Chapter 14. Service-Based Architecture Style

Service-based architecture is a hybrid variant of the microservices architectural style and is considered one of the most pragmatic styles available, mostly due to its flexibility. Although service-based architecture is distributed, it doesn't have the same level of complexity and cost as other distributed architectures (such as microservices or event-driven architecture), making it a popular choice for business-related applications.

Topology

The basic topology of service-based architecture follows a distributed macro-layered structure consisting of a separately deployed user interface; separately deployed, remote, coarse-grained services; and, optionally, a monolithic database.

Although Figure 14-1 illustrates the *basic* topology of this style, it can vary greatly, including using separate UIs and separate databases. This chapter will describe these topology variants in detail.

Services within this architectural style typically represent a specific domain or subdomain with a system, so they are called *domain services*. Domain services are generally coarse-grained and represent some portion of the system's functionality (such as order fulfillment or order shipping). They are typically independent of each other and separately deployed. Services are typically deployed much like any monolithic application would be, so they do not require containerization (although deploying a domain service in a container such as Docker or Kubernetes is certainly an option). When using a single monolithic database, an architect should try to minimize the number of domain services (we recommend no more than 12) to avoid change control, scalability, and fault tolerance issues.

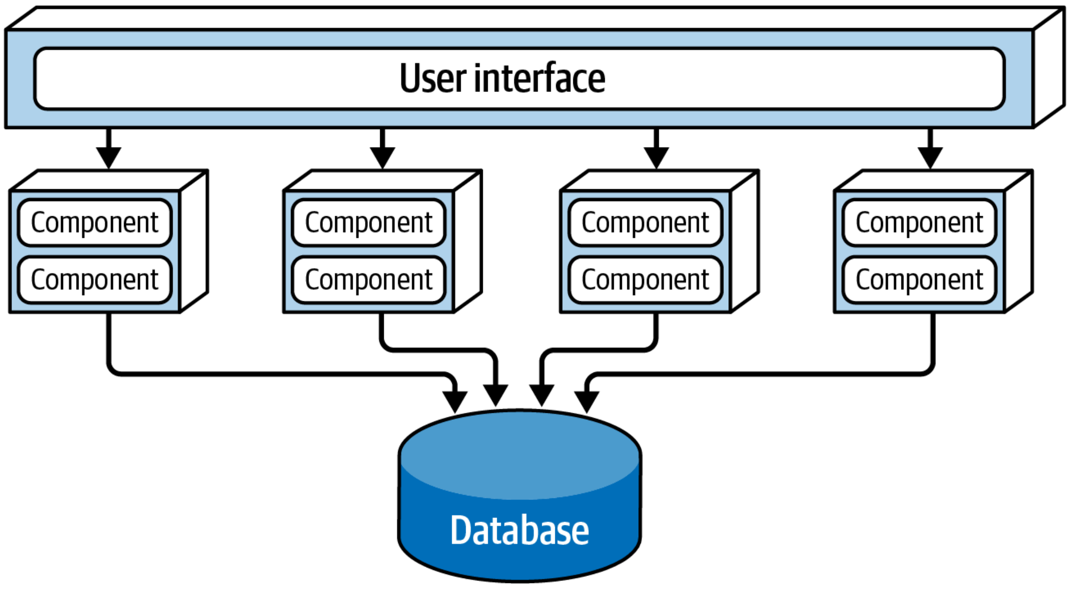


Figure 14-1. Basic topology of the service-based architectural style

Typically, each domain service is deployed as a single instance. However, based on the system's scalability, fault tolerance, and throughput needs, architects sometimes create multiple instances of a domain service. Doing so usually requires some sort of load-balancing capability between the UI and the domain service so that the UI can be directed to a healthy and available service instance.

Services are accessed remotely from a UI using a remote access protocol, typically REST. Other possibilities include messaging, remote procedure call (RPC), an API layer with a proxy or gateway, or even SOAP. However, in most cases, the UI has an embedded service locator pattern so it can access the services directly; the service locator pattern can also be embedded within an API Gateway or proxy.

The service-based architecture typically uses a centrally shared monolithic database. This allows services to leverage SQL queries and joins just as a traditional monolithic layered architecture would. Because of the small number of domain services typically found in this architectural style, exhausting the available database connections is rarely an issue. Managing database changes, however, can present a challenge. "Data Topologies" describes techniques for addressing and managing database changes within a service-based architecture.

Style Specifics

Because domain services in a service-based architecture are generally coarse-grained, each domain service is typically designed using a layered architectural style consisting of an API Facade layer, a Business layer, and a Persistence layer. Another popular design approach is to partition each domain service into subdomains, similar to the modular monolith architectural style (see Chapter 11). Both approaches are illustrated in Figure 14-2.

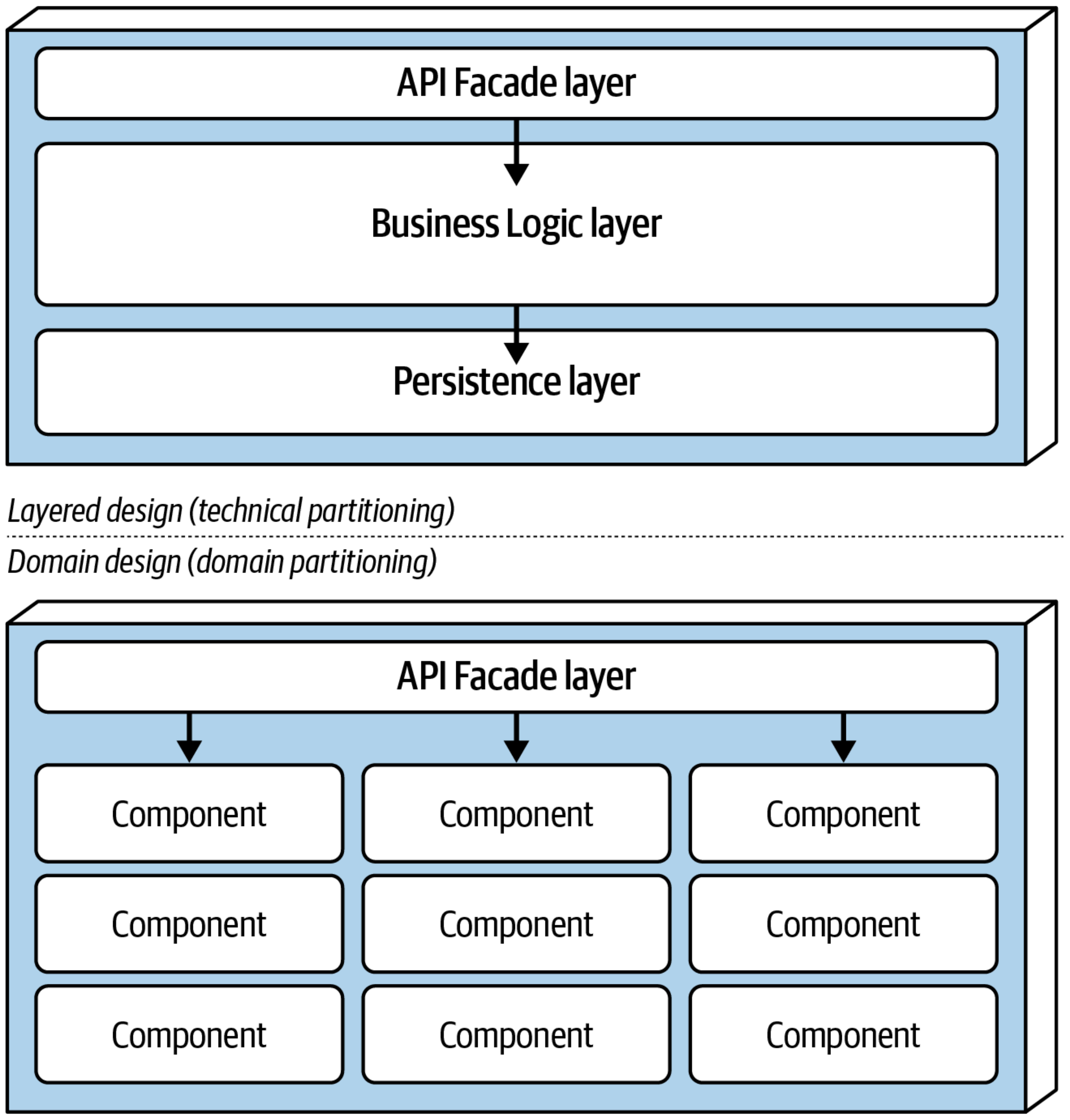


Figure 14-2. Domain service design variants

Regardless of its design, a domain service must contain some sort of API access facade that the UI interacts with to execute some sort of business functionality. The API access facade typically takes on the responsibility of orchestrating the business request from the UI.

Let’s consider an example from an ecommerce site with a service-based architecture. A business request comes from the UI: a customer is ordering some items. This single request is received by the API access facade within the `OrderService` domain service. The API access facade internally orchestrates everything that has to happen to fulfill this request: placing the order, generating an order ID, applying the payment, and updating the inventory for each product ordered. In a microservices architecture, completing this request would likely involve orchestrating many separately deployed, remote, single-purpose services. This difference in granularity—between internal class-level orchestration in a service-based architecture and external service orchestration in microservices—points to one of the many significant differences between the two styles.

Because domain services are coarse-grained, regular ACID (atomicity, consistency, isolation, durability) database transactions with standard database commits and rollbacks can ensure database integrity within a single domain service. Compare this to the fine-grained single-purpose services of highly distributed architectures like microservices, which use a distributed transaction technique known as BASE transactions (basic availability, soft state, eventual consistency). Such fine-grained services don't support the same level of database integrity that ACID transactions in a service-based architecture can support.

To illustrate this point, consider the order checkout process within our example service-based ecommerce site. Suppose the customer places an order, but the credit card they use for payment has expired. Since this payment is an atomic transaction within the same service, the service can remove everything added to the database so far using a standard transaction rollback. The service notifies the customer that the payment cannot be applied.

Now consider this same process in a microservices architecture, with its smaller, fine-grained services. First, the `OrderPlacement` service accepts the request, creates the order, generates an order ID, and inserts the order into the order tables. Once this is done, the `OrderPlacement` service makes a remote call to the `PaymentService`, which tries to apply the payment. If the payment cannot be applied because the credit card is expired, the order cannot be placed. Now the data is in an inconsistent state: the order information has already been inserted, but has not been approved. A separate action known as a *compensating update* (discussed in Chapter 9) would need to be applied to the `OrderPlacement` service to bring the data to a consistent state.

Service Design and Granularity

Because domain services are coarse-grained, they allow for better data integrity and consistency, but there's a big trade-off. In a service-based architecture, changing the order-placement functionality in an `OrderService` would require the team to test and redeploy all of the service's functionality, including payment processing. In a microservices architecture, however, the same change would only affect a smaller, fine-grained `OrderPlacement` service, requiring no testing or deployment for the `PaymentService`. Furthermore, because a domain service deploys more functionality, there's more risk that something else might break (including payment processing). With microservices, each service has a single responsibility, so there's less chance of breaking other functionalities when making changes.

User Interface Options

The service-based architecture style includes many UI variants, making it very flexible. For example, an architect could break apart the single monolithic UI illustrated in Figure 14-1 into separate UIs, even to the point of matching each domain service. This would increase the system's overall scalability, fault tolerance, and agility. These UI variants are illustrated in Figure 14-3.

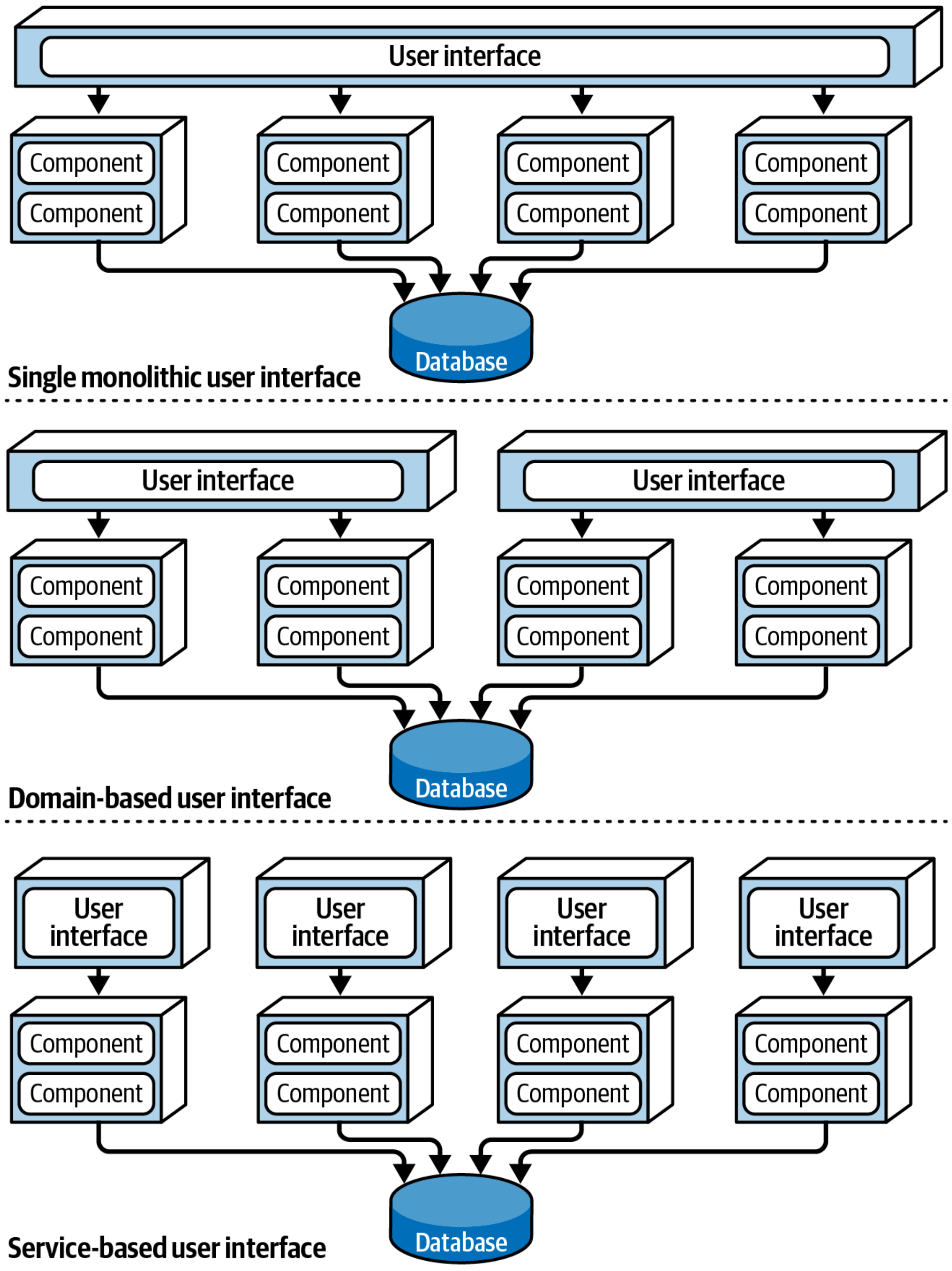


Figure 14-3. Three user-interface variants in a service-based architecture

For example, a typical ordering system could have a UI for customers to place orders, and separate, internal UIs for the order packers to view the items that need to be packed and for customer support.

API Gateway Options

Thanks to the flexibility of this architectural style, it’s possible to add an API layer consisting of a reverse proxy or API Gateway between the UI and services, as shown in Figure 14-4. This is useful for exposing domain service functionality to external systems, for consolidating shared cross-cutting concerns (such as metrics, security, auditing requirements, and service discovery) and moving them into the API Gateway, and for load-balancing domain services that have multiple instances.

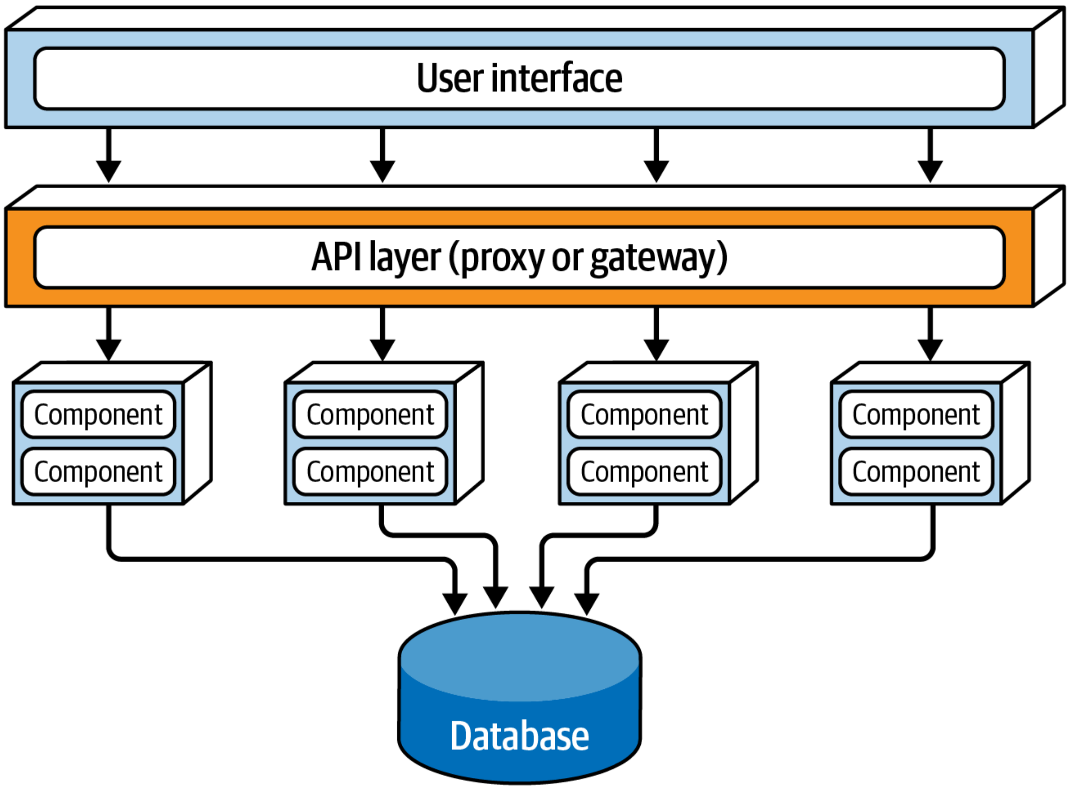


Figure 14-4. Adding an API layer between the UI and the domain services

Data Topologies

Service-based architectures offer architects many choices of database topology, further illustrating their flexibility. This style is unique in being a distributed architecture that can effectively support a monolithic database. However, that single monolithic database could be broken apart into separate databases, even going as far as to create a domain-scoped database to match each domain service (similar to microservices). If using multiple separate databases, the architect must make sure that no other domain service needs the data in each database, which might incur interservice communication between domain services. It's usually preferable to share data rather than call another domain service in this architectural style. These database variants are illustrated in Figure 14-5.

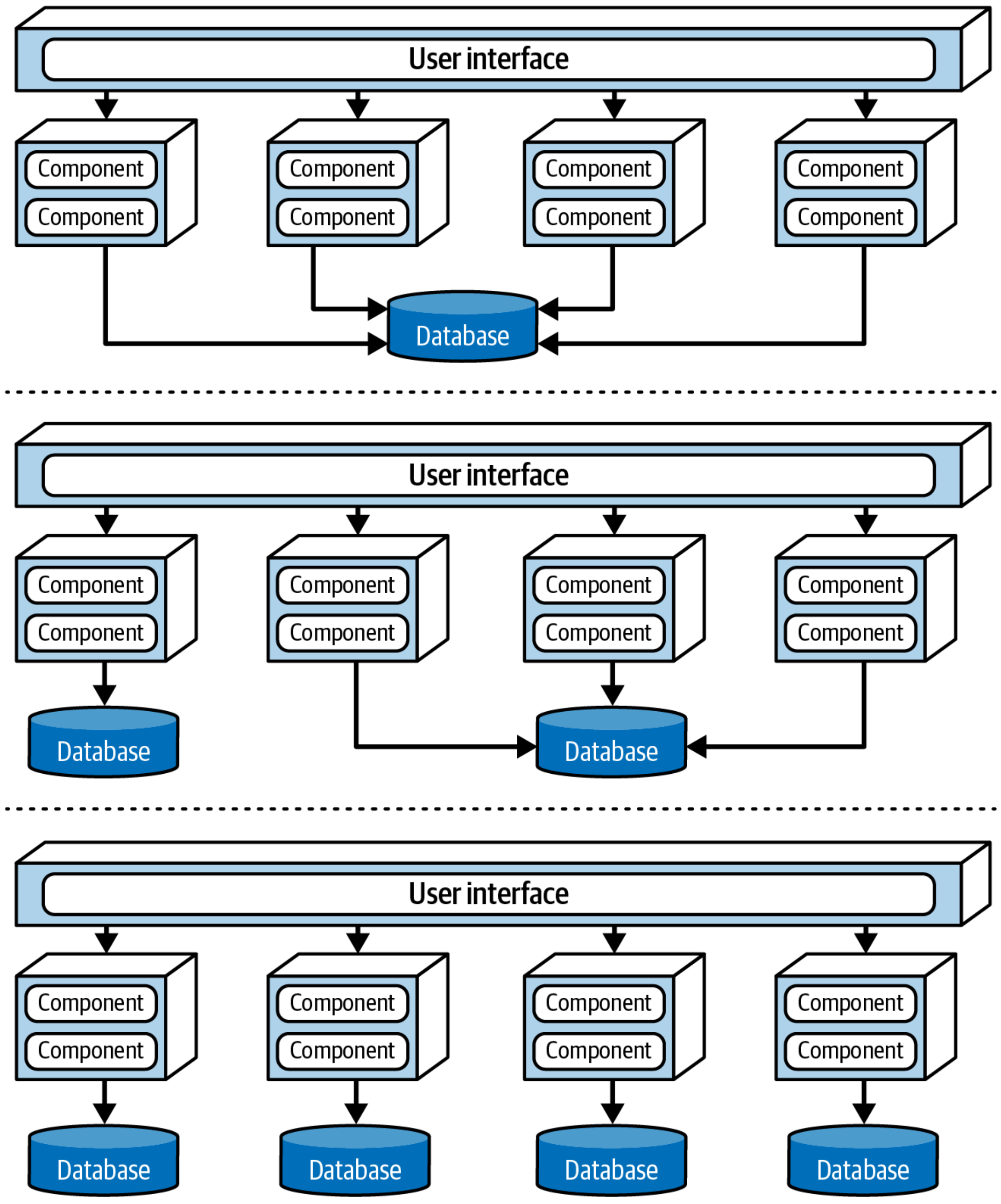


Figure 14-5. Service-based architecture database topologies

Although this architectural style supports a monolithic database, schema changes to the database table can, if not done properly, affect every domain service. This makes changing the database a very costly, risky task that demands a great deal of effort, coordination, and overall reliability.

In a service-based architecture, the shared class files representing the database table schemas (called *entity objects*) usually reside in a custom shared library used by all of the domain services, such as a JAR or DLL file. These shared libraries can also contain SQL code. Creating a single shared library of all of the entity objects is the *least* effective way to implement a service-based architecture. Any change to the database table structures also requires changing that library of the corresponding entity objects, which means changing and redeploying *every* service, regardless of whether it actually accesses the changed table. Shared library versioning can help address this issue, but it's still difficult to know which services the table change will affect without doing detailed manual analysis. This scenario, which is considered an antipattern in service-based architecture, is illustrated in Figure 14-6.

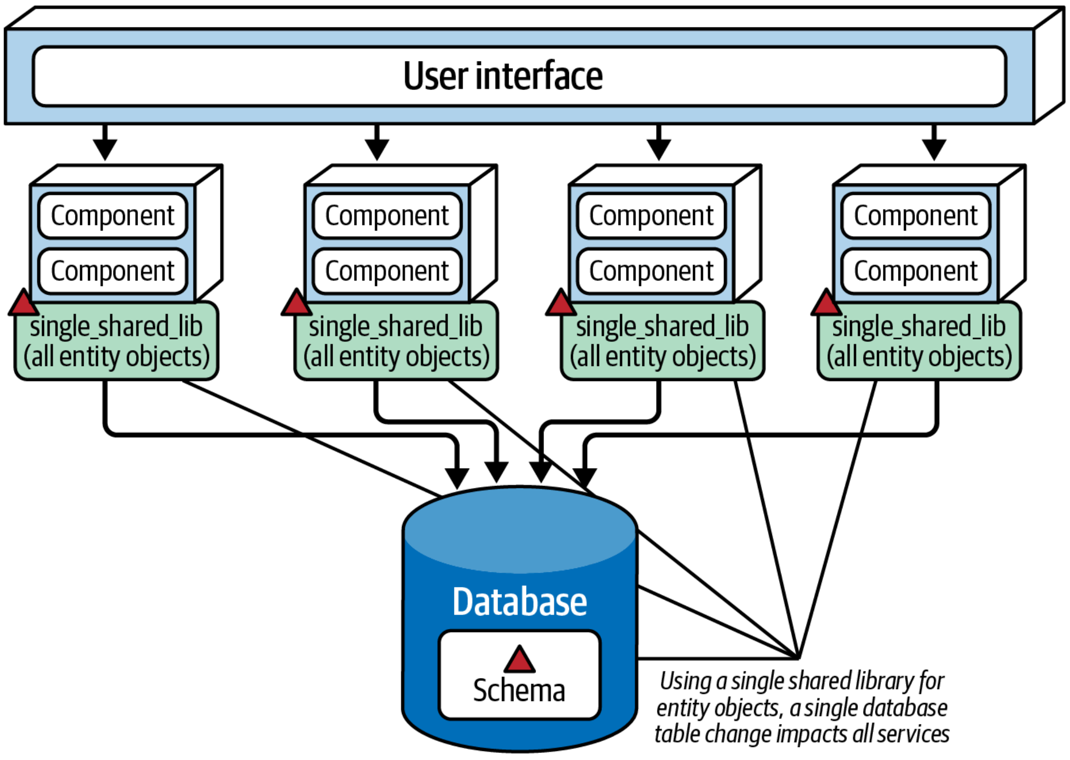

User interface
Component
Component
single_shared_lib (all entity objects)
Database
Schema
Using a single shared library for entity objects, a single database table change impacts all services

Figure 14-6. Using a single shared library for database entity objects impacts all services when a change occurs and is considered an antipattern in service-based architecture

One way to mitigate the impact and risk of database changes is to logically partition the database and represent that logical partitioning through separate shared libraries. Notice that in Figure 14-7, the database is logically partitioned into five separate domains: common, customer, invoicing, order, and tracking. The domain services use five corresponding shared libraries that match the logical partitions in the database. Using this technique, any changes the architect makes to a table within a particular logical domain (in this case, `Invoicing`) would match the corresponding shared library containing the entity objects (and possibly SQL as well). The only services affected would be those using that shared library. No other services would be affected by this change, so there would be no need to retest and redeploy them.

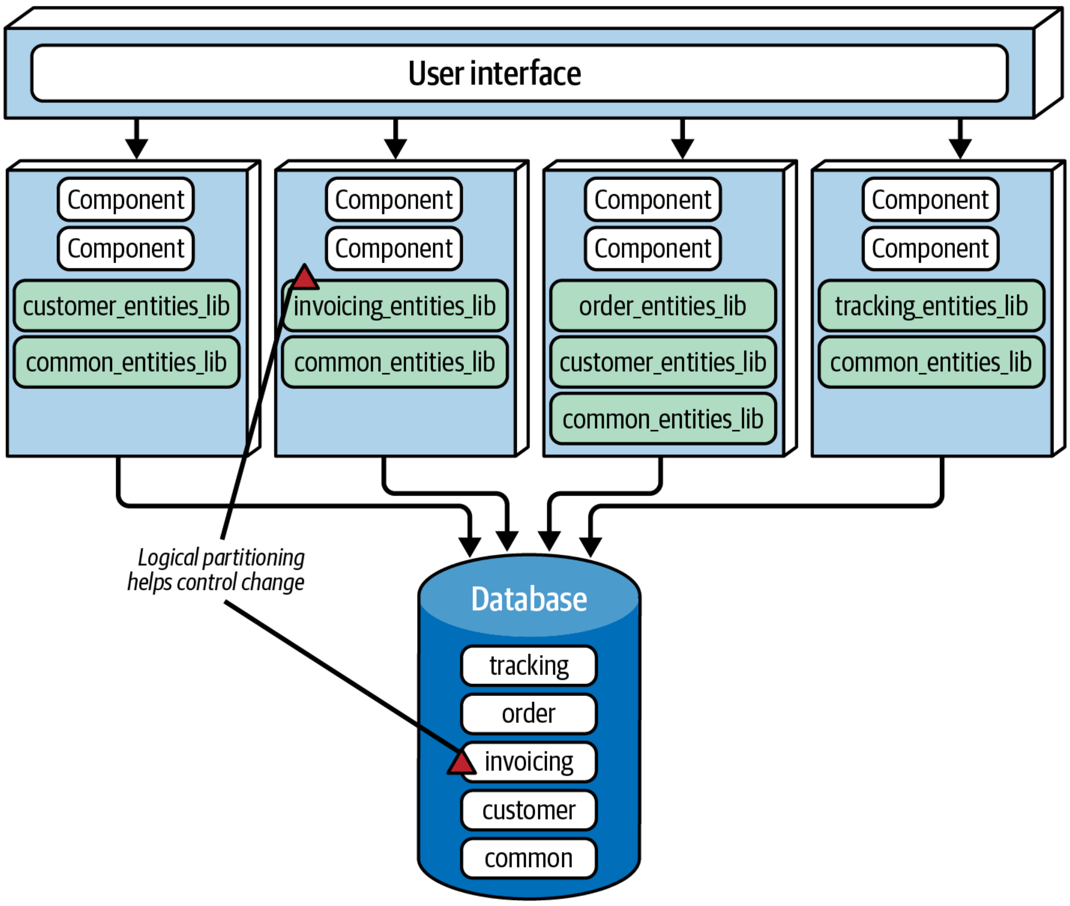


Figure 14-7. Using multiple shared libraries for database entity objects

In Figure 14-7, the database contains a `Common` domain and a corresponding `common_entities_lib` shared library, which is used by all services. This is relatively common. Since these tables are common to all services, changing them requires coordinating all services that access the shared database. One way to mitigate the potential rippling side effects of changes to these tables (and to their corresponding entity objects and domain services) is to lock the common entity objects in the version control system (if available) and allow only the database team to make changes. This helps control change and emphasizes the significance of making changes to the common tables used by all services.

Tip

To better control database changes within a service-based architecture, make the logical partitioning in the database as fine-grained as possible—while still maintaining well-defined data domains.

Cloud Considerations

Being a distributed architecture, the service-based style works well in cloud environments, even though its domain services are typically coarse-grained. Due to their large scope, domain services are typically implemented as containerized services rather than as serverless functions, and they can easily leverage cloud file storage, database, and messaging services.

Common Risks

While interservice communication is typical with microservices, it's something architects try to avoid in the service-based architectural style. Ideally, domains should be as independent as possible, with coupling focused only at the database level. Too much communication between domain services is a good indication that either the architect didn't partition the domains correctly or this isn't the right architectural style for the problem they are solving.

Another common risk is creating *too many* domain services. The practical upper limit is around 12. Any more than that is likely to start causing issues with testing, deployment, monitoring, and database connections and changes.

Governance

In addition to the common structural and operational architectural governance techniques discussed throughout this book, such as cyclomatic complexity, scalability, responsiveness, and so on, architects can apply specific governance tests to ensure the structural integrity of a service-based architecture.

Because domain services should be as independent as possible, the first aspect of governing this style is ensuring that changes don't span across multiple domain services. If they do, that's a good indication that the domain boundaries aren't appropriately defined, or that service-based is not the most appropriate architectural style for the problem.

When interservice communication is unavailable, architects can also govern the amount of communication between domain services. There are certainly situations and workflows where one domain service must communicate with another: for example, an `OrderProcessing` domain might need to communicate with a `CustomerNotification` domain to email order status information to the customer. For the most part, though, domain services should be largely independent from each other, with orchestration taking place at the UI or API Gateway level.

Team Topology Considerations

Since service-based architectures are domain partitioned, they work best when teams are also aligned by domain area (such as cross-functional teams with specialization). When a domain-based requirement comes along, a domain-focused cross-functional team can work together on that feature within a specific domain service, without interfering with other teams or services. Conversely, technically partitioned teams (such as UI teams, backend teams, database teams, and so on) do not work well with this architectural style because of its domain partitioning. Assigning domain-based requirements to a technically organized team requires a level of interteam communication and collaboration that proves difficult in most organizations.

Here are some considerations for architects with regard to aligning a service-based architecture with the specific team topologies outlined in "Team Topologies and Architecture":

Stream-aligned teams

If domain boundaries are properly aligned, stream-aligned teams work well with this architectural style, particularly when their streams are focused on a specific domain. However, service-based architecture becomes more challenging when streams cross the boundaries defined by the domain services. In this case, the architect should analyze the boundaries and granularity of the domain services and either realign them to the streams or choose a different architectural style.

Enabling teams

Service-based architecture is not as effective when paired with the enabling team topology as other distributed architectures are, due to the coarse-grained nature of its domain services. However, an architect can increase this style's modularity by carefully identifying and creating appropriate components within each domain service. Specialists and cross-cutting team members can then make suggestions and perform experiments based on those components.

Complicated-subsystem teams

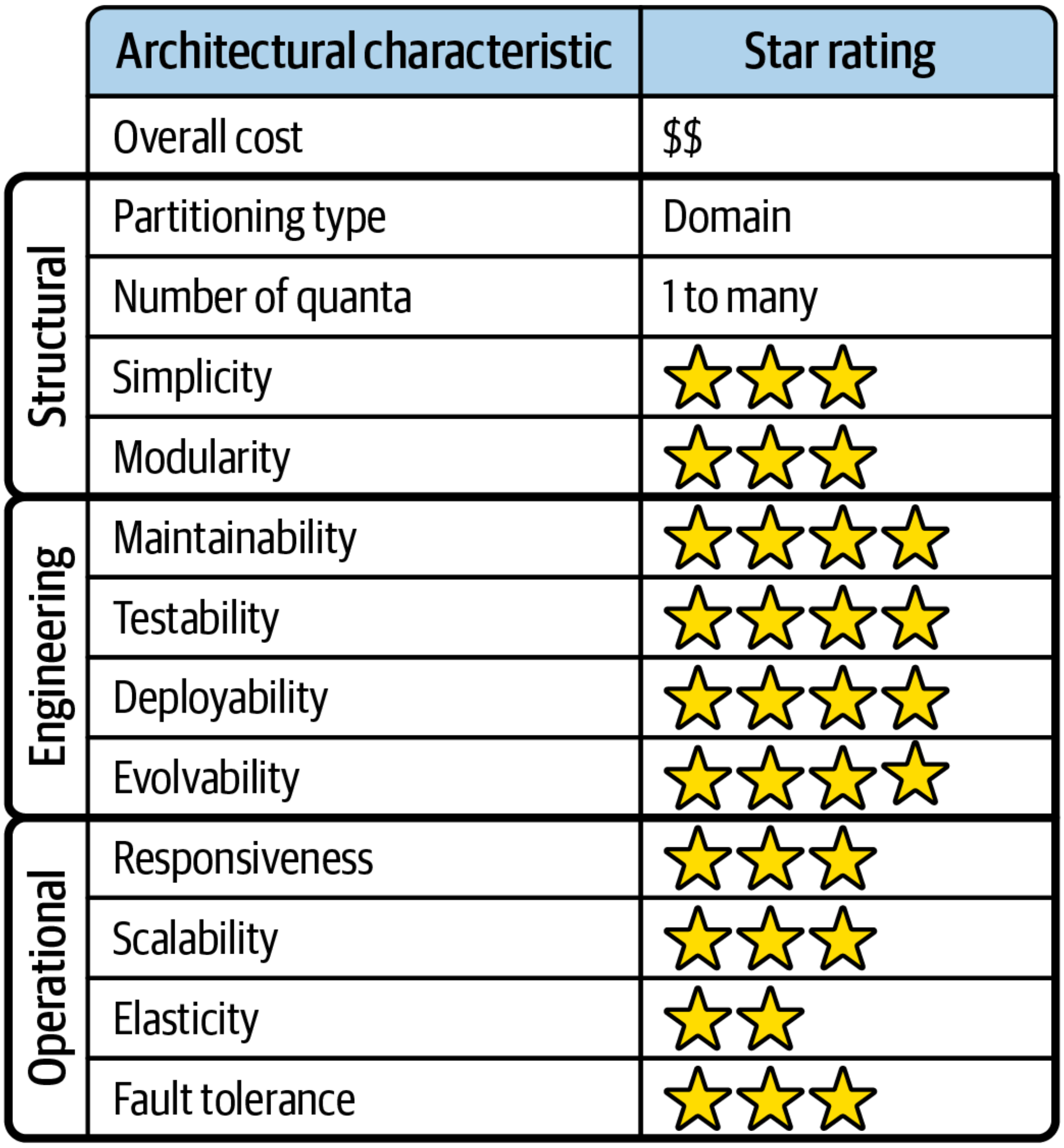
Complicated-subsystem teams can leverage this architecture style's domain-level and subdomain-level modularity to focus on complicated domain or subdomain processing, independent of other team members (and services).

Platform teams

Service-based architecture's high degree of modularity helps teams leverage the benefits of the platform team topology by utilizing common tools, services, APIs, and tasks.

Style Characteristics

A one-star rating in the characteristics ratings table in Figure 14-8 means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architectural style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.



	Architectural characteristic	Star rating
	Overall cost	$$
Structural	Partitioning type	Domain
	Number of quanta	1 to many
	Simplicity	★★★
	Modularity	★★★
Engineering	Maintainability	★★★★
	Testability	★★★★
	Deployability	★★★★
	Evolvability	★★★★
Operational	Responsiveness	★★★
	Scalability	★★★
	Elasticity	★★
	Fault tolerance	★★★

Figure 14-8. Service-based architecture characteristics ratings

Service-based architecture is a *domain-partitioned* architecture, meaning that its structure is driven by domains rather than technical considerations (such as presentation logic or persistence logic).

Consider the example of the Going Green electronics recycling application, which we introduced in Chapter 7 and revisited in Chapter 13. For the purposes of this chapter, let's imagine that Going Green uses a service-based architecture style. Each service, being a separately deployed unit of software, is scoped to a specific domain (such as item assessment). Changes made within this domain only impact that specific service and its corresponding UI and database. Nothing else needs to be modified to support a specific assessment change.

In a distributed architecture, the number of quanta can be greater than or equal to one. For example, if Going Green's services all share the same database or UI, the entire system would be only a single quantum. However, as discussed in "Style Specifics", both the UI and database can be federated (broken apart), resulting in multiple quanta within the overall system. In Figure 14-9, Going Green's system contains two quanta. One is for the customer-facing portion of the application and contains a separate customer UI, database, and set of services (`Quoting` and `Item Status`). The other deals with the internal operations of receiving, assessing, and recycling electronic devices. Even though the internal operations quantum contains separately deployed services and two separate UIs, they all share the same database, making the internal operations portion of the application a single quantum.

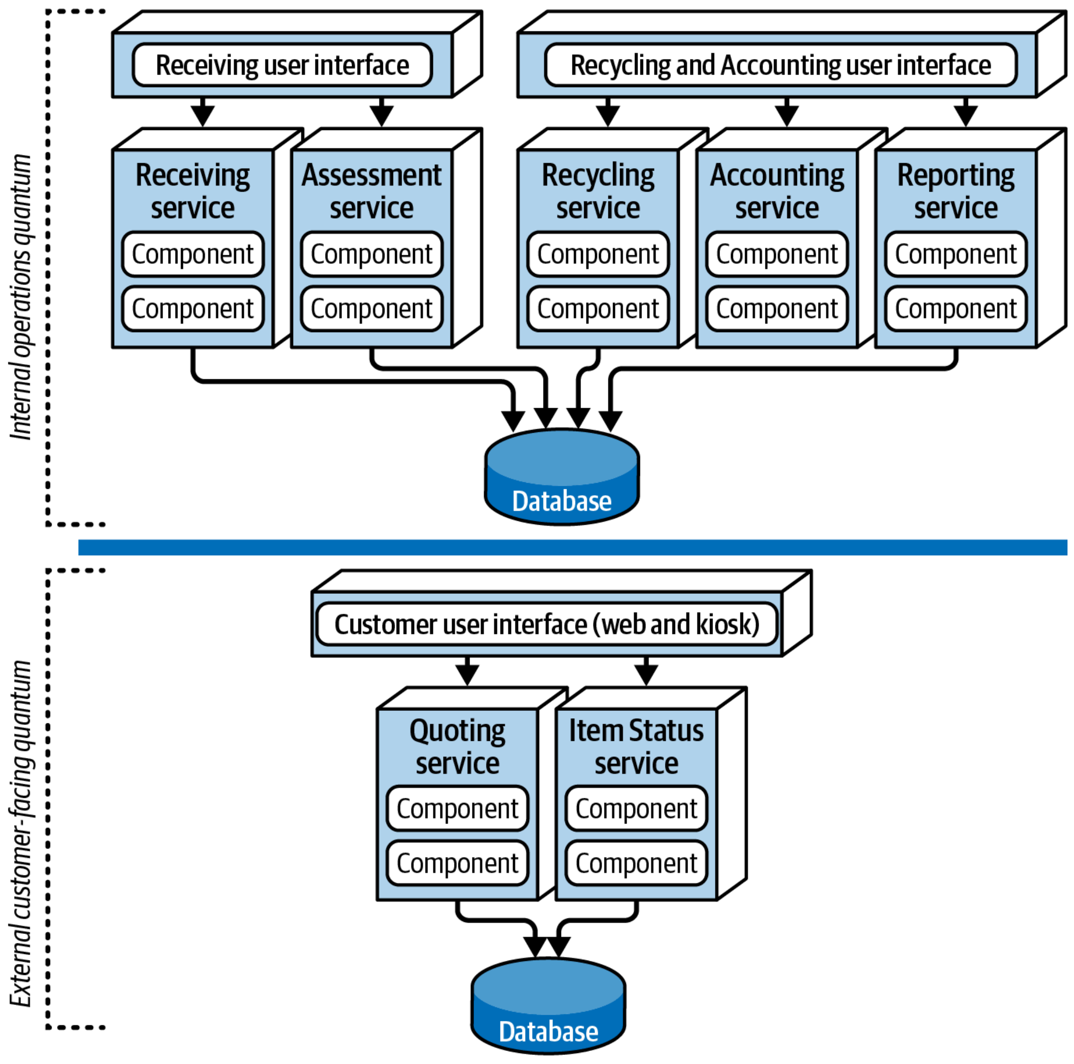


Figure 14-9. Separate quanta in a service-based architecture

Although we didn't give service-based architecture any five-star ratings, it nevertheless rates high (four stars) in many vital areas. Breaking an application apart into separately deployed domain services using this architectural style allows for faster change (agility), better test coverage due to modularity based on domain scoping (testability), and the ability to deploy more frequently with less risk than with a monolithic architecture (deployability). These three characteristics lead to better time to market, allowing organizations to deliver new features and fix bugs relatively quickly.

Fault tolerance and overall application availability also rate high for service-based architecture. Even though domain services tend to be coarse-grained, the four-star rating comes from the fact that services in this architectural style are usually self-contained and, thanks to code and database sharing, do not typically use interservice communication. As a result, if one domain service goes down (for instance, Going Green's `Receiving` service), it doesn't affect any of the other six services.

Scalability only rates three stars due to the coarse-grained nature of the services; correspondingly, elasticity rates only two stars. Although programmatic scalability and elasticity are certainly possible with this architectural style, it does replicate more functionality than architecture styles with finer-grained services (such as microservices), making it less cost-effective and less efficient in terms of machine resources. Typically, service-based architectures use only a single instance of each service, unless there is a need for better throughput or failover. A good example of this, illustrated in Figure 14-9, is Going Green—only its `Quoting` and `Item Status` services need to scale to support high customer volumes. The other operational services only require single instances, making it easier to support things like single in-memory caching and database-connection pooling.

Simplicity and overall cost are two other drivers that differentiate this architectural style from other, more expensive and complex distributed architectures, such as microservices, event-driven architecture, or even space-based architecture. This makes service-based one of the easiest and most cost-effective distributed architectures to implement. While that's an attractive proposition, there is, as always, a trade-off. The higher the cost and complexity, the better these four-star characteristics (such as scalability, elasticity, and fault tolerance) become.

Its flexibility, combined with its many three-star and four-star architecture characteristics, make service-based architecture one of the most pragmatic styles available. While there are much more powerful distributed architectural styles, many companies find that such power comes at too steep a price. Others find that they simply don't *need* that much power. It's like buying a Ferrari but only using it to commute to work in rush-hour traffic—sure, it looks cool, but what a waste of power, speed, and agility!

Service-based architecture is also a natural fit for domain-driven design. Because services are coarse-grained and domain scoped, each domain fits nicely into a separately deployed service encompassing that particular domain. Compartmentalizing that functionality into a single unit of software makes it easier to apply changes to that domain.

Maintaining and coordinating database transactions is always an issue with distributed architectures, which typically rely on *eventual consistency* (meaning that independent database updates will eventually be in sync with each other) rather than traditional *ACID transactions* (meaning database updates are coordinated and performed together in a single unit of work). Service-based architecture leverages ACID transactions better than any other distributed architecture style because its domain services are coarse-grained. This means the transaction scope is set to a particular domain service, allowing for the traditional commit-and-rollback transaction functionality found in most monolithic applications.

Finally, service-based architecture is a good choice for architects who want to achieve a good level of modularity without getting tangled up in the complexities of granularity and service coordination (see "Choreography and Orchestration" in Chapter 18).

Examples and Use Cases

To illustrate the flexibility and power of the service-based architectural style, we'll revisit our example of Going Green, a system used to recycle old electronic devices (such as an iPhone or Galaxy cell phone).

Going Green's processing flow works as follows:

1. The customer asks Going Green (via a website or kiosk) how much money it will pay for an old electronic device (*quoting*).

2. If satisfied with the quote, the customer sends the device to the recycling company (*receiving*).

3. Going Green assesses the device's condition (*assessment*).

4. If the device is in good working condition, Going Green pays the customer for the device (*accounting*). During this process, the customer can go to the website at any time to check on the status of the item (*item status*).

5. Based on the assessment, Going Green either safely destroys the device and recycles its parts, or resells it on a third-party sales platform, such as Facebook Marketplace or eBay (*recycling*).

6. Going Green periodically runs financial and operational reports on its recycling activity (*reporting*).

Figure 14-10 illustrates this system as implemented using a service-based architecture. Each of the domain areas we just identified is implemented as a separately deployed, independent domain service. The services that need to scale (and hence would require multiple service instances) are those needing higher throughput (in this case, the customer-facing `Quoting` and `ItemStatus` services). Since the other services don't need to scale, they only require a single service instance.

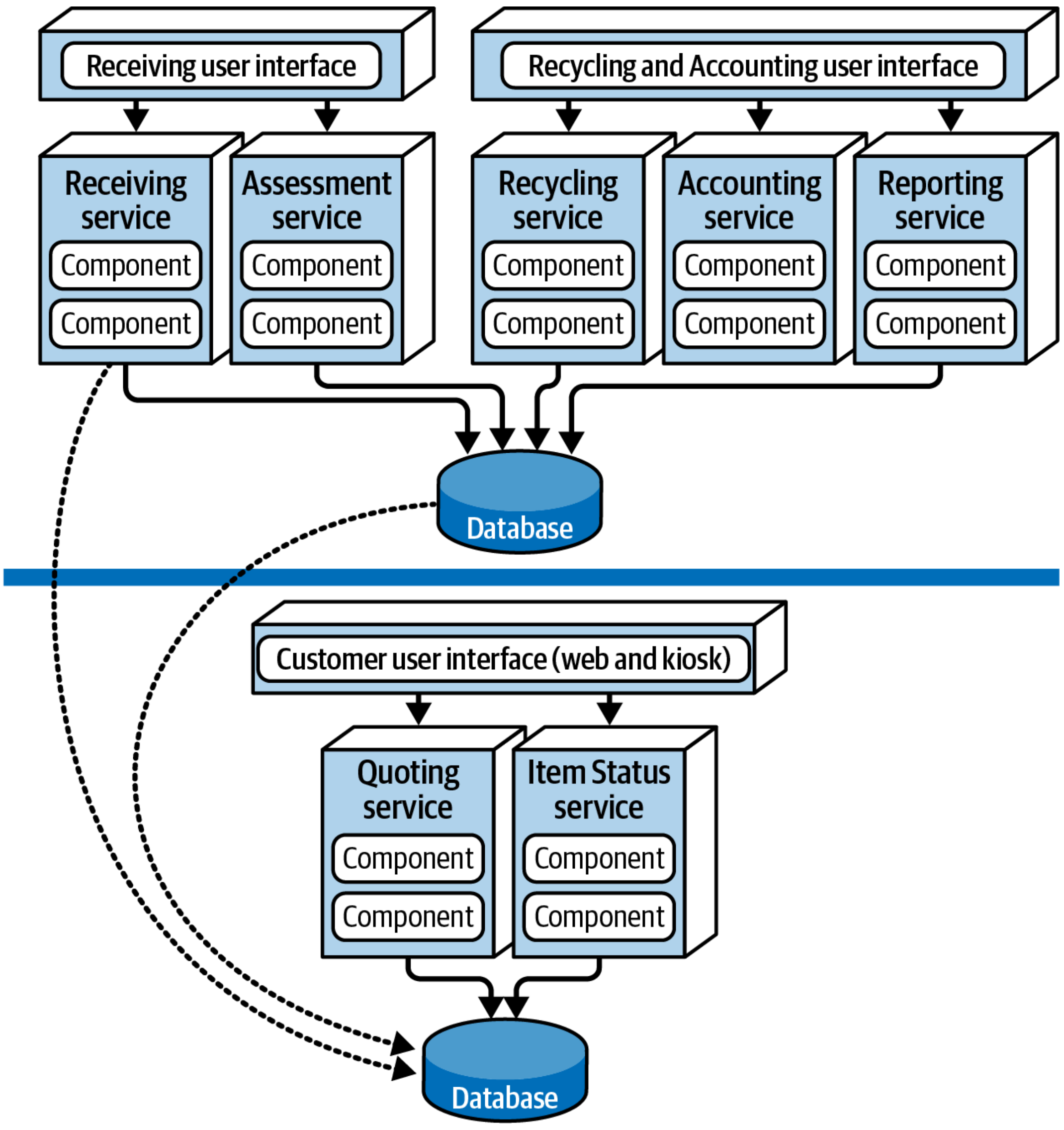


Figure 14-10. Going Green's electronics recycling application using service-based architecture

In this example, the UI applications are separated into domains: *Customer Facing*, *Receiving*, and *Recycling and Accounting*. This separation provides good fault tolerance at the UI level, good scalability, and appropriate security (since external customers have no network path to access internal functionality). Notice, too, that there are two separate physical databases: one for external customer-facing operations, and one for internal operations. This arrangement allows internal data and operations to reside in a separate network zone from the external operations (denoted by the horizontal line). It also provides much better security-access restrictions and data protection—and constitutes a separate architectural quantum). One-way access through the firewall allows internal services to access and update customer-facing information, but not vice versa. Alternatively, depending on the database, teams could also use internal table mirroring and external table synchronization to synchronize the data between the two databases.

Additionally, the `Assessment` service changes constantly, as new products are received or come on the market. With a service-based architecture, these frequent changes are isolated to a single domain service, providing agility, testability, and deployability.

Service-based architecture is a very flexible architectural style that offers domain partitioning and good levels of scalability, agility, fault tolerance, availability, and responsiveness at a fairly low price, compared to other distributed architectures. These factors make it a popular choice.

Service-based architectures also make good "stepping stone" migration targets for other distributed architectures, whether the organization is migrating to another distributed architecture style or creating a new distributed system from scratch. This drives us to our main point:

> Not every portion of an application needs to be microservices.
>
> Mark Richards

Initially moving to or creating a service-based architecture as a "stepping stone" *before* migrating to the target architectural style allows teams to analyze the domains and decide which portions of the architecture *should* be microservices. For example, in the Going Green example, the `Recycling` and `Accounting` services don't need to be broken down any further and should probably remain domain services. However, the `Assessment` service changes frequently and requires high levels of agility, so *that* service should be broken down into separate services, one for each type of electronic device. If the Going Green team skipped this step and moved *straight* to a microservices architecture, *every* piece of functionality would likely end up as a microservice, even if it didn't need to be one.

These are a few of the many reasons that service-based architectures are a favorite among architects. However, it's only one of many distributed architectural styles, and understanding all of them is helpful for determining the right fit for a particular business problem. To that end, let's look at some other distributed architectures.