**Chapter 7. API Management and Consumption Patterns**

When we are building cloud native applications, we can build the business capabilities by using a wide array of design patterns covered throughout the book. Once you have the business capabilities implemented as microservices, you then need to present those services to external or internal consumers as managed APIs by using *API management patterns*.

In this chapter, we explore some of the most commonly used patterns in API management. We also cover a few API consumption patterns, which are essential in building frontend applications such as a web, mobile, or desktop applications on top of the managed APIs. Let’s begin our discussion with API management patterns.

**API Management Patterns**

Any cloud native application must have a given set of business capabilities that are exposed to its external consumers (customers, partners, and so forth) as well as internal consumers (other teams or departments in the same organization). These business capabilities are exposed to consumers through APIs. *API management* is the process of creating, managing, securing, analyzing, and scaling APIs on top of existing microservices.

API management enables consumers of your cloud native application to do the following:

* Have better engagement with the APIs you expose
* Integrate your application with other services and build new capabilities
* Monetize the consumption of your APIs
* Generate business analytics and insights
* Expose business capabilities as managed, secure, and resilient

Before diving into API management patterns, let’s first understand some key concepts related to API management.

An *API* or an *API proxy* represents a business capability of your cloud native application that is exposed to consumers in a managed way. An API may be backed by one or more backend microservices and is usually hosted in a separate API gateway runtime. The logic of the business capability that we expose should reside with the backend microservices, while the API provides only capabilities such as security, throttling, versioning, and so on. We should try to avoid putting business logic in the API layer, as the underlying service is responsible for supporting the required business logic. A given API can offer more than one backend microservice (for example, an order API can be backed by an order management capability hosted at *<host>:<port>/order/management* services and an order status capability hosted in a different context (*order/status*). But we do not recommend creating compositions such as service orchestrations at the API layer. APIs are usually exposed via REST/OpenAPI or GraphQL.

An *API product* is a concept occasionally used when it comes to presenting an API to developers via the API developer portal. The API product has one or more APIs and is usually mapped to a high-level business capability. For example, in cloud storage services, Storage Services can be the API product, while we can have multiple APIs such as the File API, Blobs API, and Disk API under that one API product.

In the following sections, we’ll explore commonly used patterns in the API management of cloud native applications. Let’s start the discussion with the API Gateway pattern.

**API Gateway Pattern**

The *API Gateway pattern* is the most common way of exposing the business capabilities of your cloud native application to the consumer. With this pattern, you put a separate layer in front of the capabilities that are designed as microservices and that you wish to expose to consumers. With this approach, an API gateway acts as the front door to your cloud native applications.

**How it works**

An API gateway layer is often built on top of existing microservices. Any microservices or composite services that you develop can be exposed as managed APIs through the API gateway. The API gateway operates alongside the API control plane and developer portal. Before we dive into each of these components, let’s first understand the roles that you may encounter in a typical API management process:

*API creator/API developer*

A person in a technical role who understands the technical aspects of the API (including interfaces, documentation, and versions) and uses the API publisher to provision APIs into the API store. The creator or developer uses the API store to consult ratings and feedback provided by API users. Creators and developers can add APIs to the store but cannot manage their life cycle.

*API publisher*

Manages a set of APIs across the enterprise or business unit and controls the API life cycle, subscriptions, and monetization aspects. The publisher is also interested in usage patterns for APIs and has access to all API statistics. (In certain cases, the API creator and publisher roles may be combined into a single role.)

*Application developer*

Uses the API store to discover APIs, read the documentation and forums, rate/comment on the APIs, subscribe to APIs, obtain access tokens, and invoke the APIs.

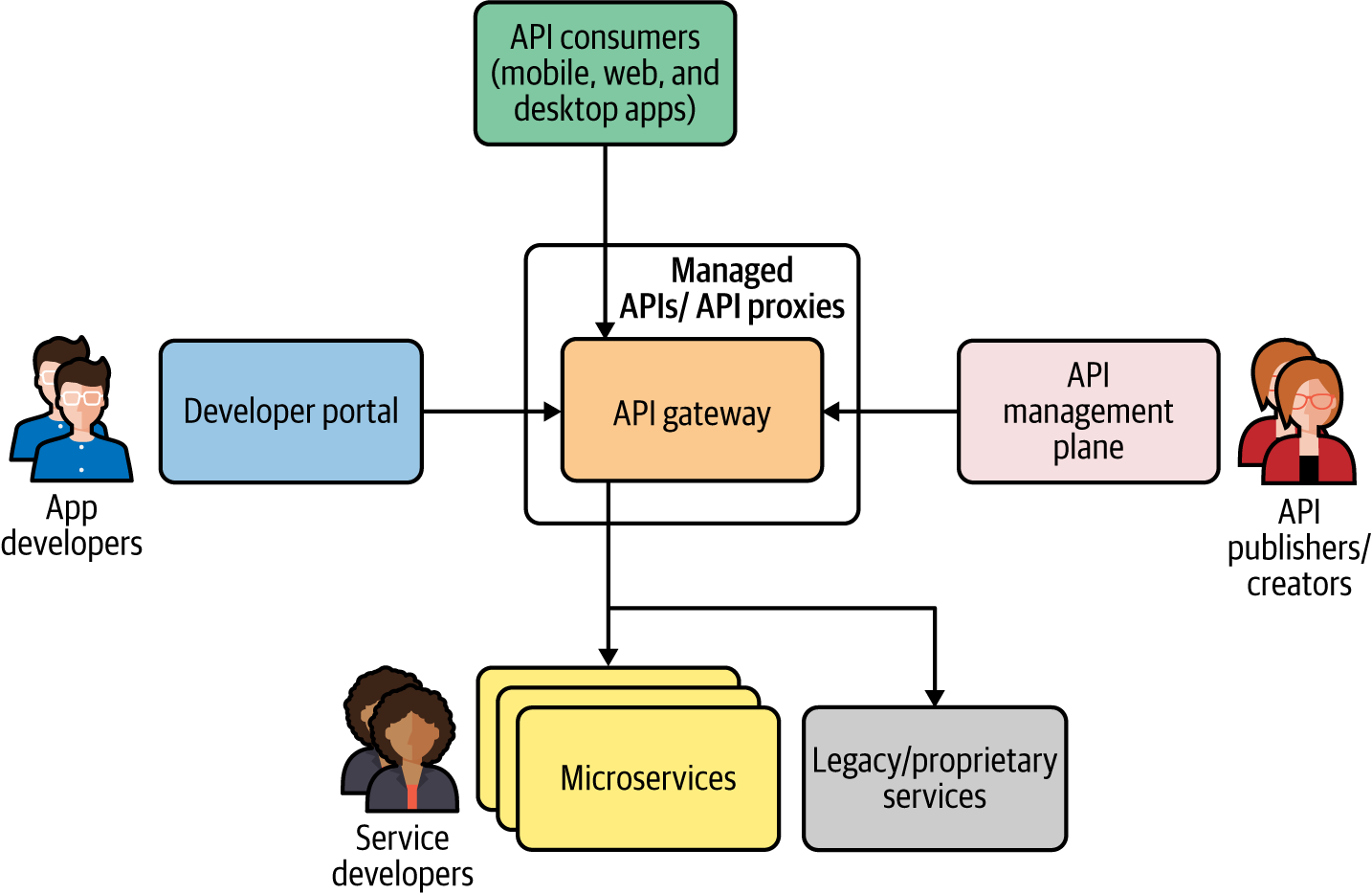
*API control plane admin*

Hosts and manages the API management solution. This person is responsible for creating user roles in the system, managing databases, and ensuring security.

If we look at a typical life cycle of an API and how these roles come into the picture at different stages, we can identify the key steps of API life-cycle management as follows:

1. We identify the business capabilities that we are going to expose as managed APIs.
2. The API creator (this can also be a person who develops services) creates a managed API (or API proxy) on top of the corresponding microservices of the business capability. The downstream services can be individual microservices or service compositions. The API is created at the API gateway.
3. API publishers can publish the APIs that are created so that they are available in the developer portal to be discovered by the application developers.
4. Application developers subscribe to APIs that they wish to consume through the API developer portal and learn more about the API so that they can consume applications on top of those APIs.
5. The applications that consume those APIs (such as mobile, web, or desktop applications) send requests via the API gateway to consume the business capabilities.
6. API publishers and creators control, observe, and manage the API exposed to the consumer.

[Figure 7-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_using_api_gateway_and_as) shows how each role maps to the key components of the API Gateway pattern.



**Figure 7-1. API management using API Gateway and associated roles**

Let’s discuss each component and their interactions in detail.

**API gateway**

The *API gateway* is the front door to your APIs. The APIs that you create in front of the downstream microservices are hosted inside the API gateway runtime. None of the roles that we’ve discussed directly interacts with the API gateway, but the API consumers do (for example, the consumers of the mobile and web applications that are built on top of the managed APIs)

The API gateway is responsible for accepting the API calls and applying security validations such as tokens, certificates, and other credential validations. It also applies various quality-of-service attributes to the API calls, such as throttling (quotas, rate limiting), caching, and versioning. In certain scenarios, you may also implement lightweight transformation logic (such as JSON to XML, or minor changes to the payload), but again we should not try to expose any logic related to the business capability of the API that we offer. An API gateway is also responsible for collecting all the data related to observability and API analytics.

**API management plane**

The *API management plane* is the main interface that the API creators and publishers use to control and administer the behavior of the APIs hosted through the API gateway and developer portal. Using the control plane, we can define and create APIs (and API products), change the life-cycle state of the APIs, and manage API access policies such as throttling, caching, security, and versioning. Management of users, schemas, and API visibility are also done at the control plane.

**API developer portal**

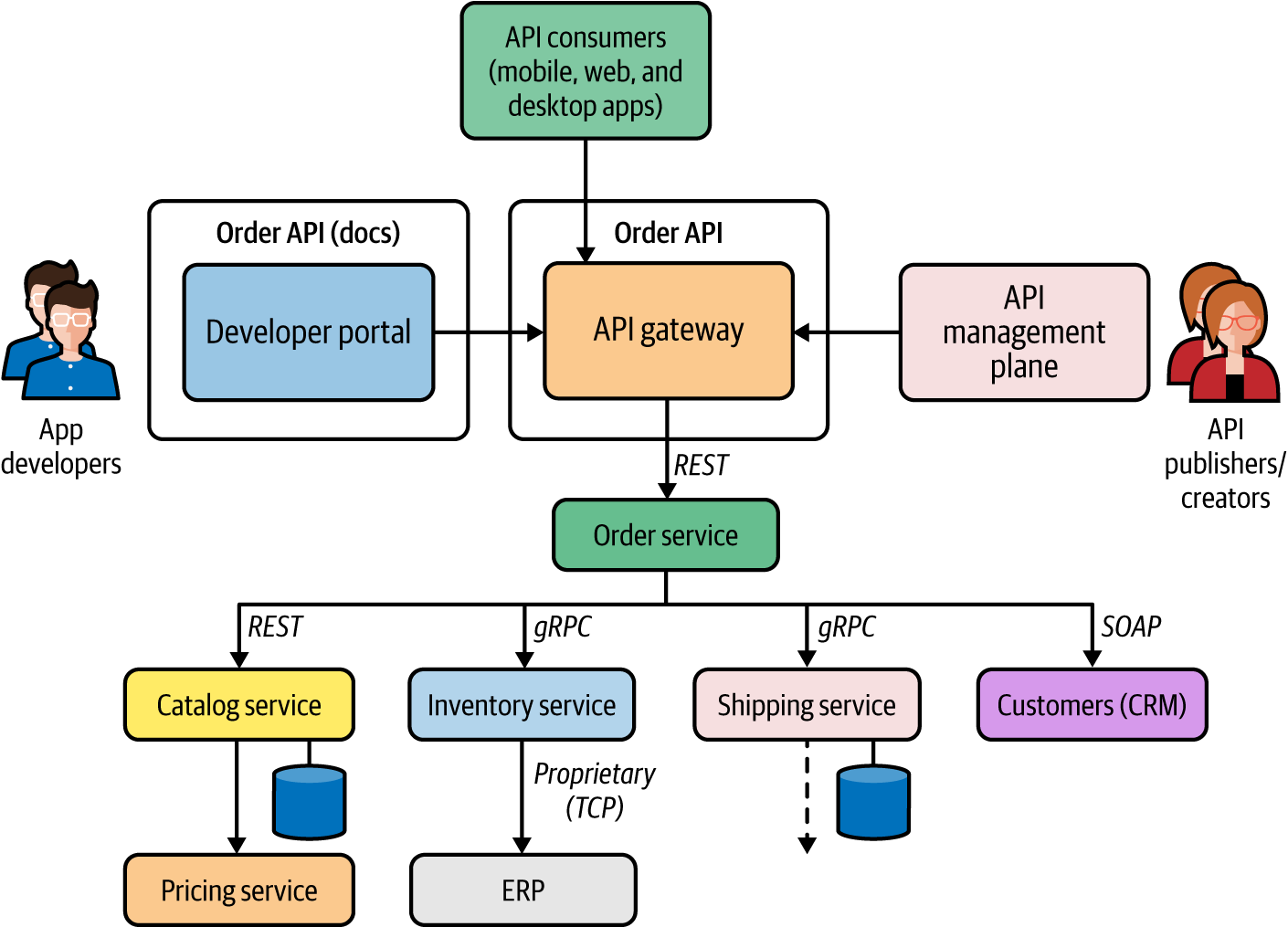
The *developer portal* is where you present your APIs to the application developers so they can discover and learn about your APIs, try them out, subscribe to them (or to API products), rate them, and provide feedback. (This is similar to the concept of an app store that you usually find with mobile devices such as iPhone or Android). The developer portal also allows developers to generate required security keys and tokens and use them when building applications that consume those APIs. Developers may also use the developer portal to obtain analytics on their own usage.

The API gateway is a mandatory component for implementing the API Gateway pattern, and the rest are optional. Based on your use case, you can choose whether you need a management plane or a developer portal. The roles of API creator and API publisher may be merged into one in most cases, as the microservices developers themselves can expose selected services as managed APIs by using the API Gateway pattern.

**How it’s used in practice**

The API Gateway pattern is usually used as a facade on top of existing microservices. Not all your microservices need to be exposed as APIs. When you want to expose a certain capability as an API, you first create a microservice that caters to that capability and then you can create an API by using that microservice as the backend.

[Figure 7-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#implementation_of_remote_proc-id00202) shows the Order API exposing the business functionalities related to order management in an online application. This requires multiple services to interact, and we may use the Service Orchestration or Service Choreography patterns to implement the service composition. Once we have the composite service (Order service, in this case), API creators/publishers can create an Order API by using the API management plane, and then publish it. Once you publish the API, it becomes available in the developer portal for the app developers. They can then build apps that consume the Order API.



**Figure 7-2. Implementation of remote procedure calls**

The applications that are built to consume the Order API send requests to the API gateway, where security, throttling, caching, versioning, and other policies are enforced. Since we start from microservices and then expose APIs later, this approach is known as a *bottom-up API management* technique.

The API Gateway pattern can also be used in a *top-down* approach, in which we start with the API design. Depending on the use case, this approach requires us to first design the API and then delve into the implementation, where we develop one or more microservices and their interactions.

Top-down API management is usually suitable when you have API management requirements at the time you start building a cloud native application. If you are adopting API management later into the development of your cloud native application, the bottom-up approach works better.

**Considerations**

Here are some of the main considerations to keep in mind when applying the API Gateway pattern:

* The API gateway layer should not have business logic–related capabilities. To have better separation of concerns, the underlying service should take care of the business capabilities.
* We can have multiple API management layers to expose APIs to internal and external consumers.
* At the initial stages, you may require only an API gateway, while the API control plane and developer portal are not mandatory. API publishing, and obtaining the information on published APIs, can be supported via dedicated interfaces of the API gateway (for example, the REST API).
* As your API management requirements become more advanced, you will need dedicated API control plane and developer portal components. Depending on the requirements, you also can adopt new components to handle API monetization, traffic management, and so on.
* The API gateway layer is often used as a monolithic runtime. If the gateway layer does any heavy lifting related to security, policy enforcement, and so forth, you may have to split it into multiple runtimes. The API Microgateway pattern that we discuss next can be used in such scenarios.

**Related patterns**

API management patterns can be used along with any of the service composition patterns that we discussed previously. We can use different variations of the API Gateway architecture to cater to specific requirements of cloud native applications. These variations are discussed in [“API Microgateway Pattern”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_microgateway_pattern) and in [“Service Mesh Sidecar as an API Gateway Pattern”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#service_mesh_sidecar_as_an_api_gateway).

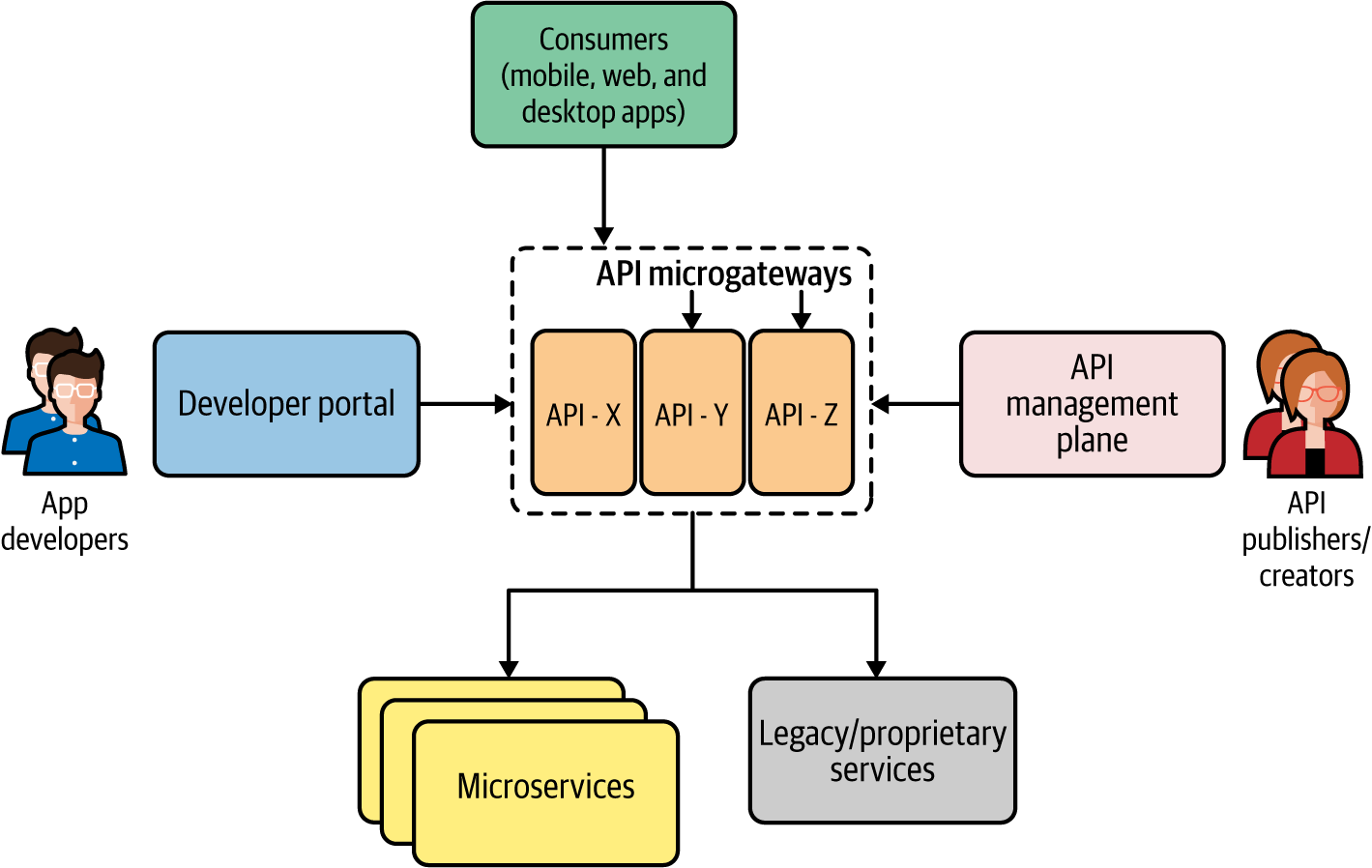
**API Microgateway Pattern**

The *API Microgateway pattern* is a simple variation of the API Gateway pattern. The key idea is to make the API gateway a distributed component so that each API deployed in the API gateway has an independent runtime. So, the API gateway is no longer a monolithic component in this pattern.

**How it works**

In the API Gateway pattern, the API gateway component hosts the runtime for all APIs that we expose through our API management layer. However, when the number of APIs that we expose grows, this monolithic API gateway layer can become really bulky and will start giving us all the problems of a conventional monolithic application. And a monolithic API gateway layer goes against some of the core principles of microservices in a cloud native application.

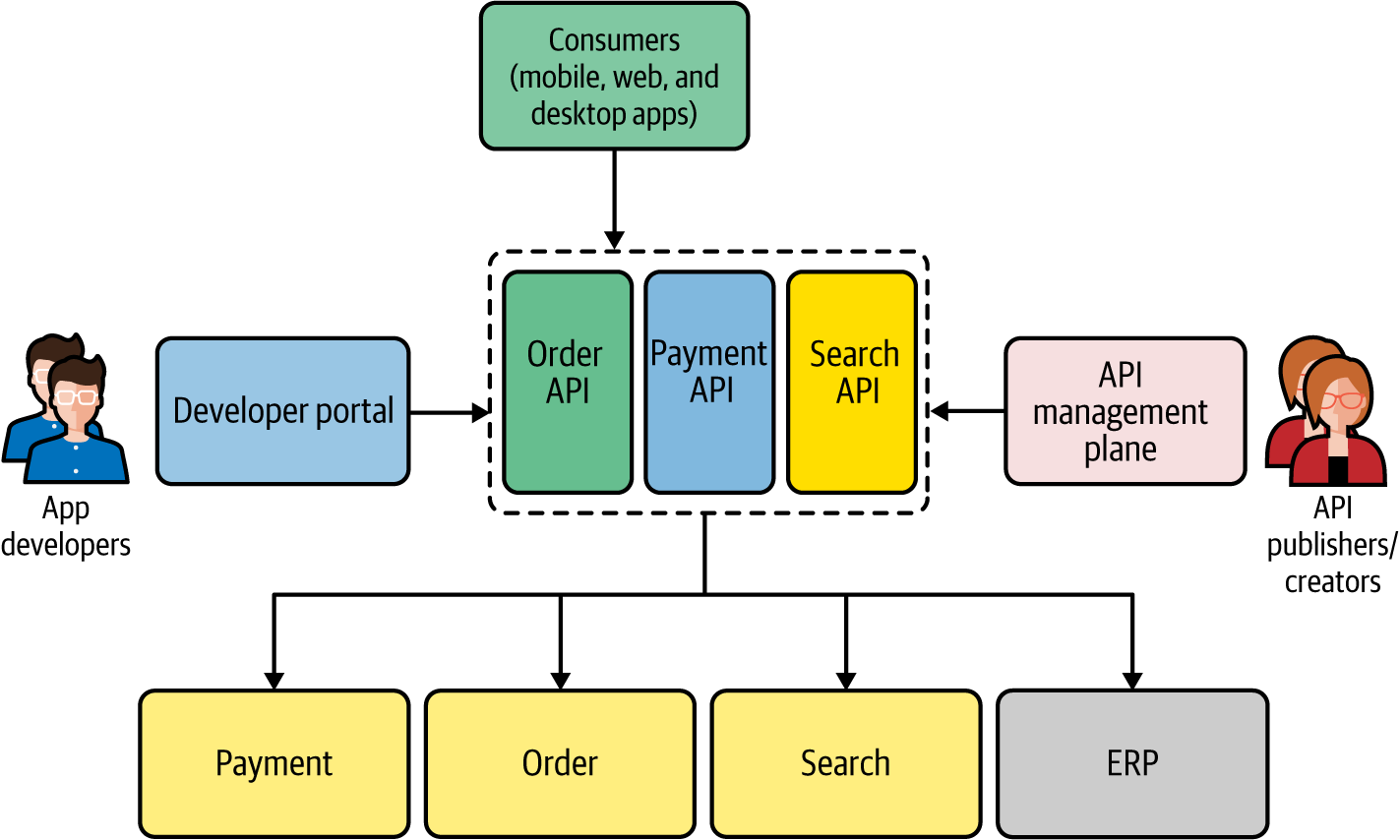
The APIs that we expose from an API management layer represent business capabilities that are designed, developed, deployed, and managed independently. Therefore, a true cloud native application shouldn’t have a monolithic or central API gateway layer. Instead, each API should have its own runtime—called a *microgateway*—and we should be able to manage all these runtimes together by using a central API management plane that is exposed via a central API developer portal. [Figure 7-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_with_microgateways) illustrates how to segregate an API gateway into multiple runtimes so that each API has its own runtime.



**Figure 7-3. API management with microgateways**

**How it’s used in practice**

The API Microgateway pattern is similar to the microservices concepts presented in [Chapter 1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch01.html#introduction_to_cloud_native). A microgateway is usually implemented using a lightweight gateway runtime deployed as a container. Therefore, we can leverage containers and container orchestration systems to run microgateways. The rest of the components of API management work exactly the same as in the API Gateway pattern. [Figure 7-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_microgateways_in_action) illustrates a typical use case of an online retail application, in which different APIs are exposed to consumers by using microgateways.



**Figure 7-4. API microgateways in action**

As you can see, each API is an independent entity that doesn’t share the API gateway runtime with other components. For example, if the Search API gets more traffic than the Order API, the Search API can be independently scaled.

**Considerations**

The API Microgateway pattern requires support from the underlying API management platform to have such a decentralized deployment of the gateway. Since we are using one gateway for each API, the gateway runtime must be extremely lightweight. When it comes to managing and scaling such a decentralized system, we need a container orchestration system such as Kubernetes to handle the operational overhead. In most API management solutions offered as cloud services, the existence of the microgateway is transparent, and the cloud service takes care of handling it seamlessly.

**Related patterns**

The API Microgateway pattern uses the same terminology as the API Gateway pattern. All the concepts that we discussed for the API Gateway pattern are equally applicable to the Microgateway pattern. The Service Mesh Sidecar as an API Gateway pattern is an application of the Microgateway pattern in the context of service mesh, which we discuss next.

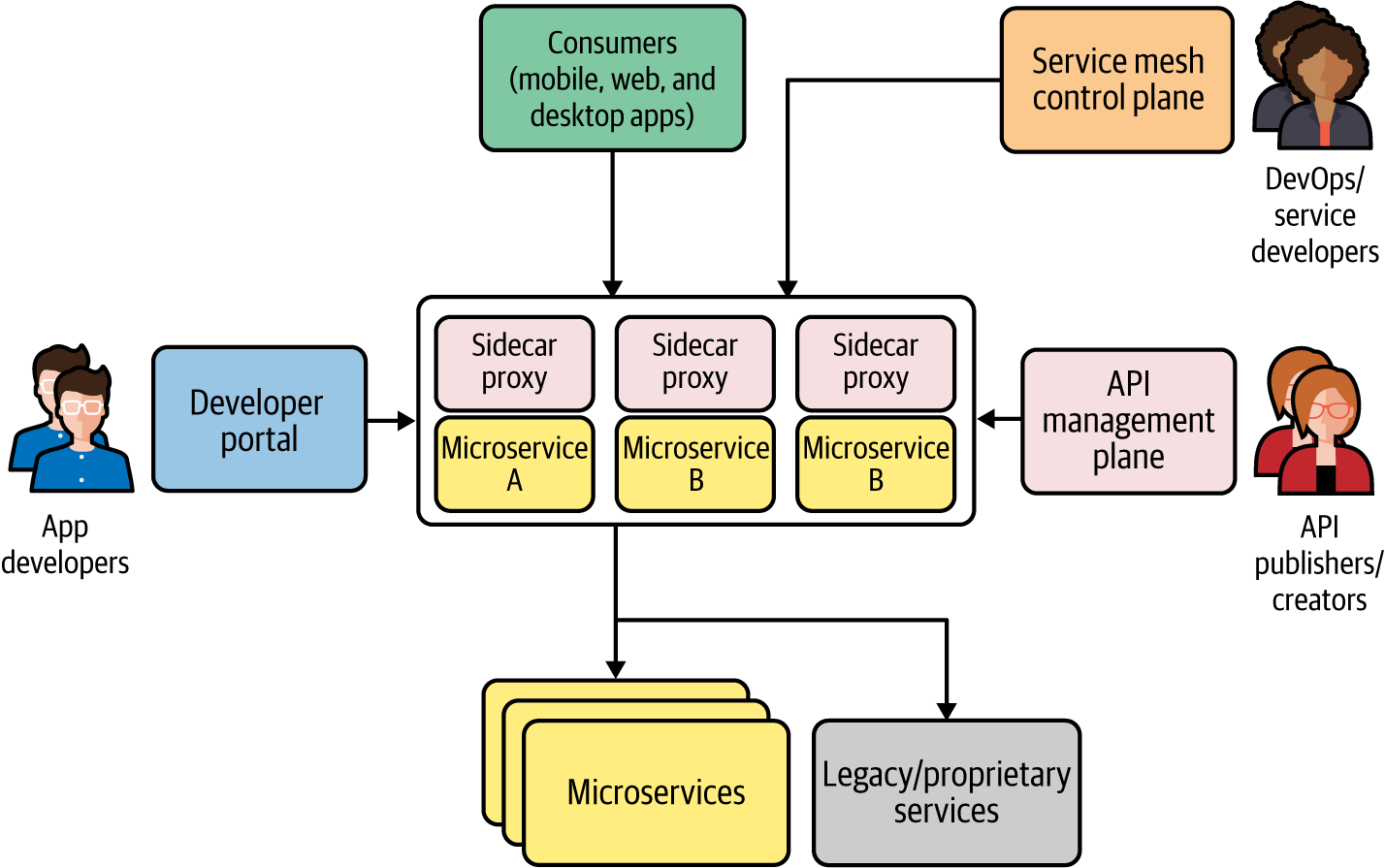
**Service Mesh Sidecar as an API Gateway Pattern**

We covered the Service Mesh and Sidecar patterns in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern). The key idea behind the Service Mesh Sidecar as an API Gateway pattern is that, in an environment where we use service mesh, rather than using a separate API gateway runtime, we can offload the tasks of the API gateway to the sidecar proxies running alongside each microservice.

**How it works**

Suppose you are already using a service mesh to run your microservices and want to include API management in the same deployment. For the microservices that you want to expose as APIs, you can leverage a sidecar proxy to function as the API gateway in addition to supporting the normal service mesh capabilities. Each service mesh sidecar proxy that is colocated with the microservices can also be controlled via the API management plane and service mesh control plane, as shown in [Figure 7-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_through_service_mesh_sid).

The type of users who interact with the service mesh control plane are different from those who interact with the API management plane. Service developers or DevOps primarily control the service mesh behavior, while API creators and publishers manage the API life cycle.



**Figure 7-5. API management through service mesh sidecars**

One of the main reasons for using a sidecar proxy as an API gateway for each microservice is that the requirements of a service mesh sidecar proxy and an API gateway are similar. Both intend to work as *proxies* and are controlled by a separate control plane. So, when we need to expose a capability as a managed service, we select the capability that is often implemented as a separate microservice (if no such service exists, we have to build it). Through the API management control plane, you can convert it to an API that is published to the developer, stored, and served via the same sidecar proxy runtime.

**How it’s used in practice**

The implementation of this pattern is taken care of by the API management solution that you use for most parts, and most of the underlying details are transparent to the users who use the API management solution. The API control plane can directly manage sidecar proxies or can use the service mesh control plane APIs to manage them. In a cloud environment that provides API management as a cloud service, the implementation details of the API gateway may become irrelevant. This pattern is used when you are already using a service mesh and want to include API management capabilities on top of it.

**Considerations**

This pattern has some of the most complex distributed architecture comprising multiple distributed components. Its operational complexity is even greater than that of the Service Mesh pattern. If you are adopting this pattern, you should be ready to handle this complexity or leverage a cloud service that is already doing it for you.

**Related patterns**

The application of this pattern requires a solid understanding of the Service Mesh pattern (covered in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern)), as well as the API Gateway and API Microgateway patterns (covered in this chapter).

**Technologies for Implementing API Management Patterns**

The current API management technology space is a crowded market. The on-premises offerings include [Kong](https://konghq.com/), [MuleSoft](https://oreil.ly/eD5Yi" \t "_blank), [Red Hat 3scale API Management](https://oreil.ly/1rPne), and [WSO2 API Manager](https://oreil.ly/qrAOU).

All the leading cloud providers have a significant footprint in cloud offerings for API management such as [Google’s Apigee](https://oreil.ly/5mSRW), [Azure API Management](https://oreil.ly/NYAye), [Amazon API Gateway](https://oreil.ly/3BnPg), and [MuleSoft Anypoint Platform](https://oreil.ly/Gtzb9" \t "_blank).

**Summary of API Management Patterns**

[Table 7-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_management_patterns) lists the API management patterns, and details when and when not to use them.

| **Pattern** | **When to use** | **When not to use** |
| --- | --- | --- |
| API Gateway | Essential in all the API management use cases. API management plane and developer portal are optional for small-scale use cases but require those components if you plan to do end-to-end API life cycle management. | (Not applicable.) |
| API Microgateway | Only when the API gateway layer requires scaling and isolation per each API. | If your cloud native application exposes a handful of managed APIs that don’t do any heavy lifting, using this pattern will incur a redundant complexity. |
| Service Mesh Sidecar as API Gateway | Only when you are using a service mesh in production and want to include API management without using a separate layer. | If you don’t have a service mesh or plan to use it at the production level, using this pattern is overkill. Even if you use a service mesh, you can still use API management as a separate component. |
| Table 7-1. API management patterns | | |

**API Consumption Patterns**

Most of the connectivity patterns that we discussed so far are applicable to backend services and systems. In this section, we will explore some of the commonly used patterns to connect frontend and backend applications when building cloud native applications.

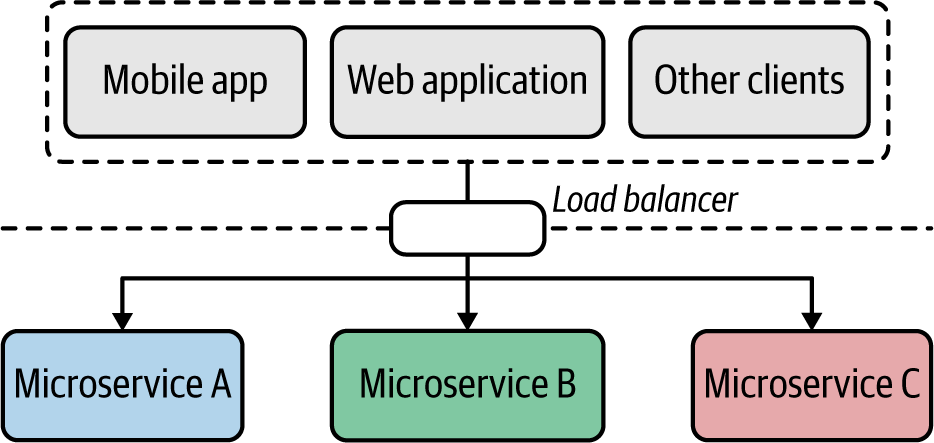
In a typical cloud native application, we have a collection of microservices, a messaging infrastructure, data stores, and frontend applications that expose capabilities directly to consumers. So, frontend and backend connectivity patterns are built around the ways that you can integrate frontend applications (such as web, mobile, and desktop applications) with the backend microservices and systems.

**Direct Frontend-to-Microservices Communication Pattern**

When you build a cloud native application’s backend by building a series of microservices that expose business capabilities, the frontend applications can directly consume those microservices.

**How it works**

To facilitate the direct consumption of microservices by the frontend application, you need to expose all the microservices required for your frontend applications as public-facing services. [Figure 7-6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#directly_exposing_microservices_to_fron) shows what this looks like. The frontend applications access these services via a load balancer, and extra care needs to be taken to secure these microservices as they are directly exposed to the public.



**Figure 7-6. Directly exposing microservices to frontends**

Each application is directly coupled to the service interface of these microservices; client applications need to change when the service gets changed. All the cross-cutting capabilities such as authentication, authorization, and observability need to be implemented at the level of each microservice.

**How it’s used in practice**

This pattern is suitable if you are building a simple cloud native application with a few services and systems. This pattern is often used in small-scale microservices deployments that can’t afford to have a dedicated API management layer to expose microservices as managed APIs, which we’ll discuss in the next section. Later, when the system needs to scale for more consumers and more microservices, we can bring in the API Gateway pattern.

**Considerations**

As discussed, this pattern has inherent limitations, including tight coupling between the frontend and backend services, security risks of directly exposing microservices to the public, and no central place to apply cross-cutting capabilities across multiple microservices.

**Related patterns**

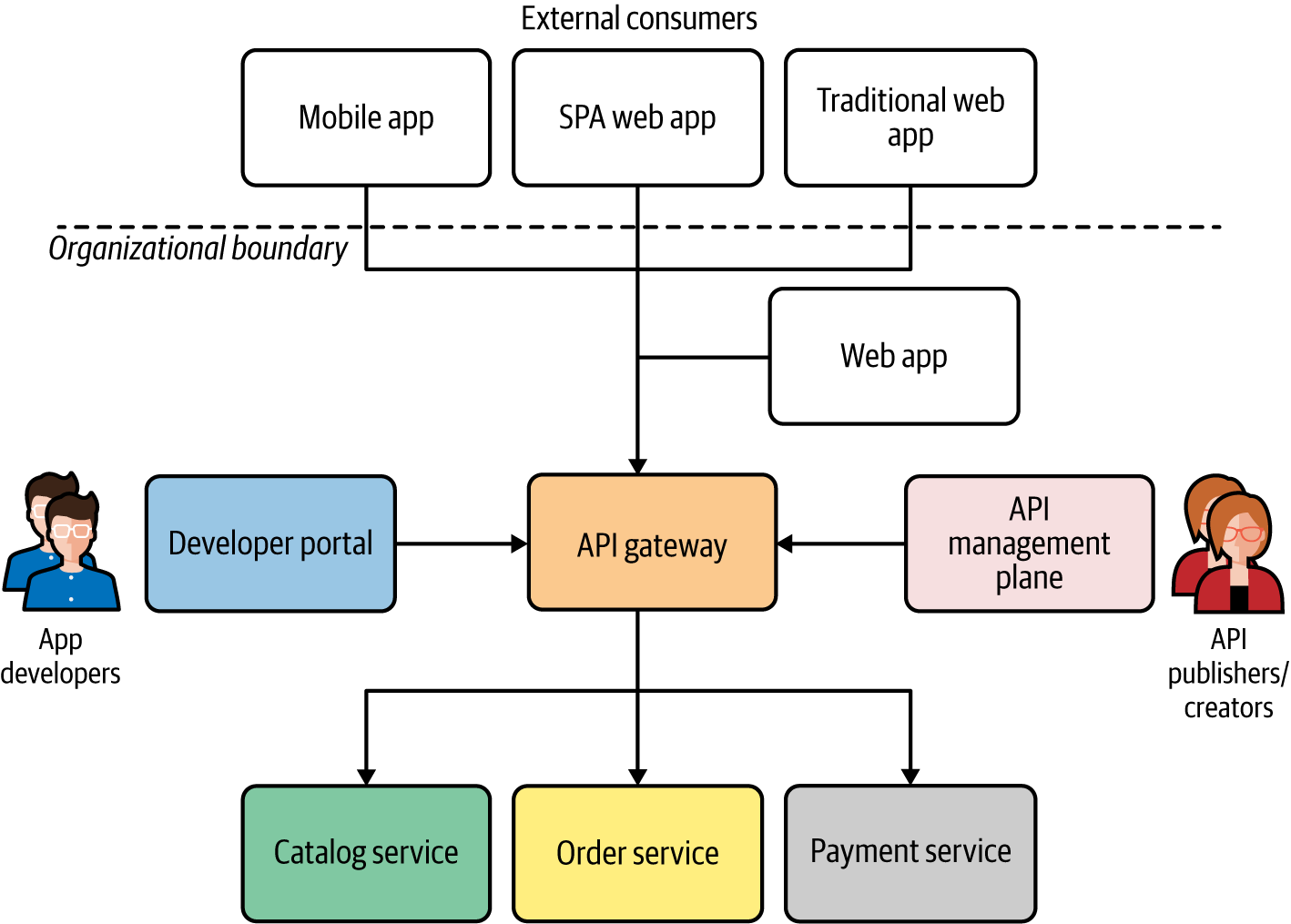
This pattern often gets extended to use an API gateway instead of directly consuming the microservices.

**Frontends Consuming Services Through API Gateway Pattern**

When you have to expose the business capabilities of your cloud native application, rather than exposing microservices directly to consumers, you can use an API management layer. Instead of accessing the microservices directly, frontends access microservices via the API management layer.

**How it works**

We implement frontend to API management layer communication by exposing all the services or business functionalities that need to be exposed to the external parties via an API management layer ([Figure 7-7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#frontends_are_consuming_business_capabi)). This layer comprises an API gateway, API management plane, and a developer portal.



**Figure 7-7. Frontends are consuming business capabilities via an API management layer**

To expose a given microservice as an API, you need to define an interface (via an API contract) that is provided to the consumers who access your API. The API gateway adheres to that contract. The API contract that you expose from the API gateway may or may not be the same as the API of the backend microservice. The API management layer acts as the facade, and the internal implementation details of the microservices can be changed without breaking the consumer frontends. Frontends completely rely on the APIs exposed via the API gateway, and have to adhere to all the cross-cutting requirements enforced by the API management layer such as security, throttling, service access policies, and so on.

**How it’s used in practice**

Most organizations increasingly adopt this API management pattern and are moving all their frontend developments to rely on the organization’s exposed APIs. When you are not using an API management layer to build your frontend applications, you should consider gradually introducing API management into the architecture and rebuild the frontends so that they use exposed APIs rather than invoking microservices directly.

**Considerations**

When building a frontend application using APIs, one of the limitations you face is that your frontend application has to depend on a general-purpose API rather than something specifically serving the needs of the frontend application (for example, the requirements of a mobile application versus desktop application may be drastically different). Proper design of the APIs hosted at the API management layer is critical, as well as being able to customize APIs for your frontends, which may also be required in certain use cases. This is discussed in the next section.

**Related patterns**

This pattern is built on top of API management patterns that we discussed in the previous sections. You may also further extend this pattern with the Backend for Frontends pattern, which creates a dedicated API for each frontend component.

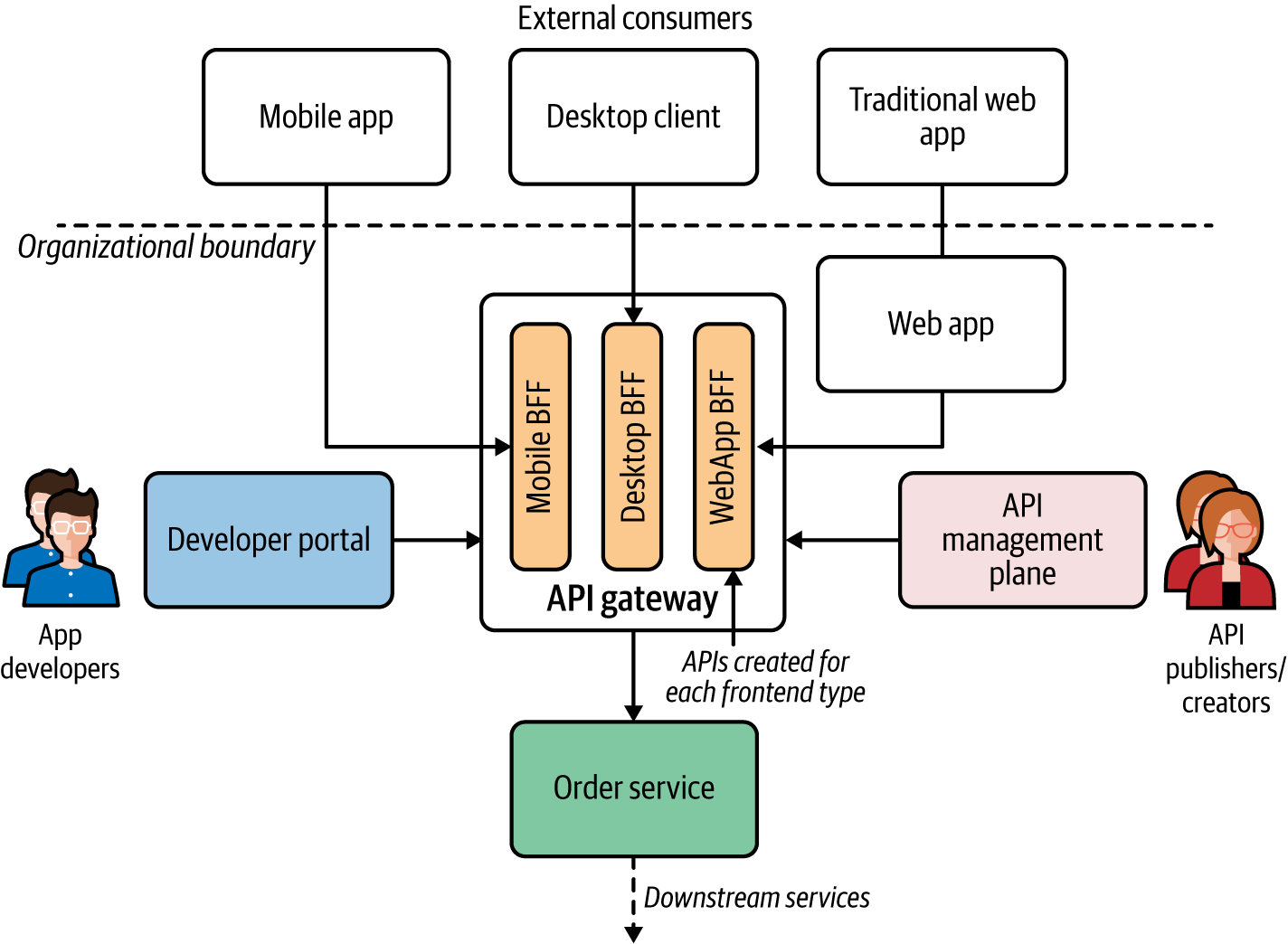
**Backend for Frontends Pattern**

With frontend applications, we expose the capabilities of our cloud native application via different frontend application types such as mobile, desktop, or web applications. When we use a general-purpose API and build all these frontend applications on top of it, we often hit roadblocks because the required functionality is not directly available in these common APIs. The *Backend for Frontends pattern* solves this problem by introducing APIs that are tailor-made for specific frontend applications.

**How it works**

The requirements of a frontend application can significantly vary from one application type to another. The frontend of a mobile application may be drastically different from that of a web application, for instance. In such scenarios, if we want to expose a general-purpose API of the business capability to consumers via the frontend application, we may face quite a lot of constraints and roadblocks. To build the best user experience for each frontend application type, we will need to use an API that is tailor-made for the requirements of the specific frontend application that we develop. These APIs are known as *backends for frontend applications* (*BFFs*).

Rather than depending on a single general-purpose API, we can build different backend APIs for each frontend application type ([Figure 7-8](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#the_backend_for_frontends_pattern_in_ac)). These APIs may directly interact with the backend microservices (in this case, the Order microservice) or can rely on a general-purpose API that exposes the core business functionality.



**Figure 7-8. The Backend for Frontends pattern in action: each frontend application has a corresponding backend for frontend API/service at the API gateway**

The APIs that are specific to a particular frontend type are usually managed by the same team that develops the frontend application. This pattern also allows each BFF the freedom to have its own life cycle so that changes in one API won’t impact the other APIs.

**How it’s used in practice**

The Backend for Frontends pattern is commonly used when we want to enable a different experience for consumers through different frontend applications. The frontend application requirements can drastically vary. Mobile applications want certain data formats, lightweight and free messages, and different security protocols compared to desktop applications, for instance. Therefore, when designing BFFs, you have the freedom of choosing the most suitable technology for building the best consumer experience for the business capability that you offer.

**Considerations**

While BFF reduces the complexity of having a general-purpose API, it also adds operational overhead, as the same capability is now being offered via multiple APIs. It’s usually a good practice to use a generic downstream service or API to build a BFF, as it can be used to track the consumption of the API and to generate analytics for the API usage. This pattern has inherent drawbacks due to the proliferation of APIs when the number of APIs that we expose increases as well as the type of frontends that we support increases. You should be using BFF only when it significantly improves the user experience for your cloud native application.

**Related patterns**

BFF is commonly used along with the API Gateway and API Microgateway patterns covered earlier in this chapter. With this pattern, we can also leverage service composition patterns such as Service Orchestration and Service Choreography (covered in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern)).

**Summary of API Consumption Patterns**

[Table 7-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch07.html#api_consumption_patterns) lists the API Consumption patterns and when and when not to use them.

| **Pattern** | **When to use** | **When not to use** |
| --- | --- | --- |
| Direct Frontend-to-Microservices Communication | API management is not a critical requirement. | Often you need to bring managed APIs into the picture at some point. So this approach is suitable only when getting started. |
| Frontends Consuming Services Through API Gateway | The consumers are homogeneous and don’t have disparate requirements to fulfill at the frontend level. The API exposed to consumers works for all the disparate sets of consumers. | The API needs frequent changes due to frontend requirements. The successful consumption of an API largely depends on the frontend application. |
| Backend for Frontends (BFF) | We want to have a unique experience for each frontend application. | You have a single type of frontend application or its requirements are homogeneous. |
| Table 7-2. API consumption patterns | | |

**Summary**

In this chapter, we explored how the business capabilities are presented to your consumers as managed APIs, which allow us to control, secure, monetize, and govern the consumption of the business capabilities of our cloud native applications. We also connected the dots between the frontend applications, API layer, and backend microservices by looking at the patterns related to frontend and backend connectivity.

In the next chapter, we’ll apply some of the cloud native patterns discussed throughout this book to a complete use case.