters that go into selecting a technology for your multi-tenant SaaS offering.

Now, as we look at SaaS through the lens of our control and application planes, it's also natural to think about how the needs of these two planes might influence your choice of technologies. If you choose an entirely container-based model for your application plane, should that mean your control plane must also be implemented with containers? The reality here is that the planes will support different needs and different consumption profiles. There is nothing that suggests that the technologies they use must somehow match.

Consider, for example, the cost and consumption profile of your control plane. Many of these services may be consumed on a more limited basis than services running in our application plane. We might favor choosing a different technology for our control plane that yields a more cost-efficient model. Some teams might choose to use serverless technologies to implement their control plane.

The decisions can also be much more granular. I might choose one technology for some types of services and different technologies for other services. The key here is that you should not assume that the profile, consumption, and performance profile of your control and application planes will be the same. As part of architecting your SaaS environment, you want to consider the technology needs of these two planes independently.

Avoiding the Absolutes

This discussion of SaaS architecture concepts devoted lots of attention to defining SaaS architecture through the lens of the control and application planes. The planes equip us with a natural way to think about the different components of a multi-tenant architecture and they give us a good mental model for thinking about how the different features of a multi-tenant architecture should land in your SaaS environment.

While these constructs are useful, I would also be careful about attaching absolutes to this model. Yes, it's a good way to think about SaaS and it provides us with a framework for talking about how we can approach building multi-tenant solutions. It's certainly provided me with a powerful construct for engaging teams that are trying to design and architect their SaaS systems. It has also put emphasis on the need for a set of shared services that are outside the scope of the multi-tenant architecture of your application.

The key here, though, is to use these concepts to shape how you approach your SaaS architecture, allowing for the fact that there may be nuances of your environment that may require variations in your approach. It's less about being absolute about what's in each plane and more about creating an architecture that creates a clear division of

responsibility and aligns with the security, management, and operational profile of your SaaS offering.

Conclusion

This chapter was all about building a foundation of SaaS architecture concepts. We looked at the core elements of SaaS architecture with the goal of framing multi-tenant architecture patterns and strategies without getting into the specifics of any particular technology or domain. The concepts we covered here should apply to any SaaS environment and provide any team with any technology a mental model for approaching SaaS architecture.

We've really only touched the surface of multi-tenant architecture here. As we move forward, we'll start mapping these concepts to concrete examples that tease out all the underlying details and add another layer of design considerations that we'll build on the mental model that we've created here. This added layer of detail will start to illustrate the web of possibilities you'll need to navigate as you consider how best to connect the needs of your SaaS business with the realities that come with realizing these principles with languages, technology stacks, and tools that bring their own set of variables to your multi-tenant equation.

Our next step is to start looking at SaaS deployment models. This will shift us from thinking about concepts to mapping these concepts to seeing those concepts landed in different patterns of deployment. The goal here is to start thinking about and bringing more clarity to the strategies that are used to support the various SaaS models that you'll need to consider as you shape your SaaS architecture.

Chapter 3. Multi-Tenant Deployment Models

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the third chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

Selecting your multi-tenant deployment model is one of the first things you’ll do as a SaaS architect. It’s here that you step back from the details of the multi-tenant implementation and ask yourself broader questions about the fundamental footprint of your SaaS environment. The choices you make around the deployment model of your application will have a profound impact on the cost, operations, tiering, and a host of other attributes that will have a direct impact on the success of your SaaS business.

In this chapter, I'll be walking through a range of different multi-tenant deployment models, exploring how the footprint of each of these models can be used to address a variety of different technology and business requirements. Along the way, I'll highlight the pros and cons of the various models and give you a good sense of how the model you select can shape the complexity, scalability, performance, and agility of your SaaS offering. Understanding these models and their core values/tradeoffs is essential to arriving at an architecture strategy that balances the realities of your business, customers, time pressures, and long-term SaaS objectives. While there are themes in these models that are common to many SaaS teams, there is no one blueprint that everyone will follow. Instead, it will be your job to navigate these deployment models, weigh the options, and select a model or combination of models that address your current and emerging needs.

We'll also use this chapter to continue to expand our SaaS vocabulary, attaching terminology to these models and their supporting constructs that will be referenced throughout the remainder of this book. These new terms will give you more precise ways to describe the nature of SaaS environments and enable you to be more crisp and granular about how you describe the moving parts of a multi-tenant architecture. These new terms and concepts allow us to describe and classify SaaS architectures in a way that better accommodates the broad range of multi-tenant permutations that we find in the wild.

What's a Deployment Model?

In Chapter 1, I spent a fair amount of time looking at how we needed to expand our definition of SaaS and multi-tenancy. This included looking at how the notion of multi-tenancy needed to embrace a broader range of architecture patterns and strategies. Now, in this chapter, we can move from the conceptual to a more concrete view of how this definition of SaaS influences the options you have when you start defining and choosing your SaaS architecture.

Here, we'll start to identify and name specific multi-tenant architecture patterns, providing a better sense of the different high-level architecture strategies you'll want to consider when designing your own SaaS environment. I refer to these different patterns as deployment models largely because they represent how your tenants will be deployed into the application plane of your SaaS solution. Each deployment model is meant to identify a distinct approach to defining the high-level multi-tenant architectural model that could be employed by your system.

The deployment model you select will have a significant influence on the technical and business profile of your SaaS architecture. Each deployment model comes with its own unique blend of pros and cons that you'll need to weigh to figure out which pattern best aligns with

the realities and goals of your solution. Are your tenants sharing some of their infrastructure? Are they sharing all of their infrastructure? Or, is there some mix of shared and dedicated infrastructure in your SaaS environment? What domain and compliance models will you need to support? Are you planning to offer a tiered model? Your answers to these questions (and others) will all be used as data points that will guide your selection of a deployment model. From there, you'll see how the selection of a deployment model will have a cascading impact across all the moving parts of your multi-tenant implementation, fundamentally shaping how your solution is built, managed, operated, configured, and provisioned. Thus, the need to have a firm grasp of these models and their corresponding tradeoffs.

Let's look at a couple of conceptual deployment models to help clarify this concept. [Figure 3-1](#) provides examples of two sample deployment models. On the left, you'll see a deployment model that has all of its tenant resources being shared across the compute layers of our multi-tenant environment. However, the storage resources are dedicated to individual tenants. In contrast, the right-hand side of the diagram shows another variation of a deployment model where all of a tenant's infrastructure (compute, storage, etc.) is deployed in a dedicated model. These are just a small sample of two deployment models, but they give you a more concrete view of what we're talking

about when we're describing the fundamental aspects of a SaaS deployment model.

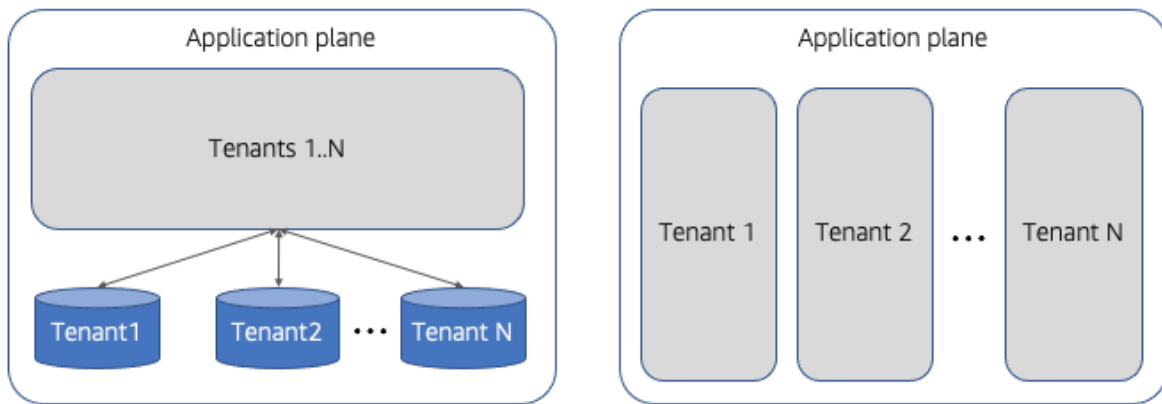


Figure 3-1. Conceptual deployment models

It's important to note that a deployment model represents a pattern that may have multiple implementations that share a common set of values and guiding principles. My goal here is to clearly identify the different categories of deployment models, acknowledging that the attributes and realities of these models will vary based on the technology stack, services, tools, and other environmental factors that are used to move from concept to implementation. Also, these models should not be viewed as being mutually exclusive. As you'll see in the sections that follow, there can be compelling business reasons that may have teams creating solutions that rely on combinations of these models. For our discussion here, you should start by understanding the principles and value proposition of each deployment model. Then,

from there, you can consider which flavors/combinations of these models best align with the requirements of your SaaS solution.

Picking a Deployment Model

Understanding the value proposition of each deployment model is helpful. However, selecting a deployment model goes beyond evaluating the characteristics of any one model. When you sit down to figure out which deployment model is going to best for your application and business, you'll often have to weigh a wide spectrum of parameters that will inform your deployment selection process.

In some cases, the state of your current solution (if you're migrating) might have a huge impact on the deployment model you choose. A SaaS migration can often be more about finding a target deployment model that lets you get to SaaS without rebuilding your entire solution. Time to market, competitive pressures, legacy technology considerations, and team makeup are also factors that could represent significant variables in a SaaS migration story. Each of these factors would likely shape the selection of a deployment model.

Obviously, teams that are building a new SaaS solution have more of a blank canvas to work with. Here, the deployment model that you choose is probably more driven by the target personas and experi-

ence that you're hoping to achieve with your multi-tenant offering. The challenge here is selecting a deployment model that balances the near- and long-term goals of the business. Selecting a model that is too narrowly focused on a near-term experience could limit growth as your business hits critical mass. At the same time, over-rotating to a deployment model that reaches too far beyond the needs of current customers may represent pre-optimization. It's certainly a tough balancing act to find the right blend of flexibility and focus here (a classic challenge for most architects and builders).

No matter where you start your path to SaaS, there are certainly some broader global factors that will influence your deployment model selection. Packaging, tiering, and pricing goals, for example, often play a key role in determining which deployment model(s) might best fit with your business goals. Cost and operational efficiency are also part of the deployment model puzzle. While every solution would like to be as cost and operationally efficient as possible, each business may be facing realities that can impact their deployment model preferences. If your business has very tight margins, you might lean more toward deployment models that squeeze every last bit of cost efficiency out of your deployment model. Others may be facing challenging compliance and/or performance considerations that might lead to deployment models that strike a balance between cost and customer demands.

These are just some simple examples that are part of the fundamental thought process you'll go through as part of figuring out which deployment model will address the core needs of your business. As I get deeper into the details of multi-tenant architecture patterns, you'll see more and more places where the nuances of multi-tenant architecture strategies will end up adding more dimensions to the deployment model picture. This will also give you a better sense of how the differences in these models might influence the complexity of your underlying solution. The nature of each deployment model can move the complexity from one area of our system to another.

The key here is that you should not be looking for a one-size-fits-all deployment model for your application. Instead, you should start with the needs of your domain, customers, and business and work backward to the combination of requirements that will point you toward the deployment model that fits with your current and aspirational goals.

It's also important to note that the deployment model of your SaaS environment is expected to be evolving. Yes, you'll likely have some core aspects of your architecture that will remain fairly constant. However, you should also expect and be looking for ways to refine your deployment model based on the changing/emerging needs of customers, shifts in the market, and new business strategies. Worry less about getting it right on day one and just expect that you'll be using data from your environment to find opportunities to refactor your de-

ployment model. A resource that started out as a dedicated resource, might end up switched to a shared resource based on consumption, scaling, and cost considerations. A new tier might have you offering some parts of your system in a dedicated model. Being data driven and adaptable are all part of the multi-tenant technical experience.

Introducing the Silo and Pool Models

As we look at deployment models, we're going to discover that these models will require the introduction of new terminology that can add precision to how we characterize SaaS architecture constructs. This relates to our earlier exploration of how the term multi-tenant had to take on a broader meaning to fit the realities of SaaS businesses. Now, as we start to look at deployment models, you'll notice that we still need terminology that can better capture and accurately convey how the resources in our architecture are consumed by tenants.

There are two terms that I'm going to introduce here to give us a more granular way to think about classifying dedicated and shared resources. Across the rest of this book, you'll see that I will use the term *silo* to refer to any model where a resource is dedicated to a given tenant. I'll use the term *pool* to reference any model where a tenant resource is shared by one or more tenants.

This may seem like a subtle nuance in language. In reality, it has significant implications on how we describe multi-tenant architecture. It allows us to describe the behavior and scope of our SaaS architecture resources without the ambiguity and legacy baggage that comes with labeling resources a multi-tenant. As we look more at deployment models and the full range of SaaS architecture concepts that span this book, I will be using silo and pool as the foundational terms that characterize the usage, isolation, deployment, and consumption of the resources in our multi-tenant architecture.

To help crystallize this concept, let's look at a conceptual architecture that includes resources that are being consumed in a combination of dedicated and shared models. [Figure 3-2](#) provides a view of a series of microservices that have been deployed into a SaaS architecture. In this image, I've created a hypothetical environment where we have a series of microservices that are using different strategies for dedicating and sharing tenant resources.

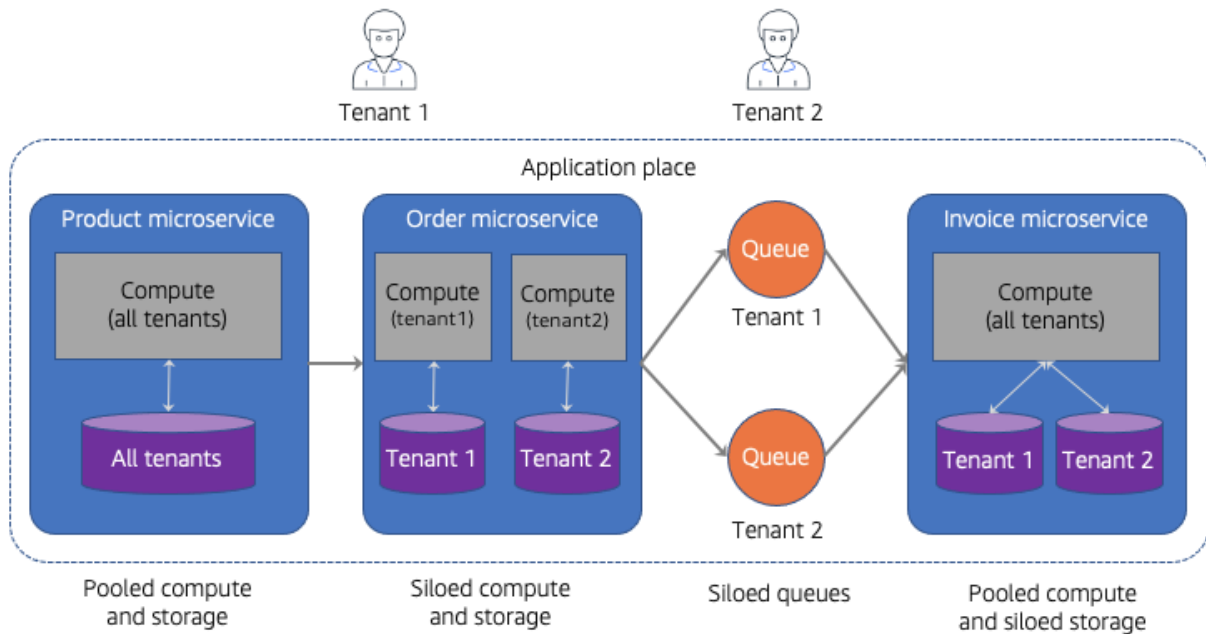


Figure 3-2. Silo and pooled resource models

At the top of the diagram, you'll see that I've put two tenants here to illustrate how the tenants in our environment are landing in and consuming resources. These tenants are running an ecommerce application that is implemented via Product, Order, and Invoice microservices. Now, if we follow a path through these microservices from left to right, you'll see how we've applied different deployment strategies for each of these microservices.

Let's start with the Product microservice. Here, I've chosen a strategy where the compute and the storage for all of our tenants will be deployed in a pooled model. For this scenario, I have decided that the isolation and performance profile of this service fits best with the values of a pooled approach. As we move to the Order microservice,

you'll see that I've chosen a very different model here. In this case, the service has siloed compute and storage for every tenant. Again, this was done based on the specific needs of my environment. This could have been driven by some SLA requirement or, perhaps, a compliance need.

From the Order service, you'll then see that our system sends a message to a queue that prepares these orders for billing. This scenario is included to highlight the fact that our siloed and pooled concepts are extended beyond our microservices and applied to any resource that might be part of our environment. For this solution, I've opted to have a siloed queues for each tenant. Finally, I have an Invoice service on the right-hand side that pulls messages from these queues and generates invoices. To meet the requirements of our solution, I've used a mix of siloed and pooled models in this microservice. Here, the compute is pooled and the storage is siloed.

The key takeaway here is that the terms silo and pool are used to generally characterize the architecture footprint of one or more resources. These terms can be applied in a very granular fashion, highlighting how tenancy is mapped to very specific elements of your architecture. These same terms can also be used more broadly to describe how a collection of resources are deployed for a tenant. So, don't try to map silo and pool to specific constructs. Instead, think of

them as describing the tenancy of a single resource or a group of resources.

This caveat will be especially important as we look at deployment models throughout this chapter, allowing us to apply to silo and pool concepts and varying scopes across our multi-tenant architecture.

Full Stack Silo Deployment

Now that you have a high-level sense of the scope and role of deployment models, it's time to dig in a bit more and start looking at defining specific types of deployment models. Let's start by looking at what I'll label as a full stack silo deployment model.

As its name suggests, the full stack silo model places each tenant into a fully siloed environment where all of the resources of a tenant are completely siloed. The diagram in [Figure 3-3](#) provides an example of a full stack silo environment. Here you'll see that we have an environment where our application plane is running workloads for two tenants. These two tenants are running in siloes where the compute, storage, and every resource that's needed for the tenant is deployed into some logical construct that creates a clear boundary between our tenants.

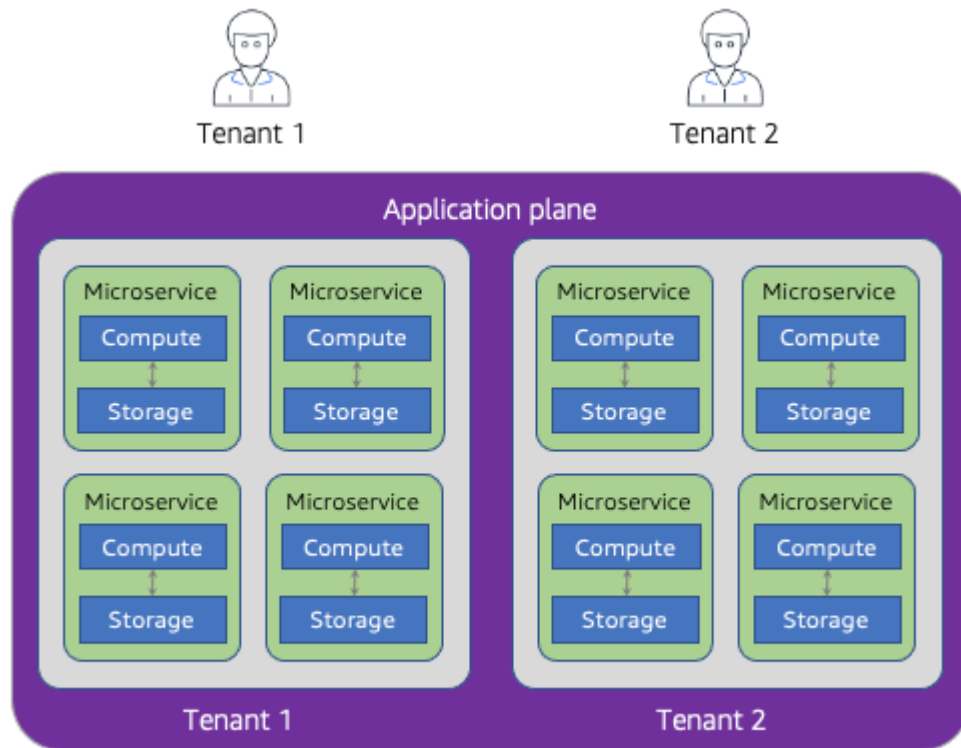


Figure 3-3. Full stack silo deployment model

In this particular example, I simplified the contents of the silo, showing a range of microservices that are running in each tenant environment. In reality, what's in the silo and how your application is represented could be represented by any number of different technologies and design strategies. This could have been an n-tier environment with separate web, application, and storage tiers. I could have included any number of different compute models and other services here as well (queues, object storage, messaging, and so on). The emphasis, at this stage, is less about what's in each of these tenant environments and more on the nature of how they are deployed.

Where Full Stack Silo Fits

The full stack silo model can feel like a bit of a SaaS anti-pattern. After all, so much of our discussion of SaaS is centered around agility and efficiency. Here, where we have fully siloed tenant resources, it can appear as though we've compromised on some of the fundamental goals of SaaS. However, if you think back to the definition of SaaS in Chapter 1, you'll recall that SaaS isn't exclusively about sharing infrastructure for economies of scale. SaaS is about operating in a model where all of our tenants are operated, managed, and deployed collectively. This is the key thing to keep in mind when you're looking at the full stack silo model. Yes, it has efficiency challenges. We'll get into those. At the same time, as long as every one of these environments is the same and as long as these are running the same version of our application, then we can still realize much of the value proposition of SaaS.

So, knowing that full stack silo meets our criteria for SaaS, the real question here is more about when it might make sense for you to employ this model. Which factors typically steer organizations toward a full stack silo experience? When might it be a fit for the business and technology realities of your environment? While, there are no absolutes here, there are common themes and environmental factors that have teams selecting the full stack silo model. Compliance and legacy considerations are two of the typical reasons teams will end up

opting for full stack silo footprint. In some heavily regulated domains, teams may choose a full stack silo model to simplify their architecture and make it easier for them to address specific compliance criteria. Customers in these domains might also have some influence on the adoption of a full stack silo, insisting on having siloed resources as part of selecting a SaaS solution.

The full stack silo model can also represent a good fit for organizations that are migrating a legacy solution to SaaS. The fully siloed nature of this model allows these organizations to move their existing code into a SaaS model without major refactoring. This gets them to SaaS faster and reduces their need to more immediately take on adding tenancy to all the moving parts of their architecture. Migrating teams will still be required to retrofit your legacy environment to align with the SaaS control plane, its identity model, and a host of other multi-tenant considerations. However, the scope and reach of these impacts can be less pronounced if your solution is moving into a full stack silo environment that doesn't need to consider scenarios where any of a tenant's resources are pooled.

Full stack silo can also be a tiering strategy. For example, some organizations may offer a premium version of their solution that, for the right price, will offer tenants a fully dedicated experience. It's important to note that this dedicated experience is not created as a one-off environment for these tenants. It's still running the same version of

the application and is centrally managed alongside all the other tiers of the system.

In some cases, the full stack model simply represents a lower barrier of entry for teams—especially those that may not be targeting a large number of tenants. For these organizations, full stack silo allows them to get to SaaS without tackling some of the added complexities that come with building, isolating, and operating a pooled environment. Of course, these teams also have to consider how adoption of a full stack silo model might impact their ability to rapidly scale the business. In this case, the advantages of starting with a full stack could be offset by the inefficiencies and margin impacts of being in a full stack silo model.

Full Stack Silo Considerations

Teams that opt for a full stack silo model, will need to consider some of the nuances that come with this model. There are definitely pros and cons to this approach that you'll want to add to your mental model when selecting this deployment model. The sections that follow provide a breakdown of some of the key design, build, and deployment considerations that are associated with the full stack silo deployment model.

Control Plane Complexity

As you may recall, I have described all SaaS architectures as having control and application planes where our tenant environments live in the application plane and are centrally managed by the control plane. Now, with the full stack silo model, you have to consider how the distributed nature of the full stack model will influence the complexity of your control plane.

In [Figure 3-4](#), you can see an example of a full stack silo deployment that highlights some of the elements that come with building and managing this model. Since our solution is running in a silo per tenant model, the application plane must support completely separate environments for each tenant. Instead of interacting with a single, shared resource, our control plane must have some awareness of each of these tenant siloes. This inherently adds complexity to our control plane, which now must be able to operate each of these separate environments.

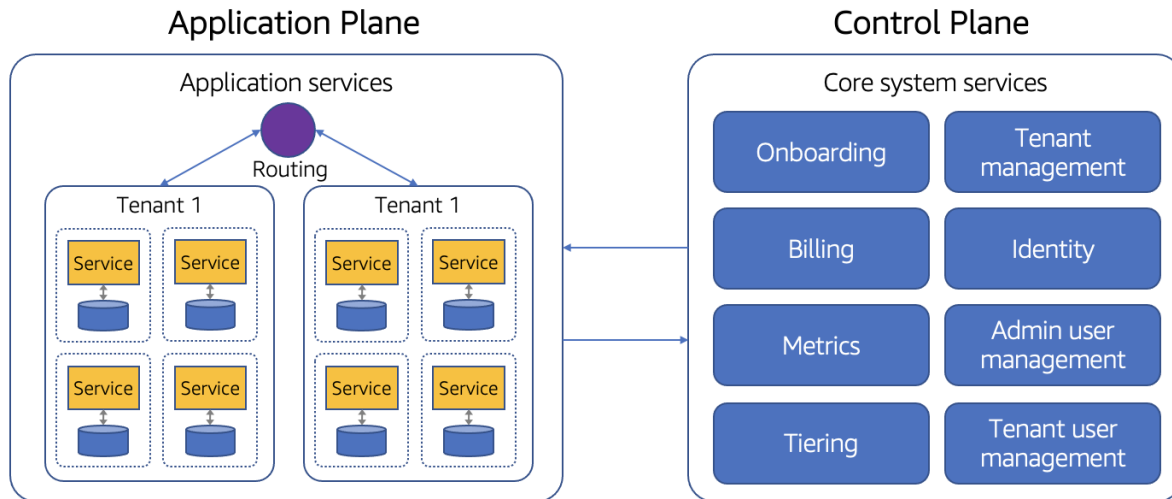


Figure 3-4. Managing and operating a full stack silo

Imagine implementing tenant onboarding in this example. The addition of each new tenant to this environment must fully provision and configure each siloed tenant environment. This also complicates any tooling that may need to monitor and manage the health of tenant environments. Your control plane code must know where each tenant environment can be found and be authorized to access each tenant silo. Any operational tooling that you have that is used to manage infrastructure resources will also need to deal with the larger, more distributed footprint of these tenant environments, which could make troubleshooting and managing infrastructure resources more unwieldy. Any tooling you've created to centralized metrics, logs, and analytics for your tenants will also need to be able to aggregate the data from these separate tenant environments. Deployment of appli-

cation updates is also more complicated in this model. Your DevOps code will have to roll out updates to each tenant silo.

There are likely more complexities to cover here. However, the theme is that the distributed nature of the full stack silo model touches many of the moving parts of your control plane experience, adding complexity that may not be as pronounced in other deployment models. While the challenges here are all manageable, it definitely will take extra effort to create a fully unified view of management and operations for any full stack silo environment.

Scaling Impacts

Scale is another important consideration for the full stack silo deployment model. Whenever you're provisioning separate infrastructure for each tenant, you need to think about how this model will scale as you add more tenants. While the full stack silo model can be appealing when you have 10 tenants, its value can begin to erode as you consider supporting hundreds or thousands of tenants. The full stack silo model, for example, would not be sustainable in many classic BC2 environments where the scale and number of tenants would be massive. Naturally, the nature of your architecture would also have some influence on this. If you're running Kubernetes, for example, it might come down to how effectively the silo constructs of Kubernetes would scale here (clusters, namespaces, etc.). If you're using separate

cloud networking or account constructs for each siloed tenant, you'd have to consider any limits and constraints that might be applied by your cloud provider.

The broader theme here is that full stack siloed deployments are not for everyone. As I get into specific full stack silo architectures, you'll see how this model can run into important scaling limits. More importantly, even if your environment can scale in a full stack silo model, you may find that there's a point at which the full stack silo can become difficult to manage. This could undermine your broader agility and innovation goals.

Cost Considerations

Costs are also a key area to explore if you're looking at using a full stack silo model. While there are measures you can take to limit overprovisioning of siloed environments, this model does put limits on your ability to maximize the economies of scale of your SaaS environment. Typically, these environments will require lots of dedicated infrastructure to support each tenant and, in some cases, this infrastructure may not have an idle state where it's not incurring costs. For each tenant, then, you will have some baseline set of costs for tenants that you'll incur—even if there is no load on the system. Also, because these environments aren't shared, we don't get the efficiencies that would come with distributing the load of many tenants

across shared infrastructure that scales based on the load of all tenants. Compute, for example, can scale dynamically in a silo, but it will only do so based on the load and activity of a single tenant. This may lead to some over-provisioning within each silo to prepare for the spikes that may come from individual tenants.

Generally, organizations offering full stack siloed models are required to create cost models that help overcome the added infrastructure costs that come with this model. That can be a mix of consumption and some additional fixed fees. It could just be a higher subscription price. The key here is that, while the full stack silo may be the right fit for some tiers or business scenarios, you'll still need to consider how the siloed nature of this model will influence the pricing model of your SaaS environment.

As part of the cost formulas, we must also consider how the full stack silo model impacts the operational efficiency of your organization. If you've built a robust control plane and you've automated all the bits of your onboarding, deployment, and so on, you can still surround your full stack silo model with a rich operational experience. However, there is some inherent complexity that comes with this model that will likely add some overhead to your operational experience. This might mean that you will be required to invest more in the staff and tooling that's needed to support this model, which will add additional costs to your SaaS business.

Routing Considerations

In [Figure 3-4](#), I also included a conceptual placeholder for the routing of traffic within our application plane. With a full stack silo, you'll need to consider how the traffic will be routed to each silo based on tenant context. While there are any number of different networking constructs that we can use here to route this load, you'll still need to consider how this will be configured. Are you using subdomains for each tenant? Will you have a shared domain with the tenant context embedded in each request? Each strategy you choose here will require some way for your system to extract that context and route your tenants to the appropriate silo.

The configuration of this routing construct must be entirely dynamic. As each new tenant is onboarded to your system, you'll need to update the routing configuration to support routing this new tenant to its corresponding silo. None of this is wildly hard, but you'll see that this is an area that will need careful consideration as you design your full stack siloed environment. Each technology stack will bring its own set of considerations to the routing problem.

Availability and Blast Radius

The full stack silo model does offer some advantages when it comes to the overall availability and durability of your solution. Here, with

each tenant in its own environment, there is potential to limit the blast radius of any potential operational issue. The dedicated nature of the silo model gives you the opportunity to contain some issues to individual tenant environments. This can certainly have an overall positive effect on the availability profiles of your service.

Rolling out new releases also behaves a bit differently in siloed environments. Instead of having your release pushed to all customers at the same time, the full stack silo model may release to customers in waves. This can allow you to detect and recover from issues related to a deployment before it is released to the entire population. It, of course, also complicates the availability profile. Having to deploy to each silo separately requires you to have a more complicated rollout process that can, in some cases, undermine the availability of your solution.

Simpler Cost Attribution

One significant upside to the full stack silo model is its ability to attribute costs to individual tenants. Calculating cost-per-tenant for multi-tenant environments, as you'll see in Chapter 14, can be tricky in SaaS environments where some or all of a tenant's resources may be shared. Knowing just how much of a shared database or compute resource was consumed by a given tenant is not so easy to infer in pooled environments. However, in a full stack silo model, you won't

face these complexities. Since each tenant has its own dedicated infrastructure, it becomes relatively easy to aggregate and map costs to individual tenants. Cloud providers and third-party tools are generally good at mapping costs to individual infrastructure resources and calculating a cost for each tenant.

Full Stack Silo in Action

Now that we have a good sense of the full stack silo model, let's look at some working examples of how this model is brought to life in real-world architecture. As you can imagine, there are any number of ways to implement this model across the various cloud providers, technology stacks, and so on. The nuances of each technology stack adds its own set of considerations to your design and implementation.

The technology and strategy you use to implement your full stack silo model will likely be influenced by some of the factors that were outlined above. They might also be shaped attributes of your technology stack and your domain realities.

The examples here are pulled from my experience building SaaS solutions at Amazon Web Services (AWS). While these are certainly specific to AWS, these patterns have corresponding constructs that have mappings to other cloud providers. And, in some instances,

these full stack silo models could also be built in an on-premises model.

The Account Per Tenant Model

If you're running in a cloud environment—which is where many SaaS applications often land—you'll find that these cloud providers have some notion of an account. These accounts represent a binding between an entity (an organization or individual) and the infrastructure that they are consuming. And, while there's a billing and security dimension to these accounts, our focus is on how these accounts are used to group infrastructure resources.

In this model, accounts are often viewed as the strictest of boundaries that can be created between tenants. This, for some, makes an account a natural home for each tenant in your full stack silo model. The account allows each silo of your tenant environments to be surrounded and protected by all the isolation mechanisms that cloud providers use to isolate their customer accounts. This limits the effort and energy you'll have to expend to implement tenant isolation in your SaaS environment. Here, it's almost a natural side effect of using an account-per-tenant in your full stack silo model.

Attributing infrastructure costs to individual tenants also becomes a much simpler process in an account-per-tenant model. Generally,

your cloud provider already has all the built-in mechanisms needed to track costs at the account level. So, with an account-per-tenant model, you can just rely on these ready made solutions to attribute infrastructure costs to each of your tenants. You might have to do a bit of extra work to aggregate these costs into a unified experience, but the effort to assemble this cost data should be relatively straightforward.

In [Figure 3-5](#), I've provided a view of an account-per-tenant architecture. Here, you'll see that I've shown two full stack siloed tenant environments. These environments are mirror images, configured as clones that are running the exact same infrastructure and application services. When any updates are applied, they are applied universally to all tenant accounts.

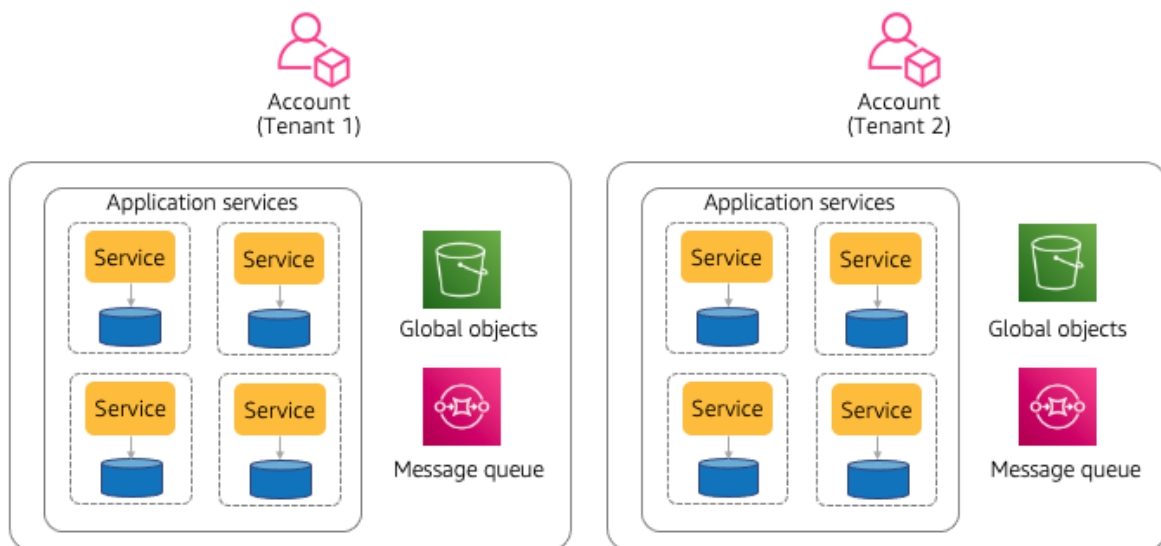


Figure 3-5. The account-per-tenant full stack silo model

Within each account, you'll see examples of the infrastructure and services that might be deployed to support the needs of your SaaS application. There are placeholders here to represent the services that support the functionality of your solution. To the right of these services, I also included some additional infrastructure resources that are used within our tenant environments. Specifically, I put an object store (Amazon Simple Storage Service) and a managed queue service (Amazon's Simple Queue Services). The object store might hold some global assets and the queue is here to support asynchronous messaging between our services. I included these to drive home the point that our account-per-tenant silo model will typically encapsulate all of the infrastructure that is needed to support the needs of a given tenant.

Now, the question is: does this model mean that infrastructure resources cannot be shared between our tenant accounts? For example, could these two tenants be running all of their microservices in separate accounts and share access to a centralized identity provider? This wouldn't exactly be unnatural. The choices you make here are more driven by a combination of business/tenant requirements as well as the complexities associated with accessing resources that are outside the scope of a given account.

Let's be clear. full stack silo, I'm still saying that the application functionality of your solution is running completely in its own account. The

only area here where we might allow something to be outside of the account is when it plays some more global role in our system. Here, let's imagine the object store represented a globally managed construct that held information that was centrally managed for all tenants. In some cases, you may find one-off reasons to have some bits of your infrastructure running in some shared model. However, anything that is shared cannot have an impact on the performance, compliance, and isolation requirements of our full stack silo experience. Essentially, if you create some centralized, shared resource that impacts the rationale for adopting a full stack silo model, then you've probably violated the spirit of using this model.

The choices you make here should start with assessing the intent of your full stack silo model. Did you choose this model based on an expectation that customers would want *all* of their infrastructure to be completely separated from other tenants? Or, was it more based on a desire to avoid noisy neighbor and data isolation requirements? Your answers to these questions will have a significant influence on how you choose to share parts of your infrastructure in this model.

If your code needs to access any resources that are outside of your account, this can also introduce new challenges. Any externally accessed resource would need to be running within the scope of some other account. And, as a rule of thumb, accounts have very intentional and hard boundaries to secure the resources in each account. So,

then, you'd have to wander into the universe of authorizing cross-account access to enable your system to interact with any resource that lives outside of a tenant account.

Generally, I would stick with the assumption that, in a full stack silo model, your goal is to have all tenant's resources in the same account. Then, only when there's a compelling reason that still meets the spirit of your full stack silo, consider how/if you might support any centralized resources.

Onboarding Automation

The account-per-tenant silo model adds some additional twists to the onboarding of new tenants. As each new tenant is onboarded (as we'll see in Chapter 4), you will have to consider how you'll automate all the provisioning and configuration that comes with introducing a new tenant. For the account-per-tenant model, our provisioning goes beyond the creation of tenant infrastructure—it also includes the creation of new accounts.

While there are definitely ways to automate the creation of accounts, there are aspects of the account creation that can't always be fully automated. In cloud environments, however, there are some intentional constraints here that may restrict your ability to automate the configuration or provisioning of resources that may exceed the default limits for those resources. For example, your system may rely on

a certain number of load balancers for each new tenant account. However, the number you require for each tenant may exceed the default limits of your cloud provider. Now, you'll need to go through the processes, some of which may not be automated, to increase the limits to meet the requirements of each new tenant account. This is where your onboarding process may not be able to fully automate every step in a tenant onboarding. Instead, you may need to absorb some of the friction that comes with using the processes that are supported by your cloud provider.

While teams do their best to create clear mechanisms to create each new tenant account, you may just need to allow for the fact that, as part of adopting an account-per-tenant model, you'll need to consider how these potential limit issues might influence your onboarding experience. This might mean creating different expectations around onboarding SLAs and better managing tenant expectations around this process.

Scaling Consideration

I've already highlighted some of the scaling challenges that are typically associated with the full stack silo model. However, with the account-per-tenant model, there's another layer to the full stack silo scaling story.

Generally speaking, mapping accounts to tenants could be viewed as a bit of an anti-pattern. Accounts, for many cloud providers, were not necessarily intended to be used as the home for tenants in multi-tenant SaaS environments. Instead, SaaS providers just gravitated toward them because they seemed to align well with their goals. And, to a degree, this makes perfect sense.

Now, if you have an environment with 10s of tenants, you may not feel much of the pain as part of your account-per-tenant model. However, if you have plans to scale to a large number of tenants, this is where you may begin to hit a wall with the account-per-tenant model. The most basic issue you can face here is that you may exceed the maximum number of accounts supported by your cloud provider. The more subtle challenge here shows up over time. The proliferation of accounts can end up undermining the agility and efficiency of your SaaS business. Imagine having hundreds or thousands of tenants running in this model. This will translate into a massive footprint of infrastructure that you'll need to manage. While you can take measures to try to streamline and automate your management and operation of all these accounts, there could be points at which this may no longer be practical.

So, where is the point of no return? I can't say there's an absolute data point at which the diminishing returns kicks in. So much depends on the nature of your tenant infrastructure footprint. I mention this

mostly to ensure that you're factoring this into your thinking when you take on an account-per-tenant model.

The VPC-Per-Tenant Model

The account-per-tenant model relies on a pretty coarse-grained boundary. Let's shift our focus to constructs that realize a full stack silo within the scope of a single account. This will allow us to overcome some of the challenges of creating accounts for individual tenants. The model we'll look at now, the Virtual Private Cloud (VPC)-Per-Tenant Model, is one that relies more on networking constructs to house the infrastructure that belongs to each of our siloed tenants.

Within most cloud environments you're given access to a fairly rich collection of virtualized networking constructs that can be used to construct, control, and secure the footprint of your application environments. These networking constructs provide natural mechanisms for implementing a full stack siloed implementation. The very nature of networks and their ability to describe and control access to their resources provides SaaS builders with a powerful collection of tools that can be used to silo tenant resources.

Let's look at an example of how a sample networking construct can be used to realize a full stack silo model. [Figure 3-6](#) provides a look at

a sample network environment that uses Amazon's VPC to silo tenant environments.

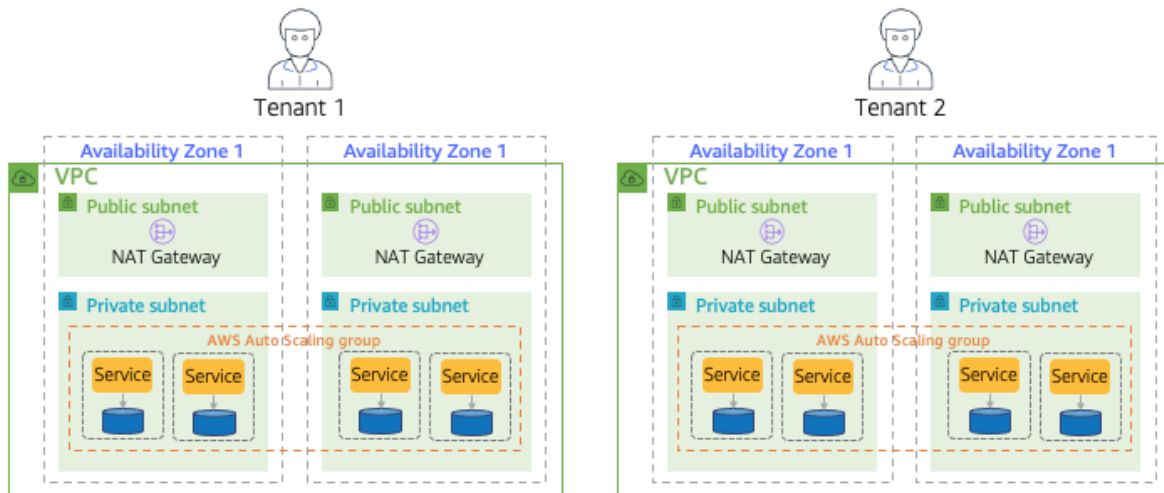


Figure 3-6. The VPC-per-tenant full stack silo model

At first glance, there appears to be a fair amount of moving parts in this diagram. While it's a tad busy, I wanted to bring in enough of the networking infrastructure to give you a better sense of the elements that are part of this model.

You'll notice that, at the top of this image, we have two tenants. These tenants are accessing siloed application services that are running in separate VPCs. The VPC is the green box that is at the outer edge of our tenant environments. I also wanted to illustrate the high availability footprint of our VPC by having it include two separate availability zones (AZs). We won't get into AZs, but just know that AZs represent distinct locations within an AWS Region that are engineered to be

isolated from failures in other AZs. We also have separate subnets here to separate the public and private subnets of our solution. Finally, you'll see the application services of our solution deployed into private subnets of our two AZs. These are surrounded by what AWS labels as an Auto Scaling Group, which allows our services to dynamically scale based on tenant load.

I've included all these network details to highlight the idea that we're running our tenants in the network siloes that offer each of our tenants a very isolated and resilient networking environment that leans on all the virtualized networking goodness that comes with building and deploying your solution in a VPC-per-tenant siloed model.

While this model may seem less rigid than the account-per-tenant model, it actually provides you with a solid set of constructs for preventing any cross-tenant access. You can imagine how, as part of their very nature, these networking tools allow you to create very carefully controlled ingress and egress for your tenant environments. We won't get into the specifics, but the list of access and flow control mechanisms that are available here is extensive. More details can be found [here](#).

Another model that shows up here, occasionally, is the subnet-per-tenant model. While I rarely see this model, there are some instances where teams will put each tenant silo in a given subnet. This, of

course, can also become unwieldy and difficult to manage as you scale.

Onboarding Automation

With the account-per-tenant model, I dug into some of the challenges that it could create as part of automating your onboarding experience. With the VPC-per-tenant model, the onboarding experience changes some. The good news here is that, since you're not provisioning individual accounts, you won't run into the same account limits automation issues. Instead, the assumption is that the single account that is running our VPCs will be sized to handle the addition of new tenants. This may still require some specialized processes, but they can be applied outside the scope of onboarding.

In the VPC-per-tenant model, our focus is more on provisioning your VPC constructs and deploying your application services. That will likely still be a heavy process, but most of what you need to create and configure can be achieved through a fully automated process.

Scaling Considerations

As with accounts, VPCs also face some scaling considerations. Just as there are limits on the number of accounts you can have, there can also be limits on the number of VPCs that you can have. The management and operation of VPCs can also get complicated as you

begin to scale this model. Having tenant infrastructure sprawling across hundreds of VPCs may impact the agility and efficiency of your SaaS experience. So, while VPC has some upsides, you'll want to think about how many tenants you'll be supporting and how/if the VPC-per-tenant model is practical for your environment.

Remaining Aligned on Full Stack Silo Mindset

Before I move on to any new deployment models, it's essential that we align on some key principles in the full stack silo. For some, the allure of the full stack silo model can be appealing because it can feel like it opens (or re-opens) the door for SaaS providers to offer one-off customization to their tenants. While it's true that the full stack silo model offers dedicated resources, this should never be viewed as an opportunity to fall back to the world of per-tenant customization. The full stack silo only exists to accommodate domain, compliance, tiering, and any other business realities that might warrant the use of a full stack silo model.

In all respects, a full stack silo environment is treated the same as a pooled environment. Whenever new features are released, they are deployed to *all* customers. If your infrastructure configuration needs to be changed, that change should be applied to *all* of your siloed environments. If you have policies for scaling or other run-time behaviors, they are applied based on tenant tiers. You should never have a

policy which is applied to an individual tenant. The whole point of SaaS is that we are trying to achieve agility, innovation, scale, and efficiency through our ability to manage and operate our tenants collectively. Any drift toward a one-off model will slowly take you away from those SaaS goals. In some cases, organizations that moved to SaaS to maximize efficiency will end up regressing through one-off customizations that undermine much of the value they hoped to get out of a true SaaS model.

The guidance I always offer to drive this point home centers around how you arrive at a full stack silo model. I tell teams that—even if you’re targeting a full stack silo as your starting point—you should build your solution as if it were going to be a full stack pooled model. Then, treat each full stack silo as an instance of your pooled environment that happens to have a single tenant. This serves as a forcing function that allows the full stack siloed environments to inherit the same values that are applied to a full stack pool (which we’re covering next).

The Full Stack Pool Model

The full stack pool model, as its name suggests, represents a complete shift from the full stack silo mindset and mechanisms we’ve been exploring. With the full stack pool model, we’ll now look at SaaS

environments where all of the resources for our tenants are running in a shared infrastructure model.

For many, the profile of a fully pooled environment maps to their classic notion of multi-tenancy. It's here where the focus is squarely on achieving economies of scale, operational efficiencies, cost benefits, and a simpler management profile that are the natural byproducts of a shared infrastructure model. The more we are able to share infrastructure resources, the more opportunities we have to align the consumption of those resources with the activity of our tenants. At the same time, these added efficiencies also introduce a range of new challenges.

[Figure 3-7](#) provides a conceptual view of the full stack silo model.

You'll see that I've still included that control plane here just to make it clear that the control plane is a constant across any SaaS model. On the left of the diagram is the application plane, which now has a collection of application services that are shared by all tenants. The tenants shown at the top of the application plane are all accessing and invoking operations on the application microservices and infrastructure.

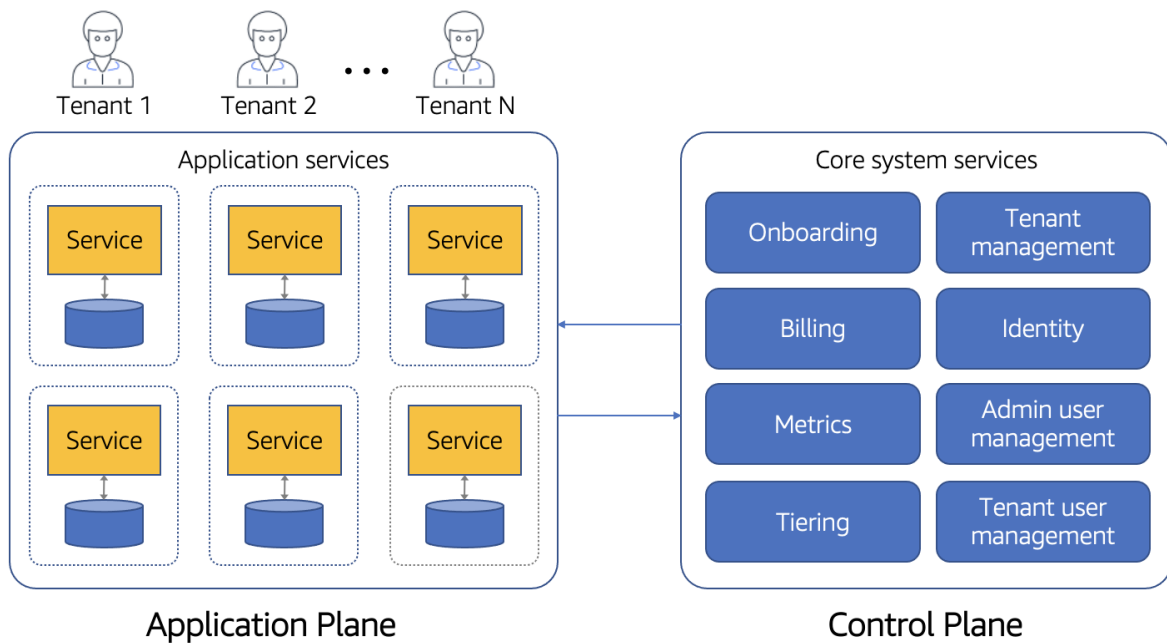


Figure 3-7. A full stack pooled model

Now, within this pool model, tenant context plays a much bigger role. In the full stack silo model, tenant context was primarily used to route tenants to their dedicated stack. Once a tenant lands in a silo, that silo knows that all operations within that silo are associated with a single tenant. With our full stack pool, however, this context is essential to every operation that is performed. Accessing data, logging messages, recording metrics—all of these operations will need to resolve the current tenant context at run-time to successfully complete their task.

[Figure 3-8](#) gives you a better sense of how tenant context touches every dimension of our infrastructure, operations, and implementation are influenced by this tenant context in a pooled model. This concep-

tual diagram highlights how each microservice must acquire tenant context and apply it as part of its interactions with data, the control plane, and other microservices. You'll see tenant context being acquired and applied as we send billing events and metrics data to the control plane. You'll see it injected in your call to downstream microservices. It also shows up in our interaction with data.

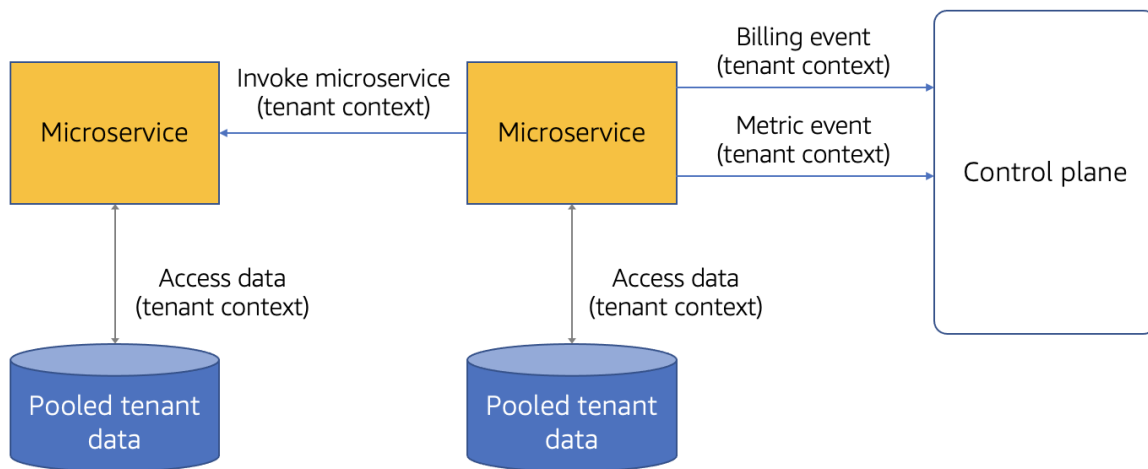


Figure 3-8. Tenant context in the full stack pooled environment

The fundamental idea here is that, when we have a pooled resource, that resource belongs to multiple tenants. As a result, tenant context is needed to apply scope and context to each operation at run-time.

Now, to be fair, tenant context is valid across all SaaS deployment models. Silo still needs tenant context as well. What's different here is that the silo model knows its binding to the tenant at the moment it's provisioned and deployed. So, for example, I could associate an envi-

environment variable as the tenant context for a siloed resource (since its relationship to the tenant does not change at run-time). However, a pooled resource is provisioned and deployed for all tenants and, as such, it must resolve its tenant context based on the nature of each request it processes.

As we dig deeper into more multi-tenant implementation details, we'll discover that these differences between silo and pool models can have a profound impact on how we architect, deploy, manage, and build the elements of our SaaS environment.

Full Stack Pool Considerations

The full stack pool model also comes with a set of considerations that might influence how/if you choose to adopt this model. In many respects, the considerations for the full stack pool model are the natural inverse of the full stack silo model. Full stack pool certainly has strengths that are appealing to many SaaS providers. It also presents a set of challenges that come with having shared infrastructure. The sections that follow highlight these considerations.

Scale

Our goal in multi-tenant environments is to do everything we can to align infrastructure consumption with tenant activity. In an ideal scenario, your system would, at a given moment in time, only have

enough resources allocated to accommodate the current load being imposed by tenants. There would be zero over-provisioned resources. This would let the business optimize margins and ensure that the addition of new tenants would not drive a spike in costs that could undermine the bottom line of the business.

This is the dream of the full stack pooled model. If your design was somehow able to fully optimize the scaling policies of your underlying infrastructure in a full stack pool, you would have achieved multi-tenant nirvana. This is not practical or realistic, but it is the mindset that often surrounds the full stack pooled model. The reality, however, is that creating a solid scaling strategy for a full stack pooled environment is very challenging. The loads of tenants are often constantly changing and new tenants may be arriving every day. So, the scaling strategy that worked yesterday, may not work today. What typically happens here is teams will accept some degree of over-provisioning to account for this continually shifting target.

The technology stack you choose here can also have a significant impact on the scaling dynamics of your full stack pool environment. In Chapter 12 we'll look at a serverless SaaS architecture and get a closer look at how using serverless technologies can simplify your scale story and achieve better alignment between infrastructure consumption and tenant activity.

The key theme here is that, while there are significant scaling advantages to be had in a full stack pooled model, the effort to make this scaling a reality can be challenging to fully realize. You'll definitely need to work hard to craft a scaling strategy that can optimize resource utilization without impacting tenant experience.

Isolation

In a full stack siloed model, isolation is a very straightforward process. When resources run in a dedicated model, you have a natural set of constructs that allow you to ensure that one tenant cannot access the resources of another tenant. However, when you start using pooled resources, your isolation story tends to get more complicated. How do you isolate a resource that is shared by multiple tenants? How is isolation realized and applied across all the different resource types and infrastructure services that are part of your multi-tenant architecture? In Chapter 10, we'll dig into the strategies that are used to address these isolation nuances. However, it's important to note that, as part of adopting a full stack pool model, you will be faced with a range of new isolation considerations that may influence your design and architecture. The assumption here is that the economies of scale and efficiencies of the pooled model offset any of the added overhead and complexity associated with isolating pooled resources.

Availability and Blast Radius

In many respects, a full stack pool model represents an all-in commitment to a model that places all the tenants of your business into a shared experience. Any outage or issues that were to show up in a full stack pool environment are likely to impact *all* of your customers and could potentially damage the reputation of your SaaS business. There are examples across the industry of SaaS organizations that have had service outages that created a flurry of social media outcry and negative press that had a lasting impact on these businesses.

As you consider adopting a full-stack pool model, you need to understand that you're committing to a higher DevOps, testing, and availability bar that makes every effort to ensure that your system can prevent, detect, and rapidly recover from any potential outage. It's true that every team should have a high bar for availability. However, the risk and impact of any outage in a full stack pool environment demands a greater focus on ensuring that your team can deliver a zero downtime experience. This includes adopting best-of-breed CI/CD strategies that allow you to release and rollback new features on a regular basis without impacting the stability of your solution.

Generally, you'll see full stack pool teams leaning into fault tolerant strategies that allow their microservices and components to limit the blast radius of localized issues. Here, you'll see greater application of

asynchronous interactions between services, fallback strategies, and bulkhead patterns being used to localize and manage potential microservice outages. Operational tooling that can proactively identify and apply policies here is also essential in a full stack pool environment.

It's worth noting that these strategies apply to any and all SaaS deployment models. However, the impact of getting this wrong in a full stack pool environment can be much more significant for a SaaS business.

Noisy Neighbor

Full stack pooled environments rely on carefully orchestrated scaling policies that ensure that your system will effectively add and remove capacity based on the consumption activity of your tenants. The shifting needs of tenants along with the potential influx of new tenants means that the scaling policies you have today may not apply tomorrow. While teams can take measures to try and anticipate these tenant activity trends, many teams find themselves over-provisioning resources that create the cushion needed to handle the spikes that may not be effectively addressed through your scaling strategies.

Every multi-tenant system must employ strategies that will allow them to anticipate spikes and address what is referred to as noisy neighbor

conditions. However, noisy neighbor takes on added weight in full stack pooled environments. Here, where essentially everything is shared, the potential for noisy neighbor conditions is much higher. You must be especially careful with the sizing and scaling profile of your resources since everything must be able to react successfully to shifts in tenant consumption activity. This means accounting for and building defensive tactics to ensure that one tenant isn't saturating your system and impacting the experience of other tenants.

Cost Attribution

Associating and tracking costs at the tenant level is a much more challenging proposition in a full stack pooled environment. While many environments give you tools to map tenants to specific infrastructure resources, they don't typically support mechanisms that allow you to attribute consumption to the individual tenants that are consuming a shared resource. For example, if three tenants are consuming a compute resource in a multi-tenant setting, I won't typically have access to tools or mechanisms that would let me determine what percentage of that resource was consumed by each tenant at a given moment in time. We'll get into this challenge in more detail in Chapter 14. The main point here is that, with the efficiency of a full stack pooled model also comes new challenges around understanding the cost footprint of individual tenants.

Operational Simplification

I've talked about this need for a single pane of glass that provides a unified operational and management view of your multi-tenant environment. Building this operational experience requires teams to ingest metrics, logs, and other data that can be surfaced in this centralized experience. Creating these operational experiences in a full stack pooled environment tends to be a simpler experience. Here, where all tenants are running in shared infrastructure, I can more easily assemble an aggregate view of my multi-tenant environment. There's no need to connect with one-off tenant infrastructure and create paths for each of those tenant-specific resources to publish data to some aggregation mechanism.

Deployment is also simpler in the full stack pooled environment. Releasing a new version of a microservice simply means deploying one instance of that service to the pooled environment. Once it's deployed all tenants are now running on the new version.

A Sample Architecture

As you can imagine, the architecture of a full stack pool environment is pretty straightforward. In fact, on the surface, it doesn't look all that unlike any classic application architecture. [Figure 3-9](#) provides an example of a fully pooled architecture deployed in an AWS architecture.

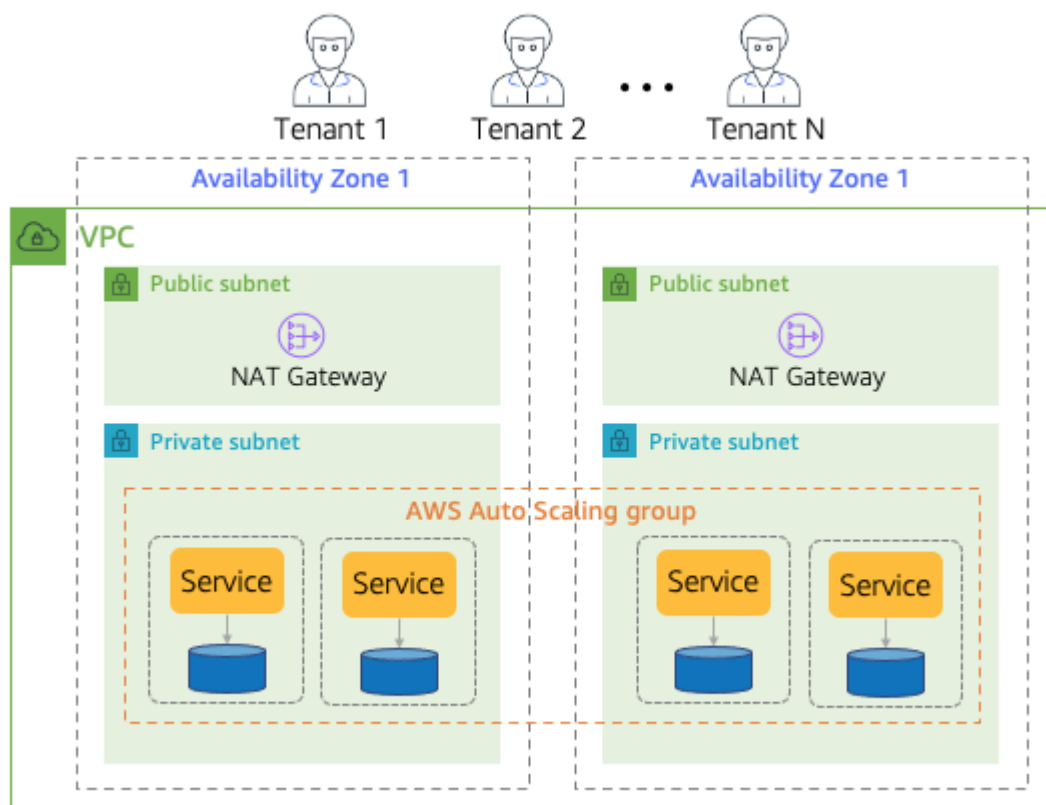


Figure 3-9. A full stack pooled architecture

Here you'll see I've included many of the same constructs that were part of our full stack silo environment. There's a VPC for the network of our environment and it includes two Availability Zones for high availability. Within the VPC there are separate private and public subnets that separate the external and internal view of our resources. And, finally, within the private subnet you'll see the placeholders for the various microservices that deliver the server side functionality of our application. These services have storage that is deployed in a pooled model and their compute is scaled horizontally using an auto-

scaling group. At the top, of course, we also illustrate that this environment is being consumed by multiple tenants.

Now, in looking at this at this level of detail, you'd be hard-pressed to find anything distinctly multi-tenant about this architecture. In reality, this could be the architecture of almost any flavor of application. Multi-tenancy doesn't really show up in a full stack pooled model as some concrete construct. The multi-tenancy of a pooled model is only seen if you look inside the run-time activity that's happening within this environment. Every request that is being sent through this architecture is accompanied by tenant context. The infrastructure and the services must acquire and apply this context as part of every request that is sent through this experience.

Imagine, for example, a scenario where Tenant 1 makes a request to fetch an item from storage. To process that request, your multi-tenant services will need to extract the tenant context and use it to determine which items within the pooled storage are associated with Tenant 1. As I move through the upcoming chapters, you'll see how this context ends up having a profound influence on the implementation and deployment of these services. For now, though, the key here is to understand that a full stack pooled model relies more on its run-time ability to share resources and apply tenant context where needed.

This architecture represents just one flavor of a full stack pooled model. Each technology stack (containers, serverless, relational storage, NoSQL storage, queues) can influence the footprint of the full stack pooled environment. The spirit of full stack pool remains the same across most of these experiences. Whether you're in a Kubernetes cluster or a VPC, the basic idea here is that the resources in that environment will be pooled and will need to scale based on the collective load of all tenants.

A Hybrid Full Stack Deployment Model

So far, I've mostly presented full stack silo and full stack pool deployment models as two separate approaches to the full stack problem. It's fair to think of these two models as addressing a somewhat opposing set of needs and almost view them as being mutually exclusive. However, if you step back and overlay market and business realities on this problem, you'll see how some organizations may see value in supporting both of these models.

[Figure 3-10](#) provides a view of a sample hybrid full stack deployment model. Here we have the same concepts we covered with full stack silo and pool deployment models sitting side-by-side.

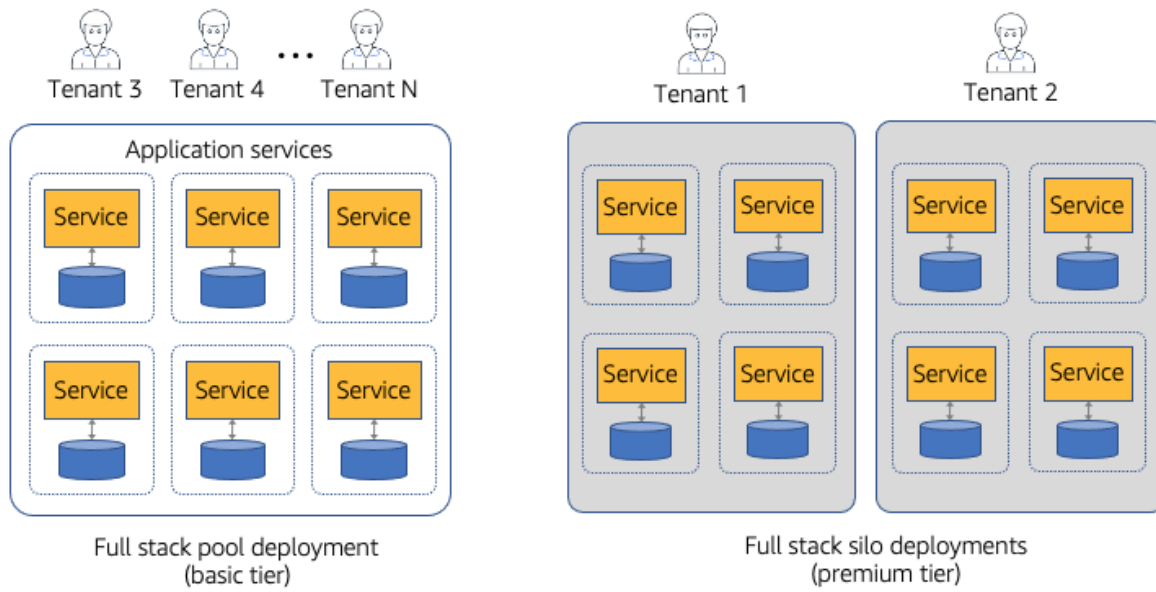


Figure 3-10. A hybrid deployment model

So, why both models? What would motivate adopting this approach? Well, imagine you've built your SaaS business and you started out offering all customers a full stack pooled model (shown on the left here). Then, somewhere along the way, you ran into a customer that was uncomfortable running in a pooled model. They may have noisy neighbor concerns. They may be worried about some compliance issues. Now, you're not necessarily going to cave to every customer that has this pushback. That would undermine much of what you're trying to achieve as a SaaS business. Instead, you're going to make efforts to help customers understand the security, isolation, and strategies you've adopted to address their needs. This is always part of the job of selling a SaaS solution. At the same time, there may be rare conditions when you might be open to offering a customer their

own full stack siloed environment. This could be driven by a strategic opportunity or it may be that some customer is willing to write a large check that could justify offering a full stack silo.

In [Figure 3-10](#), you can see how the hybrid full stack deployment model lets you create a blended approach to this problem. On the left-hand side of this diagram is an instance of a full stack pooled environment. This environment supports that bulk of your customers and we label these tenants, in this example, as belonging to your basic tier experience.

Now, for the tenants that demanded a more siloed experience, I have created a new premium tier that allows tenants to have a full stack silo environment. Here we have two full stack siloed tenants that are running their own stacks. The assumption here (for this example) is that these tenants are connected to a premium tier strategy that has a separate pricing model.

For this model to be viable, you must apply constraints to the number of tenants that are allowed to operate in a full stack silo model. If the ratio of siloed tenants becomes too high, this can undermine your entire SaaS experience.

The Mixed-Mode Deployment Model

To this point, I've focused heavily on the full stack models. While it's tempting to view multi-tenant deployments through these more coarse-grained models, the reality is that many systems rely on a much more fine-grained approach to multi-tenancy, making silo and pool choices across the entire surface of their SaaS environment. This is where we look more at what I refer to as a mixed mode deployment model.

With mixed mode deployments, you're not dealing with the heavy absolutes that come with full stack models. Instead, mixed mode allows us to look at the workloads within our SaaS environment and determine how each of the different services and resources within your solution should be deployed to meet the specific requirements of a given use case.

Let's take a simple example. Imagine I have two services in my e-commerce solution. I have an order service that has challenging throughput requirements that are prone to noisy neighbor problems. This same service also stores data that is going to grow significantly and has strict compliance requirements that are hard to support in a pooled model. I also have a ratings service that is used to manage product ratings. It doesn't really face any significant throughput challenges and can easily scale to handle the needs of tenants—even when a single single tenant might be putting a disproportionate load

on the service. Its storage is also relatively small and contains data that isn't part of the system's compliance profile.

In this scenario, I can step back and consider these specific parameters to arrive at a deployment strategy that best serves the needs of these services. Here, I might choose to make both the compute and the storage of my order service siloed and the compute and storage of my rating service pooled. There might even be cases where the individual layers of a service could have separate silo/pool strategies. This is the basic point I was making when I was first introducing the notion of silo and pool at the outset of this chapter.

Equipped with this more granular approach to silo and pool strategies, you can now imagine how this might yield much more diverse deployment models. Consider a scenario where you might use this strategy in combination with a tiering model to define your multi-tenant deployment footprint.

The image in [Figure 3-11](#) provides a conceptual view of how you might employ a mixed mode deployment model in your SaaS environment. I've shown a variety of different deployment experiences spanning the basic and premium tier tenants.

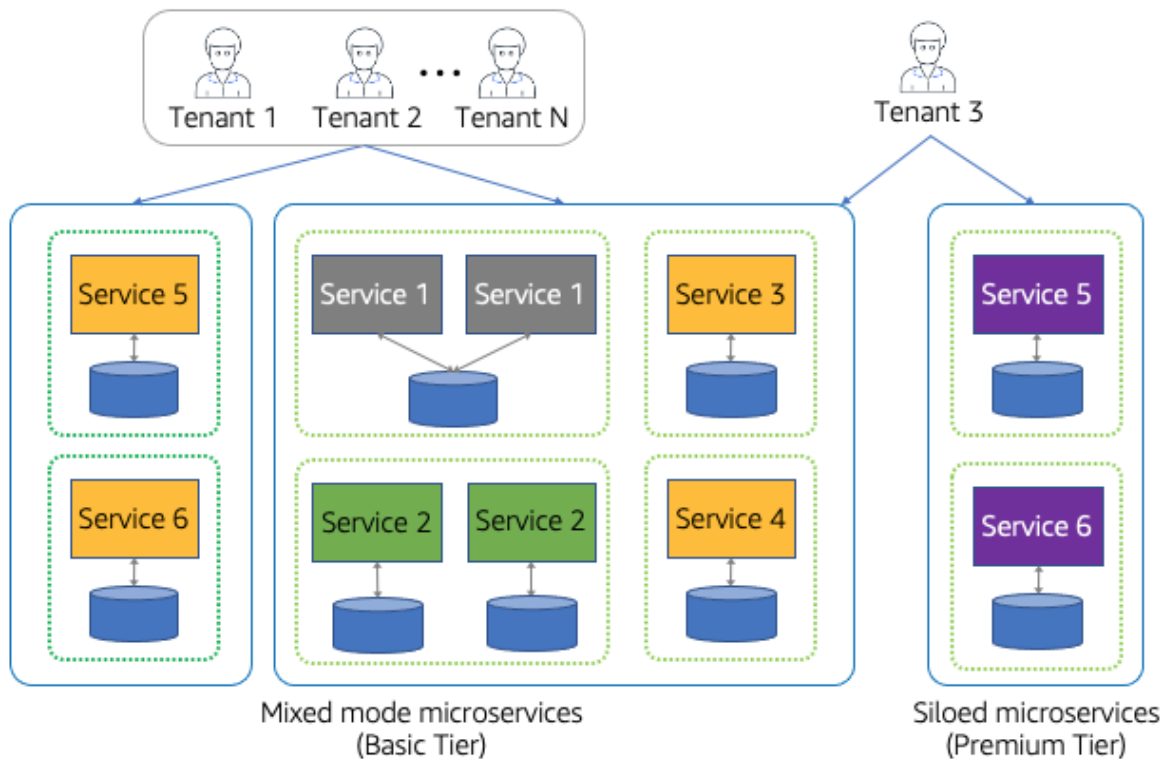


Figure 3-11. A mixed mode deployment model

On the left-hand side of this image, we have our basic tier services. These services cover all the functionality that is needed for our SaaS environment. However, you'll note that they are deployed in different silo/pool configurations. Service 1, for example, has siloed compute and pooled storage. Meanwhile, Service 2 has siloed compute and siloed storage. Services 3-6 are all pooled compute and pooled storage. The idea here is that I've looked across the needs of my pooled tenants and identified, on a service-by-service basis, which silo/pool strategy will best fit the needs of that service. The optimizations that have been introduced here were created as baseline strategies that were core to the experience of *any* tenant using the system.

Now, where tiers do come into play is when you look at what I've done with the premium tier tenants. Here, you'll notice that Services 5 and 6 are deployed in the basic tier and they're also deployed separately for a premium tier tenant. The thought was that, for these services, the business determined that offering these services in a dedicated model would represent value that could distinguish the experience of the system's premium tier. So, for each premium tier tenant, we'll create new deployments of Service 5 and 6 to support the tiering requirements of our tenants. In this particular example, Tenant 3 is a premium tier tenant that consumes a mix of the services on the left and these dedicated instances of Services 5 and 6 on the right.

Approaching your deployment model in this more granular fashion provides a much higher degree of flexibility to you as the architect and to the business. By supporting silo and pool models at all layers, you have the option to compose the right blend of experiences to meet the tenant, operational, and other factors that might emerge throughout the life of your solution. If you have a pooled microservice with performance issues that are creating noisy neighbor challenges, you could silo the compute and/or storage of the service to address this problem. If your business wants to offer some parts of your system in a dedicated model to enable new tiering strategies, you are better positioned to make this shift.

This mixed mode deployment model, in my opinion, often represents a compelling option for many multi-tenant builders. It allows them to move away from having to approach problems purely through the lens of full stack solutions that don't always align with the needs of the business. Yes, there will always be solutions that use the full stack model. For some SaaS providers, this will be the only way to meet the demands of their market/customers. However, there are also cases where you can use the strengths of the mixed mode deployment model to address this need without moving everything into a full stack silo. If you can just move specific services into the silo and keep some lower profile services in the pool, that could still represent a solid win for the business.

The Pod Deployment Model

So far, I've mostly looked at deployment models through the lens of how you can represent the application of siloed and pooled concepts. We explored coarse- and fine-grained ways to apply the silo and pool model across your SaaS environment. To be complete, I also need to step out of the silo/pool focus and think about how an application might need to support a variation of deployment that might be shaped more by where it needs to land, how it deals with environmental constraints, and how it might need to morph to support the scale and

reach of your SaaS business. This is where the pod deployment model comes into the picture.

When I talk about pods here, I'm talking about how you might group a collection of tenants into some unit of deployment. The idea here is that I may have some technical, operational, compliance, scale, or business motivation that pushes me toward a model where I put tenants into individual pods and these pods become a unit of deployment, management, and operation for my SaaS business. [Figure 3-12](#) provides a conceptual view of a pod deployment.

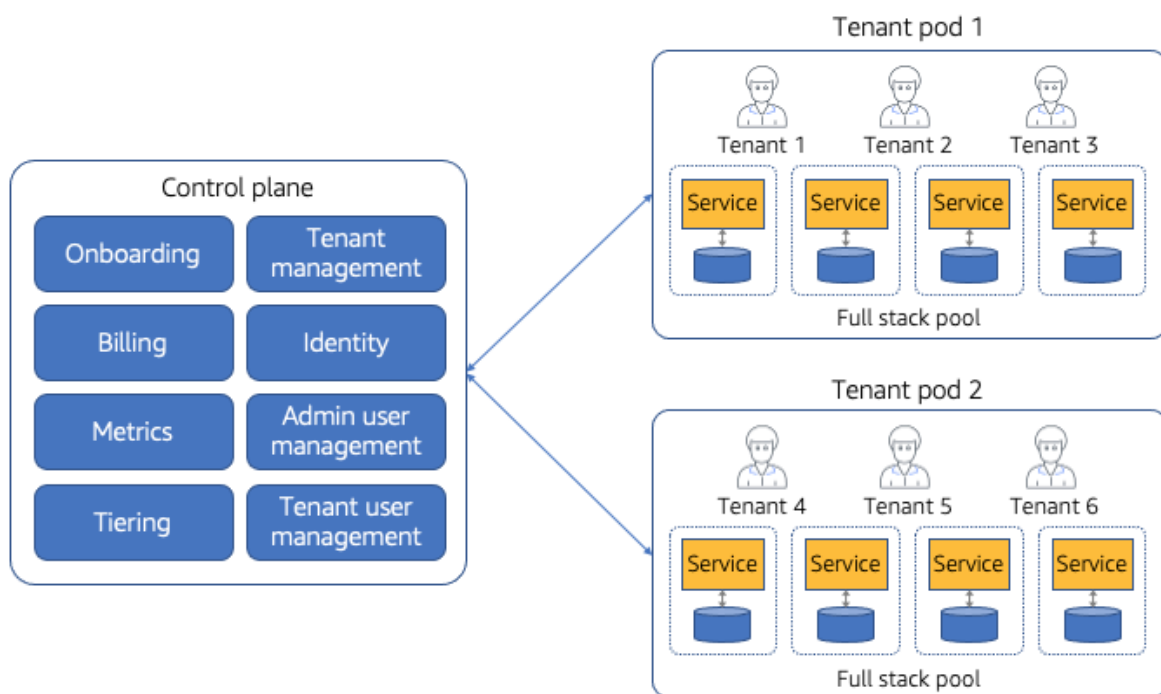


Figure 3-12. A pod deployment model

In this pod deployment model, you'll notice that we have the same centralized control plane on the left-hand side of this experience. Now, however, on the right-hand side, I have included individual pods that are used to represent self-contained environments that support the workload of one or more tenants. In this example, I have Tenants 1-3 in pod 1 and Tenants 4-6 in pod 2.

These separate pods bring a degree of complexity to a SaaS environment, requiring your control to build in the mechanisms to support this distribution model. How tenants are onboarded, for example, must consider which pod a given tenant will land in. Your management and operations must also become pod aware, providing insights into the health and activity of each pod.

There are a number of factors that could drive the adoption of a pod-based delivery model. Imagine, for example, having a full stack pooled model running in the cloud that, at a certain number of tenants, begins to exceed infrastructure limits of specific services. In this scenario, your only option might be to create separate cloud accounts that host different groups of tenants to work around these constraints. This could also be driven by a need for deploying a SaaS product into multiple geographies where the requirements of that geography or performance considerations could tip you toward a pod-based deployment model where different geographies might be running different pods.

Some teams may also use pods as an isolation strategy where there's an effort to reduce cross-tenant impacts. This can be motivated by a need for greater protections from noisy neighbor conditions. Or, it might play a role in the security and availability story of a SaaS provider.

If you choose to adopt a pod model, you'll want to consider how this will influence the agility of your business. Adopting a pod model means committing to absorbing the extra complexity and automation that allows you to support and manage pods without having any one-off mechanisms for individual pods. To scale successfully, the configuration and deployment of these pods must all be automated through your control plane. If some change is required to pods, that change is applied universally to all pods. This is the mirror of the mindset I outlined with full stack silo environments. The pod cannot be viewed as an opportunity to enable targeted customization for individual tenants.

One dynamic that comes with pods is this idea of placing tenants into pods and potentially viewing membership within a pod as something can be shifted during the life of a tenant. Some organizations may have distinct pod configurations that might be optimized around the profile of a tenant. So, if a tenant's profile somehow changes and their sizing or consumption patterns aren't aligned with those of a given pod, you could consider moving that tenant to another pod. How-

ever, this would come with some heavy lifting to get the entire footprint of the tenant transferred to another pod. Certainly this would not be a daily exercise, but is something that some SaaS teams support—especially those that have pods that are tuned to a specific experience.

While pods have a clear place in the deployment model discussion, it's important not to see pods as a shortcut for dealing with multi-tenant challenges. Yes, the pod model can simplify some aspects of scale, deployment, and isolation. At the same time, pods also add complexity and inefficiencies that can undermine the broader value proposition of SaaS. You may not, for example, be able to maximize the alignment between tenant consumption and infrastructure resources in this model. Instead, you may end up with more instances of idle or overprovisioned resources distributed across the collection of pods that your system supports. Imagine an environment where you had 20 pods. This could have a significant impact on the overall infrastructure cost profile and margins of your SaaS business.

Conclusion

This chapter focused on identifying the range of SaaS deployment models that architects must consider when designing a multi-tenant architecture. While some of these models have very different foot-

prints, they all fit within the definition of what it means to be SaaS. This aligns with the fundamental mindset I outlined in Chapter 1, identifying SaaS as a business model that can be realized through multiple architecture models. Here, you should see that—even though I outlined multiple deployment models—they all shared the idea of having a single control plane that enables each environment and its tenant to be deployed, managed, operated, onboarded, and billed through a unified experience. Full stack silo, full stack pool, mixed mode—they all conform with the notion of having all tenants running the same version of a solution and being operated through a single pane of glass.

From looking at these deployment models, it should be clear that there are a number of factors that might push you toward one model or another. Legacy, domain, compliance, scale, cost efficiency, and a host of other business and technical parameters are used to find the deployment model (or combination of deployment models) that best align with the needs of your team and business. It's important to note that the models I covered here represent the core themes while still allowing for the fact that you might adopt some variation of one of these models based on the needs of your organization. As you saw with the hybrid full stack model, it's also possible that your tiering or other considerations might have you supporting multiple models based on the profile of your tenants.

Now that you have a better sense of these foundational models, we can start to dig into the more detailed aspects of building a multi-tenant SaaS solution. I'll start covering the under-the-hood moving parts of the application and control planes, highlighting the services and code that is needed to bring these concepts to life. The first step in that process is to look at multi-tenant identity and onboarding. Identity and onboarding often represent the starting point of any SaaS architecture discussion. They lay the foundation for how we associate tenancy with users and how that tenancy flows through the moving parts of your multi-tenant architecture. As part of looking at identity, I'll also explore tenant onboarding which is directly connected to this identity concept. As each new tenant is onboarded to a system, you must consider how that tenant will be configured and connected to its corresponding identity. Starting here will allow us to explore the path to a SaaS architecture from the outside in.

Chapter 4. Onboarding and Identity

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

Now that you have a sense of the broader multi-tenant terminology and landscape, let’s look at what it means to bring these concepts to life in a working solution. So, the question is: where do you start? So many teams ask me this question. Fortunately, this is one of those areas where I think there’s a pretty uniform answer. To me, whether you’re migrating or greenfield, I’d always point you at onboarding, identity, and the control plane as the starting point of building any multi-tenant architecture. As you’ll see in this chapter, getting these foundational concepts nailed down will directly impact the down-

stream implementation of so many of the moving parts of your multi-tenant architecture. Starting here puts tenancy front-and-center from the outset of your build, forcing all the other dimensions of your multi-tenant environment to directly address and support tenant context. This spans tenant isolation, data partitioning, tiering, billing, and almost every other aspect of your multi-tenant environment. It also puts greater emphasis on building out and maturing your control plane on day one of your build, providing the early foundations for adding and building the remaining pieces of the control plane.

For this chapter, we'll walk through the logical progression that you'd typically follow to bring these concepts to life. While our focus here is on onboarding and identity, we have to start by looking at what we need to do to get the control plane provisioned and up-and-running. It's this control plane, if you recall, that will run the services that will support our onboarding and identity experience. So, we'll begin our path by digging into what it means to create a baseline environment. Here, we'll explore the common infrastructure, services, and resources that are provisioned and configured to run the baseline of our SaaS environment. This will include setting up all the high availability networking constructs, shared tenant resources, identity, and the different microservices that will support onboarding, identity, and user management.

Once we have the baseline environment up-and-running, we'll start to explore the details of the onboarding implementation. I'll highlight the different patterns, strategies, and considerations that shape your onboarding experience. This will include looking at how the onboarding microservice orchestrates and manages all the different aspects of the tenant onboarding process. This will give a better sense of the critical role onboarding plays within your SaaS architecture. You'll see, in greater detail, how onboarding stitches together so many of the multi-tenant constructs that are used to compose your SaaS experience.

As part of this exploration, we'll also get more into the details of how we bind individual users to tenants to arrive at this notion of tenant context that was discussed in Chapter 1. This will include going deeper into the specific identity mechanisms that allow us to shape how tenants are authenticated and how the tenant context from this authentication flows through all the backend services of a SaaS application. We'll see how this context ends up shaping and influencing how teams build and manage the multi-tenant features of their SaaS architecture.

The overall objective is to outline the core moving parts associated with creating a SaaS environment, providing an end-to-end look at how all the pieces come together to create the first foundational elements of a multi-tenant architecture. We'll stay at a somewhat conceptual level

at this stage, highlighting the key strategies, patterns, and considerations without getting too close to the specifics of any technology. Understanding these core concepts will equip you with the insights that will shape how you approach many of the multi-tenant topics we'll be covering in subsequent chapters. This will be especially valuable when we begin to wander into specific implementations of these strategies in Chapters 15 and 16.

Creating a Baseline Environment

Now, to get started on this journey, I want to approach onboarding and identity as if we are starting from scratch. This should give you a better sense of how you might approach implementing these strategies from the ground up. That means we have to take a step back from the specifics of onboarding and identity and first think about what foundational pieces we have to put in place before we can start onboarding tenants. If you recall, the services that support onboarding run inside of the control plane. So, we need to start by putting in place all the bits that are needed to run all the control plane microservices that support onboarding and identity.

This creation of our infrastructure, its dependent resources, and the control plane is what I refer to as creating a baseline environment. We essentially need to create the scripts and the automation that will

allow us to spin up all constructs that are needed to host our SaaS environment. While our goal is to get onboarding and identity up-and-running, the scope of the baseline environment includes all the resources that would be one-time provisioned to set up our multi-tenant environment before we start onboarding. This means we'll be setting up some resources that go beyond the scope of tenant onboarding and identity. We won't focus on those other bits right now, but it's important to note that a baseline environment would be inclusive of all of these concepts.

The actual creation of our baseline environment is achieved through a classic DevOps model, using infrastructure automation tooling to create, configure, and deploy all the assets that are required by our baseline environment. [Figure 4-1](#) provides a highly conceptualized view of this experience.

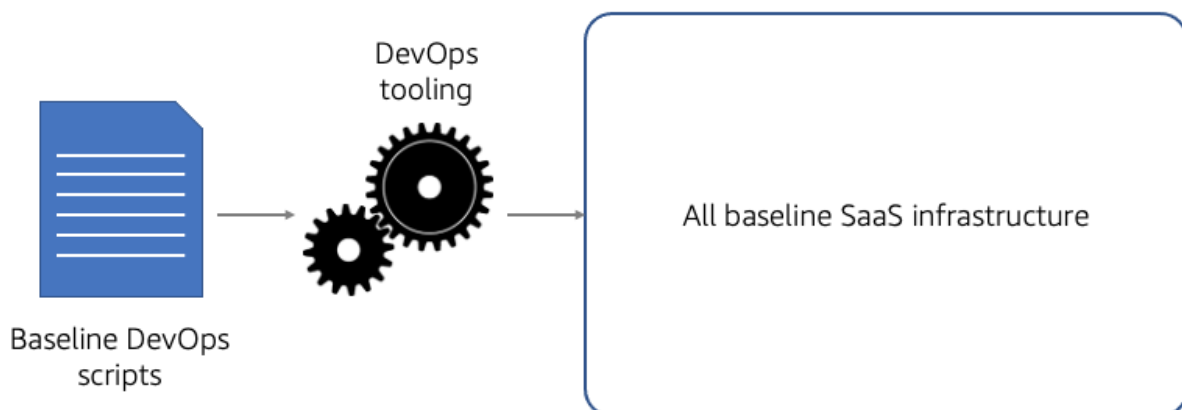


Figure 4-1. Automating creation of a baseline environment

The basic idea here is that you'll pick the DevOps tool(s) that fits your environment and create a single, repeatable automation model that can configure everything you need to get your environment moved to a state where it can begin onboarding tenants.

Of course, what's actually in your baseline environment will vary wildly based on the nature of the specific technology stack you're using for your SaaS solution. A kubernetes stack, for example, could look very different from a serverless stack. The nuances of different cloud providers would also influence the provisioning process. We'll certainly look at more specific examples to see how they land, but, for now, we want to come up a level and just focus on what needs to get provisioned in this step to prepare our system to begin onboarding tenants.

A Conceptual Baseline Environment

To get a better sense of what's in this baseline environment, let's look at a sample of what might get provisioned, configured, and deployed to bring your baseline environment to life. In [Figure 4-2](#) you'll see that I've assembled a conceptual view of the components and infrastructure that might get created in a baseline environment. The goal here was to represent some of the core baseline infrastructure concepts without getting too lost in the details of any specific technology.

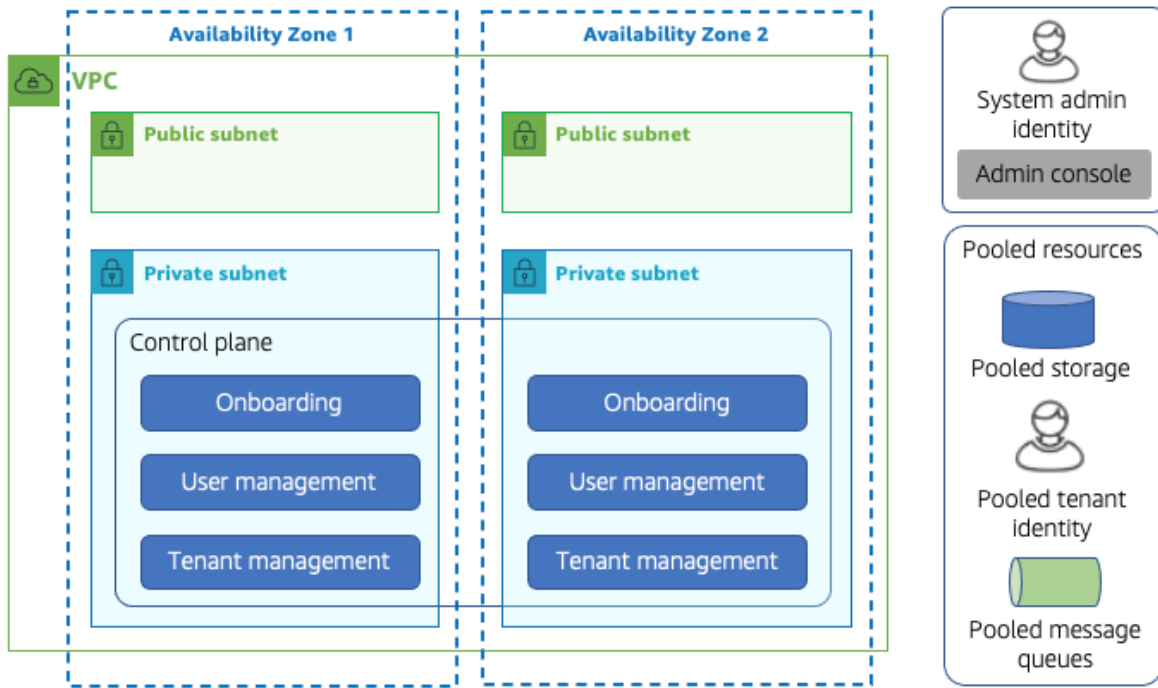


Figure 4-2. Provisioning a baseline environment

In the middle of this diagram, you'll see that I've created the foundational networking infrastructure that's needed to host my multi-tenant SaaS environment. Here I've just grabbed some common AWS networking constructs (a VPC, Availability Zones, and some subnets) to represent the high availability network that will host my SaaS environment. These same networking constructs could be mapped to any number of different technologies. The key, at this stage, is just to focus on the fact that the configuration and setup of this baseline environment will require you to provision and configure all the core networking constructs that will be used by your control plane and, potentially, your tenants.

Within this network, I've also shown the deployment of the control plane. Since the control plane is shared by all tenants, it can be configured and deployed as part of the provisioning of your baseline environment. The control plane must also be in place for us to begin onboarding tenants and establishing their identity. Here, to simplify matters, I included a sampling of a few services. In reality the list of control plane services would include a much broader range of services. We'll see those services in more detail when we start digging into more concrete solutions.

On the bottom right-hand side of the diagram, you'll also see a collection of pooled resources. The items here represent the conceptual placeholders for any resources that might be shared by tenants. Generally, if you have pooled resources that will be shared by all tenants, you can provision them during the setup of your baseline environment (since they won't need to be created during the onboarding process). Storage often provides a good example here. Imagine having a pooled database for some microservice in your solution. If it's pooled, it could be created when the baseline environment is provisioned. You'll also see the setup of a shared identity construct and a pooled message queue. Again, these are just here to highlight the fact that you'll want to consider whether these should be provisioned during the setup of your baseline environment. I'll get into some of the trade offs here when we go deeper into the tenant onboarding experience.

Finally, on the top right, I've shown placeholders for the system admin identity and administration console. This represents the users that are logging into the specific tooling that you've created to support, update, and generally manage the state of your multi-tenant architecture. I refer to this targeted tooling as your system admin console. It's this console that serves as the single plane of glass for your SaaS environment, providing your team with a purpose built collection of features and capabilities that are essential to operating your multi-tenant environment. Yes, this tool will be used in combination with other off-the-shelf solutions that provide more generalized functionality. Even with these other tools, most SaaS teams require their own custom admin application that can address the specific multi-tenant needs of their environment.

[Figure 4-3](#) provides a snapshot of a simple SaaS administration console application that helps make this concept a bit more concrete. It's through this application that you'll have access to all core information about your SaaS solution. You'll be able to monitor the status of onboarding tenants, activate/deactivate tenants, manage tenant policies, view tenant/tier metrics, and any other functionality that's needed to manage and operate your SaaS solution. This application must get configured and deployed as part of the setup of your baseline environment.

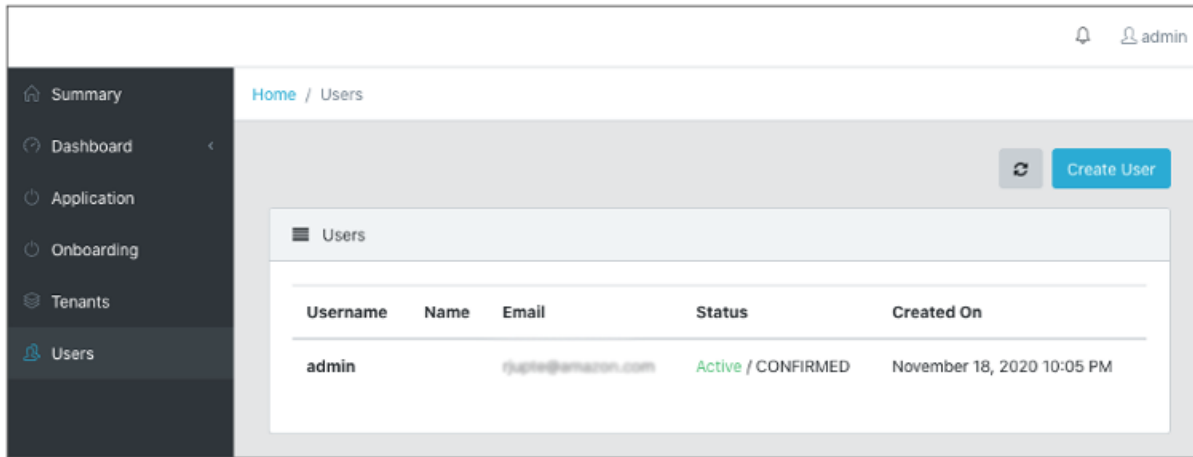


Figure 4-3. Creating and deploying a system admin console

It's worth noting that some teams tend to under invest in their admin consoles, deferring to ready made solutions in favor of building something themselves. Generally, this tradeoff rarely seems worth it. While you might be able to use third-party solutions to compose a console experience, there are specific operations, insights, and configuration options that can only be addressed effectively through the creation of a targeted experience.

Creating and Managing System Admin Identities

As part of setting up your baseline environment and configuring your administration application, you'll see that your provisioning process must also set up your system admin identity model. Each time you trigger the creation of a baseline environment, you'll be required to provide the profile of the initial administrative user. Creation of this identity is entirely separate from the creation of a tenant identity. This

also means you'll need to have a completely separate authentication experience to allow these system admin users to access the admin console or any command line tooling you might be using to manage your multi-tenant environment.

To support this system admin identity, you'll need to have some identity provider that owns and authenticates these users. The identity provider you use here could be the same identity provider that will be used for your tenant identities. Or, it could be a separate identity provider that is used as part of a more global enterprise administration strategy. Regardless of which identity provider you use, the basic mechanics of introducing a system admin identity are going to be very similar.

The key takeaway here is that you'll need some steps on your baseline provisioning automation to create and configure your system administration identity model. This automation will include the creation/configuration of the identity provider along with the creation of the initial system admin users. Once that user is set up, you should be able to use this identity to access your system admin console. Once you're made it into the system admin console, you'll be able to manage and create more system admin users.

The example in [Figure 4-3](#) happens to show a view of system admin user management. Here, I've accessed and authenticated into the

console after provisioning my environment. I can now use this same page to create and manage other system admin users.

Triggering Onboarding from the Admin Console

Once you've established your system admin user and you have your admin console up-and-running, you actually have all the pieces in place to create and onboard tenants. Now, in the final version of your offering, your onboarding could be invoked as part of some self-service experience or it could be driven by some internal process. Obviously, if this is an internally driven process, then you'll want your onboarding functionality to be managed through your system admin console. This would mean having some operation within your console that collected all the data needed for a new tenant before invoking the onboarding operation.

Some teams find lots of value in being able to onboard tenants from within the system admin console. Even if onboarding, for example, were to eventually be a self-service model, you could still have the ability to test and validate your onboarding experience from the admin console. This can be especially helpful to teams that are validating and testing the onboarding experience of your application.

Control Plane Provisioning Options

In [Figure 4-2](#), I showed the control plane being deployed into the same baseline infrastructure where your tenants would also land. This is a perfectly valid option. However, it's worth noting that how and where this control plane is placed can vary based on the needs of your environment and the technology stack that's being used for your multi-tenant architecture. In Kubernetes, for example, I could have a separate namespace for the control plane, placing my tenant environments alongside the control plane within the same cluster and networking infrastructure. I could also choose to land the control plane in a completely separate set infrastructure that is dedicated to the control plane.

[Figure 4-4](#) provides a conceptual view of these two options. On the left, you'll see the shared control plane model where the control is deployed into the same environment with your tenant infrastructure. And, on the right, you'll see an approach where the control gets its own dedicated control plane environment. Here, the tenants are running in a completely separate network or cluster that draws a harder line between the control and application planes.

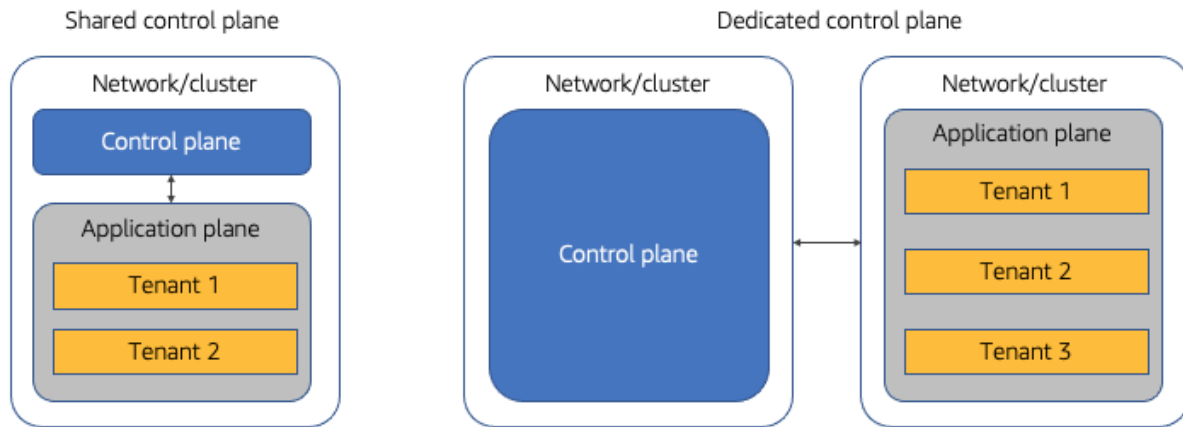


Figure 4-4. Picking a control plane deployment model

The tradeoffs of these two choices are pretty straightforward. You might choose to have a dedicated control plane environment if you want to scale, manage, and operate these environments completely independently. Compliance could represent another factor that might tip you toward this approach. Of course, putting the control in the same environment with the application plane does simplify things a bit. It reduces the number of moving parts you have to manage, configure, and provision. It might also reduce your cost footprint. If you do opt for the dedicated model, you'll need to decide how you'll integrate these separate constructs to allow the control plane to interact with your application plane.

Your technology stack choices might also have some influence on how you deploy your control plane. Some teams, for example, might opt for different technology stacks for the control and application planes. I might, for example, choose serverless for the control plane

and containers for the application plane. This might steer you more toward a dedicated control plane model.

The Onboarding Experience

Now that our baseline environment is provisioned and configured, we can turn our attention to the onboarding of tenants. It's through onboarding that you'll find that you're establishing and exercising some of the most foundational elements of a multi-tenant architecture. In fact, when working with greenfield or migrating SaaS customers, I always suggest that they focus their initial attention on the onboarding process.

Starting here forces teams to answer many of the hard questions that will influence and shape the rest of their SaaS architecture. Onboarding isn't just about creating a tenant. It's about creating and configuring all the moving parts of your infrastructure that are needed to support that new tenant. In some cases, that might be a lightweight exercise and, in others, it might require a significant amount of code to orchestrate each step in the onboarding process. How your tenants are tiered, how they authenticate, how their policies are managed, how their isolation is configured, how they're routed—these are all areas that are touched by the onboarding experience of your multi-tenant environment.

Onboarding is Part of Your Service

Many teams fall into the trap of viewing onboarding as something that gets bolted onto their system after it's built. They'll create placeholders and workarounds to simulate the onboarding experience with the idea that they can "make it real" later in the process. This comes back to the discussion of comparing a service to a product. In a SaaS environment, onboarding isn't viewed as some script or automation that's somehow outside of the scope of your offering. Instead, it is one of the most fundamental components of your SaaS experience and getting it right should be key to any team that is building a multi-tenant solution.

Onboarding sits right in the middle of both your business and technical priorities. For the business, the experience each customer has with onboarding can have a profound impact on the border success of business. How seamless, efficient, and reliable this process is will have a direct impact on the experience and perception of the customers consuming your product. It is, in many respects, your chance to make a positive first impression. The onboarding experience is also directly connected to the notion of time-to-value which looks at how long it takes a customer to move from sign-up to actual productivity and value within your SaaS offering. Any added friction that shows up here is going to impact the impression you make as a ser-

vice and could, potentially, influence your ability to move customers from adopters to promoters.

Onboarding is also where the deployment, identity, routing, and tiering strategies are put into action. How tenants are siloed and pooled, for example, will need to be expressed and realized directly through your onboarding experience. How and where you authenticate tenants will be configured and applied as part of onboarding. How your tenants are contextually routed based on their tier and deployment model will be configured within the scope of onboarding. So many of these key multi-tenant design choices that you make in your SaaS architecture are ultimately expressed and brought to life through the onboarding process of your system. In many respects, your onboarding configuration, automation, and deployment code will be at the epicenter of realizing the multi-tenant strategies that you adopt for your SaaS environment.

The amount of effort and code that goes into automation may come as a surprise to some teams. It's not uncommon for SaaS teams to underestimate the level of effort and investment that comes with building a robust onboarding experience. In reality, onboarding represents one of the most fundamental elements of a multi-tenant environment. It's through onboarding that you can achieve the operational and agility goals that are essential to a SaaS business.

Self-Service vs. Internal Onboarding

So far, this discussion of onboarding may seem like it's mostly describing mechanisms that are used by organizations that rely on a self-service tenant registration experience. Many of us have signed up for countless B2C SaaS offerings where we filled out some form, submitted our information, and started using some SaaS service.

While this classic mode of onboarding is within our scope, we must also consider scenarios where our onboarding process may not support a self-service model. Imagine, for example, some B2B SaaS provider that only onboards after you've reached a deal and agreed to onboard them to your system. These SaaS vendors may only have some internally managed onboarding experience.

My point here is that onboarding has no binding to a particular experience. You might have self-service onboarding or you might use internal onboarding. Every SaaS solution, regardless of how it presents its onboarding experience, must still lean into the same set of values. To me, the bar for self-service and internally managed onboarding processes is the same. Both of these approaches should be creating a fully automated, repeatable, low-friction onboarding process that focuses on maximizing a customer's time to value. Yes, your internal process might be run by someone in operations. This does not mean that you'd expect less automation, scale, or durability from that onboarding process.

For any SaaS system that I build, I want to be sure that I'm treating this onboarding experience as a key part of my system that is at the center of ensuring that I have a consistent, repeatable, automated onboarding mechanism that ensures that each new tenant will be introduced without requiring any manual processes and/or one-off configuration

The Fundamental Parts of Onboarding

Now that you have a better sense of the onboarding importance, let's shift our focus more toward the details of the underlying components of an onboarding experience. While there are lots of details within the implementation of the onboarding process, my goal at this stage is to give you a top-level view of the core components of this process and outline the guiding principles that typically shape this experience.

[Figure 4-5](#) provides a conceptual view of the moving parts of a multi-tenant onboarding experience. On the left you'll see the illustration of the two common patterns that could be used to drive an onboarding process. First, I've shown a tenant administrator that is onboarding through some self-service sign-up process, presumably a web application that allows the tenant to submit their information, select a plan, and provide whatever configuration information is needed to establish themselves as a new tenant in the system. I've also shown a second onboarding flow here that, in this example, is initiated by a system ad-

administrator. This represents some internal role at the SaaS provider using an administration console (or some other tooling) to enter the onboarding data for a new tenant and triggering the onboarding process. For this example, I included both of these onboarding paths. However, in most instances, a SaaS organization will support one of these two approaches. I only showed both here to drive home the idea that onboarding, regardless of its entry point, is meant to be a fully automated process for either of these two use cases.

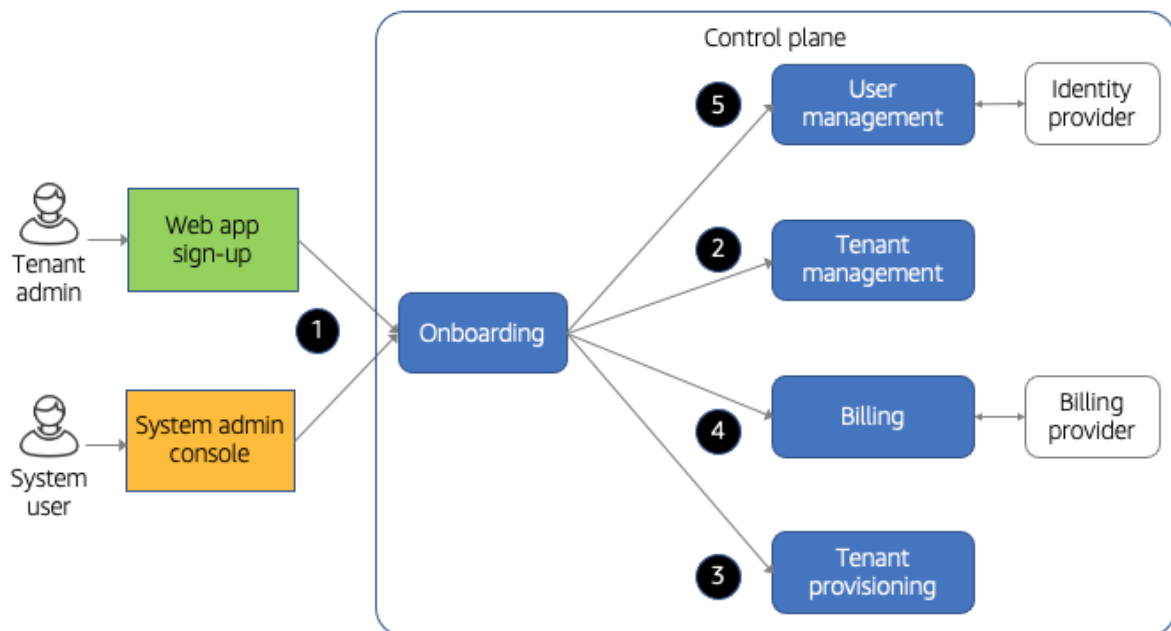


Figure 4-5. The fundamentals of tenant onboarding

For these onboarding paths, you'll see that they both send an onboarding request to the Onboarding service (step 1). For onboarding, I generally prefer to have a single onboarding service that can own all

the orchestration of onboarding. This service owns the full lifecycle of the onboarding process, managing and ensuring that all steps in the process are completed successfully. This is especially important since some aspects of an onboarding may run asynchronously and/or have dependencies on third party integrations that could have availability issues.

The onboarding process then calls a series of distributed services that are used to create and configure the tenant's settings and supporting infrastructure. The sequencing of this onboarding flow can vary based on the nature of your SaaS application. Generally, the goal here is to create and configure all of the required tenant assets before making the tenant active and/or notifying the tenant admin user that their account is active.

While there are multiple ways to implement this onboarding flow, you'll need to start with creating a tenant identifier. In our example, this tenant identifier will be created by sending a create tenant request to the Tenant Management service (step 2), passing in all the information about our tenant (company name, identity configuration, tier, and so on). It will also generate the unique identifier that will be associated with our tenant. Teams will often use a globally unique identifier (GUID) as the value for their tenant identifier, avoiding the inclusion of any attributes that might be connected to the name or other identifying information about the tenant. This prevents anyone

from being able to connect a tenant with a given identifier. This tenant also is created with some notion of an “active” status that manages the current state of a tenant. In this case, where we’re onboarding, the active state will initially be set to false. Once this tenant is created, you’ll now have a tenant identifier that can be used across the rest of the onboarding experience. I’ll get into more detail about the Tenant Management service and its role within the control plane in Chapter 5.

The next step in our tenant onboarding example will involve the provisioning of any tenant resources that are required (step 3). This provisioning step can, for some multi-tenant architectures, represent one of the most significant pieces of your onboarding implementation. For a full stack silo deployment, for example, this could mean provisioning a completely new collection of infrastructure and application services. In contrast, a full stack pooled environment might require minimal infrastructure provisioning and configuration.

As I dig into more working examples, you may be surprised to find out how much code/script/automation is devoted to this onboarding experience. In fact, this is often an area where SaaS systems blur the DevOps boundaries. While, in traditional environments, much of the DevOps lifecycle is focused on provisioning and updating your baseline infrastructure, SaaS environments may rely on the execution of DevOps code during the onboarding of each individual tenant. Essential-

ly, your system may be provisioning and configuring new infrastructure at run-time to process the creation of siloed tenant infrastructure. As you can imagine, this brings new considerations and mindsets to how you organize and build the overall DevOps footprint of your multi-tenant solution. For some, this represents a new mental model and new approaches to the tooling that is used to provision tenant environments.

At this stage, we have a tenant created and our tenant resources are provisioned. Now we can add this new tenant to the billing system (step 4). This is essentially where you'll provide information to the billing system that identifies the new tenant and any information that's needed to characterize the billing model that should be applied to this particular tenant. The assumption here is that, in advance of onboarding a new tenant, you've configured and set up the different tiers or billing plans that determine the overall pricing model of your solution. Then, during onboarding, your Billing service will correlate the tenant's onboarding profile with the appropriate (pre-configured) billing plan.

You'll notice that [Figure 4-5](#) calls out a separate billing provider. The idea here is that your Billing service will manage and orchestrate any integration you might have with your billing system. In many instances, this billing provider may be supported by a third-party system. It's in these cases where you may see value in putting a sepa-

rate Billing service between your onboarding process and the billing provider, allowing you to manage any unique considerations that might be required to support a given billing provider. In other instances, you might directly integrate with the billing provider from your Onboarding service. It's also worth noting that some SaaS companies will use an internal billing system. Even in this scenario, you'd still want your onboarding process to follow a similar pattern of integration. There's lots more about billing to consider (outside the scope of onboarding). I'll get more into those details in Chapter 16.

For the final piece of the onboarding experience, we need to create the tenant admin user (step 5). If you recall, the tenant admin role represents the first user that is created for a given tenant. This tenant will have the ability to create any additional users that will be able to access the system. At this stage, though, our main goal is to create this initial user within our identity provider to enable our tenant to authenticate and access their provisioned environment. Here, you'll need to rely on features of your identity provider to orchestrate the notification and validation of this new tenant. Most identity providers will support the generation of an email message that includes a URL and temporary password for accessing the system. This process then triggers the authenticating user to enter a new password as part of the login flow. The goal here is to push much of the automation of this sign-up process to your identity provider. Rely on these providers to send email invites, temporary passwords, and hand password resets.

There is one last bit to this onboarding flow that you'll need to consider. Earlier, when the tenant was created (step 2), I set the active status of the tenant to false. It's the job of your Onboarding service to track the state of all of these different onboarding states. Only after it determines that each process has completed successfully, will it set the tenant's active status to true. This may include process retries and other fallback strategies to address any failure that may have happened during the provisioning and configuration of the tenant environment. Assuming the onboarding succeeds, the Onboarding service can now call the Tenant Management service and update the active status to true. This is especially important to the administration console of your SaaS environment, which provides the functionality that is used to view and manage the state of tenants. During this onboarding process, the view of tenants should show the state of any tenant that is being onboarded and highlight the active status of your tenants.

Tracking and Surfacing Onboarding States

From looking at this process, it should be clear that your onboarding process includes lots of moving parts and dependencies. The more complex this process becomes, the more important it is to have useful, detailed operational insights into the various states of your onboarding flow. This is essential to analyzing progress, identifying is-

sues, and profiling the overall behavior and trends of your onboarding automation. It also means identifying the right design and tooling to effectively capture and surface the onboarding profile of your solution.

At a minimum, you could imagine having a distinct set of states that mapped to each of the steps in our onboarding flow. So, you might have separate states for `TENANT_CREATED`, `TENANT_PROVISIONED`, `BILLING_INITIALIZED`, `USER_CREATED`, and `TENANT_ACTIVATED`. Each of these states could be surfaced on through the tenant view in your administration console, allowing you to inspect the onboarding of any tenant at a given moment in time.

The real value of assigning and surfacing onboarding states is to provide richer operational insights into the status of your onboarding progress. This will be essential to troubleshooting any unexpected onboarding issues. Knowing precisely where your onboarding process is failing be of prime importance to your operational teams. This is especially important when your onboarding process includes a significant amount of infrastructure provisioning and configuration. In these cases, you might track more granular states that give you insights into the various stages that are within the moving parts of your provisioning process.

Tier-Based Onboarding

As part of looking at the onboarding flow, I outlined the role of the Provisioning service and its role in creating/configuring tenant environments. This provisioning process gets a bit more interesting when you consider how different tenant tiers could influence how you implement your provisioning lifecycle. If you recall, we use tiers to present different tenants profiles with different experiences. These different experiences often translate into a need for separate infrastructure and configurations based on the tier of your system.

To better understand this, let's look at a conceptual example of a tier-based onboarding example. [Figure 4-6](#) provides a view of an environment that supports two separate tiers (basic and premium).

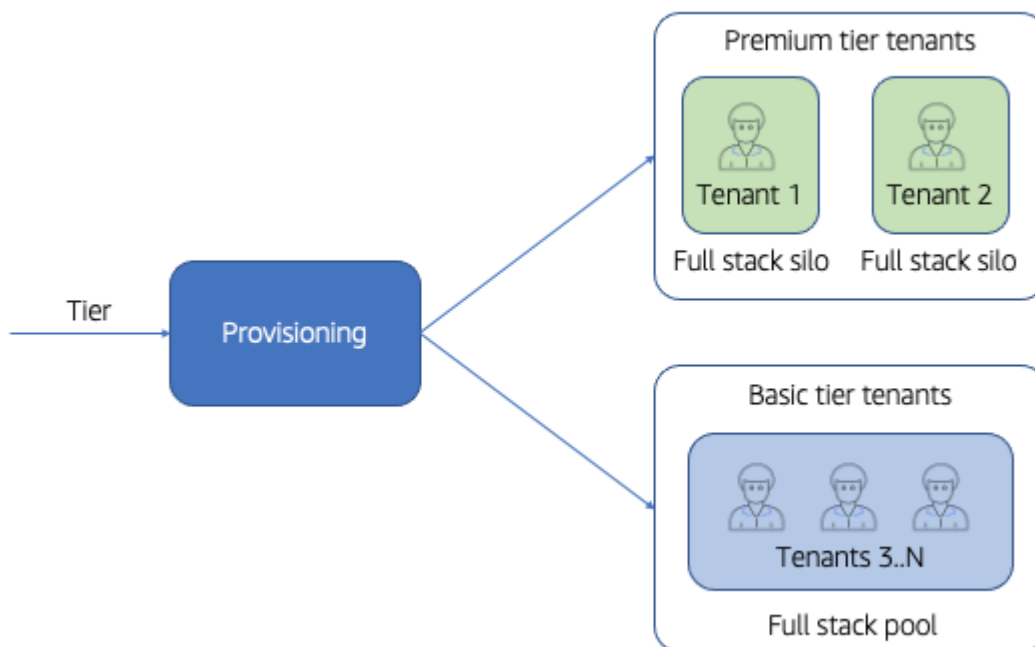


Figure 4-6. An example of tier-based onboarding

I've narrowed the view down here to focus exclusively on the Provisioning service within our control plane. Whenever the Onboarding triggers this Provisioning service, it provides the tenant contextual information that includes the tier that will be associated with your new tenant. When the Provisioning services receives this request, it will evaluate the tier and determine how/if the selected tier will influence the configuration and infrastructure that will be needed to support your tenant environment. In this example, our SaaS solution offers Premium tier tenants a full stack silo deployment mode with fully dedicated resources for each tenant. This means, each onboarding event will need to automate the provisioning of these full tenant stacks. Basic tier tenants, however, are onboarded into a full stack pool model where all the infrastructure is shared by tenants. Here, the onboarding will be a lighter weight experience, simply augmenting the configuration to add support for this new tenant.

These full stack deployment models have pretty distinct onboarding experiences that are relatively easy to digest. Where this gets more interesting is when you have a mixed deployment model. With a mixed mode deployment, your resources are siloed and pooled with much fine granularity. This means that your onboarding process will need to apply the tier-based onboarding policies to each resource based on its silo/pool configuration. [Figure 4-7](#) provides an example of how mixed mode deployment influences your Provisioning process.

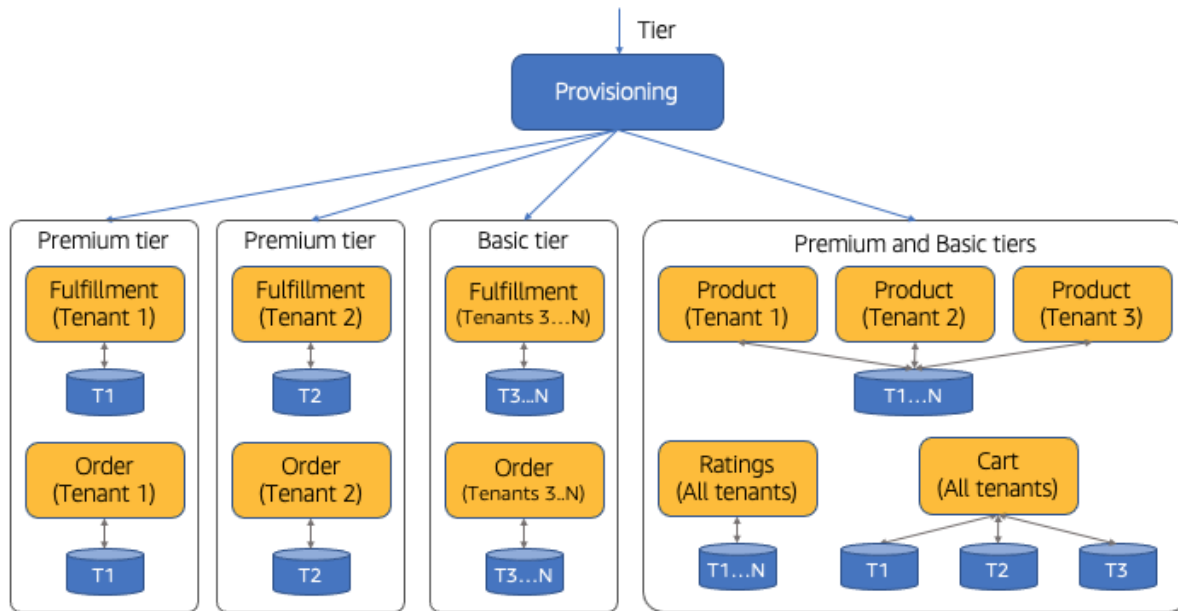


Figure 4-7. Tier-based onboarding with mixed mode deployments

Here I've intentionally made this architecture a bit busy. Our same Provisioning service is shown here, but now it has much more to consider as each tenant onboards. Let's start on the left of the diagram where you'll see that I have two services that are deployed separately for Tenant 1 and Tenant 2. So, for every Premium tier tenant, your Provisioning service will need to configure and deploy a Fulfillment and Order service in a fully siloed model. The storage and the compute of these two microservices are fully siloed.

Now, even though these two microservices are deployed separately for Premium tier tenants, these same services are also consumed by your Basic tier tenants. This is represented in the middle of the diagram where I've identified pooled versions of the Fulfillment and Or-

der microservices that are shared by all non-Premium tier tenants (in this case, Tenants 3..N). This means the Provisioning service must perform a one-time configuration and deployment of these services to support the pooled tenants. Once these services are up-and-running, the job of the Provisioning service for each new tenant will require significantly fewer moving parts. You may need to configure routing or set up some policies, but most of the heavy lifting will be done after the initial provisioning and deployment of these services.

Finally, on the right-hand side of the environment in [Figure 4-7](#), you'll see a range of services that are deployed in varying models to support the needs of both Premium and Basic tier tenants. The silo and pool choices that you make here are driven more by the universal needs of your multi-tenant architecture (instead of tiers). The idea here is that you're selecting silo and pool options based on a set of global needs (noisy neighbor, compliance, and so on).

In this example, I've intentionally created some clear variation in these services to highlight the use cases you may need to support as part of tenant onboarding. The Product microservice, for example, uses siloed compute for all tenants. That's why you see a separate instance of the service for Tenants 1-3. However, you'll also see that this same service is using pooled storage. This adds a new wrinkle to the onboarding story. Now, your Provisioning service must handle this variation, provisioning the storage a single time for all tenants

while still provisioning and deploying separate instances of the Product microservice as each tenant is onboarded.

The other services (Ratings and Cart) are just here to highlight additional patterns you could see when implementing your Provisioning service. Ratings is entirely pooled for compute and storage, while the Cart microservice has pooled compute and siloed storage. Supporting onboarding for these services is just about knowing what's siloed and what's pooled and contextually triggering the creation/configuration of these resources. This all mirrors the earlier discussion we had (in Chapter 3) around mixed mode deployment. However, here, we're looking at how that mixed mode can influence the onboarding experience of your multi-tenant environment.

One key question often comes up around the general timing of provisioning pooled resources during the onboarding process. Since these resources are configured and deployed once, many may prefer to pre-provision these resources as part of the initial setup of your entire multi-tenant environment. So, if you're setting up a brand new baseline environment, you could choose to provision all the pooled resources at this time. To me, this seems like the more natural approach. This could mean that your Provisioning service would support a separate path that is invoked by your DevOps tooling to perform the one-time creation of these resources. Then, as each new tenant onboards, this shared infrastructure would already be in place.

The other option here could be to delay the creation of these pooled resources and trigger their creation during the onboarding of your first tenant (almost like the Lazy loading pattern). While this could slow your onboarding process, the overhead of this process would only be absorbed by your first tenant. My general bias is to pre-provision these resources. However, there could be other factors that steer you toward either one of these strategies.

While it can be interesting and powerful to support these different tier-based deployment models, it's also essential to consider how/if your onboarding complexity might impact the complexity of our overall SaaS environment. Yes, you want to give the business lots of tools to be able to support different tenant profiles. At the same time, you don't want to over rotate here. Also, it is important to emphasize that this is still a tier level of customization. You should never view this mechanism as a way to support any notion of one-off customization for individual tenants.

Tracking Onboarded Resources

If you end up in a model where your onboarding process provisions and dedicated tenant resources, then you'll also have to consider how your multi-tenant environment will track and identify these resources. What you'll find here is that other aspects of your system will end up needing to locate and target these tenant-specific resources.

To really understand what I'm getting at here, let's consider a more concrete example. Imagine you've onboarded a tenant in the mixed model deployment model in [Figure 4-7](#). This model includes plenty of examples of siloed and pooled resources. Now, imagine you just onboarded a Premium tier tenant into this environment and created the individual resources that were needed to support that tenant.

Once onboarding is done and our tenant is up-and-running, you'll still be deploying updates to this environment. Patches, new features, and other changes will certainly need to get deployed through the life-cycle of your application. This is where things get a bit interesting. With the mixed mode deployment we have here, we can't simply deploy to one static location to update our system. Imagine, for example, rolling out a new version of the Order service. To get the new code deployed, your DevOps experience will need to find all the separate deployments of the Order service that span all different resources that were provisioned by the onboarding experience. Here, that would mean deploying the Order service to the Tenant 1 and Tenant 2 Premium tier siloes and to the Basic tier pooled instance that is shared by the other tenants.

So, that begs the question: How would your deployment process know how to handle this? How would it know which resources are siloed for each tenant? The only way for this to work is to have your onboarding experience capture and record the location/identity of

these per-tenant resources. While the need for this tracking information is clear, there's no clear or standard strategy that is commonly applied to address this. Some place the data in a table as new tenants are onboarded and reference this table during their deployment process. Others might use pieces of their DevOps tool chain to address this challenge. The main takeaway here is that—if your onboarding process provisions dedicated tenant resources—you'll need to capture and record the information about these resources so they can be referenced by other parts of your deployment and operational experience.

Handling Onboarding Failures

Any failure in the onboarding process can represent a significant issue for SaaS providers. However, these failures take on added importance in any multi-tenant environment that has a self-service onboarding experience. Onboarding represents the first impression you're making with a tenant and any failure in this process could translate into lost business.

While some of your reliability here will be extracted from applying solid engineering practices, there are also areas within onboarding where your dependencies on external systems can impact the durability of your onboarding process. To get a better sense of the options, let's look at a specific example of a potential external dependency

that could be part of your onboarding experience. [Figure 4-8](#) provides a conceptual view of the billing integration that could be part of your onboarding flow.

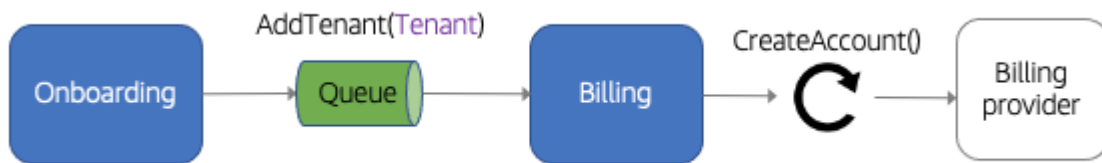


Figure 4-8. Fault tolerant integration with a billing provider

In this example, let's presume you are reliant on an integration with a third-party billing provider. By including a third-party billing solution in your onboarding (which is common), you've made the reliability of your onboarding experience directly dependent on the availability of the billing provider. If the billing system is down, so is your tenant onboarding.

Now, you might just presume that this is all just the risk associated with using third-party solutions. However, in this scenario, your system may very likely be able to continue to operate—even when the billing system is down. While it's true that you need to get the billing account created, your system could still finish its onboarding process and complete the billing configuration when the system is back online.

In [Figure 4-8](#), I've highlighted a potential approach to this problem, making the billing integration completely asynchronous. So, here, onboarding requests the addition of a new tenant through a queue. The Billing service then picks up the request and attempts to create an account in the billing system. However, it does this using an asynchronous request. If this request fails, the Billing service will capture the failure and schedule a retry. Of course, there are lots of different strategies for implementing a fault tolerant integration. So, don't get lost in the details here. The key takeaway here is that I've created an integration model with the billing provider that enables my onboarding flow to continue without waiting for the creation of the billing account to be completed. For some, it may simply be preferable to have this always be an asynchronous integration purely for the benefit of expediting the onboarding experience.

I've focused here on billing just because it provides a natural illustration of the importance of having a fault tolerant onboarding experience. However, in reality, you should look at all the moving parts of your onboarding automation and look for points of failure or bottlenecks that might require new strategies that can expedite or add durability to your onboarding process. The cost of a failed onboarding is generally high and you want to do whatever you can to make this mechanism as robust as possible.

Testing Your Onboarding Experience

At this point, the role and importance of onboarding should be clear. The potential complexity and the number of moving parts in this process can make it particularly prone to errors. With this in mind, it should also be clear that you'll want to take extra measures to validate the efficiency and repeatability of your onboarding process. Too many teams build an onboarding process and simply rely on the activity of customers to uncover any bottlenecks or design flaws that might be impacting their onboarding experience. To get around this, I always suggest that teams invest in building a rich collection of onboarding tests that can be used to exercise and push on all the dimensions of the onboarding experience.

The goal here is to ensure that the design, architecture, and automation assumptions of your onboarding experience are being fully realized in your working solution. That means pushing scale by simulating a heavy load of tenants onboarding. It means surfacing and validating metrics that measure your ability to meet onboarding SLA goals. This list goes on. The emphasis is not just on ensuring that the happy path works—it's about ensuring that onboarding meets the scale and availability requirements and delivers the service experience that meets the expectations of your customers.

Creating a SaaS Identity

So far, I've touched briefly on the role of identity as part of the onboarding process. However, there are lots of pieces to the identity puzzle that need exploring. Yes, onboarding sets up identity, but what does that mean? How does identity get configured and how does multi-tenancy affect the overall experience of our SaaS environment? Here, we'll dig more deeply into how tenancy shapes the authentication, authorization, and general multi-tenant footprint of a SaaS environment.

With multi-tenant identity, you'll have to go beyond thinking about identity purely as a tool for authenticating users. You have to broaden your view of identity to include the idea that each authenticated user must always be authenticated in the context of a tenant. It's true that users are connected to this experience, but much of the underlying implementation of your multi-tenant architecture is primarily focused on the tenant that is associated with that user. So, this means our identity model must be expanded to cover both users and tenants. The basic goal here is to create a tighter binding between users and tenants that allows them to be accessed, shared, and managed as a single unit.

In [Figure 4-9](#) you'll see a conceptual view of how a SaaS identity is composed. Here, on the left, I have the classic view of what I've labeled as a user identity. This identity is focused squarely on describing and capturing the attributes of an individual. Names, phone number, email—these are all typical descriptors that would be used to characterize the user of a system. On the right-hand side, however, I have also introduced the idea of a tenant identity. A tenant is more of an entity than an individual. A company, for example, subscribes to your SaaS service as a tenant and that tenant often has many users.

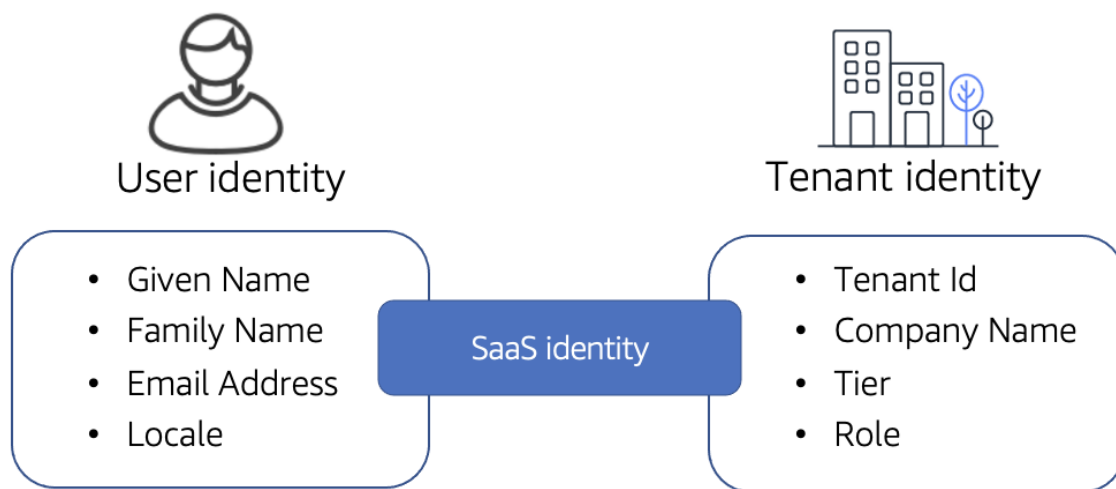


Figure 4-9. Creating a logical SaaS identity

For multi-tenant environments, these two distinct notions of identity are joined together to create what I refer to as a SaaS identity. This SaaS identity must be introduced in a way that allows it to become a first-class identity construct that is passed through all the layers of

your system. It becomes the vehicle for conveying your tenant context to all the parts of your system that need access to these user and tenant attributes. This SaaS identity maps directly to the tenant context concept that I described in Chapter 1.

The key here is that this SaaS identity needs to be introduced without somehow impacting or complicating the traditional authentication experience. Your SaaS authentication experience must retain the freedom to follow a classic authentication flow and while still enabling the merging of these user and the tenant identities. [Figure 4-10](#) provides a view of this concept in action.

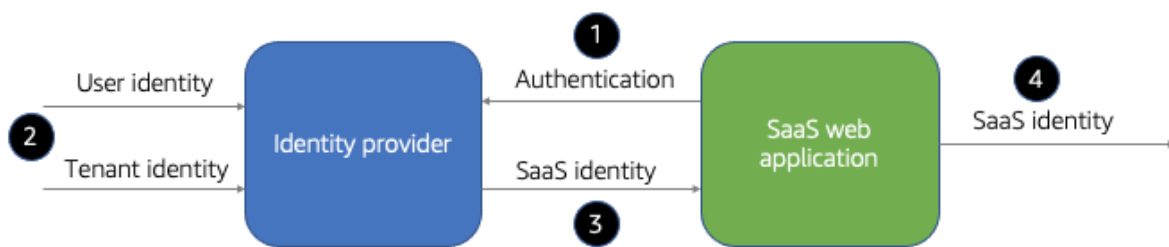


Figure 4-10. SaaS identity authentication flow

In the flow you see here, a tenant user attempts to access a SaaS web application (step 1). The application detects that the user is not authenticated and redirects them to an identity provider that has awareness of both your user and tenant identities (step 2). When the user is authenticated, the identity provider will own responsibility for returning the SaaS identity (step 3). Then, this SaaS identity is

passed downstream to all the rest of the moving parts of our system (step 4). This identity includes all of the tenant and user attributes that are needed to resolve to support the needs of the remaining elements of your SaaS application.

While this flow might vary based on the nature of your identity technology, the spirit of this experience should remain similar across different identity models. In the end, your goal here is to resolve and create this SaaS identity at the front of this process and avoid pushing this responsibility further into the details of your design and implementation.

Attaching a Tenant Identity

At this stage, I've talked about joining the user and tenant identity. While this may make sense conceptually, we still haven't talked about how you can combine these two concepts into a true, first-class identity construct. Naturally, how you do this, will vary from one identity provider to the next.

For this discussion, I'm going to focus on how the Open Authorization (OAuth) and OpenID Connect (OIDC) specifications can be used to create and configure a SaaS identity. These specifications are used widely by a number of modern identity providers, serving as an open standard for decentralized authentication and authorization. As such,

you should find that the techniques covered here should have some natural mapping to your application's identity model.

To get tenants attached to users, we first need to understand how the OIDC packages and conveys user and authorization information. Generally, when authenticating against an OIDC compliant identity provider, you'll find that each authentication returns identity and access tokens. These tokens are represented as JSON Web Tokens (JWTs) that hold all the authentication and authorization context. The identity token is meant to convey information about a user while the access token is used to authorize that user's access to different resources.

Within these JWT tokens, you'll find what are referred to as claims. There are a default set of claims that are generally included with each token to ensure that there's a standardized representation of common attributes. It's these JWTs that become the universal currency of our multi-tenant identity model.

The good news with JWTs is that they allow for the introduction of custom claims. These custom claims are essentially the equivalent of user-defined fields that can be used to attach your own property/value pairs to your JWT tokens. You can see this creates the opportunity for you to attach tenant contextual data to these tokens. [Figure 4-11](#)

provides an illustration of how these custom tenant claims would get added to your JWT tokens.

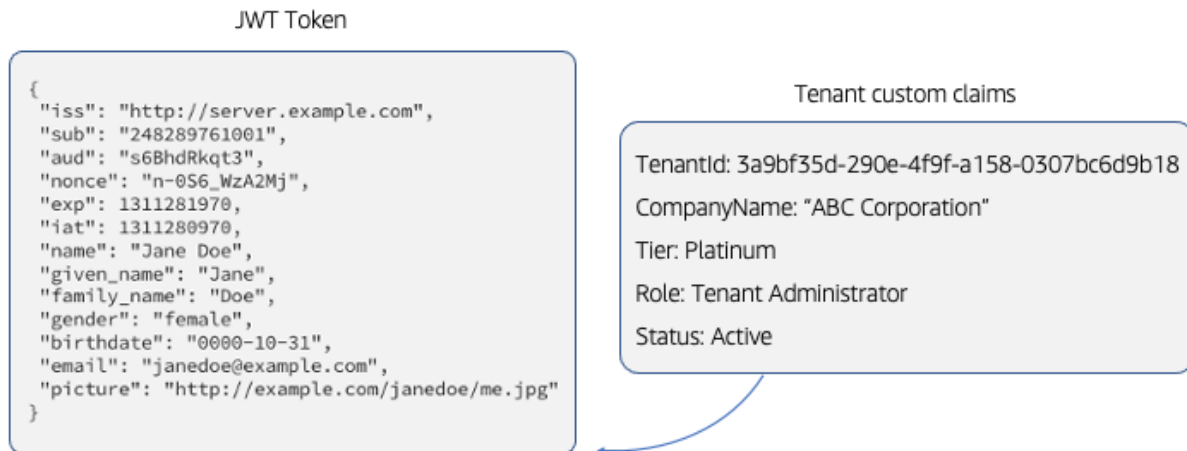


Figure 4-11. Adding tenant custom claims to a JWT token

On the left is a sample JWT token that is populated with the example claims that are part of the OIDC specification. I won't go through all of these claims, but it is worth calling out the specific user attributes that show up here. You'll see a name, given_name, family_name, gender, birthdate, and email are all in this list. On the right, however, are the attributes of our tenant that need to be merged into the JWT token. These simply get added as property value pairs to standardized representation.

While there's nothing magical or elegant about this model, being able to introduce these custom claims as first-class citizens provides a significant upside. Imagine how having these attributes embedded as

claims ends up shaping your multi-tenant authentication and authorization experience. [Figure 4-12](#) highlights how this seemingly simple construct ends up having a cascading impact across your multi-tenant architecture.

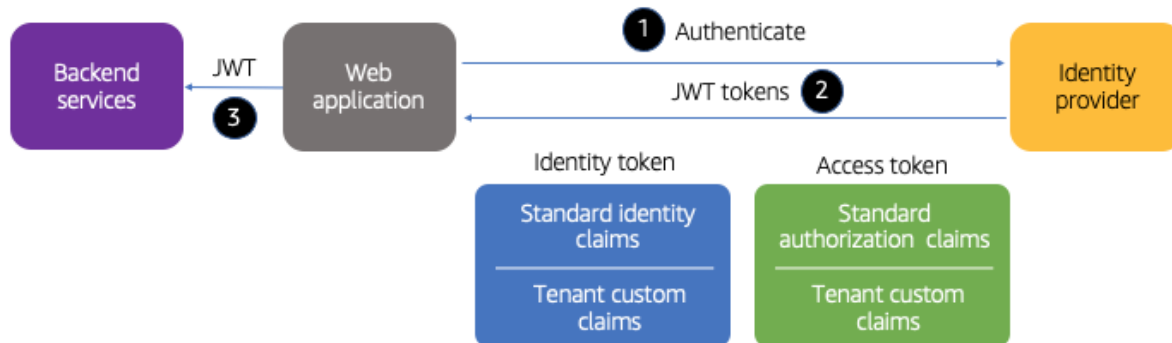


Figure 4-12. Authenticating with embedded tenant context

In this example, the flow starts at the web application. The user hits this page and is not authenticated, which sends them off to the identity provider for authentication (step 1). At this point, this represents a very familiar and vanilla flow that you've likely built multiple times. What's different here is the data comes back from this authentication experience. When you authenticate here, the identity provider is going to return its standard tokens (step 2). However, because I've configured the identity provider with tenant-specific custom claims, the tokens that are returned here now align with the SaaS identity that was discussed above. The tokens participate and behave like any other token. They're just enriched with the added tenant context that we need to create a SaaS identity.

Now, these tokens can be injected as bearer tokens and sent downstream to your backend services, inheriting all the security, lifecycle, and other mechanisms that are built into the OIDC and OAuth specifications (step 3). This strategy is particularly powerful when you look at how it impacts the broader experience of your backend services. [Figure 4-13](#) provides an example of how these tokens could flow through the different multi-tenant microservices of your application.

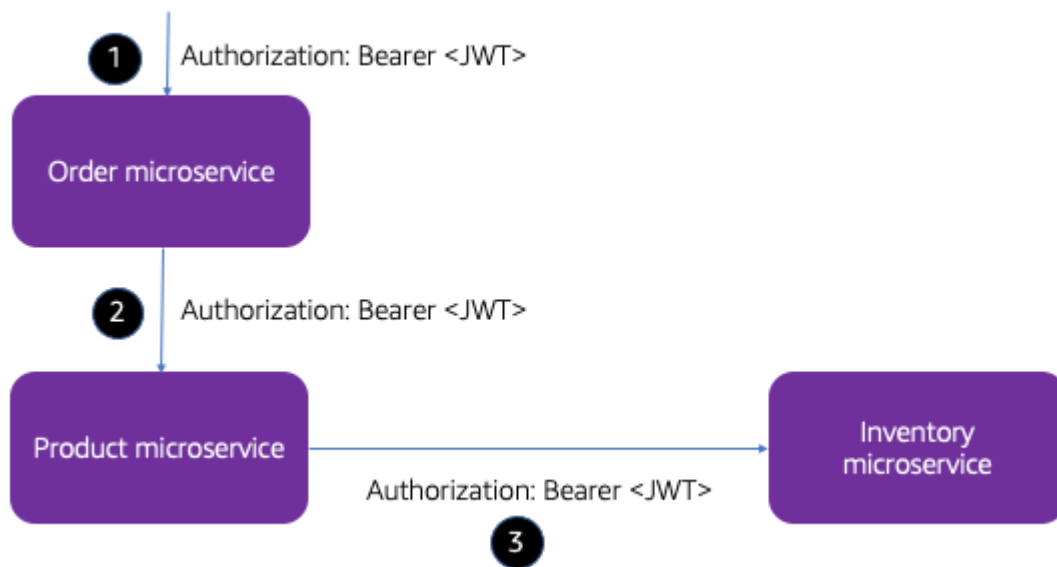


Figure 4-13. Passing tokens to downstream microservices

Here, I have three different microservices, each of which requires access to tenant context. When you authenticate and receive a tenant-aware token, this token is passed into whichever microservice you're initially calling. In this case, the call comes into the order microservice (step 1). Then, imagine that this service needs to invoke another

backend service (product) to complete its task. It can then pass this same token along to the product service (step 2). This pattern can then continue to cascade through additional downstream service invocations (step 3). In this example, I've assumed I can insert the JWT into an HTTP request as a bearer token. However, even if you're using another protocol here, there are likely ways you can inject this JWT as part of your context.

You can imagine how this very simple mechanism ends up having a rather profound impact on your overall multi-tenant architecture. This single JWT token will touch so many of the moving parts within the implementation of your multi-tenant environment. Microservices will use it for logging, metrics, billing, tenant isolation, data partitioning, and a host of other areas. Your broader SaaS architecture will use it for tiering, throttling, routing, and other global mechanisms that require tenant context. So, yes it's a simple concept, but its role within your SaaS architecture cannot be understated.

Populating Custom Claims During Onboarding

We've now seen how custom claims give us a way to connect users to tenants. What may be less clear here is how/when these claims are actually introduced. There are two pretty straightforward steps associated with adding and populating these custom claims. First, before you onboard any tenant, you'll typically need to configure your

identity provider, identifying each of the custom claims that you'd like to have added to your authentication experience. Here, you'll define each property and type that you'll want supported as a custom claim. This makes your identity provider ready to accept new tenants that can store and configure their tenant attributes.

The second half of this process is executed during onboarding. Earlier, I had discussed the creation of the tenant administration user as part of the overall onboarding flow. However, what I didn't mention here was the population of the custom claims for your newly created tenant. As you're adding the information about your user (name, email, etc.), you'll also populate all the tenant context fields for that user (tenantId, role, tier). This data must be populated for each user within the identity provider. So, even after onboarding has been completed, the introduction of additional users must include the population of these custom attributes.

Using Custom Claims Judiciously

Custom claims represent a useful construct for attaching tenant context to your tokens. In some cases, teams will get attached to this mechanism and expand its role, using it to capture and convey application security context. While there are no hard and fast rules here, I generally assume that, if something is a custom claim, it's playing an

essential role in shaping tenant context and influencing your global authorization story.

Many applications rely on role-based access controls (RBAC) that are used to enable/disable access to specific application functionality. These controls, to me, should be managed outside the scope of your identity provider. Generally, I'd view it as a mistake to bloat your tokens with custom claims that are part of your traditional application RBAC strategy. Instead, these kinds of controls should be implemented with any one of the language/technology stack mechanisms that are built exclusively for this purpose.

There's certainly some gray area here that may have some application concepts come in through custom claims. However, if you think about the lifecycle of these RBAC constructs, they tend to be constantly evolving with the features, functions, and capabilities of your application. Meanwhile, the identity profile of your application should be relatively constant. The content of your tokens, for example, should not be shifting on a weekly basis based on the addition of some new application feature or RBAC setting.

No Centralized Services for Resolving Tenant Context

Some teams try to draw a harder line between tenant identity and user identity. In these environments, the identity provider is only used to authenticate users. Here, when a user is authenticated, the tokens returned from this process do not include any tenant contextual information. In this model, these systems must rely on some downstream mechanisms to resolve tenancy. [Figure 4-14](#) provides an example of how this might be implemented.

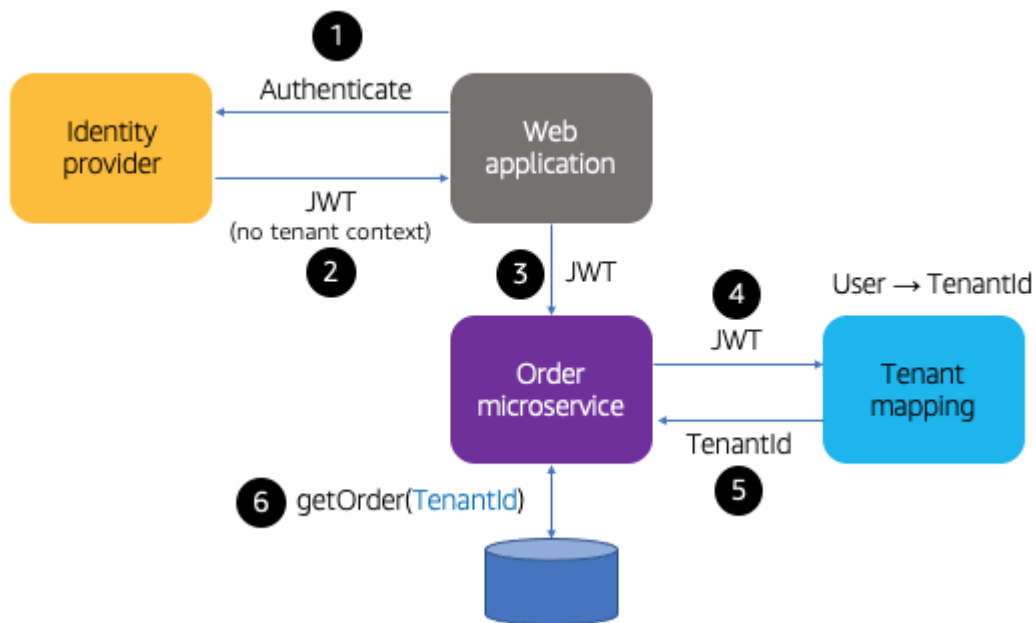


Figure 4-14. Using a separate user/tenant mapping service

In this example, the web application authenticated against an identity provider that has no awareness of tenant context (step 1). A successful authentication here will still return the JWT tokens we discussed. However, these tokens will not include any of the tenant-specific custom claims that were outlined above (step 2). Instead, the only data

here is user data. This token is then passed along to the order microservice (step 3). Now, when this order service needs to access data for a specific tenant, it needs to identify which tenant is associated with the current request. Since the JWT doesn't include this information, your code would need to acquire the context from another service (step 4). In this example, I've introduced a Tenant Mapping service that takes the JWT, extracts the user information, resolves the user to a tenant, and returns the tenant identifier (step 5). This identifier is then used to get an order for this specific tenant (step 6).

On the surface may not seem like a perfectly valid strategy. However, it actually presents real challenges for many SaaS environments. The lesser of the issues here is that it creates a hard separation between the user and the tenant, requiring teams to manage the coupled state of the user and the tenant independently. The bigger issue here, though, is that every service in the system must go through this centralized mapping mechanism to resolve tenant context. Imagine this step being performed across hundreds of services and thousands of requests. Many who adopt this approach quickly discover that this Tenant Mapping service ends up creating a significant bottleneck in their system. This then leads teams down a path of trying to optimize a service that is actually providing no business value.

This is another reason why it's so essential that the user and tenant context are bound together and shared universally across the entire

surface of your multi-tenant architecture. As a rule of thumb, my goal is to never have a service need to invoke some external mechanism to resolve and acquire tenant context. You want to have most everything you need to know about the tenant shared through the JWT that includes your SaaS identity information. Yes, there may be exceptions to this, but this should be the general mindset you take when thinking about how/where you're mapping users to tenants.

Hybrid SaaS Identity

Most of what I've described so far assumes that your SaaS system will be able to run with a single identity provider that is under your control. While this represents the ideal scenario and maximizes your options, it's also not practical to assume that every SaaS solution is built with this model. Some SaaS providers face business, domain, or customer needs that require them to support a customer-hosted identity provider.

One common case I've seen here is a scenario where a SaaS customer has some enterprise dependency on an existing, internal identity provider. Some of these customers may, as a condition of their purchase, require a SaaS provider to support authentication from these internal identity providers. In these cases, this often comes down to weighing the value of acquiring this customer and adding complexity to your environment that could impact the agility and oper-

ational efficiency of your overall SaaS experience. Still, when the right opportunity presents itself, the business parameters can push teams toward strategies that allow them to support this model.

Typically, this is achieved through some added level of tenant configuration where your tenant onboarding will add additional support for configuring this externally hosted identity provider. The goal here would be to make this as seamless as possible, limiting the introduction of any invasive or one-off code that would include tenant-centric customization. The other challenge here is that, in some cases, you'll need to provide side-by-side support for the external and internal identity providers. The reality is, most of your customers are likely to expect that your solution already includes built-in identity support.

[Figure 4-15](#) provides a view of the moving parts of this identity pattern.

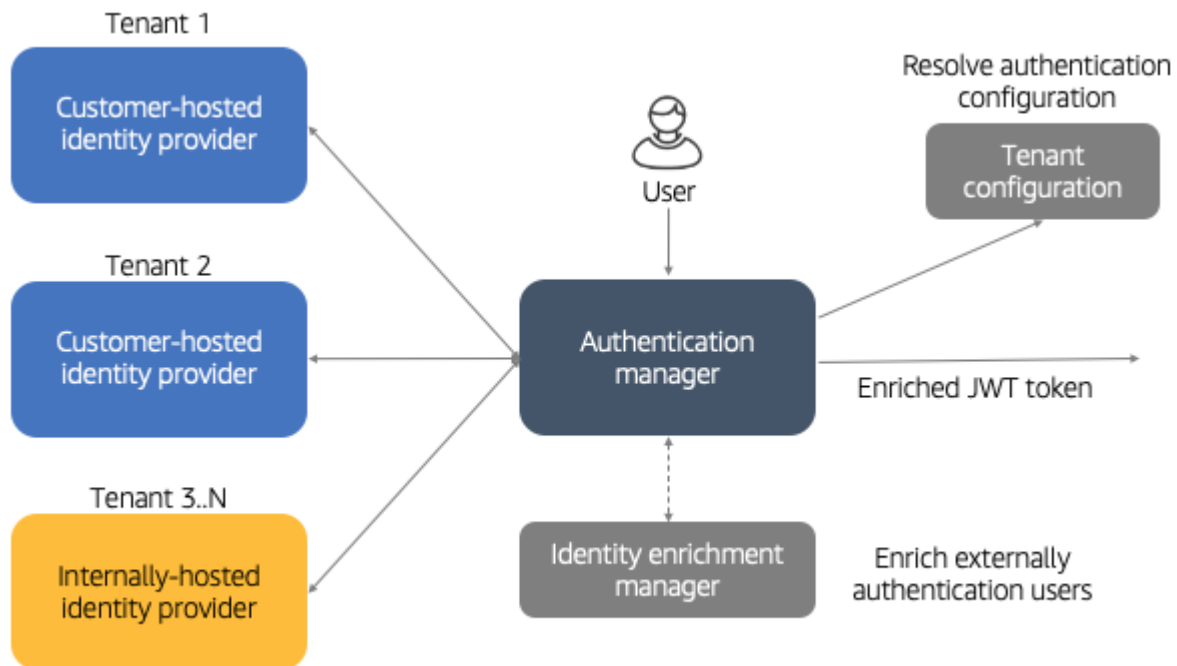


Figure 4-15. Supporting externally-hosted identity providers

Here, at the center of this example, you'll see that I have an authentication manager. This is a conceptual placeholder for introducing some service into your authentication flow that can support a more distributed set of identity providers. To make this work, your system will need to always determine how/where an identity provider is hosted. Each time a user needs to authenticate, you'll need to examine that user up and retrieve the identity configuration which will include data that describes the location/configuration of a given tenant.

On the left-hand side of [Figure 4-15](#) I've included a mix of internally and externally hosted identity providers that need to be supported by a single SaaS experience. Two tenants are using their own identity

provider. The remaining tenants are using your internally hosted identity provider.

This model seems pretty straightforward. However, the twist here is that your system has no control over these external identity providers. As such, you can't configure the claims of these providers or have your onboarding process add additional tenant context to the identity data that's managed by these providers. This means that the JWTs returned from your authentication requests will not include any of the tenant context that is essential to your multi-tenant environment. To resolve this, your solution will need to introduce new functionality that can enrich the tokens returned from these external identity providers, assuming responsibility for enriching these tokens with tenant context that is managed within your SaaS environment. This allows all downstream services to continue to rely on tenant-aware JWT tokens regardless of which identity provider was used to authenticate your user.

I've included this model because it represents an inevitable pattern that appears in the wild. At the same time, it's worth noting that there are clear downsides to this approach. Any time you have to insert yourself into the authentication flow, you are taking on an added role within the security footprint of your multi-tenant architecture. You may also be required to address scale and single point of failure requirements that come with sitting in the authentication flow. So, while this

may be necessary, it comes with real baggage that you'll want to consider carefully.

Federated SaaS Identity

There are some identity tools and strategies that can allow you to support a more diverse identity landscape without introducing all the complexity that comes with the hybrid identity model described above. With the AWS identity provider (Amazon Cognito), for example, you can use its federated identity to authenticate against another identity provider and still use Cognito to bolt tenant context onto the tokens that are returned from another provider.

With this approach you configure Cognito with the same custom claims that are needed to hold your tenant context. However, you also configure Cognito in a federated mode where it uses another identity provider to perform the authentication. The flow here is one where Cognito authenticates against the federated identity and merges your custom claims into the tokens that are returned from the authentication process.

This federated model allows you to still configure and manage the custom claims part of this experience through Cognito while still using a separate identity provider to drive authentication. This provides a good compromise for environments that have a dependency on an

external identity but still want the goodness that comes with using custom claims to manage tenant context.

Tenant Grouping/Mapping Constructs

While identity providers often conform to well established specifications (OIDC, OAuth), the constructs that are used to organize and manage identities can vary from one identity provider to the next. These providers offer a range of different constructs that are used to group and organize users. This is especially important in multi-tenant environments where you may want to group all the users that belong to a tenant together. These group constructs can have implications that will influence how you land tenants within your identity provider. In some cases, you might also be able to use these groups to apply tiering policies to tenants to shape their authentication and authorization experience.

If we look at Amazon Cognito again, for example, you'll see that it offers multiple ways to organize your tenants. Cognito introduces the idea of a User Pool. These User Pools are used to hold a collection of users and they can be individually configured, allowing pools to offer separate authentication experiences. This might lead to a User Pool per tenant model where each tenant would be given its own pool. The alternative here would be to put all tenants in a single User Pool and use other mechanisms (groups, for example) to associate users with

tenants. You'd also want to consider how any limits from your identity provider might factor into choosing a strategy.

There are tradeoffs you'll want to consider as you pick between these different identity constructs. The number of tenants you have, for example, might make it impractical to have separate User Pools for every tenant. Or, you may not need much variation by tenant and would prefer to have all tenants configured and managed collectively. You might also be thinking about how the choices you make here could impact the authentication flow of your SaaS solution. If you have separate User Pools for each tenant, you'd need to think about how you'd map tenants to their designated pools during the authentication process. This may add a level of indirection that you may not want to absorb as part of your solution.

Scale, identity requirements, and a host of other considerations are going to shape how you choose to map tenants to whichever constructs are supported by your identity provider. The key here is that, as you start to lay out your SaaS identity strategy, you'll want to identify the different units of organization that can be used to group your tenants and determine how those will shape the scale, authentication, and configuration of your multi-tenant authentication experience.

Tenant-Level Identity Customization

With the different organizational constructs also come different identity configuration options. Identity providers generally provide a range of configurable options that can be used to configure your authentication experience. Multi-factor authentication (MFA), for example, is offered as an identity feature that can be enabled or disabled. You can also configure password formatting requirements and expiration policies.

The settings for these different configuration options do not have to be globally applied to all of your tenants. You may want to make different identity features available to different tenant tiers. Maybe you'll only make MFA available to your Premium tier tenants. Or, you might decide to surface these configuration options within the tenant administration experience of your SaaS application, allowing each tenant to configure these different identity settings. This can be a differentiating feature that can add value for your tenants and allow them to create the identity experience that best fits the needs of their business.

How or if you can offer this identity customization will depend on how your specific identity provider organizes and surfaces these options. Some providers will allow you to configure this separately for individual tenants and others will only allow this to be configured globally. You'll need to dig into the constructs of your specific identity provider to figure out how/if you can associate these identity policies with individual tenants.

Sharing User Ids Across Tenants

Each user of your SaaS system has some user id that identifies that user to a tenant. This user identifier is often represented by an email address. In many cases, a single user will be associated with a single SaaS tenant. However, there are times when SaaS providers have interests in associating a single email address with many tenants. This, of course, adds a level of indirection to your authentication. Somewhere in your login flow, your SaaS system will need to determine which tenant you're accessing.

While I've seen requests for supporting this mode, I have yet to uncover any out-of-the-box strategy for handling this use case. That being said, there are some patterns that I have seen applied here. The most brute force way I've seen is one that pushes the tenant resolution to the end user. So, during sign-in, the system will detect that a user belongs to multiple tenants and will prompt the user to select a target tenant. This is anything but elegant and it does provide a bit of an information leak in that anyone can use an email address to see which tenants you belong to (if and only if you belong to more than one). In the model, you'd have a mapping table that connected users to tenants and you would use this as a lookup in advance of starting the authentication flow.

A cleaner approach to this would be to rely on an authentication experience that supplied context more explicitly. Here, the best example is probably domains and/or subdomains. If each of your tenants is assigned a subdomain (tenant1.saasprovider.com), then your authentication process can use this subdomain to acquire the tenant context. Then, the system would authenticate you against the specified tenant. This would allow the user to authenticate without any intermediate process to identify the target tenant.

There are other complications that are part of the scenario. Imagine, for example, all of your users are running in a shared identity provider construct. In that mode, the identity provider is going to require each user to be unique. This would make it impossible to support having a single user id associated with multiple tenants. Instead, you may want to consider relying on a more granular construct to hold each tenant's data (like the User Pool mentioned above).

Tenant Authentication != Tenant Isolation

As part of this discussion of authentication and JWT tokens, I sometimes find teams that will equate authentication to tenant isolation. The assumption here is that authentication is the barrier to entry for tenants and that, once you've made it beyond that challenge, you have met the criteria for tenant isolation in multi-tenant environments.

This is definitely an area of disconnect. Yes, authentication certainly starts the isolation story by issuing a JWT with tenant context. However, the code in your microservices can still include implementation that—even when working on behalf of an authenticated user—can access the resources of another tenant. Tenant isolation builds upon the tenant context that you get from an authenticated user, implementing a completely separate layer of controls and measures to ensure that your code is not allowed to cross a tenant boundary. You'll get a deeper look at these strategies in Chapter 10.

Conclusion

Hopefully, at this stage, you have a good sense of all the moving parts that come with setting up the foundational elements of your SaaS environment. This chapter outlined the fundamental process of creating your baseline environment, deploying the first bits of your control plane, and creating the moving parts of your onboarding and identity experience. This should have given you a clearer view of the role that onboarding and identity play within a multi-tenant architecture. These two mechanisms lay the foundation for introducing tenancy into an environment, creating the fundamental building blocks that will influence how this tenancy lands and is applied within your application. It's through onboarding that you'll bring the policies and strategies of your deployment model to life, configuring and creating the

various resources of your multi-tenant application plane. The principles here are focused completely on automating the introduction of tenants in a way that will support the scale, agility, and innovation goals of the business.

The role and implementation of identity was also reviewed in depth, providing a clearer view of how users are bound to tenants. The emphasis was on illustrating how creating a first-class construct for SaaS identity would have a cascading impact across the rest of your multi-tenant architecture. I looked at how tokening and connecting a user to your identity model provided a natural vehicle for injecting tenant context with the services of your application, enabling each service to access and apply this context without consulting other parts of your system.

At this stage, I've intentionally kept this more conceptual. The implementation of these models will vary based on your technology stack and the deployment footprint of your application. In Chapters 11 and 12, we'll dig into working Kubernetes and serverless SaaS architecture to get a better handle on what it means to bring these concepts to life. We'll also see some of the influence of these strategies surfacing across a range of upcoming chapters where I'll use the tenant context concepts introduced here to implement data partitioning, tenant isolation, multi-tenant microservices, and so on.

First, though, we're going to look a little deeper inside the control plane and examine the Tenant Management component of the control. This chapter already hinted at how Tenant Management surfaced as part of the onboarding experience. Now, I want to look more exclusively at the role of this service within your control plane. While this service is not exotic or overly complex, it often sits at the middle of our multi-tenant story. I'll look at what it means to create this service and outline some of the key considerations that can influence its implementation.

Chapter 5. Tenant Management

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fifth chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

In the previous chapter, we started our foray into the control plane, looking at the broader role that onboarding and identity play in bringing your multi-tenant architecture to life. As part of that process, I touched on how the Tenant Management service is used to introduce new tenants into the system. Now, it’s time to dig more into this service and get a better sense of the inner workings of this service and examine the full scope of its responsibility. This will give you a better sense of the data, operations, and constructs that put Tenant Man-

agement at the center of configuring aspects of your tenant architecture and managing the lifecycle of key tenant events.

We'll start by looking at the fundamentals of what it means to build your Tenant Management service, exploring the elements of its core design and implementation. As part of this, I'll get into some of the common tenant attributes that are managed by this service. You'll see that storing and managing these attributes is mostly straightforward. However, what you choose to store here and how it's influenced by other events in your environment can end up adding additional layers of complexity of your overall tenant management footprint.

To better understand the broader role and experience of tenant management, we also have to look at how the system administration console of your SaaS environment is used to monitor and manage your tenants. It's here that you'll get another view into how your Tenant Management service fits into the control plane experience. We'll look at how this administration console works directly with your service to support multi-tenant operational needs and manage the state of your system's tenants.

The other dimension of tenant management we'll review is tenant lifecycle management. The goal here is to look beyond the initial configuration of a tenant and examine the different events that can happen that will impact the state of a tenant. What does it mean for tenants to

change from an active to inactive state? How does your system manage and apply a tenant's move from one tier to another? How does the state of other systems (billing, for example) get conveyed to your Tenant Management service? These are all areas that can have some interaction with Tenant Management and, potentially, have some cascading impact across different layers of your SaaS architecture.

The key focus of this chapter is to highlight all these nuances of Tenant Management and equip you with a better sense of the considerations that can influence the multi-tenant strategies and patterns that are applied within your solution.

Tenant Management Fundamentals

To get a better understanding of the scope and nature of tenant management, let's start by looking at the core moving parts of the tenant management universe. [Figure 5-1](#) provides a conceptual view of the various elements that are often part of the overall tenant management footprint.

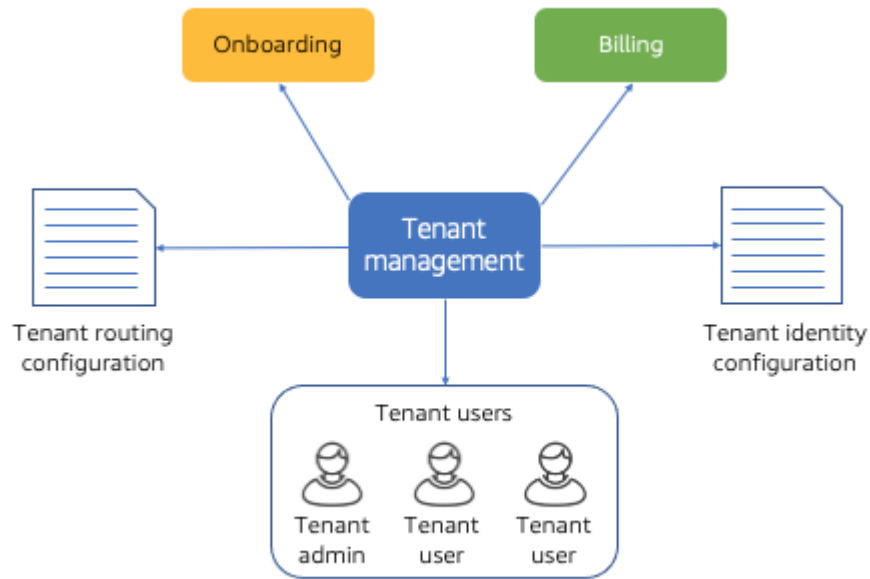


Figure 5-1. Tenant management's influence

Here you'll see that I have placed tenant management in the center of the diagram and identified the areas that are commonly configured and/or managed through their connection to a tenant. Onboarding, as discussed in Chapter 4, has the most significant influence on tenant management. When a new tenant is created, this can trigger the configuration and setup of a number of different resources that are related to your new tenant. I've shown some specific examples here.

On the right-hand side of the [Figure 5-1](#), you'll see tenant identity configuration. This is where you may have tenant-specific configuration data that is used to map your tenant's identity to a specific identity provider construct. This shows up in scenarios where each tenant may rely on a per-tenant defined identity construct. This is also where

you might see different authentication profile settings for a tenant (MFA, password expiration policy, and so on).

The left-hand side of the diagram has tenant routing configuration information. This is where you might store any per-tenant URLs or routing configuration data that is used to map tenants to specific architecture constructs. You'll see more of this as we dig into specific SaaS architectures that have per-tenant constructs that must be routed on a per tenant basis. The configuration data may change over the life of the tenant and would be centrally configured/updated through tenant management.

At the top right of the diagram, you'll also see that I've shown billing here. Certainly the tier of a tenant that's configured via tenant management will have a direct influence on how that tenant is billed. The broader billing relationship to tenant management, though, is more driven by changes in state that can be driven through the billing system.

Finally, at the bottom of [Figure 5-1](#) is a representation of the different users that can be associated with a tenant. Here I've shown a Tenant Administrator and a few Tenant Users. These are here to illustrate the relationship between a tenant and the various users that may be associated with that tenant. The Tenant Administrator is created when the tenant is first introduced. However, after onboarding, the tenant

can also create additional users for their system, including other Tenant Administrators. All of these users are logically connected to a tenant. Any change in tenant configuration will be applied to all of these users.

As I get into the details of specific multi-tenant architectures later in this book, you'll see how tenant management interacts with these different services. You'll also see more examples of how tenant management is used to shape the per-tenant configuration options of your multi-tenant infrastructure and identity experience.

Building a Tenant Management Service

Let's shift from the conceptual view of tenant management and its role and look more at what it actually means to implement a tenant management microservice. The interface of this service is typically broken down into two logical categories. First, you'll have a range of operations that are focused on the basic management of configuration data. These operations are typically exposed via a create, read, update, and delete (CRUD) interface. The other category of operations that land here are centered around broader tenant management operations. Deactivate a tenant, decommission a tenant, and so on. These operations tend to contribute the most to the overall complexity of your tenant management service.

[Figure 5-2](#) provides a simple view of the moving parts of a sample tenant management microservice. What I've tried to do here is provide a conceptual view into the common interfaces and experience that could be included in your tenant management service. On the left, you'll see all the entry points that mirror the summary I provided above. The top set of functions manage configuration and the bottom entry points that support the different management related operations.

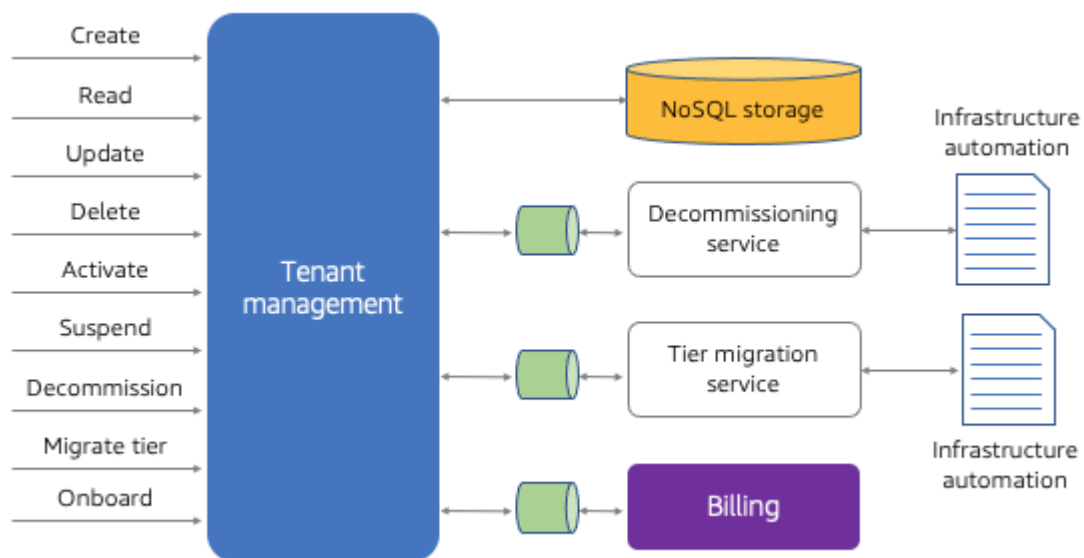


Figure 5-2. A sample tenant management implementation

The right-hand side of this diagram highlights the various backend resources and integrations that would be part of your tenant management experience. At the top right, I've shown some storage. For this example, I chose to use NoSQL storage to hold my tenant configuration information. Generally, the size and consumption patterns for this

data tends to fit well with a schemaless storage model. This also makes it easy to apply changes to the tenant configuration structure without needing to update schemas or migrate data.

I've also put placeholders here to represent some of the tenant lifecycle components that may be part of your tenant management service. I'll be covering these lifecycle concepts in more detail below, but I wanted to show them here to make it clear that this service may need to support decommissioning of tenants, tier migration, and other tenant lifecycle events. Each of these may require some level of infrastructure configuration, provisioning, or removal depending on the nature of the operation. This often means invoking the infrastructure automation scripts and/or code that are associated with these operational events. I've shown these concepts as being integrated via a queue only to highlight the idea that the actual work of processing these events might be triggered by an event and executed as part of some asynchronous job.

The last piece that I included here was billing. Your service may have multiple integrations with your control plane's billing service (or directly with your billing provider). Your billing system may trigger events that make updates to the state of tenants. Or, your tenant management service might trigger events that end up sending requests to the billing service.

You can see here that the tenant management service, on its own, doesn't introduce a significant amount of complexity into your control plane. At the same time, it sits at the center of managing key elements of your tenant state that directly influence the implementation and behavior of your multi-tenant architecture. It also provides a centralized home for processing tenant lifecycle events.

Generating a Tenant Identifier

Within the implementation of your tenant management service, you'll be responsible for generating the tenant identifier that represents the universal unique identity of any tenant in your system. The most common mechanism that is used for tenant identifiers is a GUID. It provides a natural way to have a globally unique value that has no dependency on other attributes of the tenant. The goals here are twofold. First, you just need to have some separate immutable value that can universally represent your tenant across your multi-tenant environment. Second, you want to ensure that any other consumer of this identity couldn't map it back to a specific tenant/entity in your system.

It's important to note that some multi-tenant environments may include alternate, more friendly ways to identify a tenant. For example, If my system uses a subdomain or vanity domain for individual tenants, then you'll need some way to map from that entity name to the

tenant identifier. In this case, I might have a **mycompany.saasprovider.com** where the “mycompany” subdomain represented an externally facing name that would then get mapped to an internal tenant identifier. Even when you have some other name you’re using as part of the entry into your system, these entity names or references should not be viewed as your tenant identifier. There are good reasons to keep these separate. The most obvious of these is the need to ensure that you can change these friendly names without having any impact on the rest of the system. This goes back to the basics of data management that led to the use of GUIDs to identify the different items in a database.

Storing Infrastructure Configuration

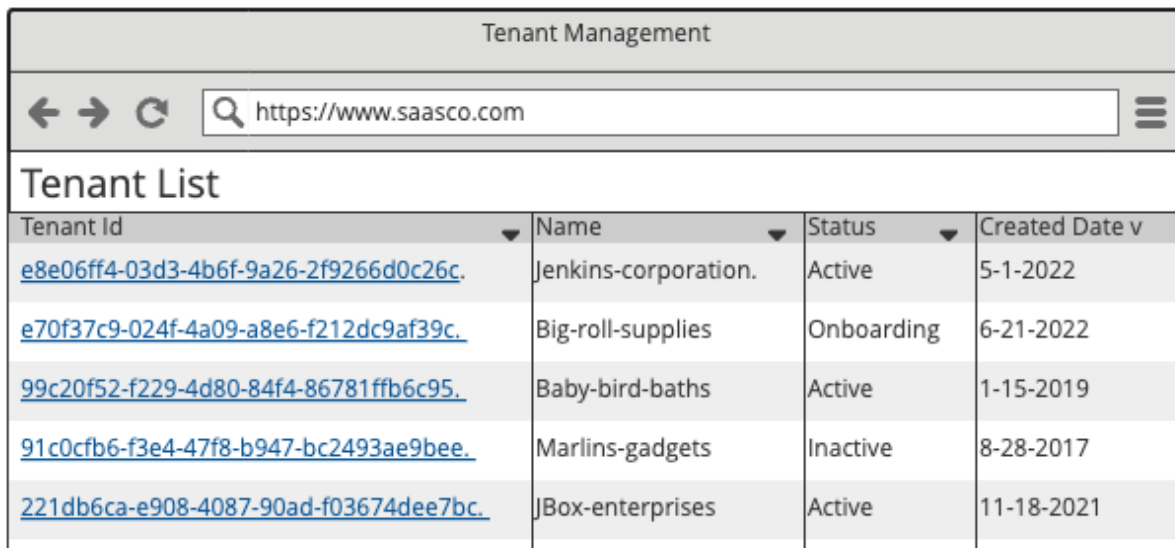
In addition to storing basic tenant attributes, tenant management can also be used to store tenant specific infrastructure configuration information. Depending on the deployment model of your multi-tenant application, you might have identity configuration, routing patterns, and other infrastructure options stored and managed by your tenant management service. This data is not typically managed directly by your administration experience. Instead, it’s stored here during the configuration of your multi-tenant infrastructure then referenced by different parts of your experience that need this information to configure/resolve tenant mappings that are part of your environment.

The data that you end up putting here is shaped by the specifics of your SaaS architecture. If, for example, you have separate identity constructs for each of your tenants, you may need to use tenant management to store a tenant's mapping to its identity construct. If you have siloed tenant environments and you have mapping to tenant specific entry points/URLs, those could be stored by the tenant management service. These are just a few examples. The key here is that tenant management may manage data that goes beyond the scope of basic tenant attributes.

While the tenant management service makes a good, centralized home for tenant state and configuration, you always need to ensure that this service does not become a bottleneck of your system. If the data that's stored here is being frequently accessed by all the moving parts of your system, you'll need to consider alternate strategies for managing and accessing this data. Ideally, the state that's here will not be used heavily by the application services of your application. This is part of why we put the critical tenant attributes into the JWT token, limiting your need to continually return to any single, centralized service to continually acquire this context.

Managing Tenant Configuration

Much of the initial focus of tenant management is on its role in creating and configuring tenants as part of the onboarding experience. It's important to note that tenant management has a life beyond onboarding. This service is also used to support different operations and use cases throughout the lifetime of a tenant. To better understand this, let's return to the system admin console that we discussed in Chapter 4. This console provides your administrative view into your multi-tenant environment, allowing you to inspect, configure, and manage your tenants.



The screenshot shows a web browser window with the address bar displaying 'https://www.saasco.com'. The page title is 'Tenant Management'. Below the address bar is a section titled 'Tenant List'. This section contains a table with four columns: 'Tenant Id', 'Name', 'Status', and 'Created Date v'. The table lists five tenants with their respective IDs, names, statuses, and creation dates.

Tenant Id	Name	Status	Created Date v
e8e06ff4-03d3-4b6f-9a26-2f9266d0c26c.	Jenkins-corporation.	Active	5-1-2022
e70f37c9-024f-4a09-a8e6-f212dc9af39c.	Big-roll-supplies	Onboarding	6-21-2022
99c20f52-f229-4d80-84f4-86781ffb6c95.	Baby-bird-baths	Active	1-15-2019
91c0cfb6-f3e4-47f8-b947-bc2493ae9bee.	Marlins-gadgets	Inactive	8-28-2017
221db6ca-e908-4087-90ad-f03674dee7bc.	JBox-enterprises	Active	11-18-2021

Figure 5-3. Managing tenant from the admin console

You'll see that this is a very straightforward experience that essentially grabs a list of tenants and lists them on the screen. This list, of course, is acquired from your tenant management service, which fetches all the tenants that are stored by your service and lists them

in this page. With each tenant, you'll see its identifier, name, status, and any other attributes that you might deem worthy of including in this view.

At a minimum, this view would be used to lookup and resolve the status of tenants. It's also commonly used to locate the unique identifier when you're searching logs or doing any other troubleshooting that might rely on having the tenant information.

In addition to surfacing information about the tenant, the console is also where you would edit/manage any policies that you might be managing for a given tenant. Perhaps the most common functionality that's configured here could be the tier or status of the tenant. In this particular example, you can see additional configuration details by selecting the link to the tenant identifier. [Figure 5-4](#) provides an example of a detailed view of a specific tenant.

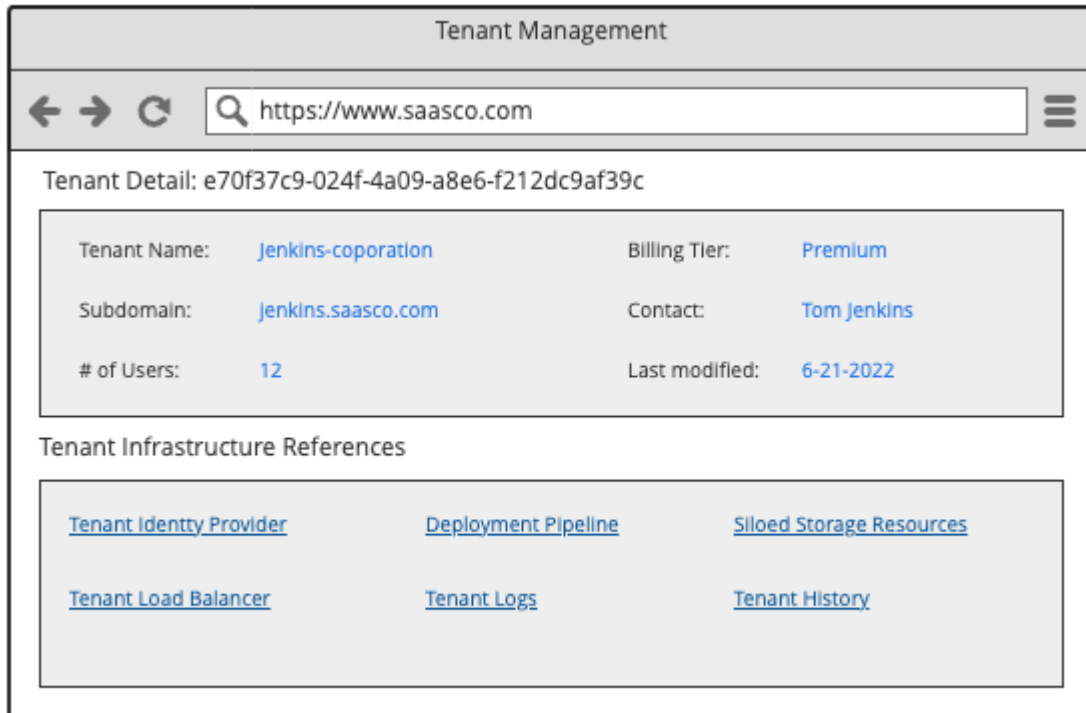


Figure 5-4. Managing tenant details from the console

Again, this is just an example, but it highlights some of the additional options you might attach to your tenant management view. Here you see the next level of detail associated with your tenants, including subdomain, onboarding status, and so on. You can imagine how this screen would be filled with more detail based on the presence of any additional tenant configuration options that are part of your multi-tenant environment.

There are two key things to note here as conceptual placeholders. First, in the top portion of the image, you'll see additional details that provide more data on the state of a single tenant. Then, at the bottom

of the image, you'll see a section that includes hyperlinks that take you to key infrastructure resources that are associated with the current tenant. These links take you directly to the admin or cloud provider infrastructure page for each resource, allowing you to rapidly access resources in the context of a specific tenant. This can streamline your operational experience, allowing you to rapidly navigate (with tenant context) to any specific tenant infrastructure resources.

It's important to note that you may or may not choose to include these links. They are especially valuable if you are running full stack siloed environments. However, the more pooled your environment is, the less likely that you'll get lots of value out of providing this tenant contextual access to infrastructure resources.

Finally, you'll also see that there's an Actions pulldown button in the top right of [Figure 5-4](#). This is where you'll perform specific operations on your tenant. At a minimum, you'll have functionality here to update the active status of a tenant. Here, an operator can access the system and temporarily deactivate a tenant. There may be additional options here to decommission a tenant or move them between tiers (these options are explored more in the section that follows).

Triggering Onboarding

In our discussion of onboarding in Chapter 4, I talked about the different models for triggering the onboarding of a tenant. I emphasized the point that onboarding of new tenants could be executed by self-service or internal processes. The self-service flavor of onboarding is pretty straightforward. You land on the sign-up page, fill in the data, and submit your request. The question is, though, where should you surface an internally managed onboarding experience? There is no one answer to this. You might have your own CLI or other tools that allow you to onboard tenants. However, certainly one common approach here would be to manage this onboarding through your tenant management console experience.

This would be achieved by adding an “Onboarding Tenant” action to your console. Triggering that action would open a form that had you fill in all the data needed to onboard a new tenant. [Figure 5-5](#) provides an example of what this form might look like.

Register

Create your account

Please select your service plan

Figure 5-5. Internally onboarding a tenant

This is a basic form that just collects and submits your data. Every multi-tenant onboarding process will likely include a tenant entity name, an email address for the tenant admin user, and the plan/tier they're signing up for. From there, the rest would depend on the configuration options and nature of your multi-tenant environment. You might need to collect a subdomain here, for example. Or, you might have a more elaborate process that allows you to bring in a full vanity domain for a tenant. The key at this point is just to highlight the fact that the tenant management experience may provide a good home for surfacing your internal process via the admin console.

Managing Tenant Lifecycle

Up to this point, the focus of tenant management has largely been on the mechanics of the front of the tenant management process where you're onboarding and configuring tenants. Now, I want to shift gears and look at the role tenant management plays beyond this initial phase. The focus now transitions to thinking about the various states that a tenant might go through during its time within your system.

To me, teams often overlook the entire notion of managing a tenant's lifecycle. They focus squarely on what it means to get a tenant introduced/configured and view that as the one and only role of their tenant management service. It's true that, for some tenants, you'll create them once and that will be it for them. However, there will also be tenants in your system that will need to go through changes in state that are more involved than just changing the value of some attribute in your tenant database.

When you set out to build your multi-tenant environment, you have to consider all the phases that a tenant could go through within your system. How will these state changes be conveyed to you? Will they be in response to some external event or driven directly from your tenant management service? What are the policies associated with these state changes? These are amongst the questions you'll want to ask yourself as you assemble the tenant management elements of your SaaS solution.

While there are any number of phases that could be part of your tenant lifecycle, there are a few common states that you'll want to consider when building any multi-tenant SaaS environment. The sections that follow will enumerate these lifecycle changes.

Activating and Deactivating a Tenant

The first and most obvious state you'll want to focus on is activation/deactivation. In general, your SaaS environment should include some ability to enable and disable tenants. This whole state is about managing a tenant's access to your system. When a tenant is deactivated, your tenant management service will need to own responsibility for figuring out what actions may need to be orchestrated to ensure that a tenant is blocked from accessing the system. This could be as simple as making a call to the user management service to disable authentication for all users for that tenant or it could be more involved.

Managing a tenant's active state can also be connected to the billing experience of your application. It's the billing system that can, in some cases, be at the front line of managing your tenant's active state. Imagine a scenario where a tenant has stopped making payments. In this case, this information might surface first within your billing system which identifies delinquent tenants. When this event is triggered by the billing system, you have to determine how your sys-

tem will respond to these events. You might have policies that send messages during some grace period that allow tenants to continue using the system while the billing issues are being resolved. At some point, though, you can reach a point where you've determined that a tenant needs to be deactivated. In this case, if the billing system originates this event, you'll need some way for this to be conveyed to your tenant management service.

[Figure 5-6](#) provides a view of how this connection between billing might be connected within the control plane of your SaaS environment.

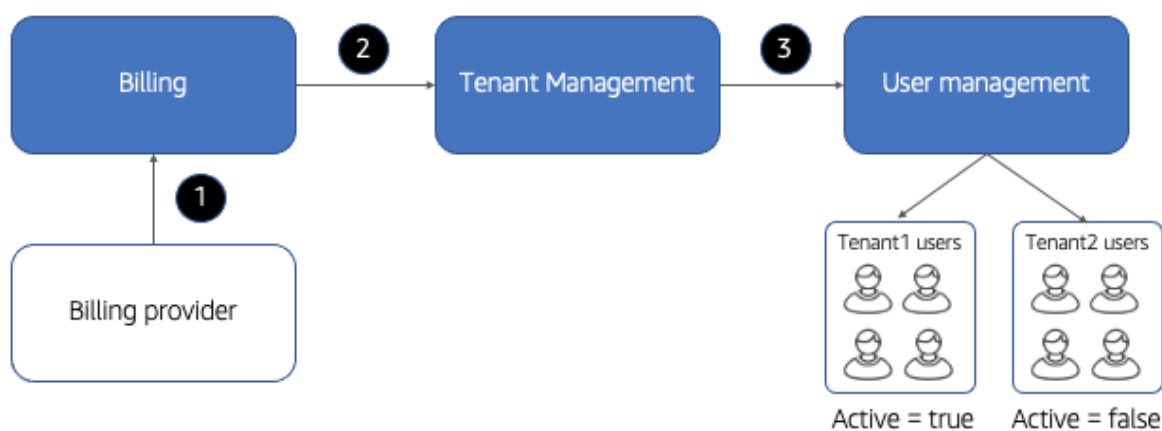


Figure 5-6. Deactivation triggered by your billing service

In this example, I've pulled in the various services that could be used to deactivate a tenant based on an event that originates from your billing provider. So, here, I have a third-party billing system that's being used to manage the billing state of my tenants. For this scenario

let's assume that a customer success person or some automation policies within the billing provider has deactivated a tenant. Now, the billing service within your control plane needs some way to acquire and react to this event. How this is achieved will depend largely on the nature of the billing providers API and integration model. Ideally, the billing provider will generate an event when a tenant is deactivated. If not, your billing service may need to have some process that periodically looks for these changes in state. In this scenario, I've presumed the billing provider is sending a message to my billing service (step 1).

Once the billing system detects this event, it will call your tenant management service with a deactivate tenant request (step 2). The service will update the state of the tenant to be inactive and convey this new state change to any part of your system that might be impacted by this change in status. In this diagram, I've presumed that deactivation is achieved by preventing tenants from authenticating. Here this is represented by a call to your control plane's User Management service, which updates the active status of all users associated with your tenant (step 3). Ideally, if your identity provider supports a group construct for your tenants, you may be able to apply this change at the group level. This would certainly be much simpler than toggling the states of individual users.

As part of deactivation, you also have to determine how/if this impacts the state of any users currently logged into the system. Do you allow them to continue or do you immediately terminate their active session? Many SaaS providers tend to lean toward a model that is more passive, allowing tenant users to finish any active sessions.

Of course, whatever can be deactivated may also be reactivated. So, you'll want to consider that path as well. This is likely just a matter reversing whatever was done to deactivate a tenant. In our example, this would just update the status of the tenant to active and re-enable authentication for your tenant.

In some cases, deactivation may be initiated by your own services (instead of via a billing system). This would mean that your admin console (as highlighted earlier) would provide a specific operation that would be used to trigger tenant deactivation. Here, your tenant management service would originate the deactivation event and update any downstream services that would be impacted by this deactivation event.

The key takeaway here is that active status of tenants must be centrally managed by your tenant management service. It should be viewed as the single source of truth for managing the state of tenants, ensuring that any impacts of changes in status are synchronized with dependent parts of your system.

Decommissioning a Tenant

With deactivation behind us, let's think more about what it would mean to decommission a tenant. You might be wondering where the line is between deactivation and decommissioning. Deactivation is only meant to suspend a tenant's account. It does not impact that existing footprint of the tenant's environment. Essentially, it's just disabling access, allowing for the fact that the tenant may be reactivated at some point.

Decommissioning would typically come after deactivation. Imagine a scenario where a tenant decides not to renew its subscription. When they reach the last day of the subscription period, your system might choose to deactivate the tenant and leave them in a deactivated state for some window of time. This strategy allows a tenant to return, reactivate, and continue to resume using their system with no impact. Here, it's as if they never left the system. This certainly creates a better customer experience. However, after a certain amount of time, the unused resources of this tenant may be contributing to costs and complexity that are not adding any value for the business. Now, you have to consider how you move from a deactivated state to decommissioning the tenant's resources.

There is no one-size-fits all approach to choosing your system's decommissioning policies. The strategy you choose will be influenced

by a variety of factors. How much overhead is a deactivated tenant adding to your system? How frequently are tenants deactivating? How are these deactivated tenants impacting the complexity of your management and operations experience? There are a wide range of factors you'll need to consider to determine how/when/if you choose to decommission tenants.

Now, let's assume you've actually reached a point where you've decided tenants do need to be decommissioned. You have a few choices here when it comes to picking a decommissioning strategy. You could choose to simply delete any resources that are associated with the tenant, essentially removing them completely from the system. Or, you could choose to archive the tenant's state before decommissioning its resources. As part of this, you'll want to reconsider what it means to rehydrate a tenant that has been decommissioned. You might, for example, leave elements of the tenant in place that have minimal impact on your system (the tenant, its users, and so on). This could make bringing the tenant back to life a somewhat simpler process without adding much cost or complexity to your environment. Ultimately, this all part of the balancing act that comes with creating your decommissioning strategy.

[Figure 5-7](#) provides a conceptual view of some of the moving parts that might be included in your decommissioning model.

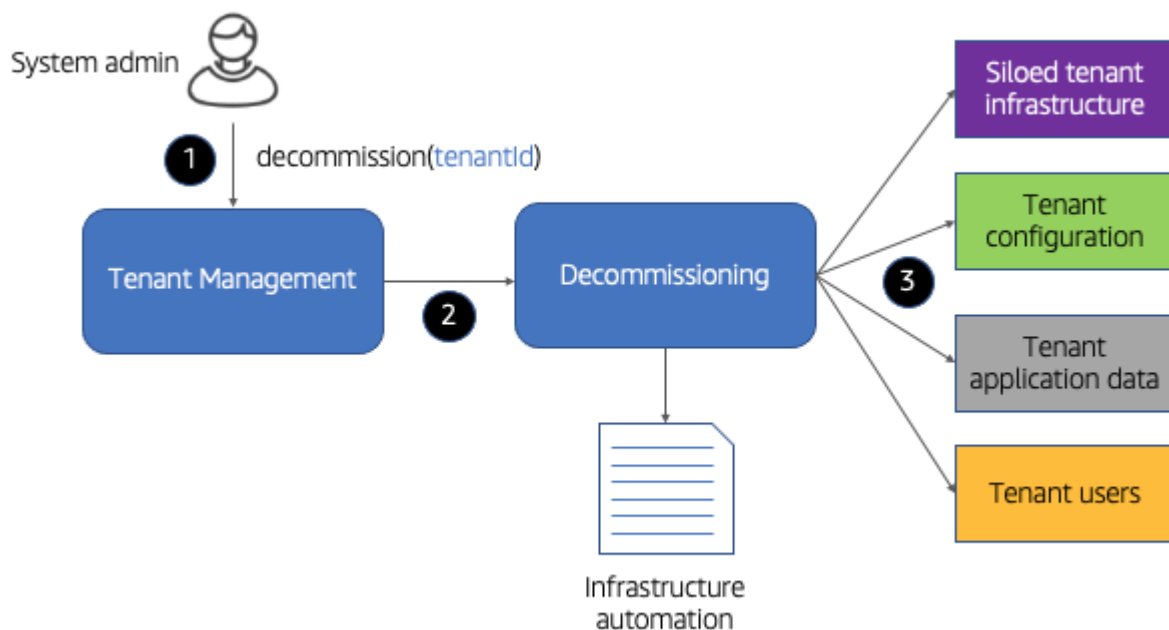


Figure 5-7. Decommissioning tenant resources

In this example, I've shown a scenario where a system administrator starts the decommissioning process which sends a decommission request to the tenant management service (step 1). At this point, you could have decommissioning implemented within the scope of the tenant management service. However, instead, I've shown decommissioning as a completely separate service that owns orchestration of all the steps needed to remove a tenant from your system. This service is then called by the tenant management service to begin the decommissioning process (step 3).

To me, it feels more natural to carve this out and allow decommissioning to exist as its own process that would be deployed, managed, and executed outside the context of the tenant management service.

The decommissioning service has the job of iterating over all the different tenant constructs and removing each one. This will be achieved through a combination of infrastructure automation tooling and scripts as well as API calls. The nature of each tenant resource may require a different decommissioning strategy.

While each SaaS solution will have its own unique blend of tenant resources, I did include a few examples in the diagram here to highlight the kinds of resources that might be touched by your decommissioning service. Naturally, if we have any siloed tenant infrastructure here, those siloed resources will be removed from your system. This may be about removing all the pieces of a full stack siloed deployment or just the individual resources that might be deployed in a siloed model. Tenant configuration and tenant users are also listed here as examples of resources that would get touched as part of this process.

Tenant data, which is shown here, can certainly be one of the more challenging areas to address when you're decommissioning a tenant. Imagine all the places where you might have pooled storage in your system. When you have pooled data, that means that your decommissioning process will need to be able to locate and selectively remove the data that sits alongside the data of other tenants. [Figure 5-8](#) provides a conceptual view of the nature of this decommissioning challenge.

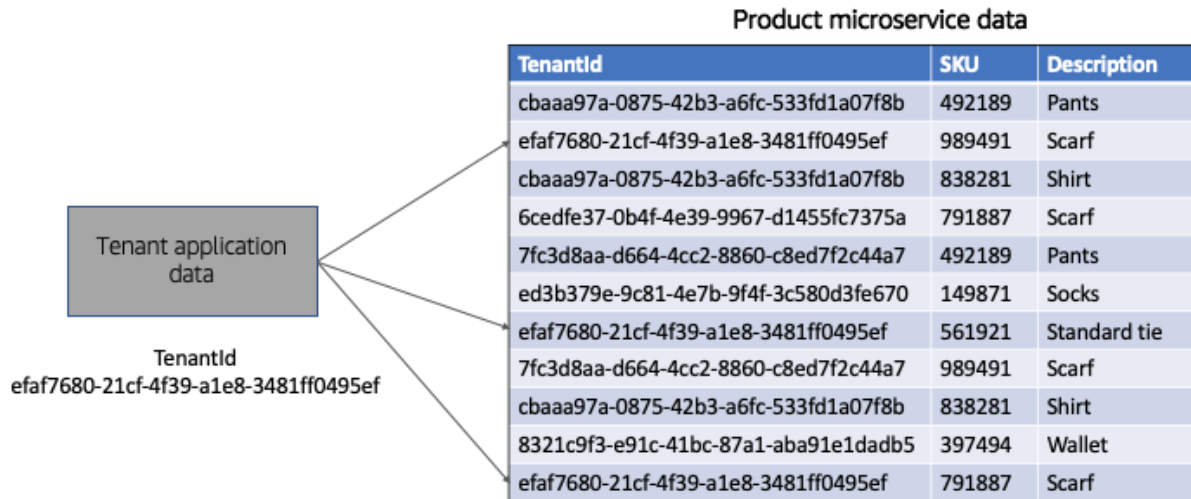


Figure 5-8. Decommissioning tenant data

Here, I've provided an example of the data that might be associated with our product microservice. Within this table, I have a `TenantId` column that includes the GUIDs that associate each product with a tenant. Now, let's assume we want to decommission the tenant with the Id `efaf7680-21cf-4f39-a1e8-3481ff0495ef`. Our decommissioning process must locate and remove all the items in this table that are associated with this tenant.

On the surface, this may not seem all that challenging. However, consider that this table is one of many in our system that may hold data in a pooled model. Each microservice in our system could be managing pooled tenant data and each of these services may rely on different storage technologies. This means that your decommissioning process may need separate code to remove data from each of these sources.

In some instances, decommissioning can be more involved than your onboarding process. The nuances of identifying and gracefully removing these tenant resources can be quite involved. Automating this process can be daunting, requiring teams to be hyper vigilant about ensuring that their decommissioning strategy doesn't impact existing tenants. This includes ensuring that the load of your decommissioning process avoids creating bottlenecks or performance issues for your existing tenants. Generally, if you can run this as an asynchronous process that has a very conservative consumption profile, you'll be better positioned to limit tenant impacts. This falls very much into the old school batch mentality where we try to run processes of this nature after hours to limit impacts. For some SaaS providers, that will work. Others may not have this luxury.

Archiving Decommissioned Data

The last piece here around decommissioning focuses on archiving tenant state and data. For some SaaS providers, this can allow them to continue to preserve existing tenant state without retaining all the other moving parts of a tenant's environment. This is especially valuable when you have high profile tenants or tenants with a significant investment in their data.

While the concept of archiving decommissioned tenant state and data is not all that complex, the variations in multi-tenant architecture

models make it difficult to prescribe any one approach for this problem. There is no standard “take a snapshot” of my tenant environment model—especially if you rely on pooled resources. Instead, this is usually more about taking on the heavy lifting of building your own tools and strategies that can navigate the nuances of your tenant environment.

Ultimately, how or if you choose to decommission data will vary based on the nature of your SaaS offering. This is often about weighing the business, cost, and complexity tradeoffs. For some businesses, the value of being able to reactivate with minimal friction could be essential to reacquiring key customers. For others, the nature of their data and the tendencies of their customers may suggest that there’s not enough upside in investing in the build out of this capability.

Changing Tenant Tiers

The last piece of our tenant lifecycle management story looks at what it means to move from a tenant from one tier to another. For many, this change represents one of the more challenging dimensions of managing tenant state.

For simple use cases, the move from one tier to another may not be all that exotic. Let’s consider a scenario where you have basic and premium tier tenants where the primary difference between these

tiers is throughput and features. Essentially, the premium tier tenants would have better overall throughput and be granted access to additional features based on their move from the basic tier. [Figure 5-9](#) provides a view of how these concepts might land in multi-tenant architecture.

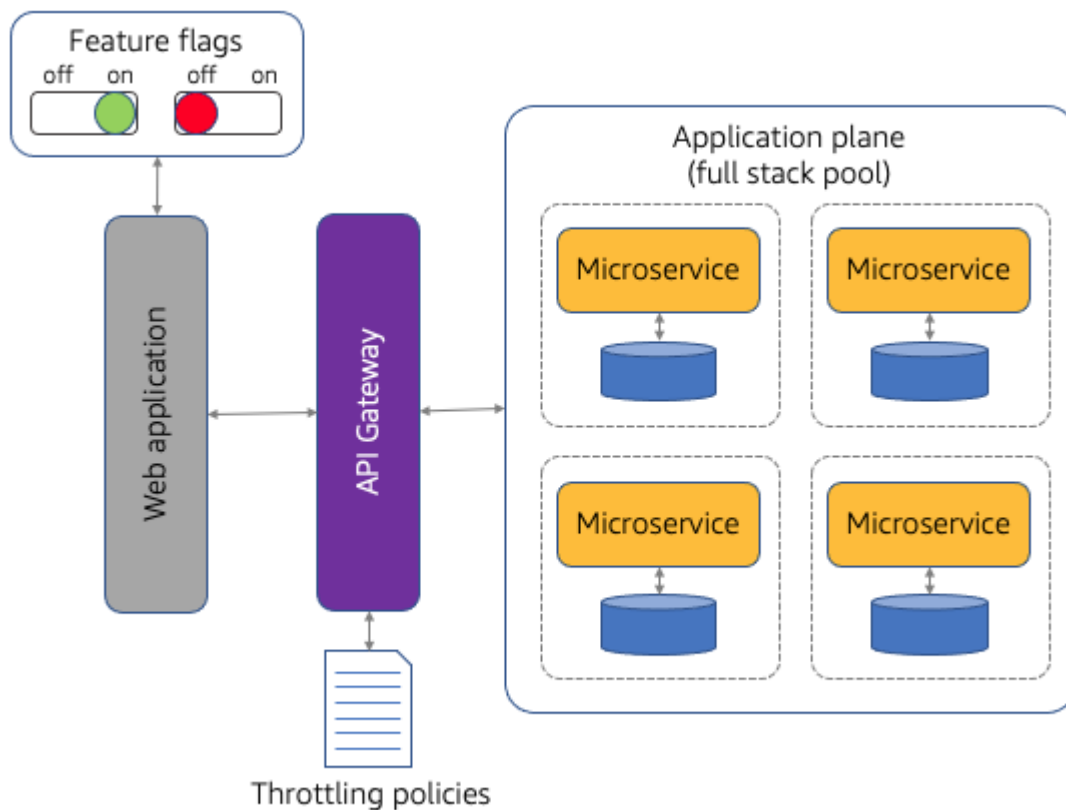


Figure 5-9. Switching tiers in a full stack pool model

Moving between basic and premium tiers in this example is limited to a few very isolated areas of this multi-tenant environment. Here, features flags are used within our application to enable paths, features, and workflows that are available to the premium tier tenant. Also, the

tier-based throttling policies that are configured as part of our API gateway will simply apply the throttling policies of the premium tier tenant to our requests (instead of the basic tier policies). This will prevent the tenant from being throttled based on the more restrictive constraints of the basic tier and, generally, be ensured of better SLAs.

You can see how moving between tiers here is a pretty straightforward model. With all of your tenant resources running in a pooled model, your system will simply update the tenant configuration to reflect the new tier. From there, the system will just use the existing feature flag and throttling policies to apply the new tier to your tenant's experience. This is the upside of having a pooled experience. You can apply changes of this nature with minimal complexity and overhead.

Now, let's consider what it would mean to move between tiers in an environment that had a more complex footprint. [Figure 5-10](#) looks at what it would mean to migrate between tiers that are using the full stack pool and full stack silo deployment models that we talked about in Chapter 3.

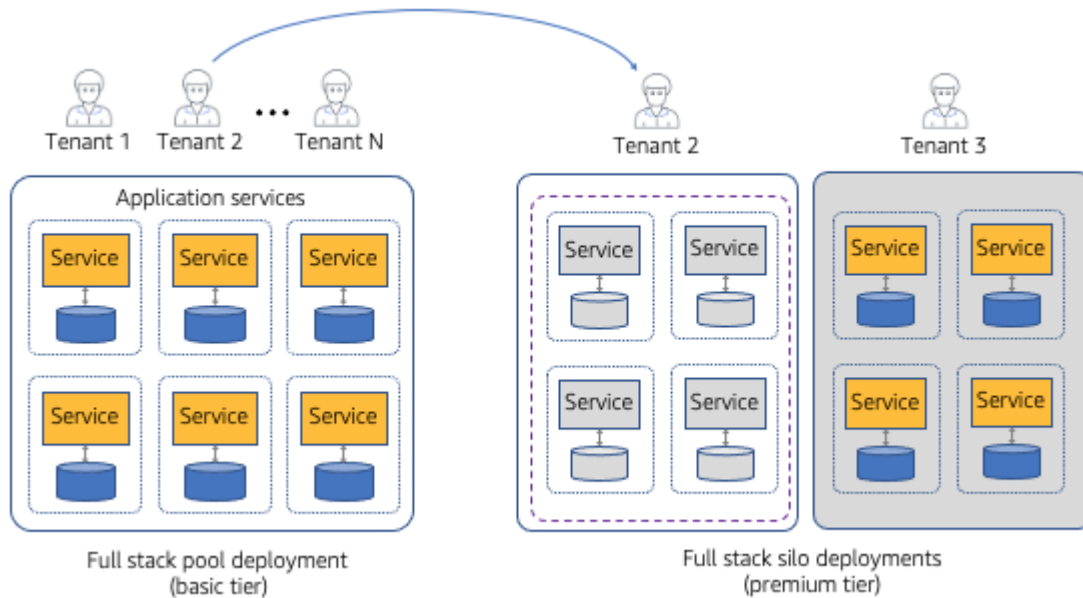


Figure 5-10. Migrating from full stack pool to full stack silo

In this example, Tenant 2 is migrating from the basic tier where it is running in a full stack pool model sharing all of its resources with other tenants. Now, when this tenant migrates to the premium tier, they are essentially moving into the full stack deployment model on the right. This migration starts by provisioning a full stack of resources for Tenant 2 and configuring any routing that's needed to send traffic to this silo. If you think about it, parts of this migration will very much simulate your onboarding experience of any premium tier tenant. In fact, it would not be unheard of to leverage parts of your onboarding code to facilitate this migration.

Where this gets a bit more tricky is when you have to move the existing state of the tenant to the full stack silo environment. Now you have to think about how/if this will be a zero downtime migration or if

you'll require tenants to deactivate to move to the new tier. You'll also have to write the new code that moves all the data and state from the pooled environment to your new full stack silo. This is where lifting gets heavy. In many respects you're taking on lots of the classic challenges that come with any environment migration. Many of the principles and strategies that are commonly used to migrate any software environment apply here—they're just being scoped at the tenant level.

While this migration to a full stack silo model has lots of moving parts, the logic and path forward is relatively straightforward. The data movement is the biggest piece of this effort. Now, let's add another wrinkle here. Let's look at what it would mean to change tiers in an environment that uses a mixed mode deployment model where each microservice in our environment may have more granular implementations of silo and pool based on the tier of a given tenant.

Consider what it might mean to migrate from a basic to premium tier in an environment that might look something like the model shown in

[Figure 5-11](#).

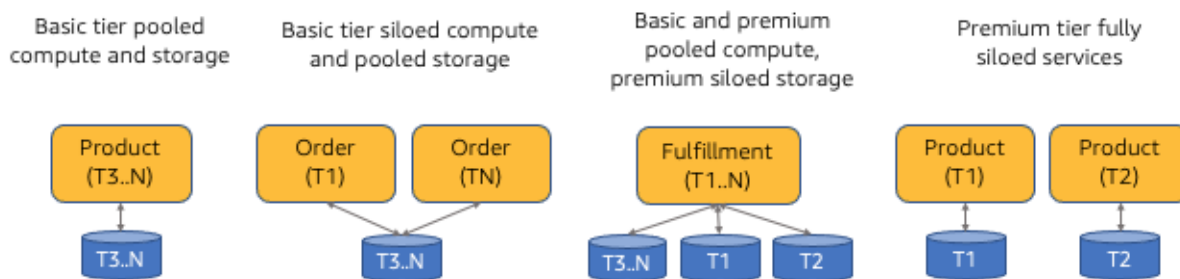


Figure 5-11. Mixed mode tier migration

Here, I've made this a bit more convoluted to highlight some of the complexities that come into the picture when you're migrating in a mixed mode deployment environment. If you look across these microservices, you'll see that they have much more granular mappings to tiers. On the left, you'll see services that are for our basic tier tenants. One interesting twist here is that the Order microservice has siloed for all tiers. In the middle, fulfillment support pooled compute for both basic and premium tiers. However, it has pooled storage for the basic tier and siloed storage for premium tiers. Then, the premium tier tenant gets fully siloed compute and storage for the Product microservice.

This very much mimics the patterns we looked at when we were examining the basics of the mixed mode deployment model. Now, though, we need to consider for a tenant to migrate from the basic to the premium tier. Let's say, for example, Tenant3 (T3) was moving to the premium tier. The steps here are less obvious. You must now

think about how each service is applying the tiering model and figure out which new resources might be needed to make this transition.

There would be no changes to the Order microservice to achieve this migration. For fulfillment, you'd need to migrate the storage from the shared model to the siloed storage model. And, finally, for the Product microservice, you'd need to provision new compute and storage for Tenant3 and migrate the data from the pooled database to this new siloed Product footprint.

Lots of the same challenges that we saw with the migration to a full stack silo follow us to this mixed mode deployment environment. Movement of the data remains a hot spot of challenges. However, you will have a more interesting web of changes that are influenced based on how you've mapped tiering to the various layers of your environment. The good news here is that, like the full stack deployment migration, this migration can also lean on some of the tooling and mechanics that are connected to your onboarding experience. The complexities associated with migration often overlap with the nuances of your onboarding experience.

For any flavor of tier migration that you consider, you'll also have to consider how this migration could impact tenants that are currently using your system. You'll want to be sure that the extraction of tenant data and state doesn't have any adverse impact on existing work-

loads. If this migration is at all disruptive to your environment, it can undermine the overall stability and availability of your environment.

Downgrading a Tenant Tier

While all the focus here has been migrating to a higher level tier, your environment must also support a model where tenants can downgrade to a lower level tier. In this mode, we're essentially looking at a model that is basically the inverse of what was done for upgrading tiers. It might just be about updating configuration to reflect your new tier or it could be more about migrating your infrastructure and data to the new tier.

If you're moving from premium to the basic tier and this move has some parts of your system moving from siloed to pooled infrastructure, then your migration now will look at how to move compute and data into their pooled constructs.

Conclusion

For this chapter, I looked at the core moving parts of managing all the moving parts of a tenant. This included looking at the overall role that tenant management plays as a microservice within the control plane of your SaaS architecture, examining the data and state that is typically managed by this service. The goal here was to highlight the role

this service plays in your broader multi-tenant environment, providing the single home and source of truth for key tenant information that is used across all the moving parts of your SaaS architecture.

As part of this topic, I also looked at how tenants are surfaced and managed through your admin console experience. The emphasis here was on outlining how this console connects to your tenant management service and is used by the administrators to manage, configure, and update your tenant information. Creating this centralized console for managing tenants often represents an essential component of your overall administration experience.

The last bits of this chapter looked at managing your tenant's lifecycle. This had us exploring the different events that can happen across the entire lifetime of your tenants, reviewing how your tenant management service would support key events that alter the state of a tenant. Some of these events are as simple as activating and deactivating tenants. Other events (like tier migration) often require a much more intensive effort to orchestrate these transitions.

Now that we've looked at how tenants are introduced and managed, we have all the elements in place for representing and managing tenants. The next logical area to explore is tenant authentication and routing. Chapter 6 will look at how tenants enter through the front door of our application and leverage the bits we've put in place in

your control plane to authenticate users. The authentication story will consider all the steps that are needed for a tenant to access your application, route to the appropriate resources, and inject tenant context into that experience. This will set up our ability to continue to push deeper into the underlying implementation of your SaaS application services.

Chapter 6. Tenant Authentication and Routing

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the sixth chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

At this stage, our focus on the control plane has been squarely on building a foundation that allowed us to introduce multi-tenancy into our architecture. Onboarding, user management, and tenant management all allowed us to configure, capture, and prepare our tenant for entry into a SaaS environment. Now it’s time to start thinking about how a tenant uses these constructs (and others) to enter the front door of our multi-tenant environment.

It's at this point, where you need to tenant authenticate a user, that all the pieces of our onboarding and tenant management come together. Here you'll see how the configuration information that was stored in tenant management can play a role in the flow and implementation of your authentication experience. We'll also see how the work that was done to connect our users to tenants will yield the tenant context that becomes essential to the downstream services that are part of your multi-tenant architecture.

For this chapter, I'll begin by looking at the fundamentals of how you expose the entry point to your multi-tenant solution. There are multiple strategies that can be used to access a SaaS environment, some of which explicitly identify the tenant that is entering the system and others that rely on internal mechanisms to determine which tenant is accessing the system. Each of these have implications on how your tenant is authenticated and connected with the appropriate identity provider.

We'll also look at how your path in the front door influences the authentication model of your multi-tenant environment. As part of this, I'll also examine how you might use different identity provider constructs to support various tenant authentication experiences. You'll see how the identity strategies we discussed in Chapter 4 will come into play as you begin to authenticate individual tenants.

The last bit of the chapter will look at how this authentication experience projects into the downstream elements of your multi-tenant architecture. This includes examining how JWT tokens are injected into your microservices and examining how the context of authentication can be used to route your tenant requests to specific elements of your multi-tenant environment. This routing context often ends up tightly connected to how you access your SaaS application.

The basic goal here is to go to the next level of bringing your SaaS environment to life, building on the foundation that I established in Chapters 1-5. It's here that we see how an actual tenant enters a SaaS environment and how the constructs we put in place allow us to authenticate tenants and acquire the context that needed to shape the rest of the downstream multi-tenant footprint of your SaaS architecture.

Entering the Front Door

Now that we have a sense of the scope of this authentication topic, let's start our discovery at its most natural point—the front door. The core elements of your authentication always begin with determining how tenants will access your application. While these access patterns seem trivial, you'll begin to see how the strategy you choose here

goes beyond the URL that's used to expose your application to tenants.

There are multiple options available to you when thinking about how you might have tenants access your system. You may want the domain of your system, for example, to include a tenant name and rely on this domain as part of your tenant-mapping and routing strategy. Or, you might just allow tenants to have their own unique domains based on branding or other considerations. In either case, the domain still typically ends up playing some role in identifying the tenant that is accessing your system. Some SaaS solutions, however, have no dependency on the domain, using a single domain for all tenants. In this approach, you'll need the authentication flow of your environment to inject the tenant context into the system.

The key here is that—even as you're selecting a path into your application—you're making choices that can have a cascading impact across other dimensions of your SaaS architecture. This access strategy may even have impacts on the technologies and services that are used to implement specific elements of your solution.

In the sections that follow, I'll look at some of the common patterns that are used when selecting an access model for your SaaS environment. With each of these patterns, I'll review some of the considerations that are associated with each approach.

Access Via a Tenant Domain

One of the common ways that tenants enter the front door of your application is through a domain. More specifically, tenants can enter with a domain that includes information that can be used to identify a tenant. [Figure 6-1](#) provides a conceptual view of how the context of this domain influences the downstream footprint of your multi-tenant architecture.

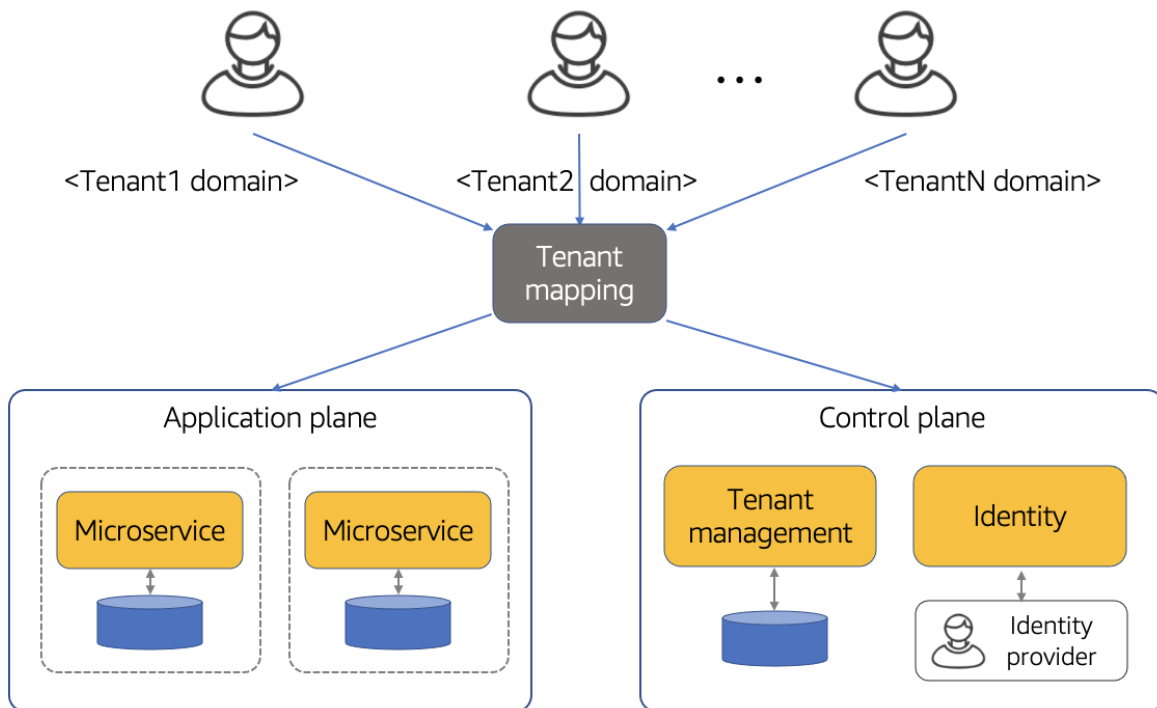


Figure 6-1. A domain driven access model

Here you'll see that I've shown a range of different tenants that are accessing a SaaS environment using a domain. In this model, each

tenant's domain is configured during the onboarding process. Once the domain is configured, the URL is shared with the tenant as their entry point into your SaaS service.

With this approach, all inbound tenant requests from these different domains are routed through what I've labeled as tenant mapping. This box represents a conceptual placeholder for the different technologies, infrastructure, and/or services that would be used to map a tenant's domain to the appropriate backend services. This mapping service is usually needed to extract the incoming tenant context from the domain and use that context to resolve any mappings to dedicated tenant resources.

There are at least two common places this mapping may be applied. The first of these is authentication. When a tenant user is being authenticated, your system may need to map an incoming authentication request to its corresponding identity construct. This applies, primarily, to tenant environments that don't have a single, shared identity provider. In these cases, your authentication may be supported by a separate identity provider or they may have distinct identity constructs within a single identity provider (groups, user pools, etc.) that are bound to individual tenants to provide specific features or experiences.

In [Figure 6-1](#) you can see how this authentication mapping unfolds. In this example, the tenant accesses your system with a tenant-specific domain. Then, that domain goes through the tenant mapping service, which extracts the tenant information from the origin of the incoming request. This information is then used to lookup the tenant identity setting in tenant management, which is then used to authenticate the tenant user against the target identity provider/construct.

The other area where tenant mapping is applied is related to the routing of application requests. In instances where your architecture is using one or more siloed tenant resources, your incoming tenant requests will need to be routed to each of these specific resources. Here, the mapping service must use the tenant context to identify that route that will be taken for a given request. We'll cover this use case later in this chapter.

While the technologies used to support the unique tenant domains can vary, the fundamentals are typically not all the different. How, where, and when it is applied will vary based on what you need to do with the context of this domain. It might be used to lookup a tenant identifier, map your tenant user to an identity provider, configure routing, or perform any other operation that may need tenant context to process a request.

It's also worth noting here that the domain name (or subdomain) represents a friendly, publicly visible name that you've exposed as the URL entry of a tenant. This name may change over the life of the tenant and, as such, is maintained entirely separate from the internal notion of a tenant identifier which will not change over the life of the tenant.

The Subdomain-Per-Tenant Model

At this point, the fundamentals of the domain-per-tenant model should be clear. Of course, even though I've talked about domains generically, the reality is there are multiple ways to associate domains with tenants. In many SaaS environments, there's a desire to use domains to identify tenants without requiring each tenant to have a completely unique domain. In this scenario, you might have a SaaS company (ABCSoftware, for example) that owns the abc-software.-com domain and they want to use that domain as part of the presence and branding of their SaaS experience. Their customers, in this example, also don't see value in having their own domains.

In this scenario, it's very natural to use subdomains as a way to identify tenants without creating an entirely new domain for each tenant. So, with our abc-software.com example, you would prepend a friendly tenant name to the existing domain. The end result would yield domains like tenant1.abc-software.com and tenant2.abc-soft-

ware.com. You've likely seen this pattern implemented across existing SaaS solutions that you consume today.

This approach is often very appealing to SaaS providers, enabling them to provide a unique access point for each tenant without absorbing the complexity and overhead of creating a unique domain for each tenant.

The Vanity Domain-Per-Tenant Model

The other option here is to use a vanity domain for each tenant. This approach is often used in scenarios where a tenant's are presenting a branded experience to their customers. In fact, in these scenarios, the tenant's users may have no visibility/awareness of the underlying SaaS system that they are running.

To better understand this model, consider an example where a SaaS provider offers an ecommerce solution. With this offering, tenants use the SaaS environment to configure and host their own, privately branded stores. In this scenario, tenants would have their own unique domains that are used to access your multi-tenant environment.

In most respects, this vanity domain model looks very much like the subdomain model that we outlined. It still uses the origin to extract the

tenant name and then map it to tenant context for downstream consumption.

Onboarding with Tenant Domains

When you use a domain to identify tenants, you have to consider how this will influence the design of your tenant onboarding model. Now, as each new tenant is created, you'll need to configure the setup of these domains and create any mappings that will be needed to support a tenant's entry into your environment.

The subdomain-per-tenant model has a lighter weight impact on your onboarding flow. Here, your effort is primarily focused on setting up the various DNS configuration options that are part of your environment. [Figure 6-2](#) provides an example of some of the elements that could be configured as part of onboarding a new tenant in a subdomain-per-tenant model.



Amazon CloudFront

Origin	DomainName	Alternate Names
App-Bucket	https://abc123.cloudfront.net	app.saasco.com
		tenant1.saasco.com



Amazon Route 53

Record Name	Type	Route to
app.saasco.com	A	https://abc123.cloudfront.net
tenant1.saasco.com	A	https://abc123.cloudfront.net

Figure 6-2. Configuring tenant subdomains during onboarding

Here, I've outlined a specific AWS example that shows how you'll need to configure the moving parts of your network routing to support the subdomain-per-tenant model. At the top of the diagram I have shown the configuration of the Content Delivery Network (CDN) that will be processing our tenant requests. It uses Amazon's CloudFront service. The table I've shown here illustrates how you'll configure the CDN to support tenant subdomains. The first row in this CDN table includes the settings that are configured when you initially provision your baseline SaaS environment. In this example, the general domain that was assigned the alternate name of app.sassco.com, where saasco represents the branded domain name of our fictitious SaaS company. Now, when a new tenant is onboarded to the system, you must add a new row to this CDN table. In this case, I've added one new tenant that gets configured as tenant1.sassco.com.

In addition to configuring the CDN, we must also set up the DNS routing for this new tenant subdomain. The table at the bottom of [Figure 6-2](#) provides an example of how you might configure your DNS service. For this example, we're looking at how this is achieved with the Amazon Route 53 service. The table here shows two entries. The first row contains the baseline set of values that are populated when you first set up your environment. The second row gets populated when you onboard tenant1. This creates the A record that points the tenant1.saasco.com subdomain to the general domain of your SaaS environment.

While many of the steps to make this work are mirrored in the vanity domain model, there is one key difference you'll need to factor into your domain onboarding story. When a tenant requires a vanity domain, your onboarding process will need to support a way to bring that domain to life within your multi-tenant environment. If the domain exists and is being migrated to your environment, your onboarding flow will need to include the steps required to make this migration happen. However, there may also be instances where the tenant creates their domain as part of their onboarding process. As you can imagine, this would be a much more involved process. The complexity here is often centered around the various steps required to validate and register a new domain. Choosing to support automation of this entire experience can represent a significant investment for some teams.

While AWS services were used here to illustrate the onboarding experience, the fundamentals of configuring this experience would likely be similar with other tools and services. The key here is that you'll need to consider how you will automate the configuration of these domain settings as part of onboarding new tenants.

Access Via a Single Domain

So far, I've focused mostly on strategies that rely on domains to identify tenants that are entering the front door of your application. However, some SaaS providers—especially those using a B2C model—will use a single domain for all tenants. In these instances, all tenants will use the same domain to access your SaaS environment. [Figure 6-3](#) provides a conceptual view of an environment with a single domain.

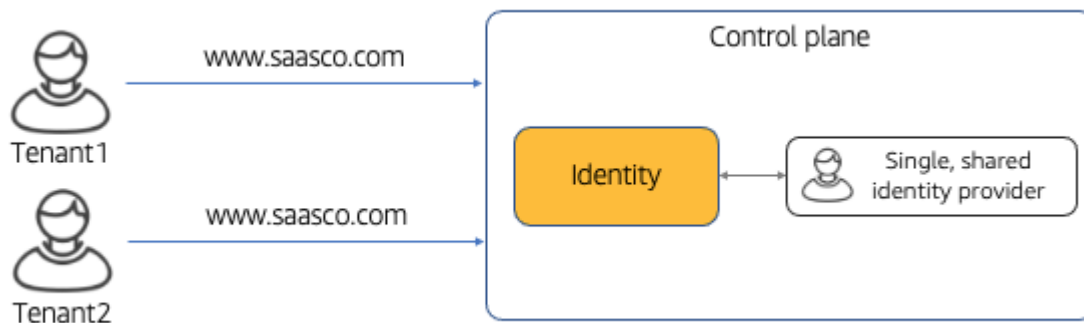


Figure 6-3. Single domain with a shared identity provider

In this example, I have two tenants that are accessing my system via a single, shared domain (www.saasco.com). Now, as these tenants

attempt to authenticate an existing tenant user, they are directed to the identity provider hosted within our control plane. Since this model uses a global identity construct to house all of our tenant users, it can authenticate all tenants against this single endpoint without any real challenges. Of course, having all of your users in a single identity provider construct will also mean that each user of your system can be associated with one and only one tenant.

Where this gets more interesting is when your architecture supports more flavors of your authentication experience. If you recall, in Chapter 4 I talked about how identity providers may offer different grouping constructs for users that allowed you to have separate authentication policies for each tenant. If your identity provider supports these grouping constructs, this can enable you to offer authentication options for the different tiers of your solution.

[Figure 6-4](#) provides an updated view of the single domain strategy that employs separate identity constructs for your tenant.

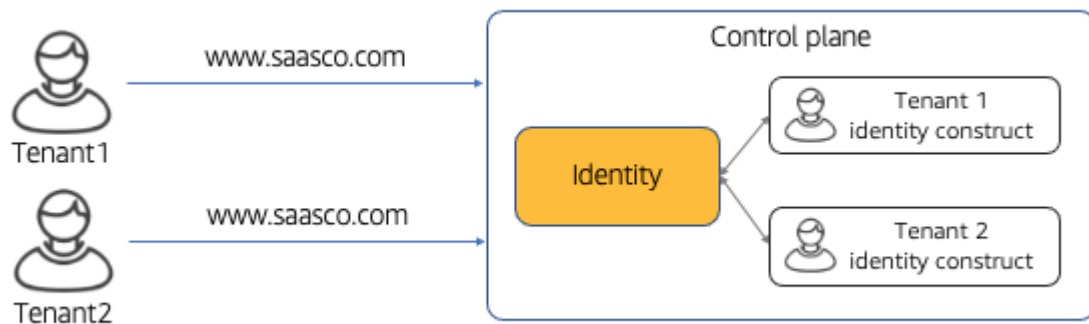


Figure 6-4. Single domain with separate identity constructs

This is mostly a mirror of the diagram in [Figure 6-3](#). However, the one change here is that we now have separate identity constructs for each of our two tenants. This is where things get a bit more complicated with the single domain model. Without a domain to identify tenants as they enter the environment, you have no context that will allow you to determine which identity construct should be used to authenticate your users. Each tenant request looks the same to your environment regardless of which tenant initiated the request.

There are a few ways you might approach resolving tenant context for your authentication flow. One strategy here is to use the domain of your user email address to associate the user with a given tenant. In this model, you'd presume that tenants coming from a specific domain/customer would be mapped to the tenant for that customer. This can work if you can presume that all tenants from a domain belong to a specific tenant. However, this does impose limits on your ability to support a broader range of users with various email domains. Anoth-

er possibility here is to consider mapping individual user identifiers to specific tenants. The diagram in [Figure 6-5](#) provides a conceptual view of this approach.

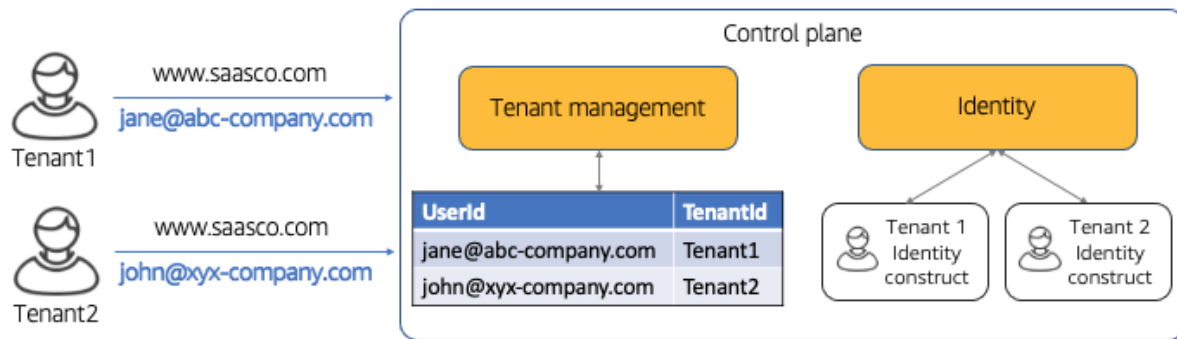


Figure 6-5. Authentication mapping based on tenant identifiers

In this example, you'll see that we inherit the same single domain model I've used throughout this section, using www.saasco.com as the single entry point for all tenants. I've also added two sample user identifiers here that represent the different values that are used during the authentication experience. To be able to authenticate in this model, you'll need to first resolve the user identifier to its corresponding tenant. This is represented here with the table that is associated with the tenant management service, mapping individual users to tenants. In reality, this mapping is likely a mapping from the user to a tenant's identity construct. Once you have that identity construct, you can authenticate the user against the target identity construct. This model still relies on having a one-to-one relationship between a

userId and a tenant. However, it does let you have separately configured identity policies for tenant tiers.

The Man in the Middle Challenge

In looking at the solution in [Figure 6-5](#), you'll notice that this strategy relies on a level of indirection to successfully map tenants to their identity providers. This does present some challenges if you're trying to precisely conform to standard authentication models. To better understand this, let's step back and consider a more traditional web application authentication implementation.

[Figure 6-6](#) provides a conceptual view of a classic authentication experience that you may have implemented multiple times. This flow starts with a tenant user attempting to access your web application (Step 1) and is redirected to an identity provider to be authenticated (Step 2). When the user successfully authenticates, the identity provider returns tokens that will provide identity and authorization information (Step 3). The identity provider then directs you back to the web application as an authenticated user (Step 4) before calling one or more downstream microservices with this authentication context (Step 5).

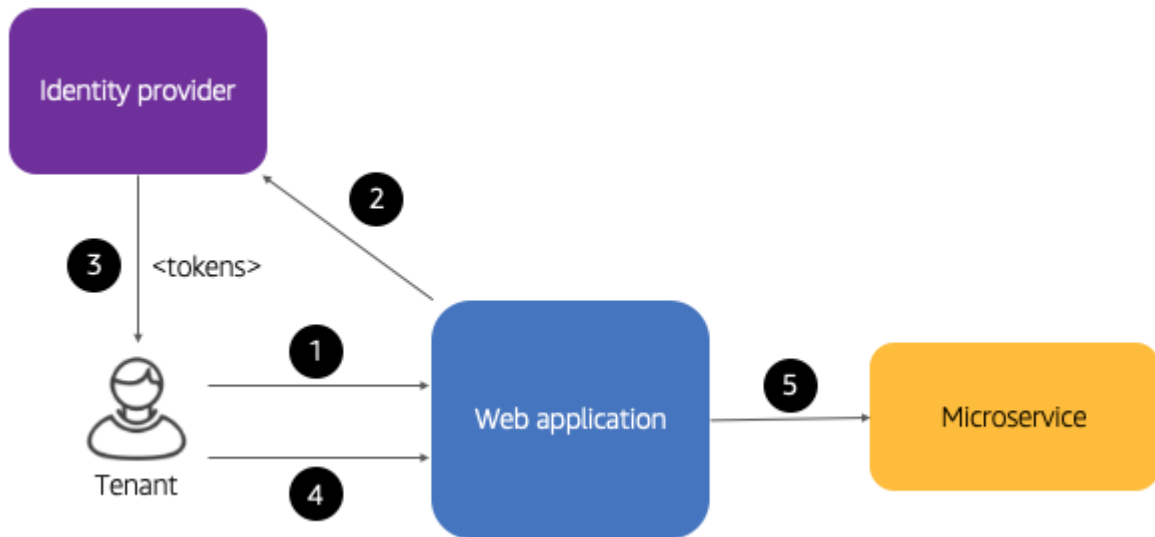


Figure 6-6. A classic web application authentication flow

The beauty of this flow is that it is orchestrated entirely by the identity constructs of your environment. The path to the identity provider and back into your web application is mostly outside of the control of your code and conforms to the standard authentication flows that are supported by identity providers.

Ideally, you'd be able to implement your SaaS solution without straying too far from this flow. At the same time, I've also talked about how different identity configurations and access patterns may rely on code that can map a tenant to its corresponding identity construct. These strategies often require the injection of additional mapping constructs that reside within the flow of your authentication experience.

In [Figure 6-5](#), for example, you saw an instance where the tenant management service was used to lookup and map a user to an identity construct. This approach means that your authentication flow cannot go directly to the identity provider to process your authentication. Instead, it must first determine the target identity construct and then drive the redirection to the appropriate identity model.

Any time you add this layer of indirection to your authentication flow, you're essentially adding a point of scale and failure to the authentication model of your environment. While this may be exactly what you need to do, you want to be sure you consider the tradeoffs associated with injecting yourself into the authentication flow.

The Multi-Tenant Authentication Flow

It should be clear at this point that there's a strong connection between how you enter the front door of your application and how that influences the overall authentication experience of your multi-tenant architecture. Now, let's assume you have a path into your environment that enables you to direct authentication requests to the appropriate identity provider construct. Assuming those bits are lined up, we can look more closely at the remaining steps in the authentication process.

It's important to note that this process is squarely focused on authenticating a tenant user and returning the SaaS identity that was described in Chapter 4. Much of the onboarding experience that was covered in that chapter was setting the stage for this moment of authentication, configuring our identity model with all the pieces needed to authenticate and user and return the tokens that will represent the tenant context that we'll need for all the downstream elements of our multi-tenant implementation.

Generally, the moving parts of this experience align with the OAuth and OIDC specifications that are implemented by most identity providers. Still, it's helpful to see the end-to-end flow of this experience to give you a better sense of what's happening. It also connects key dots in illustrating the injection of tenant context that's been referenced across several dimensions of our multi-tenant architecture discussion.

A Sample Authentication Flow

While we've poked around elements of authentication concepts, let's look at a sample flow to see all the moving parts in one end-to-end flow (shown in [Figure 6-7](#)).

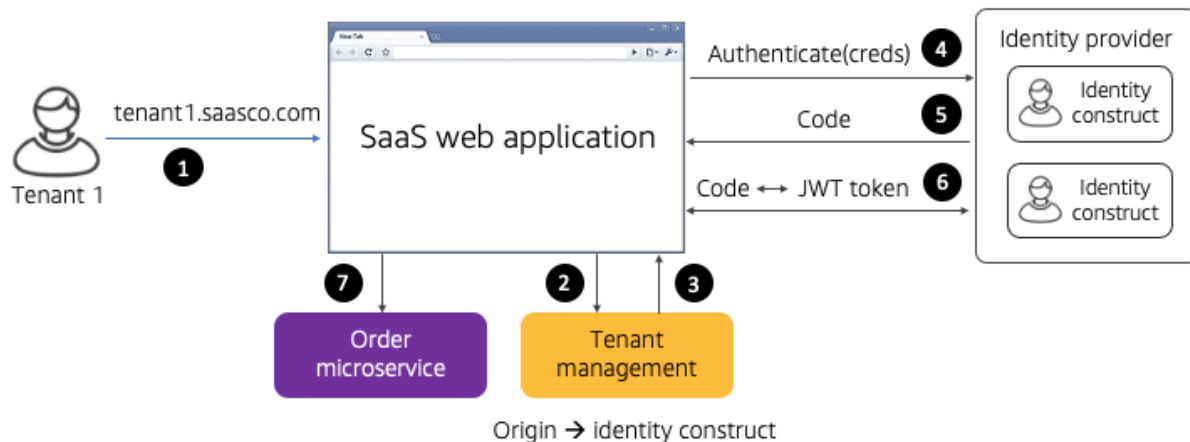


Figure 6-7. A sample multi-tenant authentication flow

In this example, I have used a subdomain-per-tenant model as part of this authentication flow. Tenants that are authenticating into this environment enter via their assigned subdomain, in this case `tenant1.saasco.com`. The tenant user hits our web application (Step 1) and our application determines that the user is not authenticated. Since the user is not authenticated, your web application will redirect your user to the application login experience where the user will enter their credentials. This is a bit different from the classic authentication flow in that we'll rely on the web application to detect and direct the user to the login form.

Now, before you can authenticate the user, you'll need to determine the specific identity provider construct that is associated with this tenant. This is achieved by calling the tenant management service (Step 2) and requesting the information for the target identity constructs that will be used to authenticate the user. The tenant management ser-

vice will inspect the origin (subdomain) of the tenant and lookup its mapping to the identity provider and return this information to the web application (Step 3). Now, you have everything you need to authenticate the tenant user. The next steps in this process follow the classic bits of an OAuth flow. First, we call the identity provider, passing along the tenant user credentials (Step 4). The identity provider will then return a code (Step 5) before exchanging this code for a JWT token (Step 6). Finally, now that you have the JWT token with our tenant context, this token can then be injected into the calls to our microservices (Step 7).

The tokens that come out of this process, as discussed in Chapter 4, are injected into the downstream services as bearer tokens where the authorization header of your HTTP request is set to be a “bearer” token and assigned the value of the access token that comes back from your authentication flow. Then, your downstream services can use this token to implement the authorization of your microservices and provide access to a request’s tenant context.

Federated Authentication

Where the federated authentication story gets more complicated is when we start to consider scenarios where your identity footprint might be more distributed. If your solution, for example, needs to authenticate against some externally hosted identity provider that’s out-

side of your control, this can add a layer of complexity to your overall multi-tenant authentication model.

When you control all aspects of the identity experience, you can control the custom claims and policies through your identity provider. However, when a third-party is authenticating your user in some federated model, it's less clear how these essential moving parts of the multi-tenant identity experience can be supported. You can't really require that third-party to include the custom claims that provide tenant context. At the same time, our multi-tenant design relies heavily on its ability to issue JWT tokens that include this tenant context.

The good news is that there are federated identity solutions that can fill in bits of the multi-tenant experience. With Amazon Cognito, for example, you can have custom claims configured within Cognito for users that will be authenticated from a third-party provider. In this model, when you authenticate against the federated provider, Cognito can seamlessly stitch its custom claims into the JWT tokens that are returned from this authentication. This gives you the ability to support a third-party provider while still retaining the ability to manage custom claims for each user.

There are any number of different federated identity models that you may need to support in your SaaS environment. Each identity provider also tends to have its own unique approach to federating

these users. Some may allow you to inject the custom claims and others may not. The key here is that—if you’re federating to a third-party provider—you’ll have to determine which strategy you’ll be using to acquire tenant context. In some cases, this may require you to manipulate or inject JWT tokens to achieve the desired experience.

No One-Size-Fits-All Authentication

This flow in [Figure 6-7](#) represents one of many approaches that can be taken to authenticate a user in a multi-tenant environment. This is part of the general challenge for SaaS solutions. The path into your application, the identity constructs you’re using, and other factors may require you to consider a variety of different approaches to connect the dots of your identity flow.

The key here is that this is rarely as simple as hitting an identity provider and following any one of the typical authentication flows that are outlined by various identity providers. Instead, you have to wrap your multi-tenant requirements around these identity flows to align them with the multi-tenant patterns that are implemented in your environment. In the end, you’re definitely conforming to the OAuth and OIDC specifications that are supported by identity providers, but your path into those flows is shaped by how tenants are mapped into the overall identity scheme of your multi-tenant environment.

Routing Authenticated Tenants

After you've made it through the front door of your application and you're ready to invoke backend services, you still need to consider how tenant context can influence the routing of these requests. Now, you could argue that this is not part of authentication and, technically, you'd be correct. However, the context that comes out of the authentication process has a direct impact on the downstream routing story of your application. So, it seemed natural to make routing part of this discussion. You'll also see that, in some cases, your routing strategy might also influence the authentication model that you select.

When I talk about routing here, I'm generally referring to how tenants are mapped to their corresponding resources on a request-by-request basis. [Figure 6-8](#) provides a conceptual view of the routing mental model.

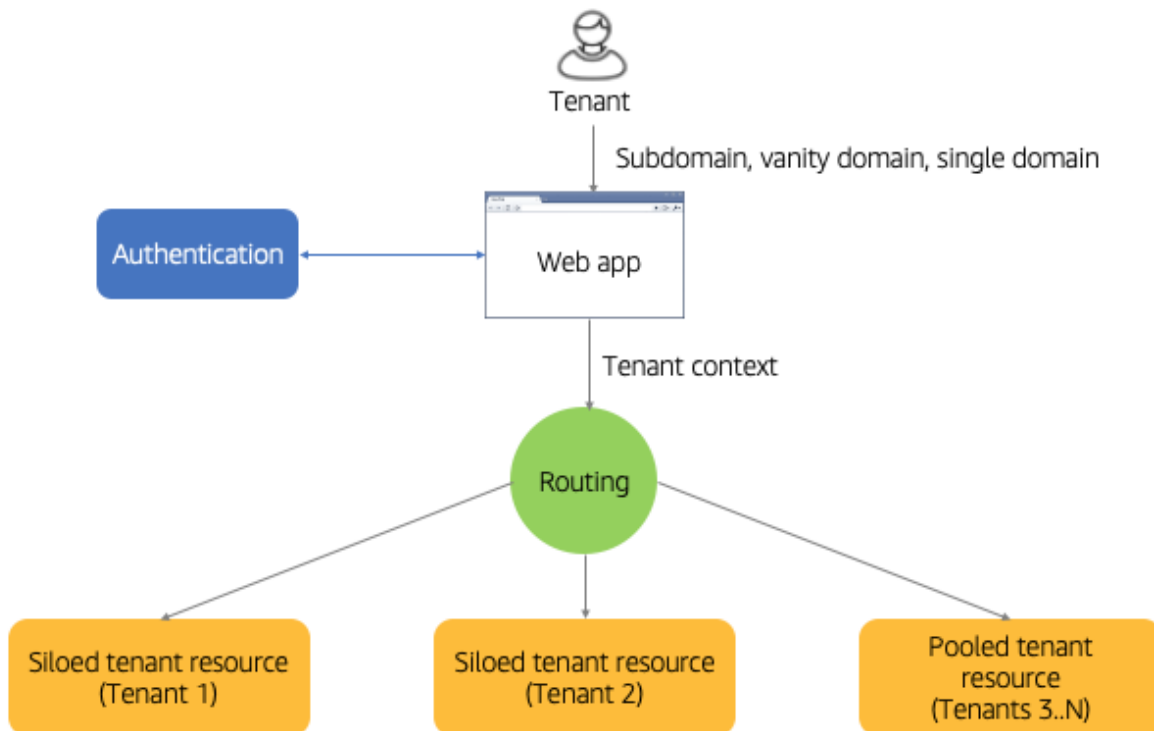


Figure 6-8. Tenant routing basics

Here, you'll see that I have a tenant entering the front door of our SaaS environment and authenticating with any one of the models discussed earlier. Once you've authenticated, your application will begin consuming the microservices and functionality of your SaaS application. Now, if all of your application services are running in a pooled model, you can just make these calls directly. However, it would not be uncommon for you to have a more diverse architecture footprint for your application services, some of which are running in a silo model and some of which are running in a pooled model.

When you have a mix of tenant deployment models, you'll now have to consider how you'll route requests to the appropriate tenant re-

sources. In [Figure 6-8](#), I've shown a simple conceptual model where we have Tenants 1 and 2 running with siloed resources and the remaining tenants running with pooled resources. With this split of silo and pooled resources, you'll be required to introduce some notion of routing in your multi-tenant architecture to determine how this routing will be achieved.

As you can imagine, the approach you take here will vary based on the technology stack you're using, the front door entry point of your application, and the deployment model of your application. Certainly, though, the approach you choose for users to enter and authenticate will have a significant impact on the options you'll have available when you're looking at routing strategies. A tenant specific domain, for example, might be able to be combined with other networking tools/services to implement your routing strategy. So, as you're thinking about the entry into your application, you should also be thinking about how it will align with the routing requirements of your SaaS environment.

The routing model of your application also has implications for the onboarding experience of your SaaS solution. As each new tenant is onboarded to your system, you may need to update the configuration of your routing infrastructure to provision/configure the constructs that will be needed to route the workloads of this new tenant.

Routing with Different Technology Stacks

I mentioned that the technology stack you're using can have an influence on the routing model of your environment. While there's far too many permutations here to cover all the possibilities, I thought I'd review a few examples of different multi-tenant technology stacks and look at how you might implement routing in these different models.

For this discussion, I'm going to look at two of the more common SaaS technology models: serverless and containers. The sections that follow will examine some of the nuances associated with routing for each of these stacks to give you a better sense of some of the variables that come into play as part of developing a tenant-aware routing model.

To get into the details of any routing strategy, I'll have to discuss specific technologies. The tools and mechanisms available for serverless on AWS for example might look somewhat different when realized on Azure or GCP. The routing models for Kubernetes, however, are likely to be more similar (with caveats).

Serverless Tenant Routing

Let's start by looking at how you might implement a routing strategy that uses serverless technology to implement its multi-tenant application. In a serverless environment, you will have a set of functions that

are composed to create the various microservices that represent the functionality of your application.

With AWS Lambda, these functions are typically accessed through an API Gateway. This gateway describes and exposes the HTTP entry points into your services, mapping requests to their corresponding functions. In this respect, you can view the API Gateway as an essential path through which all activity flows to your application services. In fact, as we look more at a detailed serverless SaaS implementation in Chapter 15, you'll see how this gateway plays multiple roles in your overall multi-tenant architecture. For now, though, let's just focus on how the gateway(s) are provisioned and configured to support the basic routing needs of your SaaS environment.

[Figure 6-9](#) provides a conceptual view of a basic serverless SaaS environment constructed with AWS Lambda.

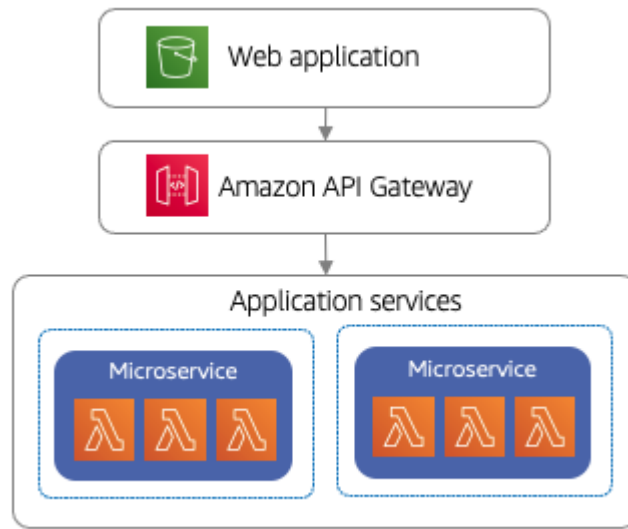


Figure 6-9. API Gateway routing in serverless environments

This figure represents the most basic of serverless SaaS architectures. Here, we have a web application that's hosted in an Amazon S3 bucket. This application makes requests through the Amazon API Gateway which are then routed to the Lambda functions that are composed as part of the various microservices that are part of your application.

Now, if the functions of our serverless application are all pooled, then the role of the gateway is pretty straightforward here. Here, all requests are simply directed to their target function with regard for tenant context. However, imagine a scenario where some or all of your microservices (and functions) are running in a siloed model. This is where you'll need to evaluate the tenant context and route requests to the correct tenant functions. [Figure 6-10](#) provides an example of a

scenario where we have a mix of siloed and pooled compute resources that require targeted routing.

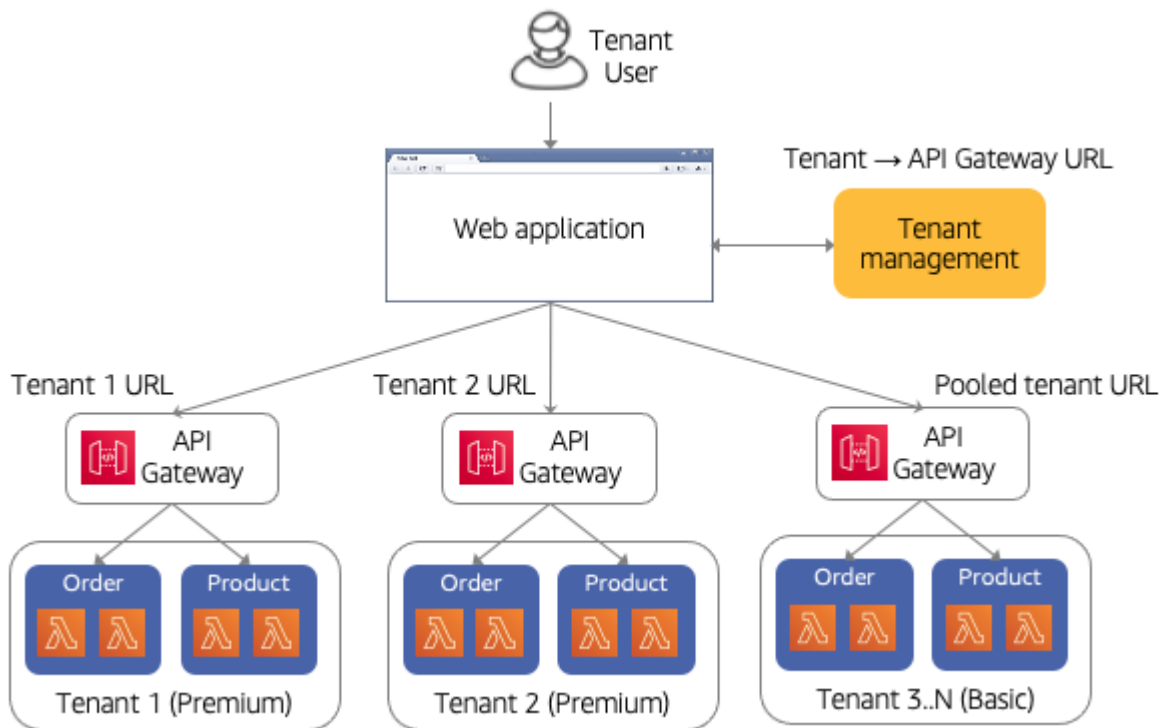


Figure 6-10. Routing to tenant-specific gateways

In this diagram, you'll see that I've opted for an API Gateway per tenant model where I provision and configure separate gateways for the different tiers of my application. I have two premium tier tenants (Tenants 1 and 2) that have siloed Lambda functions that implement their microservices. I also have basic tier tenants running in a pooled model where their functions are shared by all tenants. If you have a mixed mode deployment (some silo and some pool), then your routing will be a bit more contextual. Based on the nature of the request, you

could be sending some traffic to a dedicated API Gateway URL and some traffic to a shared API Gateway URL.

As each request is processed in this model, I need to use tenant context to identify the tenant and route these requests to the appropriate gateway URL. There are multiple ways you might achieve this mapping. If, for example, your tenants used a subdomain to access your system, you could use the origin of your HTTP request header to map the tenant to a specific gateway URL. In [Figure 6-10](#), I presumed that I had a single domain for all tenants which required the use of the tenant management service (in the top right of the diagram) to lookup the API Gateway URL for each tenant.

The downside of this particular model is that it requires the client to play a role in this mapping exercise. The client must acquire the URL from the tenant management service and apply that URL as part of making requests.

There are certainly downsides that you'll need to consider with this mapping model. Performing these mappings on every request may add overhead and latency that impacts the performance of your solution. Some address this by resolving this mapping on the client side, but it feels unnatural to distribute this problem to the client. The more typical approach here is to introduce a caching strategy at the mapping or gateway levels to hold recently mapped tenants. The key here

is to be sure that you're factoring the performance impacts of this mapping into the overall strategy.

Scale is also an area that deserves attention here. Having a gateway-per-tenant may not scale well in environments with a large number of tenants. This is where limiting the number of siloed tenant resources becomes an important piece of the puzzle. If you have a few premium tier tenants and the rest are basic, you'll likely be fine. However, if the number of premium tier tenants is expected to grow substantially, you'll want to reevaluate your options.

Container Tenant Routing

While the serverless routing example was more focused on mapping tenants to specific API Gateway entry points, this next strategy is more driven by the fundamental nature of how serverless applications are built and deployed with AWS Lambda. For contrast, let's look at how you might implement tenant-aware routing in an environment where your application plane is primarily built with Kubernetes.

Within a Kubernetes multi-tenant architecture, you have lots of native constructs that can be used to shape the routing footprint of your SaaS application. The list of options is pretty extensive, but, for now, I want to illustrate how you might use a service mesh to build out this

routing experience. If you're new to the idea of a service mesh, you can think of it as a Kubernetes platform mechanism that lets you configure/implement different security and observability aspects of your system (outside of your application). There are multiple service mesh implementations. For this solution, I've chosen to use Istio to implement our routing model.

[Figure 6-11](#) provides a conceptual view of a multi-tenant Kubernetes environment that uses a service mesh to control the flow of both your authentication and your routing to tenant environments.

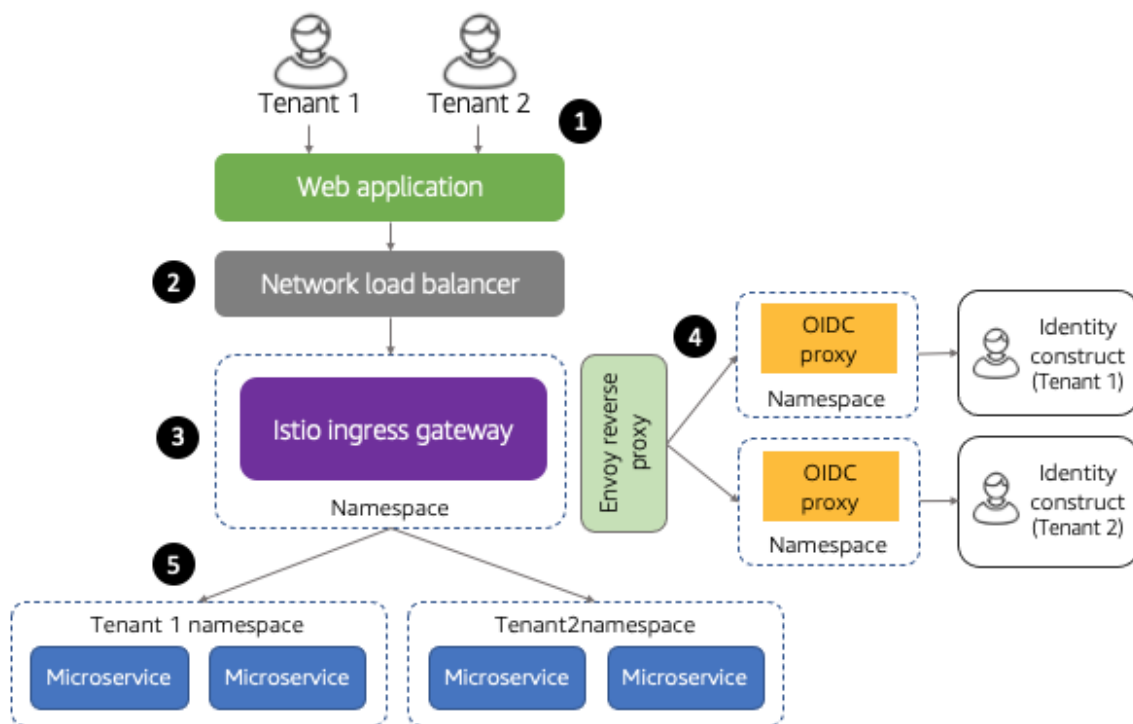


Figure 6-11. Routing tenant requests with a service mesh

Here, you'll see that our tenants enter through the front door using the subdomain-per-tenant model described earlier in this chapter (Step 1). Requests from your web application are, for this example, sent through network load balance that provides a durable endpoint that can be easily referenced from the web application (Step 2). The request then passes through the network load balancer and arrives at the Istio ingress gateway that runs in its own namespace within your Kubernetes cluster. It's this gateway that orchestrates and applies all the policies that are required to route to our identity providers as well as our application microservices.

Let's presume that, in this example, the tenant is not authenticated. In this scenario, the gateway will send the requisition off through an Envoy reverse proxy (Step 4) which will use the incoming origin subdomain to route authentication requests to the tenant-specific OIDC proxy that is also running in a separate Kubernetes namespace. Each of these OIDC proxies forward authentication requests along to the corresponding tenant identity construct. This could be separate identity providers. It could be separate grouping constructs within a provider.

Once your tenant user has been authenticated, the gateway can send your request to the services of your application (Step 5). In this example, you'll see that I have two tenants running in separate Ku-

bernetes namespaces. This means our gateway must examine the origin and route each request to the appropriate tenant namespace.

While there are a lot of moving parts to this solution, to me it still feels a bit more graceful than those strategies that rely on your services lookup and map tenants. Here, we're able to lean on the natural routing and proxy mechanisms of the gateway to redirect requests, moving much of the heavy lifting out of the code of our microservices. Generally, if you can offload these routing responsibilities to other bits of infrastructure that already manage routing, this often represents a cleaner, more manageable routing experience.

CONSIDERING SCALE

The nuances of the authentication and routing mechanisms that I've covered here are clearly influenced by how/if you might have siloed resources in your multi-tenant architecture. I discussed scenarios where you might have separate identity constructs for tenants. I also talked about how the siloed resources in your application plane can impact your routing model. While these are entirely valid strategies, you must also consider how the scale of your environment will shape your approach to authentication and routing. The more you introduce separate tenant constructs to support tenant-specific identity models or siloed deployment patterns, the more you have to think about how effectively these will scale based on the number of tenants you have in your system. If you're dealing with large numbers of tenants, some of the concepts may not align well with your cost, scale, or operational goals.

Conclusion

In this chapter, we continued our path into bringing up the foundational bits of our SaaS architecture, looking into how onboarded tenants enter the front door of a multi-tenant application. The goal here was to examine the different considerations that are associated with getting a tenant authenticated, acquiring their tenant context, and injecting that tenant context into the downstream services of our environment.

We explored some of the key elements that go into designing and shaping the authentication experience. This included looking at how something as simple as how the path into your SaaS environment influences the broader architectural footprint of your multi-tenant architecture. The use of subdomain, vanity domains, and a single shared domain all have implications on how you will identify tenants during your authentication flow. As we got further into the elements of authentication, we also saw how supporting different per-tenant identity constructs shaped the overall authentication flow of your environment.

Once we got beyond the front door, we moved more into examining how the authentication context could be used as part of routing tenant requests. More specifically, we looked into how this routing was

applied in different technology stacks. It was here that we got a better sense of the connection between authentication and the different deployment models and technology stacks employed by your system.

Now that we have the foundational bits in place and we're authenticated tenants, we can start to look into the landscape of multi-tenant microservices and see how this tenant context influences how we decompose and construct the services of our application. In the next chapter, we'll look at how multi-tenancy influences your approach to designing and building microservices, getting more into the details of how you can realize the benefits of multi-tenancy while still limiting complexity for developers.

Chapter 7. Building Multi-Tenant Microservices

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the seventh chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

Much of our focus up to this point has been centered around building out all the foundational elements of our multi-tenant architecture. This meant digging into the control plane and figuring out how to put the core services in place that would allow us to introduce the notion of tenants into a SaaS environment. We looked at how tenants are onboarded, how their identity is established, how they are authenticated, and—most importantly—how all of this ends up injecting tenant context into the microservices of our application. This should have

given you a healthy respect for the role that the control plane plays in a SaaS environment and illustrated just how critical it is to invest in creating a seamless strategy for introducing foundational tenant constructs into your multi-tenant architecture.

Now, we can start to shift our attention to the application plane. It is here where we can start to think about how we will apply multi-tenancy to the design and implementation of the microservices that will bring our application to life. In this chapter, we will begin to look at how the nuances of multi-tenant workloads will influence the way we approach the design and decomposition of microservices. Isolation, noisy neighbor, data partitioning—these all represent new parameters that you'll need to factor into the design of your microservices. What we'll see here is that multi-tenancy adds new wrinkles to the classic microservices design discussion, forcing you to take new approaches to the size, deployment, and footprint of your microservices.

The introduction of tenancy also has a direct impact on how you implement your microservices. We'll look closely at how and where multi-tenancy will weave its way into the code of your microservices, highlighting different strategies that can be used to prevent tenancy from adding complexity and/or bloat to the overall footprint of your microservices. I'll explore a few sample microservices implementations and outline tools and strategies that can be used to push multi-tenant

constructs into helper and libraries that simplify the overall developer experience.

The broader goal here is to give you a better sense of the landscape of considerations that should be on your list when you're starting to build out the multi-tenant microservices. Making this a priority from the outset can have a significant impact on the efficiency, complexity, and maintainability of your SaaS solution.

Designing Multi-Tenant Microservices

Before we can talk about how multi-tenant microservices are built, we need to come up a level and look at the size, shape, and general decomposition strategies that you need to think about as you identify the different services that will be part of your system. The boundaries of your services and how you distribute load/responsibilities to those services adds a dimension of complexity and forethought in a multi-tenant model.

Microservices in Classic Software Environments

To better understand this dynamic, let's start by looking at a classic application where the entire footprint of an application is installed, deployed, and managed separately. [Figure 7-1](#) provides a simplified ex-

ample of how microservices land in one of these classic, installed software environments.

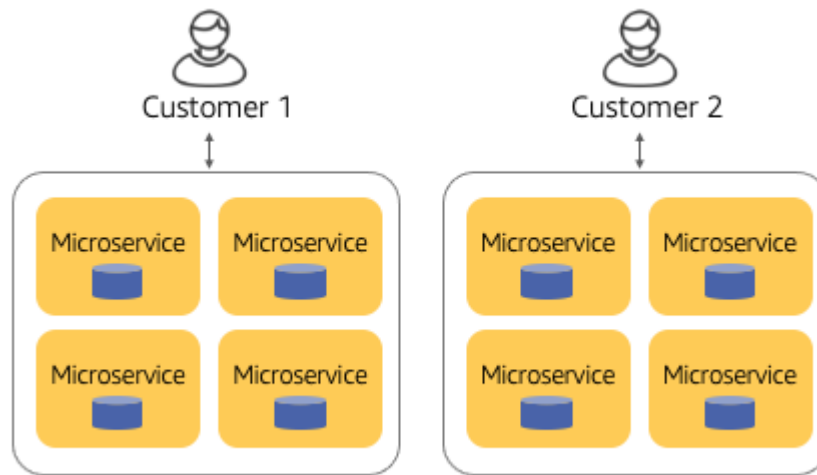


Figure 7-1. Microservices in a classic software environment

Here, you'll see that microservices are entirely dedicated to individual customers. When you're designing microservices for these environments, your focus is mostly on finding a good collection of services that can meet the scale, performance, fault tolerance, and scaling needs of a single customer. Yes, there may be some variation in how customers use your system, but the general focus is often on creating an experience that is limited to the behaviors and profile of a single customer.

This narrower focus makes it somewhat simpler to find the boundaries of your microservices. Much of the focus here is often more on the single responsibility design principle where you attempt to ensure

that you have decomposed the services in a way that ensures that each microservice has a clear, well-defined scope and functional role.

Microservices in Multi-Tenant Environments

Now, let's look at a full stack pooled multi-tenant environment.

[Figure 7-2](#) provides an example of a SaaS architecture that is supporting the needs of multiple tenants that are sharing their infrastructure resources.

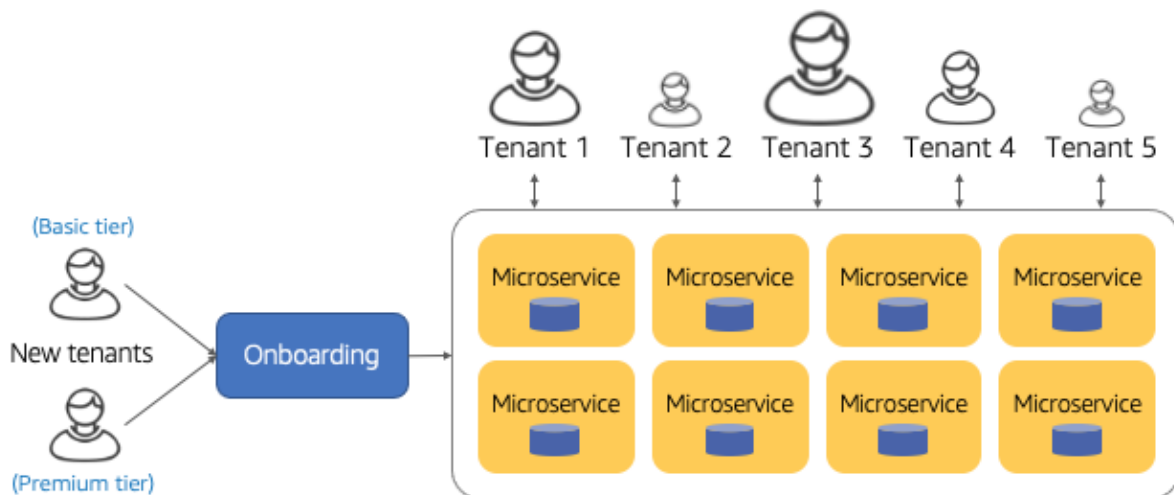


Figure 7-2. Microservices in a pooled multi-tenant environment

On the surface, it may seem like all that has changed here is the number of tenants that are consuming these microservices. However, the fact that these tenants are all exercising these microservices at the same time as shared resources has significant implications for

how you approach the size, decomposition, and footprint of each of these microservices.

The first thing you'll see here is that, at the top of the diagram, I've intentionally introduced tenants of varying sizes. This was done to illustrate the fact that there can be huge variation in how tenants put load on your system. One tenant may saturate part of your system. Another tenant may consume the entire surface of your solution, but place minimal load on the environment. The permutations here can be all over the map.

On the left, I've also shown the onboarding of new tenants. This is here to convey the idea that new tenants may be introduced into your environment at any time. There is little you can do to anticipate the workload and profiles of these new tenants. I've also highlighted the fact these new, incoming tenants may belong to different tiers with different experience and performance expectations.

So, now step back and think about what we really have here. The microservices in this environment, which are shared by all of the tenants, must somehow anticipate all the scaling, performance, and consumption needs of each of the tenant personas. You'll need to be hyper-focused on ensuring that these tenants are not creating noisy neighbor conditions where one tenant is impacting the experience of another tenant. These microservices will also need to dynamically

scale based on what could be a fairly elusive set of parameters. The scaling strategy you use today may not align with how your microservices need to scale tomorrow (or in the next hour). SLAs, tiering profiles, compliance, and other considerations may also be in this equation.

This is essentially where the benefits of shared infrastructure collide with the realities of supporting a constantly shifting landscape of customer consumption profiles. For some, this leads to over provisioning of resources to account for shifting needs and load profiles, which is exactly counter to the efficiency and economies of scale goals that are associated with the SaaS business model.

In some respects, this is all part of having a multi-tenant pooled architecture. Even if your services are over-provisioned, the collective value of having these resources shared is likely much higher than having dedicated, per-tenant infrastructure. The design of your microservices, however, can play a big role in giving you more tools and strategies for addressing the shifting needs of tenant workloads and profiles. Generally, our approach to microservice here is focused on giving yourself more knobs and dials to address the diverse range of dynamics that your tenants are imposing on your environment.

Extending Existing Best Practices

The process of arriving at your multi-tenant microservices will still mirror many of the accepted methodologies and strategies that are typically used to identify candidate services. The theme here is that, as part of applying those concepts, you'll also want to add additional multi-tenant design considerations to the list of factors that shape your microservice design, blending the foundational best practices with a collection of multi-tenant considerations (as shown in [Figure 7-3](#)).

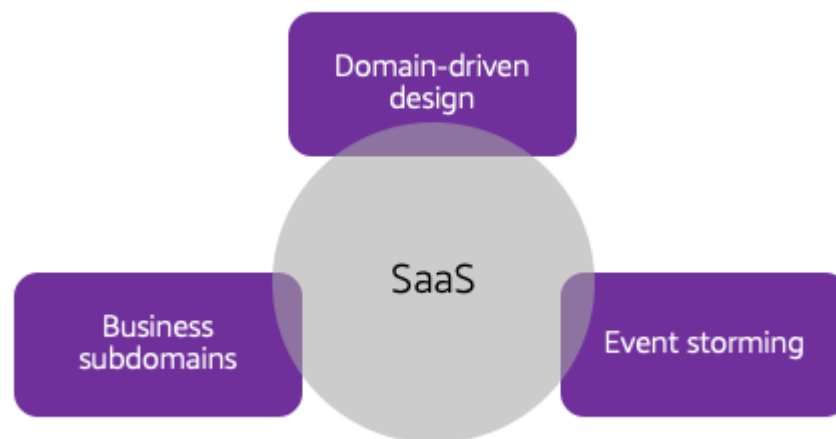


Figure 7-3. Blending microservice design methodologies

This conceptual diagram provides a clearer mental model of the approach that I'm advocating. Here you'll see examples of some of the common microservice design methodologies that teams often use to identify the different candidate microservices. While these methodologies have clear value, they don't always include discussion of the multi-tenant realities that must be factored into the design of my mul-

ti-tenant services. Yes, I know I will end up with services that map to many of the logical entities and operations that are part of my domain. The challenge here, though, is that the multi-tenant profile of my environment might end up leading me to introduce services and deployment patterns that wouldn't naturally be uncovered if you're just looking at domain objects, operations, and spheres of interaction.

To acknowledge this, I've put SaaS as a placeholder at the center of [Figure 7-3](#). The idea here was that you'd take all the design considerations that come with multi-tenancy and overlay them with other methodologies, ensuring that these concepts are front-and-center as you are beginning to model your application's services.

To drive this point home, let's dig into some of the common areas where multi-tenancy could have an influence on the design of your microservices.

Addressing Noisy Neighbor

Noisy neighbor is a concept that is not unique to multi-tenancy. Builders generally have to consider how and where users might impose load on your system that could saturate your system or degrade performance. While this is a general area of concern, you can imagine how the nature of multi-tenancy and shared infrastructure put more focus, weight, and complexity on the noisy neighbor problem. A

noisy neighbor in a SaaS environment has the potential to bring your whole system down or, minimally, degrade the experience of the other tenants in your multi-tenant environment. So, as you sit down to design the microservices of your multi-tenant environment, you'll want to be sure that you're testing assumptions about how/if your microservices address potential noisy neighbor conditions.

Noisy neighbor can show up in multiple forms in a multi-tenant environment. There may be specific operations in your environment that have high latency or consume resources in patterns that have a high potential to create bottlenecks. You may have areas where certain tenant personas are prone to saturating a particular set of services that are part of your system.

The basic challenge here is often all about scaling. Certainly, if a microservice can scale effectively enough to address the multitude of personas and workloads without over-provisioning or impacting other tenants, then you probably have a reasonable scope for your microservice. Our focus is on those scenarios where horizontal scale alone may not be effective or efficient enough to deal with the multi-tenant realities of your environment. Consider the sample service shown in [Figure 7-4](#).

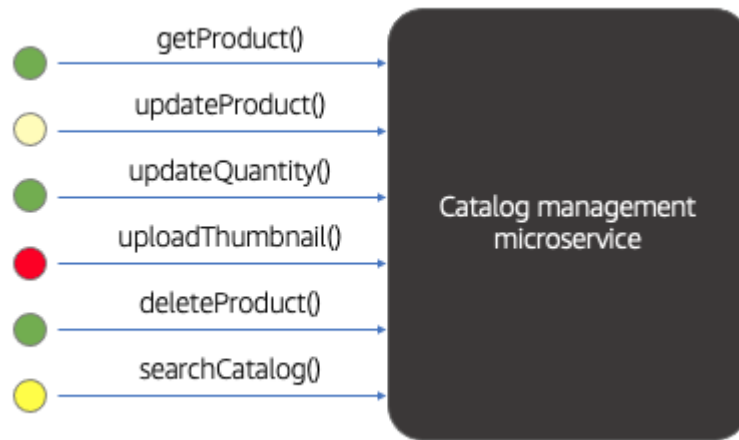


Figure 7-4. A noisy neighbor bottleneck

Here, you'll see that I've created a catalog management microservice that manages all the products in my ecommerce SaaS solution. If you focus exclusively on the API signature of this microservice, the operations listed here seem to fit naturally within the scope of managing catalog data. However, if you look at the far left, I've also highlighted the operational profile of each of the microservice's API entry points, using colors to convey their current status. You'll notice most of the operations here are healthy or healthy enough. However, the `updateThumbnail()` operation appears to be suffering from some kind of performance issue that, in this case, is yielding a noisy neighbor condition.

It turns out that this particular function happens to do some heavy lifting that is creating bottlenecks for this service. Callers are uploading images and triggering an image scaling mechanism to generate different sized thumbnails that are used in multiple contexts across the

application. Left as-is, your primary approach to solving for this might be to simply scale the microservice out, potentially over-provisioning, and hoping that this will limit any cascading impacts on your tenants. Essentially, you may be scaling this entire service when only one operation of the service needs better throughput. The better option here might be to think about whether this operation could be extracted and moved to a separate microservice where it can scale more proportionately to tenant activity without absorbing the inefficiencies of scaling out the entire catalog management microservice to address your performance issues.

The general mindset here is that you'll want to think differently about the scope of your microservice's responsibility and consider how these services will scale to meet the varying loads of a multi-tenant environment. As you're identifying your services, look for those areas that stand out as potential noisy neighbor candidates. [Figure 7-5](#) provides a conceptual view of the overall theme here.

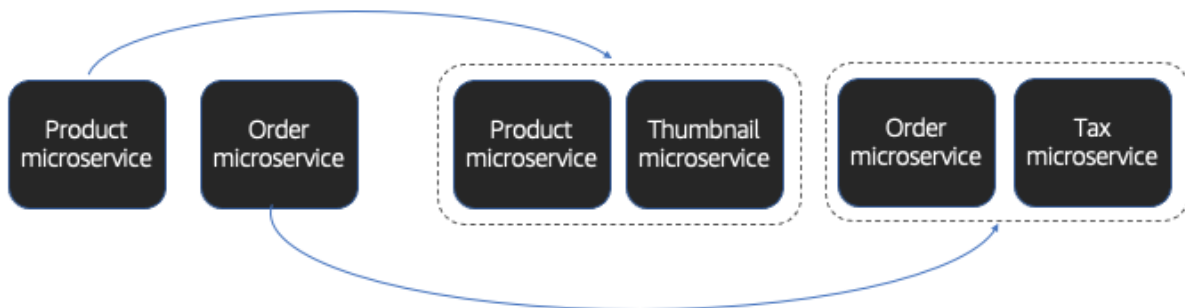


Figure 7-5. The noisy neighbor decomposition mindset

Here I've just illustrated how a given service could be further decomposed into smaller services that give you more targeted scaling options that, ideally, would limit noisy neighbor conditions and limit over provisioning. The product microservice separates out a thumbnail service to better distribute the load and the order microservice breaks out a standalone tax service. The core idea here is that we must look at the scope of our microservices through the lens of these noisy neighbor and multi-tenant scaling efficiency and look for opportunities where a more granular decomposition strategy might better target your noisy neighbor profile.

It's fair to ask if this approach is really unique to SaaS? The answer is no. As a rule of thumb, any environment should be looking at ways to better address performance and scale through more granular microservices. What's different here, though, is the diversity of tenant personas and workloads that end up requiring SaaS architects to be much more diligent about addressing these kinds of challenges. An inefficiently scaled, bottlenecked, or over-provisioned service is likely to show up much more in a multi-tenant environment, which can have much more profound impacts on your tenants and the operational profile of your SaaS environment. So, while this is a good general approach, it deserves much greater focus and attention when you're designing microservices for a multi-tenant environment.

It's worth noting that it's fully expected that your noisy neighbor strategy is going to evolve over time. The microservices you pick on day one should be expected to morph as your system evolves and you have richer insights into how and where you're observing noisy neighbor conditions. Start with the microservices that make sense and then use the strategies outlined here based on the operational profile of your environment.

Identifying Siloed Microservices

In Chapter 3, we talked about different deployment models and how these models might require you to have some or all of a tenant's resources deployed in a siloed (dedicated) model. On the surface, it may seem as though there's no real connection between siloing resources and the design of your microservices. However, there can actually be a strong correlation here between the microservices you choose and how/when/if those services get deployed in a siloed experience.

Whenever you choose to silo a microservice, you're often doing so to support a specific system or tenant need. You might, for example, need to silo some microservice to support a compliance requirement that's part of your domain. Tiering, performance, and isolation may also have some influence on which microservices that you opt to deploy in a siloed model.

Of course, any time you're siloing a resource you're making a compromise that can impact the operational, cost, deployment, and management complexity of your environment. So, if you are going to have siloed resources, it's ideal to limit the number of microservices that need to be deployed in siloes. [Figure 7-6](#) provides an example of how the isolation requirements of your environment might influence the footprint of your microservices.

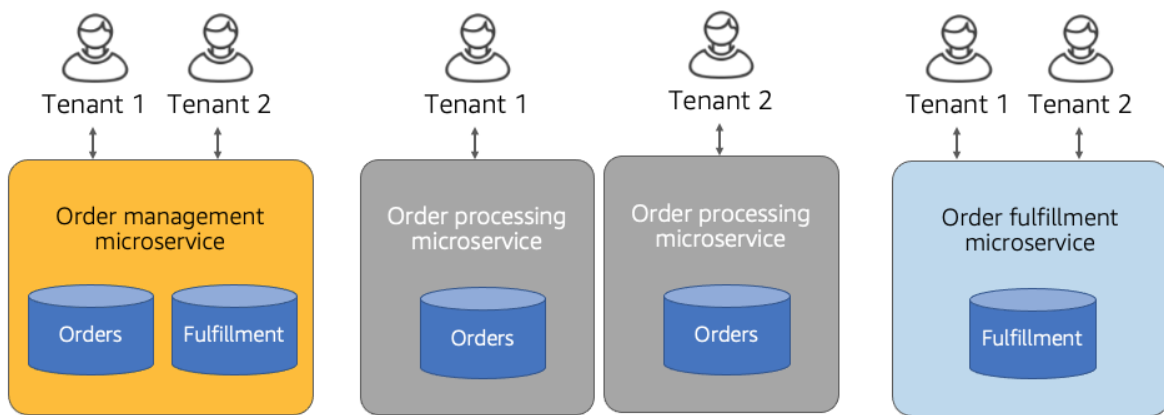


Figure 7-6. Designing microservices based on isolation needs

This diagram includes two different approaches to designing an order management microservice that address all the order processing and fulfillment needs of your solution. On the left, you'll see this microservice deployed in a pooled model that shares compute and storage for all tenants. Now, let's presume that our tenants have expressed a concern about having this microservice running in a pooled model. Your first instinct might be to move this microservice to a fully siloed deployment model to ensure that each tenant would run a dedicated

copy of this microservice. However, after probing further, you find that your tenants are actually only concerned about having dedicated compute and storage for the order processing portion of the service.

Instead of fully siloing this service as-is, you may be able to break the service down into one or more services to address the isolation needs of your customer. This is precisely what I've done in this example, breaking the original microservice into two separate microservices. Here, our new order processing service is deployed in a siloed model where each tenant gets access to a dedicated service, directly addressing our tenant's isolation requirement. As part of this move, I also introduced a new order fulfillment service that continues to run in a pooled configuration.

You can imagine how this same approach could be applied to any number of scenarios where a customer may require siloed resources. You might, for example, apply some variation of this same mindset for compliance, noisy neighbor, or general performance reasons, breaking larger microservices to give yourself more granular control over what is siloed and what is not.

This approach of picking what's pooled and what's siloed doesn't have to be about breaking services into more microservices. It might just be about grouping microservices based on their need to be siloed. [Figure 7-7](#) provides a conceptual view of how you might align

the boundaries of your microservices based on their silo vs. pool requirements.

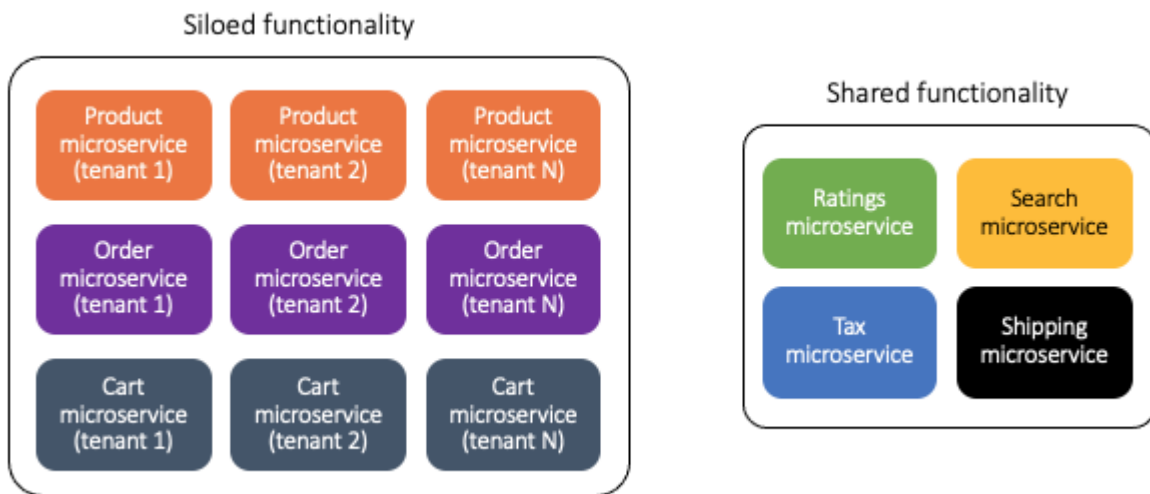


Figure 7-7. Aligning service with silo/pool requirements

For this scenario, my tenants have indicated that they need specific functionality to be deployed in a siloed model. In the diagram, you'll see that the product, order, and cart microservices are all deployed in a siloed model where each tenant has a dedicated instance of these services. Then, on the right-hand side, I have a set of services that are running in a pooled model.

You could just view this purely through the lens of deployment and say some services are siloed and some are pooled and that would be accurate. However, the idea here is that you want to be as thoughtful as possible about which microservices land in the siloed side of this experience. So, if you're designing microservices that you know are

going to have to support this mix of silo and pool, you should be thinking about how you can decompose these services in a way that will allow you to land as much as you can in the pooled model.

This siloing strategy is something you should make a core part of your microservice design mindset, identify the use cases and requirements that might justify deploying a microservice in a siloed model. This list typically includes compliance, isolation, security, tiering, and performance. It's also important to note that you may silo microservices entirely based on your own internal operational needs. For example, you may have some services that simply can't meet the demands of tenants in a pooled model. In this case, you might opt to carve that piece of functionality out and deploy it in a siloed model purely based on the operational realities of your environment.

In some instances, the siloed boundaries of your services may stand out early in your design process. In other instances, however, you may need to collect data and iterate some to arrive at a set of microservices that balance this siloed/pooled universe. For me, the main thing I'm trying to avoid here is just moving services into a silo without challenging myself to see if there are more creative ways to decompose my microservice. Again, as I mentioned above, you may not discover these boundaries until after you've collected more operational insights from the working system.

It's important to note that this silo strategy should be adopted with caution. It's not exactly operationally or cost efficient. So, you'll want to be selective about how/when you would consider this approach. For systems with a smaller population of tenants, this could be a good strategy. However, for systems with a larger pool of tenants, this would become challenging to scale and support.

The Influence of Compute Technologies

While it may be less obvious, there are also scenarios where the compute technology you're using can have some influence on the footprint of your microservices. Suppose, for example, you are considering having some microservices running with a serverless compute model and others running in a container compute model. It turns out that the nature of these different compute models can play a direct role in size, scope, and boundaries of your microservices. To better understand this, let's look at [Figure 7-8](#).

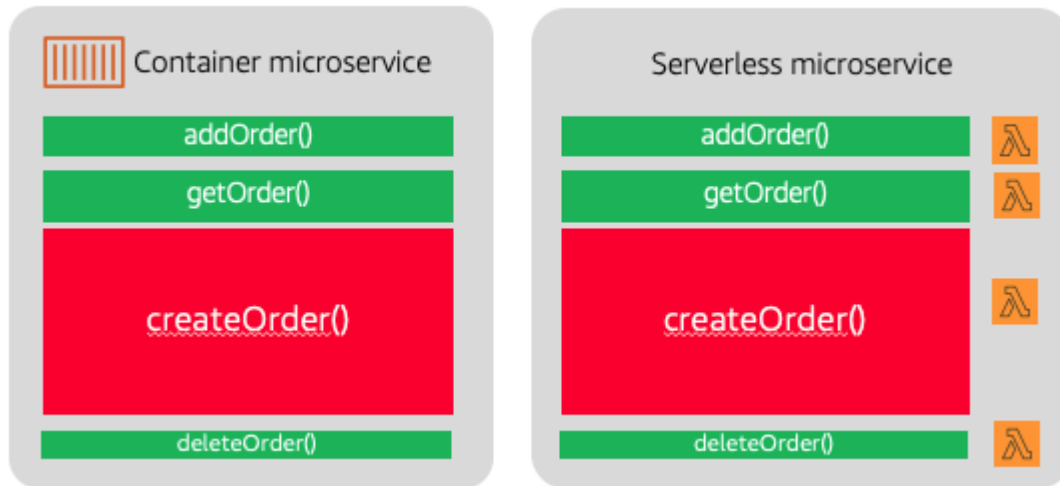


Figure 7-8. Compute and microservice design

In this example, I have included two mostly identical instances of an order microservice. On the left, the service is running in a container compute model. Meanwhile, on the right-hand side, this same service is running in a serverless model (in this case using AWS Lambda). I've highlighted the different operations that are part of this service and sized the box of each operation based on the load associated with these operations. In this case, it should be clear that the createOrder() operation is receiving the bulk of the requests. So much so, that you might ask whether this service will be able to scale efficiently or whether it's essentially going to scale based on this one operation—even though the other operations are not really being pushed.

Now, when we look at the container-based deployment, we might consider looking at how/if we'd want to consider refactoring this service to try to improve the overall scaling efficiency of our environment.

With containers, all of this functionality is packaged, scaled, and deployed collectively so the container becomes our unit of scale.

With the serverless model on the right, though, each of the operations that are deployed as part of our microservice represents a separate function that can be deployed, managed, and scaled independently. So, if `createOrder()` or any other operation here is receiving a disproportionate level of load, that function will scale on its own. This represents one of the significant advantages of the serverless model where the scaling is more granular and becomes someone else's problem to manage. The better news here is—if the load profile shifts tomorrow and another operation starts taking all the load—there's nothing I need to do to adjust the scaling policies or profile of my microservice. This also makes it much easier to accommodate and optimize for the continually shifting workloads of multi-tenant environments.

The key takeaway here is that the compute model you're using may have some influence on your microservice decomposition model. It's not the primary factor, but it does add another wrinkle to your design mindset.

Mapping Storage to Microservices

When we're designing microservice, we're also often thinking about the scope and the nature of the data that will be accessed and managed by these microservices. Since each microservice is expected to encapsulate the data that it manages, you must consider how this data will be consumed. Splitting a microservice along the wrong data boundary could end up leading to extremely chatty services that are constantly in need of data that lives in the scope of some other microservice.

These are all general considerations that are associated with designing any microservice. Now, as we look at multi-tenant microservices, we have some new factors to add to our design considerations. Our multi-tenant data can be stored in a siloed model where each tenant has its own dedicated storage structure or it could be stored in a pooled model where tenants data is co-mingled within a shared storage construct. You may also need to think about how the different operations on your data will scale effectively when customers are competing for a shared storage resource. Imagine thousands of tenants all querying some relational database that has stored all of its tenant data in a shared table. Will these tenants saturate the compute of the storage technology? Will you end up creating another flavor of a noisy neighbor condition? These are all just examples of how the footprint of your storage might influence the scope and granularity of your services. In some cases, a more coarse-grained service might be your

preferred model. In others, there may be compelling reasons to break your services into smaller bits to support different data profiles.

In many respects, the storage needs of your microservice must also look at many of the factors that shaped the noisy neighbor and siloing discussion outlined above. With storage, we're going inside the microservice and thinking about how multi-tenant requirements, tenant personas, and workloads might shape the footprint of your microservice. This is essentially a mirror of the compute considerations that we've discussed earlier. Storage has its own compute and data footprint that must also address noisy neighbor, compliance, tiering, and isolation considerations.

The key takeaway here is that storage can and often does play a significant role in the decomposition of your services. So, as you're sitting down to identify your microservices and their scope/granularity, you'll want to be sure to give storage the attention it deserves. In some instances this will be straightforward and in others the storage profile of your microservices may be the driving force that shapes a microservice's design.

Using Metrics to Analyze Your Design

The design of your microservices will always be constantly evolving. New features, new tenants, new tiers, and new workloads will have

your team continually evaluating the performance, scale, and efficiency of your multi-tenant architecture. Of course, in a multi-tenant environment, it can be more challenging to get a handle on how/if the design of your microservices is delivering the experience you intend. You might be able to use some basic monitoring data to draw some high-level conclusions about how your system is behaving, but this data won't typically allow you to evaluate the consumption and activity patterns of individual tenants and tiers. This makes it difficult to perform any kind of deeper analysis of the factors that are shaping the operational profile of your SaaS environment. Is the consumption profile of a particular tenant or tier impacting the scaling profile of a given microservice? Are basic tier tenants pushing your microservices in ways that are impacting your premium tier tenants? These are just examples of insights you really need to assess the efficacy of your microservice design.

It's only when you have these richer insights that you can really begin to evaluate how/if the design of your microservices is successfully addressing the various factors that we've been exploring here (noisy neighbor, tiering, performance, and so on). Getting these metrics means adding instrumentation to your microservices that will surface the data you need to analyze the operational profile of your microservices. [Figure 7-9](#) provides a conceptual illustration of this instrumentation model.

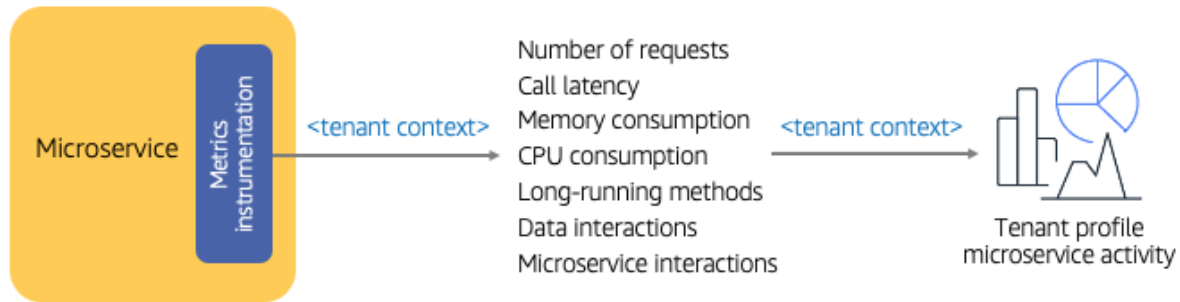


Figure 7-9. Surfacing tenant-aware microservice metrics

Here, on the left, is a sample microservice that includes the instrumentation that is needed to publish the data that can be used to assess the performance, scale, and operational profile of your microservice. In the middle there are some examples of data that you might capture for a microservice. What you choose to instrument here will depend on the nature of the microservices and the nature of the data that would best characterize its activity. All the data that's recorded here will minimally include tenant context and tenant tier (if you're using tiers).

Finally, on the right is a placeholder for the aggregation and analysis of this data. It's here that you'll use the tool of your choice to analyze this metric data and evaluate the run-time profile of your microservices. We'll dig into the whole area of multi-tenant operations and metrics in Chapter 14. The key here is that—in a multi-tenant environment—you really need to invest in capturing the metrics and analytics that can tell you how your design is performing. Without this data,

you'll have limited ability to examine how the variations in tenant workload and activity are exercising your architecture.

One Theme Many Lenses

Across this entire discussion of microservices design, I focused on finding the granularity and deployment model that best addresses the compliance, isolation, storage model, noisy neighbor, tiering, and performance requirements of your environment. While each one of these factors brings its own nuances to the microservices design discussion, you can also see that the strategies used to address these needs definitely overlap.

There were two basic themes that were sprinkled throughout our design discussion. In some cases your design strategies will focus on creating more granular services that can better respond to your multi-tenant scaling and performance needs. In other situations, you may look at using siloed deployments to create a profile that addresses your system's requirements. The key here is to add these possibilities to your design mindset and look for opportunities to support the realities of your multi-tenant workloads.

Inside Multi-Tenant Microservices

With this backdrop of multi-tenant microservice design in place, we can now start looking at what it actually means to build a multi-tenant microservice. As a rule of thumb, I tell teams that they should make every effort to limit a developer's awareness of multi-tenancy when they are coding the business capabilities of their SaaS application. Our focus here, then, is on what strategies and techniques you can introduce into your builder experience to limit the overhead they'll absorb when they're building multi-tenant microservices.

To get a better sense of how multi-tenancy lands in your microservices, let's start with a basic service that has no support for any notion of tenancy. The following code provides a snippet of a order microservice that is responsible for fetching all orders matching a given status:

```
def query_orders(self, status):
    # get database client (DynamoDB)
    ddb = boto3.client('dynamodb')

    # query for orders with a specific status
    logger.info("Querying orders with the status of %s", status)
    try:
        response = ddb.query(
            TableName = "order_table",
            KeyConditionExpression = Key('status', 'EQ', status)
        )
    except ClientError as err:
        logger.error("Error querying orders: %s", err)
```

```
logger.error(
    "Find order error, status: %s. Info: %s:
    status,
    err.response[ 'Error' ][ 'Code' ],
    err.response[ 'Error' ][ 'Message' ])
raise
else:
    return response[ 'Items' ]
```

For this service, I happen to choose an AWS NoSQL storage service (Amazon DynamoDB) to store my orders. You'll also see that I've coded this example in Python and Boto3, a library that's used to integrate with AWS services. The order data will land in DynamoDB with a "status" key that will be used to access the orders in our system.

Overall, the code here represents a relatively vanilla service that essentially takes an incoming status as a parameter and queries a database for orders that match that status. You've likely seen and/or written some variation of this function at some point along the way.

For our purposes, we're more interested in what's not here. Since this code is not running in a multi-tenant environment, there's nothing in the code that we see here that has to concern itself with multi-tenancy. The logging data it emits, the data it's accessing—none of it has to consider which tenant is actually invoking these operations.

As a multi-tenant architect, it should be your goal to have this code remain as straightforward and familiar as it is here. You must find a way to introduce tenant context and support for multi-tenant constructs without adding bloat and overhead to the builder experience. The more you can move tenant context outside the view of builders, the more opportunities you will give yourself to centralize these strategies/policies for all of your microservices.

Extracting Tenant Context

Now we can start to look at how our microservice code will begin to morph as tenancy is injected into our code. Before we can even think about applying tenant context, we have to think about how that tenant context lands in our microservices. This starts by first looking back at the identity and authentication topics that were discussed in chapters 4 and 7, respectively. In these chapters, we looked at how tenant context was bound to individual tenant user and injected as a JWT token that flowed into the microservices of our solution. We can now get into what we do to leverage this JWT token as it arrives in the context of our services.

If you recall, the JWT token gets embedded as a header within each HTTP request that is sent to your microservices. This token is passed as what is known as a “bearer” token. The term bearer maps to the idea that you are granting access to the bearer of this token. For your

microservice, it indicates that you're authorizing you to perform an operation on behalf of the tenant associated with that bearer token.

If you were to crack open one of these HTTP requests, you'd see the bearer token represented as part of the request's authorization header. The request would resemble the following format:

```
GET /api/orders HTTP/1.1
Authorization: Bearer <JWT>
```

You can see this is a basic GET request to the `/api/orders` URL with an authorization header that has the value of "Bearer" followed by the contents of your JWT token. Let's look at the code we need to add to our microservice to access the tenant context that's embedded in this JWT token. It's important to note that this token is encoded and signed so we'll need to unpack it to get access to the claims that we're interested in. The following example adds code to the prior example, introducing the steps needed to extract the tenant context from the incoming JWT token.

```
def query_orders(self, status):
    # get tenant context
    auth_header = request.headers.get('Authorization')
    token = auth_header.split(" ")
    if (token[0] != "Bearer")
        raise Exception('No bearer token in request')
```

```
bearer_token = token[1]
decoded_jwt = jwt.decode(bearer_token, "secret",
                        algorithms=["HS256"])
tenant_id = decoded_jwt['tenantId']
tenant_tier = decoded_jwt['tenantTier']

# query for orders with a specific status
logger.info("Finding orders with the status of
...")
```

I've trimmed out the actual query execution code here since it is, at the moment, unchanged. The code we want to focus on there is the snippet where I'm accessing and extracting the tenant context from the incoming request. This block of code first pulls the authorization header out of the overall HTTP request, setting the `auth_header` equal to "Bearer <JWT>" where the JWT represents your encoded token. The next bit of code performs the basic string operations needed to get the contents of the JWT copied into a separate string. This string is then decoded using a JWT library. The end result is the decoded JWT token ends up in the `decoded_jwt` variable. The last step is to acquire the `tenantId` from the JWT's custom claims. You might also be accessing other claims here (role, tier, etc.) based on the nature of your solution.

In this particular example, I'm assuming that your microservice would own responsibility for decoding each token. However, there are other options here. You could, for example, have an API gateway that would sit in front of all your microservices, processing each inbound request. This gateway could crack open these JWT tokens, access the tenant context, and inject that into each microservice. This could allow you to implement more interesting strategies to deal with the latency that comes with accessing tenant context on each request. This is just one of the alternate strategies you might consider. The key here is that somewhere at the front of these requests, you'll need code that can go through the motions of acquiring this tenant context for every request (whether it's cached or extracted from the JWT each time)..

Once this code is executed—wherever it resides—your microservice will now have access to the tenant context it needs for other downstream operations. This processing of tenant context illustrates the payoff of the onboarding and authentication flows we discussed earlier chapters, illustrating how microservices can begin to address multi-tenancy without calling other services or mechanisms to get their tenant context.

Logging and Metrics with Tenant Context

At this stage, our code now has access to tenant context. However, it's not doing anything with that context. Let's start by looking at one of the areas where you can apply context in your multi-tenant microservices: logging. Logging represents one of those foundational mechanisms every microservice is going to use, emitting messages that create an informational and debugging audit trail that is essential to troubleshooting and analyzing the activity in your system.

Now, imagine using these logs in a SaaS environment where multiple tenants are exercising your microservice at the same time. By default, if you did nothing to your logs, they would contain a mixed collection of insights that had no correlation to any specific tenant. This would make it nearly impossible to piece together a view of the activity of any one tenant. If you're on the operations team and you're told that Tenant1 is having issues that nobody else is reporting, you would have a very difficult time using your logs to identify the log messages and events that were contributing to that tenant's specific problem. Even if you found an error message, it's unlikely you'd be able to explicitly associate that error with a specific tenant.

The good news is, now that we have tenant context at our fingertips, we can inject this context to our log messages. This will introduce the tenant context that will allow your operations team to analyze logs through the lens of individual tenants, tiers, and so on. Let's look at

what our code would look like with the addition of this tenant-aware logging.

```
def query_orders(self, status):
    # get tenant context
    auth_header = request.headers.get('Authorization')
    token = auth_header.split(" ")
    if (token[0] != "Bearer")
        raise Exception('No bearer token in request')
    bearer_token = token[1]
    decoded_jwt = jwt.decode(bearer_token, "secret",
                             algorithms=["HS256"])
    tenant_id = decoded_jwt['tenantId']
    tenant_tier = decoded_jwt['tenantTier']

    # query for orders with a specific status
    logger.info("Tenant: %s, Tier: %s, Find orders
                tenant_id, tenant_tier, status);
    ...
```

Here, I've just changed one of the logging messages that is part of the order microservice. Our message simply prepends the tenant context to the front of our logging message. This context would be added to all of the logging messages within your microservices, introducing the data that equips teams with much richer insights into the specific behavior of their tenants. If you're querying logs, you can now

filter by a specific tenant's context and assemble a more complete view of how individual tenants are interacting with your system. There's no magic to this at all, but it's one of those small changes that can have a huge impact on the operational profile of your environment.

The same logging mindset should also be applied to the metrics instrumentation of your multi-tenant architecture. Yes, we want logs to build a forensic view of tenant activity, but we also need data that is used by the business to profile the consumption and activity of tenants that doesn't quite fit into the operational profile of log messages.

The mental model here is that the metrics emitted from our microservices represent insights that are focused on providing the data that can be used to analyze and answer questions that shape your business, operational, and architecture strategy. Here you're profiling how your service is influencing the tenant experience and tracking your ability to measure a range of key metrics that business and technical teams can use to evaluate the system's efficacy, agility, efficiency, and so on. We'll cover the use of these metrics more in Chapter 14. For now, though, we need to consider how the publishing of these metrics fits into the footprint of our multi-tenant microservices.

Let go back to the order microservice and add a metrics call just to provide a more concrete example of publishing a metric event.

```

def query_orders(self, status):
    # get tenant context
    ...
    tenant_id = decoded_jwt['tenantId']
    tenant_tier = decoded_jwt['tenantTier']

    # query for orders with a specific status
    logger.info("Tenant: %s, Role: %s, Finding orders with status: %s",
                tenant_id, tenant_role, status);
    try:
        start_time = time.time()
        response = ddb.query(
            TableName = "order_table",
            KeyConditionExpression = Key(status)
        )
        duration = (time.time() - start_time)
        message = {
            "tenantId": tenant_id,
            "tier": tenant_tier,
            "service": "order",
            "operation": "query_orders",
            "duration": duration
        }
        firehose = boto3.client('firehose')
        firehose.put_record(
            DeliveryStreamName = "saas_metrics",
            Record = message
        )
    except ClientError as err:

```

```
except Exception as err:
    logger.error(
        "Tenant: %s, Find order error, status: %s.
        tenant_id, status,
        err.response[ 'Error' ][ 'Code' ],
        err.response[ 'Error' ][ 'Message' ])
    raise
else:
    return response[ 'Items' ]
```

For this example, I've added the recording of a metric to the order service query. To keep it simple, I just added something to track the duration of the query. Then, I created a JSON object that included all the data about my tenant context and the operation being performed. Now I need to publish this metric to some service that can ingest and aggregate these metric events. For this example, I used an AWS streaming data pipeline (Amazon Kinesis Data Firehose) to ingest my metrics data, constructing the Firehose client and calling the `put_record()` method to send the metrics event into the service.

Again, you can see here the instrumentation of metrics, on their own, doesn't really represent a particularly complex process. The effort here is more about determining what you want to capture and how/where you'll introduce the code to publish that metric data. The investment here is small if it's adopted widely by your teams, but the return can be substantial.

The challenge of telling the metrics story here is that there is no single, universal metrics approach that everyone should apply to their microservices. The value is clear, but the specifics are hard to nail down here. This often needs to be driven by your own desire to identify the metrics that will add the most value for your business. At the same time, I will also say that some of the most effective SaaS companies are those that prioritize metrics and work to identify the insights that will best inform their ability to assess the internal and external experience of their systems.

Accessing Data with Tenant Context

Logging and metrics are relatively straightforward and are more focused on capturing insights about the activity of your services. Let's shift to looking at how tenant context will influence how data is accessed for individual tenants.

At the moment, the data returned by our order microservice hasn't really done anything to account for tenant context. In fact, without any further modification, this service would return the same data for every tenant that requested orders. This, of course, is not the intended behavior of our system. To address this, we need to apply the incoming tenant context to our query, limiting the view of orders to those that are associated with the calling tenant.

The natural and simplest way to apply tenant context here is to add the tenant to the parameters that are part of your search. We already have our tenant identifier and simply need to decide how to use this tenant identifier to access our data. You have multiple options here. Let's presume, for the moment, that you have a pooled database model where your tenant data is co-mingled in the same table. When the data is pooled, we can simply add a TenantId key to our order table that will associate each order with a specific tenant. This tenant identifier will become the key of our table. This means that the status we were using will now become a secondary search parameter that returns all orders for a tenant that match the supplied status.

The code to apply this tenant context to your query is pretty straightforward. In the example below you'll see how I've augmented the query portion of the microservice, using the tenant identifier as the key and the status as a filter.

```
response = ddb.query(  
    TableName = "order_table",  
    KeyConditionExpression = Key('TenantId'),  
    FilterExpression=Attr('status').eq(status)
```

This slight tweak to your database search is all that's needed to ensure that the orders that are returned here are limited to just those that are associated with the current tenant.

In this scenario, I started with the simplest of use cases. Where data access discussion gets more interesting is when we start thinking about the various combinations of storage strategies your microservice might need to support. Suppose, for example, your system offered different storage for different tiers as shown in [Figure 7-10](#).

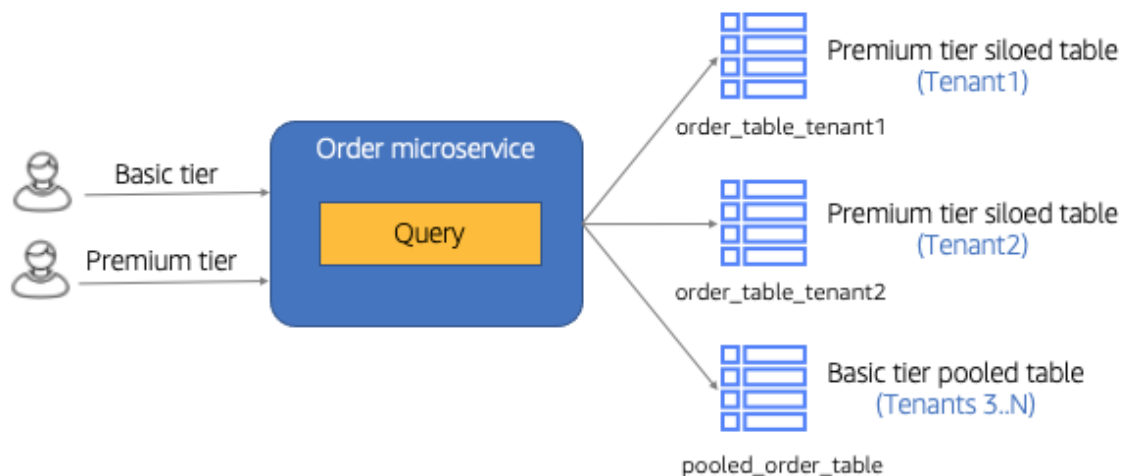


Figure 7-10. Supporting tiered storage models

In this scenario, our order microservice is processing requests from basic and premium tier tenants. The compute is completely shared by these tenants. However, on the right-hand side of the diagram, you'll see that the microservice employs different storage strategies for each of these tiers. The basic tier tenants are all stored in a single ta-

ble that's indexed by `tenantId` (as was the case in our prior example). However, the premium tier tenants store their data in a siloed model where each tenant has its own, dedicated storage. In this example, you'll see that each of these dedicated tables are assigned a name that conveys their binding to a specific tenant.

Now, with this new wrinkle in the mix, let's think about what this means for the implementation of your microservice. Somewhere within the code of our microservice, you must have logic that examines the tier of each tenant to determine which table will be used to process their requests. And, depending on how their data is stored, you may need multiple paths of execution within your microservice to support identifying and interacting with the tenant's order data.

Let's start with a brute force approach to this, knowing that we'll need to refine to make this simpler. To make this work, we'll essentially need to add some mapping operation to our query to resolve the name of the table that will be used (based on a tenant's tier). I've revisited the query we had within our microservice above, adding a new `getTenantOrderTable()` function that examines a tenant's tier and returns the name that will be used for a given tenant request. Here's the snippets of code that add this functionality.

```
response = ddb.query(  
    TableName = getTenantOrderTable(tenantId),  
    KeyConditionExpression = Key('TenantId')
```

```

keyConditionExpression = Key( 'tenantId')
FilterExpression=Attr('status').eq('active')

# helper function to get generate tier-based table name
def getTenantOrderTableName(tenant_id, tenant_tier):
    if tenant_tier == BASIC_TIER:
        table_name = "pooled_order_table"
    elif tenant_tier == PREMIUM_TIER:
        table_name = "order_table_" + tenantId
    return table_name

```

This approach, however, presumes that the tables for basic and premium tier tenants will be identical. And, for the most part, they would be the same. However, our pooled tenants rely on a TenantId key that is used to access the orders for individual tenants. This key has no value or meaning in the siloed tables. Many teams will keep this key in their siloed tables simply to avoid having to support additional one-off behavior. If you choose to remove this key from the siloed resources, you'll need to have more specialized code to compose your interaction with the data to account for the presence/absence of this key.

Naturally, the type of the data that your microservice stores and the technologies it uses are going to vary significantly. The example we covered here represents just one of many ways that multi-tenant data might shape how you implement your microservice.

Supporting Tenant Isolation

The data access example we just covered relied on inserted tenant context into our query to scope the data to a given tenant. It's easy to assume that, if we're filtering these queries by tenant, then you've put all the measures in place to ensure that one tenant can't access the data of another tenant. And, in theory, it's not an unreasonable expectation. However, in multi-tenant environments—where tenant isolation is essential to the trust of your tenants—filtering data access by tenant isn't really enough.

It's critical here that we draw a clear line between the strategies that are used to partition/access data and the strategies that are used to enforce tenant isolation. How data is stored and accessed is what we would consider your “data partitioning” strategy, which is covered in depth in chapter 8. How we protect resources (including data) from cross-tenant access, is referred to as “tenant isolation”, which is covered in detail in chapter 9. When we're talking about isolating tenant resources, we're talking about the measures that we use to surround the code within our microservices to ensure that developers don't intentionally or unintentionally cross tenant boundaries. So, regardless of what tenant parameter might be in your query, for example, the tenant isolation policies that surround that query will prevent that code from accessing the resources of another tenant.

This, of course, means that we need to introduce new constructs and mechanisms into the implementation of our microservices to apply the tenant isolation strategies. The goal here is to have your code somehow acquire an isolation context before it accesses any resource and use that context to scope your resource access to the current tenant. With this context applied, any attempt to interact with a resource will be constrained to just those resources that belong to the current tenant.

Now, let's look at how we can take this theory and turn it into something more concrete to give you a better idea of how this might land in your multi-tenant microservices. For this particular example where we're accessing DynamoDB, we can achieve our isolation goals by configuring our session with a set of credentials that will scope data access based on tenant context. If you look back to the starting for our order microservice, you'll see where the Boto3 client was initialized as the client library that would be used to access our order data. The initialization code is as follows:

```
def query_orders(self, status):  
    # get database client (DynamoDB)  
    ddb = boto3.client('dynamodb')  
    ...
```

This initialization of the Boto3 library here used a broader, default set of credentials to initialize the client. In this state, your client is initialized with a much wider scope, allowing it to access any item in your order table. That means any query here could access data for any tenant, regardless of what tenant context was passed into our microservice.

Our goal, then, is to scope down the access of this client for each request that is attempting to get orders, initializing the client with a scope that includes the context of the calling tenant. To achieve this, we'd need to change the way our client is initialized. The code that would apply this scope would resemble the following:

```
def query_orders(self, status):
    # get database client (DynamoDB) with tenant scope
    sts = boto3.client('sts')

    # get credentials based on tenant scope policy
    tenant_credentials = sts.assume_role(
        RoleArn = os.environ.get('IDENTITY_ROLE'),
        RoleSessionName = tenant_id,
        Policy = scoped_policy,
        DurationSeconds = 1000
    )

    # get a scope session using assume role credentials
    tenant_scoped_session = boto3.Session(
```



```
aws_access_key_id =
    tenant_credentials['Credentials']['AccessKey']
aws_secret_access_key =
    tenant_credentials['Credentials']['SecretAccessKey']
aws_session_token =
    tenant_credentials['Credentials']['SessionToken']
)

# get database client with tenant scoped credentials
ddb = tenant_scoped_session.client('dynamodb')
...
```

You can see that there are a few moving parts to this solution. First, you'll see that our first block of code here is focused on getting a narrower set of credentials that are scoped based on the current tenant identifier. In this particular example, we're staying within the family of AWS services, relying on the AWS Security Token Service (STS) to facilitate this scoping exercise. With STS, I can define a policy that restricts access to my order table. We won't dig into the details of this policy here, but just know that it essentially restricts access to just those items in the database that match a given tenant id. So, when I call the `assume_role()` function and supply my policy and tenant identifier (extracted from the JWT), this service will return a set of credentials that will limit access to just those items that belong to the current tenant. These credentials are stored in the `tenant_credentials` variable.

Once we have these credentials, we can declare and initialize a session with the specific credential values that were returned from our `assume_role()` call. Here you'll see the typical credential values that are used by AWS services when accessing resources.

All that remains now is to declare our DynamoDB client (as we did before). However, the client is now created using the `tenant_scoped_session` variable. This essentially tells Boto3 to initialize the client with the credential values that we set up in the prior steps. Now, when we invoke the query command using this client, it will inherit the scoping policies and apply these to any call that is made with this client.

This mechanism creates a true tenant isolation experience, providing a tenant-scoped context for any call to the database. Now, no matter what value or configuration a developer puts into their query, the system will prevent the microservice from accessing data that's not valid for the current tenant.

This example should give you a better sense for how tenant isolation can influence the footprint of your multi-tenant microservices. Getting this right is essential to building a robust isolation story for your solution. The challenge here, though, is that there are a number of factors that make it difficult to have a one-size-fits-all approach to isolation. The technology you're using, the cloud you're on, the services you're consuming, the silo/pool footprint of your resources—each of these el-

ements may all require a different strategy to describe and enforce tenant isolation within your microservice. The spirit and mindset of what we've done here is still valid for any microservice. It's really in the implementation and realization of these concepts where you'll run into significant variation.

It's also important to note that your microservices are likely to be interacting with a range of different types of resources. The isolation policies and approach we've covered here is meant to be applied across any resource that might be managing or touching tenant-specific constructs. If you have queues, for example, those queues may require some form of isolation.

Hiding Away and Centralizing Multi-Tenant Details

When I kicked off our discussion of building multi-tenant microservices, I put great emphasis on being able to introduce these constructs without bloating or adding complexity to your developer experience. My goal was to hide away and centralize many of the multi-tenant constructs and keep your microservice code focused squarely on implementing the business logic of your application.

Up to this point, I really haven't achieved this goal. In fact, if I were to put all the concepts we discussed into one final version of my order microservice, it would likely have tripled in size and complexity. You can also imagine how having this code in each microservice would be inefficient, distributing common concepts and constructs to every microservice in my system. At a minimum, this would represent a bad bit of programming. It would also limit my ability to centrally manage my multi-tenant strategies and policies.

This is where we put our basic builder skills to work and look for the natural opportunities to move these concepts out of our microservices and into libraries that can hide away much of the detail that we have been covering. There's nothing uniquely multi-tenant to this approach. It's more that, as a multi-tenant architect, I want to be sure I'm doing what I can to streamline the microservice developer's experience.

If you look back to our example, you can see where there is code that could easily be moved into helper libraries. Consider, for example, how the code that we added to acquire tenant context from the JWT token could be moved into a separate function. The code would simply be lifted out of our microservice and turned into a function that resembled the following:

```
def get_tenant_context(request):
```

```
auth_header = request.headers.get('Authorization')
token = auth_header.split(" ")
if (token[0] != "Bearer")
    raise Exception('No bearer token in request')
bearer_token = token[1]
decoded_jwt = jwt.decode(bearer_token, "secret",
                        algorithms=["HS256"])

tenant_context = {
    "TenantId": decoded_jwt['tenantid'],
    "Tier": decoded_jwt['tenantid']
}

return tenant_context
```

This new `get_tenant_context()` function takes an HTTP request and does all the work we described above to extract the JWT token, decode it, and pull out the custom claims that have our tenant context. I tweaked the function some, putting all the custom claims into a JSON object. How and what you chose to return here will depend on what's in your custom claims. You might have separate functions to get specific custom claims (`get_tenant_id()`, for example). This is more a matter of style and what works best for your particular environment.

The key here, though, is that this library now means that any microservice needed to extract tenant context can simply make a single call to this library and shrink the amount of code that lands in your microservices. It also allows you to alter JWT policies without having

these changes cascade across your solution. Imagine choosing a different approach to encoding or signing your JWT. With this in a centralized function, you can now make these alterations outside the view of microservice developers.

This same theme can also be applied to the logging, metrics, data access, and tenant isolation code that we covered above. Each one of these areas can be addressed through the introduction of libraries that standardize the handling of these multi-tenant concepts.

With logging and metrics, the key here is really just removing the added overhead of injecting tenant context every time you log a message or record a metric. Now you can simply share the request context with each of these calls and let some external function determine how to acquire tenant context and inject into your message/events.

Data access is one area that may be a bit less generic and may require helpers that are local to a specific microservice. If you recall, we discussed a use case where our order service might need to support a tier-based storage model where each tier might need to route its requests to a different order table. In this scenario, you might lean on the traditional Data Access Library (DAL) or Repository pattern to create a targeted construct that abstracts away the details of interacting with our microservice's storage. Here, this DAL could encapsulate

all the multi-tenant requirements, including applying isolation within that layer (completely outside the view of microservice developers).

Now, let's assume we moved all of this multi-tenant code into a library. The code of our microservice would be streamlined substantially, returning it to the version that resembles the copy that we had before multi-tenancy was introduced. The code below introduces a new function to get tenant context, introduces a logging wrapper that injects tenant context, adds a function to get a tenant-scoped database client, and employs an order DAL to hide away the details of mapping tiers to specific tables.

```
def query_orders(request, status):
    # get tenant context from request
    tenant_context = get_tenant_context(request)

    # get scoped database client
    ddb = get_scoped_client(tenant_context, policy)

    # query for orders with a specific status
    log_helper.info(request, "Find order with the status %s" % status)
    Try:
        response = get_orders(ddb, tenant_context, status)
    except ClientError as err:
        log_helper.error(
            request,
            "Find order error, status: %s. Info: %s:" % (status, err.message)
```

```
        status,  
        err.response[ 'Error' ][ 'Code' ],  
        err.response[ 'Error' ][ 'Message' ] )  
    raise  
else:  
    return response[ 'Items' ]
```

While there are lots of nuances in how this concept could end up getting applied in your microservices, this example gives you a sense for just how this approach can influence the implementation of your microservices. In many respects, this comes down to following fundamental programming best practices. The elegance here is less in the detail of what's in these libraries and more about the value they bring to microservice builders.

Interception Tools and Strategies

At this point, you can see how simply moving these shared multi-tenant concepts into libraries makes sense. However, in addition to moving this code outside of your microservices, you should also consider the different technology and language constructs that can be used to streamline your developer experience and centralize your multi-tenant strategies.

The basic idea here is that we want to look at how we might leverage the built-in capabilities of a given technology construct to support these horizontal multi-tenant needs, allowing you to introduce and configure multi-tenant operations/policies without minimal cooperation of your microservice builders.

Consider, for example, our approach to tenant isolation. If my language/tools provide me with a mechanism that allows me to insert processing between my microservice and the resources I am accessing, this might enable me to enforce aspects of my isolation model completely outside the view of my microservices.

The hard part of providing guidance here is that the list of possible strategies that enable this approach is quite long. Each language and their supporting frameworks, for example, bring their own unique blend of options to this discussion. Also, the different technology stacks and cloud services that are part of your architecture may include their own constructs that can be applied in this scenario. How and where these constructs are applied will vary significantly based on the specifics of each of these different options. While it would be counterproductive to review the spectrum of possibilities here, I do want to highlight a few sample strategies to give you a better sense of the different types of mechanisms that could fit with this mindset.

Aspects

Aspects are generally introduced as a language or framework construct. They allow you to weave cross-cutting mechanisms into your code that enable you to inject pre- and/or post-processing logic into the footprint of your microservices. This allows you to introduce global policies and strategies into your microservices that may align well with some of the multi-tenant mechanisms that are part of your environment.

The tenant context processing that we discussed, for example, might represent a great fit with aspect-oriented programming. You could use an aspect to intercept each request that comes into your microservice, adding pre-processing that would extract the JWT from the HTTP header, decode it, and initialize the tenant context for the rest of your request. You could also consider implementing elements of your tenant isolation model here, acquiring and injecting the tenant-scoped credentials that are needed to enforce your isolation policies.

The key here is that this becomes a standard mechanism for all of your microservices that is woven into your microservices as part of a global strategy (instead of relying on developers to call the appropriate helper functions in their code).

Sidecars

If your multi-tenant architecture is built with Kubernetes, you may want to consider how/if sidecars could be used to apply multi-tenant strategies to your microservices. A sidecar runs in a separate container within your Kubernetes environment and has the ability to sit between your microservice and other resources/microservices. The nice part of these sidecars is that they are entirely outside the view of your microservice. This allows you to apply any global multi-tenant policies in a way that may not require the cooperation of your microservice.

You can see how sidecars fit within this broader theme of hiding away multi-tenant tenant details. Being able to separately deploy and configure this sidecar allows me to create a much more robust multi-tenant enforcement story, allowing me to have greater control over my microservice's interactions with other resources.

Middleware

Some development frameworks support the notion of middleware. The idea here is that you can introduce code that sits between your inbound request and your target operation. This allows you to intercept and apply any global policies that would be applied across your microservice.

This middleware mechanism is commonly used in the Node.js Express framework. The framework provides all the built-in constructs that are needed to implement the many of the multi-tenant microservice strategies that we've covered here (tenant context, isolation, etc.).

AWS Lambda Layers/Extensions

I mentioned that different cloud providers and their services may include constructs that could be a good fit for realizing some of these cross-cutting multi-tenant strategies. I thought it would be worth highlighting one example here. If you're building a serverless SaaS environment on AWS, you have the opportunity to use the Lambda Layers and/or Lambda Extensions to move your shared libraries into a standalone mechanism.

With Lambda Layers, you can essentially move all of your helpers into a shared library that is then deployed independently. Each of the Lambda functions that are part of your microservice can reference the code in this shared library, allowing them to access the different helper functions you have without having that code being part of each microservice. This allows you to manage, version, and deploy these globally shared constructs entirely separately. Now, when you want to update your isolation mechanism, for example, you could update

the code in your Lambda Layer, deploy it, and have each of your microservices get updated with this new capability.

Lambda Extensions, on the other hand, fit more into the aspect pattern that we discussed above, allowing you to associate custom code with the lifecycle of a Lambda function. For example, you could use an Lambda Extension to pre-process a request upon entry to a function of your microservice. The code for a Lambda Extension can also live within a Lambda Layer.

Conclusion

At this point, you should have a much better sense of what it means to design and build a multi-tenant microservice. In this chapter, we started by looking at how the needs of multi-tenant environments directly influences the shape, size, and footprint of the microservices that support the core functionality of your SaaS offering. This required us to examine the various factors that must be added to your mental model when identifying the boundaries of your microservices.

Much of our design discussion was focused on the continually shifting workloads and consumption profiles of your tenants and how this influences your approach to picking the combinations of microservices that will allow you to best address these realities—especially in envi-

ronments that have pooled resources. This led to a much deeper dive into the different ways that noisy neighbor possibilities might impact the design of your microservices, potentially driving new service decomposition strategies that can better address the tendencies of a given microservice/workload. We also looked at how you might apply siloing strategies across your system's microservices to address targeted performance, experience, and tiering requirements.

As part of looking at design, I also highlighted the importance and value of investing in metrics that can continually inform the design of your microservices. The key here was to acknowledge that you should expect your design to evolve based on the operational insights that will come once it's alive and running under real workloads. It's only with solid multi-tenant metrics and insights that you can understand how your microservice design is meeting the current and emerging needs of your tenants.

Once we had a grasp of the design considerations, we shifted our attention to looking inside our microservices to better understand how multi-tenant shapes the actual implementation of these services. The focus here was first on determining how and where multi-tenancy needs to land within the implementation of your microservice. Here, I highlighted how your service would acquire tenant context and then apply it across logging, metrics, data access, and tenant isolation. A key point of emphasis here was on making every effort to ensure that

the introduction of these multi-tenant strategies/policies would not come at the cost of adding complexity or bloat to your services. We looked at specific ways to address this goal, outlining different ways to move these multi-tenant constructs outside the view of developers and moved them to a more centrally developed and managed model.

Overall, the broader theme here is that multi-tenancy can have a significant impact on both the design and implementation of your microservices. At the design level, it's all about anticipating the dynamic, shifting, and elusive profile of your tenant consumption tendencies and multi-tenant requirements. Within the microservices, it's more about focusing on thinking about how you can ensure that your microservices are implementing the core best practices without undermining the experience/productivity of your builders.

Now that we have a sense of how multi-tenant is implemented in our microservices, we can start digging into the various strategies that are used to represent the data that is used by these microservices. While I've provided glimpses of how you might store data in multi-tenant environments, we mostly looked at this through the lens of other concepts. Now, in the next chapter, we'll focus exclusively on the different approaches and considerations that are associated with representing, operating, and managing multi-tenant data. This will give you a more complete view of the different factors you should consider when working with multi-tenant data and highlight key points of inflec-

tion that will influence how/where you store the data that's part of your multi-tenant environment..

About the Author

Tod Golding is a cloud applications architect who has spent the last seven years immersed in cloud-optimized application design and architecture. As a global SaaS lead within AWS, Tod has been a SaaS technology thought leader, publishing and providing SaaS best practices guidance through a broad set of channels (speaking, writing, and working directly with a wide range of SaaS companies). Tod has over 20 years of experience as an architect and developer, including time at both startups and tech giants (AWS, eBay, Microsoft). In addition to speaking at technical conferences, Tod also authored *Professional .NET Generics*, was coauthor on another book, and was a columnist for *Better Software* magazine.