7 Software Architecture Patterns

Software architecture patterns are one of the most useful tools that can be leveraged for designing a software architecture. Some of the design issues we face as software architects already have proven solutions. Experienced software architects are knowledgeable about available architecture patterns and can recognize when one can be applied to a given design scenario.

This chapter begins by explaining what software architecture patterns are and how they can be used. It then goes into detail about some commonly used architecture patterns, including **layered architecture**, **event-driven architecture** (**EDA**), **Model-View-Controller** (**MVC**), **Model-View-Presenter** (**MVP**), **Model-View-ViewModel** (**MVVM**), **Command Query Responsibility Segregation** (**CQRS**), and **service-oriented architecture** (**SOA**).

In this chapter, we will cover the following topics:

- Software architecture patterns
- Layered architecture
- Event-driven architecture , including event notifications, event-carried state transfer, and event-sourcing
- Model-View-Controller pattern
- Model-View-Presenter pattern
- Model-View-ViewModel pattern
- Command Query Responsibility Segregation
- Service-oriented architecture

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Software architecture patterns

A **software architecture pattern** is a solution to a recurring problem that is well understood, in a particular context. Each pattern consists of a context, a problem, and a solution. The problem may be to overcome some challenge, take advantage of some opportunity, or to satisfy one or more quality attributes. Patterns codify knowledge and experience into a solution that we can reuse.

Using patterns simplifies design and allows us to gain the benefits of using a solution that is proven to solve a particular design problem. When working with others who are familiar with patterns, referencing one of them provides a shorthand with which to reference a solution, without having to explain all its details. As a result, they are useful during discussions to communicate ideas.

Software architecture patterns are similar to design patterns, except that they are broader in scope and are applied at the architecture level. Architecture patterns tend to be more coarse-grained and focus on architectural problems, while design patterns are more fine-grained and solve problems that occur during implementation.

A software architecture pattern provides a high-level structure and behavior for software systems. It is a grouping of design decisions that have been repeated and used successfully for a given context. They address and satisfy architectural drivers and as a result, the ones that we decide to use can really shape the characteristics and behavior of the architecture. Each pattern has its own characteristics, strengths, and weaknesses.

Software architecture patterns provide the structure and main components of the software system being built. They introduce design constraints, which reduce complexity and help to prevent incorrect decisions. When a software architecture pattern is followed consistently during design, we can anticipate the properties that the software system will exhibit. This allows us to consider whether a design will satisfy the requirements and quality attributes of the system.

Using software architecture patterns

Much like design patterns, software architecture patterns come into being after they are successfully repeated in practice. As a software architect, you will discover and use an existing pattern when appropriate.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

The majority of the time, you will not be inventing or creating a new pattern. It is possible that you may find yourself tackling challenges and problems in a novel domain, requiring you to create truly new solutions that do not currently exist. However, even in that case the solution will not be a pattern yet. Only after it is repeated in practice and becomes known as a solution for a particular context and problem does it become a pattern.

Software architecture patterns can be applied to the entire software system or to one of the subsystems. Consequently, more than one software architecture pattern can be used in a single software system. These patterns can be combined to solve problems.

Overusing architecture patterns

While leveraging architecture patterns is a valuable tool, don't force the use of a particular pattern. A common mistake with design and architecture patterns is to use one even if it is not appropriate to do so.

A developer or architect may become aware of a particular pattern and then become overzealous in their desire to use it. They may apply a pattern just due to their familiarity with it.

The key is to gain knowledge of the patterns that are available and an understanding as to the scenarios in which they should be applied. This knowledge allows a software architect to select and use the appropriate pattern. A software architecture pattern should only be used if it is the best solution for a given design issue and context.

Understanding the difference between architecture styles and architecture patterns

You may come across the term **architecture style** and compare its meaning with the term **architecture pattern**. For the most part, these two terms are used interchangeably.

However, some people make a distinction between the two, so let's take a moment to explain the difference. It is not completely clear-cut, as the definitions vary depending on who you ask. One of the ways an architecture style is defined is that it is a set of elements, and the vocabulary to be used for those elements, that is available to be used in an architecture. It constrains an architecture design by restricting the available design choices. When a software system adheres to a particular architecture style, it will be expected to exhibit certain properties.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

For example, if we were to follow the microservice architecture style for an application, which comes with the constraint that each service should be independent of the others, we can expect such a system to have certain properties. A system that follows a microservice architecture style will be able to deploy services independently, isolate faults for a particular service, and use a technology of the team's choosing for a given service.

A software architecture pattern is a particular arrangement of the available elements into a solution for a recurring problem given a certain context. Given a particular architecture style, we can use the vocabulary of that style to express how we want to use the elements available in that style in a certain way. When this arrangement is a known solution to a common, recurring problem in a particular context, it is a software architecture pattern.

This book does not focus on making a distinction between these two terms and, for the most part, uses the term **software architecture pattern**, as evidenced by the title of this chapter.

Now that we know what software architecture patterns (and styles) are, let's explore some of the commonly used ones, beginning with the layered architecture.

Layered architecture

When partitioning a complicated software system, layering is one of the most common techniques. In a **layered architecture**, the software application is divided into various horizontal layers, with each layer located on top of a lower layer. Each layer is dependent on one or more layers below it (depending on whether the layers are open or closed), but is independent of the layers above it.

Open versus closed layers

Layered architectures can have layers that are designed to be open or closed. With a closed layer, requests that are flowing down the stack from the layer above must go through it and cannot bypass it. For example, in a three-layer architecture with presentation, business, and data layers, if the business layer is closed, the presentation layer must send all requests to the business layer and cannot bypass it to send a request directly to the data layer.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Closed layers provide layers of isolation, which makes code easier to change, write, and understand. This makes the layers independent of each other, such that changes made to one layer of the application will not affect components in the other layers. If the layers are open, this increases complexity. Maintainability is lowered because multiple layers can now call into another layer, increasing the number of dependencies and making changes more difficult.

However, there may be situations in which it is advantageous to have an open layer. One of them is to solve a common problem with the layered architecture, in which unnecessary traffic can result when each layer must be passed even if one or more of them is just passing requests on to the next layer.

In our example of a three-layer architecture with presentation, business, and data layers, let's say that we introduce a shared services layer between the business and data layers. This shared services layer may contain reusable components needed by multiple components in the business layer. We may choose to place it below the business layer so that only the business layer has access to it. However, now all requests from the business layer to the data layer must go through the shared services layer even though nothing is needed from that layer. If we make the shared services layer open, requests to the data layer can be made directly from the business layer.

The important point for software architects to understand when designing a layered architecture is that there are advantages to closed layers and achieving layers of isolation. However, experienced software architects understand when it might be appropriate to open a layer. It is not necessary to make all of the layers open or closed. You may selectively choose which layers, if any, are open.

Tiers versus layers

You may have heard the terms **tier** and **layer** in reference to layered architectures. Before we proceed with discussing layered architectures, these terms should be clarified.

Layers are logical separations of a software application and tiers are physical ones.

When partitioning application logic, layers are a way to organize functionality and components. For example, in a three-layered architecture, the logic may be separated into presentation, business, and data layers. When a software architecture is organized into more than one layer, it is known as a **multi-layer architecture**. Different layers do not necessarily have to be located on different physical machines. It is possible to have multiple layers on the same machine.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Tiers concern themselves with the physical location of the functionality and components. A three-tiered architecture with presentation, business, and data tiers implies that those three tiers have been physically deployed to three separate machines and are each running on those separate machines. When a software architecture is partitioned into multiple tiers, it is known as a **multi-tier architecture**.

Keep in mind that some people use the two terms interchangeably. When communicating with others, if the distinction is important, you may want to be precise in your language and you may need to confirm with the other person what they mean when they use one of the terms.

Advantages of layered architectures

There are some key benefits to using a layered architecture. This pattern reduces complexity by achieving a **Separation of Concerns** (**SoC**). Each layer is independent and you can understand it on its own without the other layers. Complexity can be abstracted away in a layered application, allowing us to deal with more complex problems.

Dependencies between layers can be minimized in a layered architecture, which further reduces complexity. For example, the presentation layer does not need to depend directly on the data layer and the business layer does not depend on the presentation layer. Minimizing dependencies also allows you to substitute implementations for a particular layer without affecting the other layers.

Another advantage of layered architectures is the fact that they can make development easier. The pattern is pervasive and well known to many developers, which makes using it easy for the development team. Due to the way that the architecture separates the application logic, it matches up well with how many organizations hire their resources and allocate tasks during a project. Each layer requires a particular skill set and suitable resources can be assigned to work on each layer. For example, UI developers for the presentation layer, and backend developers for the business and data layers.

This architecture pattern increases the testability quality attribute of software applications. Partitioning the application into layers and using interfaces for the interaction between layers allows us to isolate a layer for testing and either mock or stub the other layers. For example, you can perform unit testing on classes in your business layer without the presentation and data layers. The business layer is not dependent on the presentation layer and the data layer can be mocked or stubbed.

Applications using a layered architecture may have higher levels of reusability if more than one application can reuse the same layer. For example, if multiple applications target the same business and/or data layers, those layers are reusable.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

When an application using a layered architecture is deployed to different tiers, there are additional benefits:

- There is increased scalability as more hardware can be added to each tier, providing the ability to handle increased workloads.
- A multi-tier application can experience greater levels of availability when multiple machines are used per layer. Uptime is increased because, if a hardware failure takes place in a layer, other machines can take over.
- Having separate tiers enhances security as firewalls can be placed in between the various layers.
- If a layer can be reused for multiple applications, it means that the physical tier can be reused as well.

Disadvantages of layered architectures

Although layered architectures are commonly used, and for good reasons, there are disadvantages to using them. Although the layers can be designed to be independent, a requirement change may require changes in multiple layers. This type of coupling lowers the overall agility of the software application.

For example, adding a new field will require changes to multiple layers: the presentation layer so that it can be displayed, the business layer so that it can be validated/saved/processed, and the data layer because it will need to be added to the database. This can complicate deployment because, even for a change such as this, an application may require multiple parts (or even the entire application) to be deployed.

Another minor disadvantage is the fact that more code will be necessary for layered applications. This is to provide the interfaces and other logic that are necessary for the communication between the multiple layers.

Development teams have to be diligent about placing code in the correct layer so as not to leak logic to a layer that belongs in another layer. Examples of this include placing business logic in the presentation layer or putting data-access logic in the business layer.

Although applications with good performance can be designed with layered architectures, if you are designing a high-performance application, you should be aware that there can be inefficiencies in having a request go through multiple layers. In addition, moving from one layer to another sometimes requires data representations to be transformed. One way to mitigate this disadvantage is to allow some layers to be open but this should only be done if it is appropriate to open a layer.

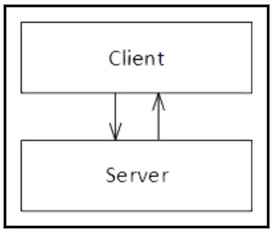
Copyright © 2018. Packt Publishing, Limited. All rights reserved.

There are some additional disadvantages to layered architectures when they are deployed to multiple tiers:

- The performance disadvantage of layered architectures has already been mentioned, but when those layers are deployed to separate physical tiers, as is common, there is an additional performance cost. With modern hardware, this cost may be small but it still won't be faster than an application that runs on a single machine.
- There is a greater monetary cost associated with having a multi-tier architecture. The more machines are used for the application, the greater the overall cost.
- Unless the hosting of the software application is handled by a cloud provider or has otherwise been outsourced, an internal team will be needed to manage the physical hardware of a multi-tier application.

Client-server architecture (two-tier architecture)

Layered architectures became very prevalent with the popularity of client-server software systems. In a distributed application that uses a **client-server architecture**, also known as a two-tier architecture, clients and servers communicate with each other directly. A client requests some resource or calls some service provided by a server and the server responds to the requests of clients. There can be multiple clients connected to a single server:



The **Client** part of the application contains the user interface code and the **Server** contains the database, which traditionally has been a **relational database management system** (**RDBMS**). The majority of application logic in a client-server architecture is located on the server, but some of it could also be located in the client. The application logic located on the server might exist in software components, in the database, or both.

When the client contains a significant portion of the logic and is handling a large share of the workload, it is known as a thick, or fat, client. When the server is doing that instead, the client is known as a thin client.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

In some client-server applications, the business logic is spread out between the client and the server. If consistency isn't applied, it can make it difficult to always know where a particular piece of logic is located.

If a team isn't diligent, business logic might be duplicated on the client and the server, violating the DRY principle.

There may be instances in which the same piece of logic is needed on both the client and the server. For example, there may be business logic needed by the user interface to validate a piece of data prior to submitting the data to the server. The server may need this same business logic because it also needs to perform this validation. While centralizing this logic may require additional communication between the client and the server, the alternative (duplication) lowers maintainability. If the business logic were to change, it would have to be modified in multiple places.

Using stored procedures for application logic

If application logic does exist in the database, it is commonly found in stored procedures. A stored procedure is a grouping of one or more **Structured Query Language** (**SQL**) statements that forms a logical unit to accomplish some task. It can be used to do a combination of retrieving, inserting, updating, and deleting data.

It used to be popular to use stored procedures with client-server applications because their use reduced the amount of network traffic between the client and the server. Stored procedures can contain any number of statements within them and can call other stored procedures. A single call from the client to the server is all that is needed to execute a stored procedure. If that logic was not encapsulated inside a stored procedure, multiple calls would need to be made over the network between the client and the server to execute the same logic.

Stored procedures are compiled the first time they are executed, at which time an execution plan is created that a database's query engine can use to optimize its use on subsequent calls. In addition to some performance benefits, there are security advantages to using stored procedures. Users and applications do not need to be granted permissions to the underlying database objects that a stored procedure uses, such as database tables. A user or application can execute a stored procedure but it is the stored procedure that has control over what logic is executed and which database objects are used. Stored procedures increase reusability in that once one is written and compiled, it can be reused in multiple places.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Although there are benefits to using stored procedures, there are drawbacks. There are limited coding constructs, as compared with high-level programming languages, that are available for use in application logic. Having some of your business logic located in stored procedures also means that your business logic isn't centralized.

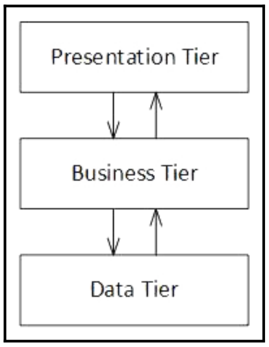
In modern application development, application logic should not be placed in stored procedures. It belongs outside the data layer, independent and decoupled from the mechanism used to store the data. Based on that point, some have relegated stored procedures to simple CRUD operations. However, if that is the case, stored procedures are not providing much of a benefit.

Although application logic should not be placed in stored procedures, they can still be of use in some situations. For complex queries (for example, SQL queries with complex table joins and WHERE clauses) and queries requiring multiple statements and large amounts of data, the performance advantages of stored procedures can be useful.

N-tier architecture

With an **n-tier architecture**, also known as a **multitier architecture**, there are multiple tiers in the architecture. One of the most widely-used variations of this type of layered architecture is the three-tier architecture. The rise of the web coincided with a shift from two-tier (client-server) architectures to three-tier architectures. This change was not a coincidence. With web applications and the use of web browsers, rich client applications containing business logic were not ideal.

The three-tier architecture separates logic into presentation, business, and data layers:



Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Presentation tier

The **presentation tier** provides functionality for the application's UI. It should provide an appealing visual design as it is the part of the application that users interact with and see. Data is presented to the user and input is received from users in this tier. Aspects of the usability quality attribute, which were discussed in `Chapter 4`, *Software Quality Attributes*, should be the concern of the presentation tier.

This tier should contain logic to render the user interface, including the placement of data in the appropriate UI components, formatting the data that is presented appropriately, and hiding/showing UI components as required.

It should also provide some basic validation to help users avoid or minimize mistakes, such as ensuring the correct type of data is being entered for a given control and that the data is in the correct format. Developers should be careful not to introduce business logic into the validation, which should be handled by the business tier.

The presentation tier should provide users with useful feedback, such as friendly and informative messages, tooltips, visual feedback, such as a progress bar for long-running processes, and notifications to inform users about the completion or failure of asynchronous operations.

Software architects should strive to design *thin clients* that minimize the amount of logic that exists in the presentation tier. The logic in the presentation tier should focus on user interface concerns. A presentation tier devoid of business logic will be easier to test.

Business tier

The **business tier**, which is sometimes referred to as the application tier, provides the implementation for the business logic of the application, including such things as business rules, validations, and calculation logic. Business entities for the application's domain are placed in this tier.

The business tier coordinates the application and executes logic. It can perform detailed processes and makes logical decisions. The business tier is the center of the application and serves as an intermediary between the presentation and data tiers. It provides the presentation tier with services, commands, and data that it can use, and it interacts with the data tier to retrieve and manipulate data.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Data tier

The **data tier** provides functionality to access and manage data. The data tier contains a data store for persistent storage, such as an RDBMS. It provides services and data for the business tier.

There are variations of n-tier architectures that go beyond just three tiers. For example, in some systems, there is a data access or persistence layer in addition to a data or database layer. The persistence layer contains components for data access, such as an **object-relational mapping** (**ORM**) tool, and the database layer contains the actual data store, such as an RDBMS. One reason to separate these into two distinct layers is if you wanted the ability to switch out your data access or database technology for a different one.

Event-driven architecture

An **event** is the occurrence of something deemed significant in a software application, such as a state change, that may be of interest to other applications or other components within the same application. An example of an event is the placement of a purchase order or the posting of a letter grade for a course that a student is taking.

An **event-driven architecture** (**EDA**) is a distributed, asynchronous software architecture pattern that integrates applications and components through the production and handling of events. By tracking events, we don't miss anything of significance related to the business domain.

EDAs are loosely coupled. The producer of an event does not have any knowledge regarding the event subscribers or what actions may take place as a result of the event.

SOA can complement EDA because service operations can be called based on events being triggered. The converse can also be designed, such that service operations raise events.

EDAs can be relatively complex given their inherent asynchronous, distributed processing. As with any distributed architecture, issues may occur due to a lack of responsiveness, performance issues, or failures with event mediators and event brokers (these components will be described shortly).

Event channels

Before we cover the two main event topologies for EDAs, let's go over the concept of event channels because both topologies make use of them.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Event messages contain data about an event and are created by event producers. These event messages use **event channels**, which are streams of event messages, to travel to an event processor.

Event channels are typically implemented as message queues, which use the point-to-point channel pattern, or message topics, which use the publish-subscribe pattern.

Message queues

Message queues ensure that there is one, and only one, receiver for a message. In the context of event channels, this means that only one event processor will receive an event from an event channel. The point-to-point channel pattern is utilized for message queue implementations.

The point-to-point channel pattern

The point-to-point channel pattern is a messaging pattern used when we want to ensure that there will be exactly one receiver for a given message. If a channel has multiple receivers so that more than one message can be consumed concurrently, and more than one receiver attempts to consume a message, the event channel will ensure that only one of them succeeds. Since the event channel is handling that, it removes any need for coordination between the event processors.

Message topics

Message topics allow multiple event consumers to receive an event message. The publish-subscribe pattern is used for the implementation of message topics.

The publish-subscribe pattern

The publish-subscribe pattern, which is sometimes referred to as pub/sub for short, is a messaging pattern that provides a way for a sender (publisher) to broadcast a message to interested parties (subscribers).

Rather than publishers sending messages directly to specific receivers as in the point-to-point channel pattern, the messages can be sent without any knowledge of the subscribers or even if there are no subscribers. Similarly, it allows subscribers to show interest in a particular message without any knowledge of the publishers or even if there are no publishers.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

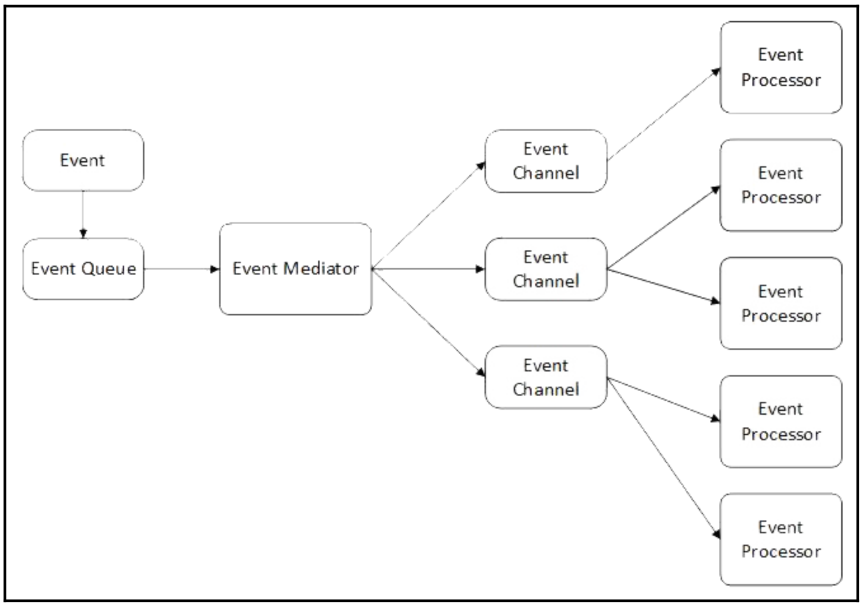
Event-driven architecture topologies

The two main topologies for EDAs are the mediator and broker topologies.

The mediator topology

A **mediator topology** for EDAs uses a single event queue and an event mediator to route events to the relevant event processors. This topology is commonly used when multiple steps are required to process an event.

With the mediator topology, event producers send events into an **event queue**. There can be many event queues in an EDA. Event queues are responsible for sending event messages on to the **event mediator**. All of these events, referred to as *initial events*, go through an event mediator. The event mediator then performs any necessary orchestration:



After the event mediator orchestrates each event from the event queue, it creates one or more asynchronous *processing events* based on the orchestration. These processing events get sent out to an **event channel**, which can either be a message queue or a message topic.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Message topics are more commonly used with the mediator topology due to the orchestration involved, which allows multiple event processors, which will perform different tasks, to receive an event message. This can be seen in the preceding diagram where some of the event channels are sending an event message to multiple event processors.

Event processors listen in on event channels to pick up events and process them in line with their design. In the *Event processing styles* section later in this chapter, we will cover the different processing styles typically used by event processors.

Event mediator implementations

An event mediator can be implemented in a number of different ways. For simple orchestrations, an integration hub can be used. These typically allow you to define mediation rules using a **domain-specific language** (**DSL**) for the routing of events. Domain-specific languages, unlike a general-purpose language, such as C#, Java, or UML, allow expressions to be written for a particular domain.

For a more complex orchestration of events, **Business Process Execution Language** (**BPEL**) can be used in conjunction with a BPEL engine. BPEL is an XML-based language that is used to define business processes and their behavior. It is frequently used with SOA and web services.

Large software applications with complex orchestration needs, which may even include human interactions, may opt to implement an event mediator that uses a **business process manager** (**BPM**). Business process management involves modeling, automating, and executing business workflows. Some business process managers use **Business Process Model and Notation** (**BPMN**) to define business processes. BPMN allows for business process modeling using a graphical notation. Business process diagrams created with BPMN are similar to activity diagrams in UML.

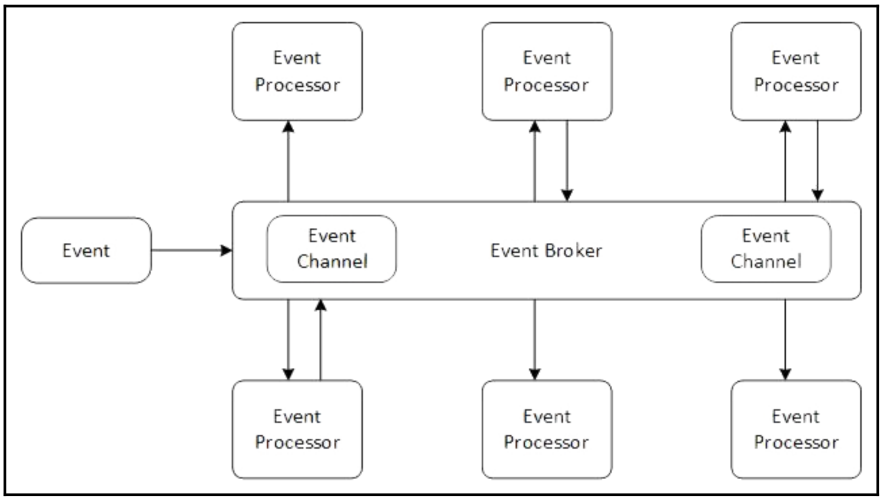
Software architects must understand the needs of the software application in order to select an appropriate event mediator implementation.

The broker topology

In a **broker topology**, the event messages created by event producers enter an event broker, sometimes referred to as an event bus. The event broker contains all of the event channels used for the event flow. The event channels may be message queues, message topics, or some combination of the two.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Unlike the mediator topology, there is no event queue with the broker topology. The event processors are responsible for picking up events from an event broker:



This topology is ideal when the processing flow is fairly simple and there is no need for centralized event orchestration. **Events** flow to the **event processor** from the **event broker** and, as part of the processing, new events may be created.

This can be seen in the preceding diagram, where in some cases, events are flowing from the event processors back to the event broker. A key to the broker topology is chaining events in order to execute a particular business task.

Event processing styles

Event processors are components that have a specific task and contain logic to analyze and take action on events. Each event processor should be independent and loosely coupled with other event processors.

Once event messages reach event processors, there are three prevalent styles for processing events: **simple event processing** (**SEP**), **event stream processing** (**ESP**), and **complex event processing** (**CEP**). The type of event processing needed by a particular application depends on the processing complexity that is required. EDAs may utilize a combination of these three styles.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Simple event processing (SEP)

In SEP, notable events are immediately routed in order to initiate some type of downstream action. This type of processing can be used for real-time workflows to reduce the amount of time between the event taking place and the resulting action.

Simple event processing also includes functionality such as transforming event schemas from one form to another, generating multiple events based on the payload of a single event, and supplementing an event payload with additional data.

Event stream processing (ESP)

ESP involves analyzing streams of event data and then taking any necessary action based on that analysis. Events are screened based on various conditions and then either an action may be taken for the event or the event may be ignored. This type of processing is ideal for real-time flows in which decisions are also involved.

An example is a stock trading system in which an event takes place and enters an event stream when a stock ticker reports a change in price. Based on the price, an algorithm determines whether a buy or sell order should be created, and notifies the appropriate subscribers, if necessary.

Complex event processing (CEP)

In CEP, analysis is performed to find patterns in events to determine whether a more complex event has occurred. A complex event is an event that summarizes or represents a set of other events. Events may be correlated over multiple dimensions, such as causal, temporal, or spatial.

CEP can take place over a longer period of time as compared to the other types of event processing. This type of event processing might be used to detect business threats, opportunities, or other anomalies.

An example of functionality that uses CEP is a credit card fraud engine. Each transaction on a credit card is an event and the system will look at a grouping of those events for a particular credit card to try to find a pattern that might indicate fraud has taken place. If processing fraud is detected, downstream action is initiated.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Types of event-driven functionality

EDA can mean different things. There are three main types of functionality that can typically be found in systems that have an EDA: event notification, event-carried state transfer, and event sourcing. Event-driven software systems can provide a combination of the three.

Event notification

An architecture that provides **event notification** is one in which the software system sends a message when an event takes place. This functionality is the most common in software systems that have an EDA. The mediator and broker topologies allow us to implement event notifications.

There is a loose coupling between the event producer and any event consumers as well as between the logic that sends event messages and logic that responds to the events. This loose coupling allows us to change the logic in one without affecting the other. Event processor components are single-purpose and independent of other event processors, allowing them to be modified without affecting others.

The drawback to the loose coupling between event producers and event consumers is that it can be difficult to see the logical flow of event notifications. This added complexity also makes it more difficult to debug and maintain. There aren't specific statements you can look at to see what logic will be executed. A variety of event consumers, including ones in software systems other than the one that produced the event notification, may react to an event. Sometimes the only way to understand the logical flow is to monitor your systems to see the flow of event messages.

Event-carried state transfer

Event-carried state transfer is a variation on event notification. Its use is not as common as regular event notification. When an event consumer receives an event notification, it may need more information from the event producer in order to take the action that they want to take.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

For example, a sales system may send a new order event notification and a shipping system may subscribe to this type of event messages. However, in order to take appropriate action, the shipping system now needs additional information about the order, such as the quantity and type of line items that are in the order. This requires the shipping system to query the sales system in some way, such as through an API, for this information.

While the event publisher may not need to know anything about their subscribers, the subscriber is coupled to the producer in the sense that it needs to be aware of the producer and have a way to get more information from the producer.

Callbacks to the system that produced an event notification for more data in order to handle an event increase network load and traffic. One way to resolve this is to add state information to the events so that they contain enough information to be useful for potential consumers. For example, an event notification for a new order could contain the line item details needed by the shipping system so that no callback is required. The shipping system can keep its own copy of only the order details that it needs.

Although more data is being passed around, we gain a greater level of availability and resilience. The shipping system can function, at least with orders it has already received, even if the order system is temporarily unavailable. The shipping system does not need to call back to the order system after the initial event notification is received, which can be particularly beneficial if contacting and receiving data from the order system is slow.

However, with greater availability comes lower consistency. The replication of some data between the order and shipping systems lowers the consistency of the data.

Event-sourcing

A system can use a data store to read and update the application's current state, but what if there are requirements to know the details of the state changes that got us to the current point? With **event-sourcing**, the events that take place in a system, such as state changes, are persisted in an event store. Having a complete record of all the events that took place allows it to serve as a source of truth. Replaying events from an event log can be used to recreate an application's state.

Event-sourcing works in a similar way to a transaction log in a database system. The transaction log records all of the modifications that have been made to a database. This allows for rollbacks of transactions and also allows us to recreate the system state up to a particular point, such as right before a failure occurred.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Events should be immutable as they represent something that has already taken place. Actions may take place downstream as the result of an event, so if an event could be changed after the fact, it could put your system in an inconsistent state. If an update or cancellation of an event is necessary, a compensating event should be created. Compensating logic can be executed based on such events and can apply the necessary business rules to apply counter-operations. This will ensure that the event store is still a source of truth and that we can replay all of the events to recreate an application's state.

The benefits of event-sourcing include the fact that it can aid in debugging a system. It provides the ability to take events and run them through the system to see how the system will behave. This can be used to determine the cause of a problem. Event-sourcing also provides detailed auditing. The complete record of events allows us to see what happened, how it happened, when it happened, and other details.

Although event-sourcing can be very useful, it does introduce some added complexity into a software system. Multiple instances of an application and multithreaded applications might be persisting events to an event store. The system must be designed to ensure that events are processed in the correct order.

The code that processes events and the event schema can change over time. Consideration must be given to ensure that older events, possibly with different event schemas, can still be replayed with the current logic.

If part of the sequence of events includes the use of an external system, consideration has to be given to storing the responses from the external system as events. This will ensure that we can replay events accurately to rebuild the application's state without having to call the external system again.

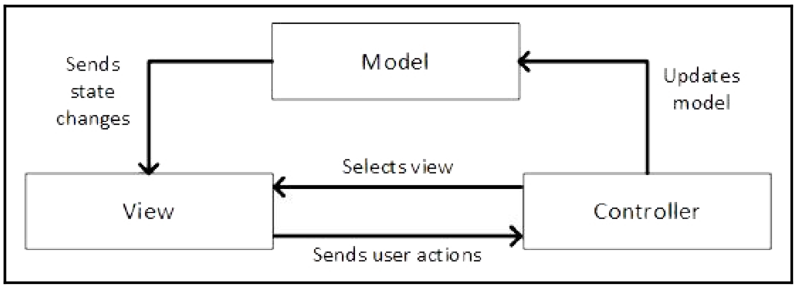
The Model-View-Controller pattern

The **Model-View-Controller** (**MVC**) pattern is a software architecture pattern that is widely used for the UI of an application. It is particularly well suited to web applications, although it can also be used for other types of applications, such as desktop applications.

The pattern provides a structure for building user interfaces and provides a separation of the different responsibilities involved. A number of popular web and application development frameworks make use of this pattern. A few examples include Ruby on Rails, ASP.NET MVC, and Spring MVC.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

The MVC pattern consists of the **Model**, **View**, and **Controller**:



The model, view, and controller all have distinct responsibilities for the user interface. Let's take a look at each of them more closely.

Model

The model manages the application data and the state. Among its responsibilities is the processing of data to and from a data store, such as a database. A model is independent of the controllers and the views, allowing them to be reused with different user interfaces. This also allows them to be tested independently.

Models receive directives from controllers to retrieve and update data. Models also provide application state updates. In some variations of MVC, the model is *passive* and must receive a request to send out an application state update. In other variations, a view may be *active* and push notifications of model state changes to a view.

View

The view is responsible for the presentation of the application. It is the part of the application that is visible to the user. The view displays data to the user in an appropriate interface based on information received from the controller. If the model is providing application state updates directly to views, the views may also be updated based on these notifications.

As users manipulate a view, such as providing input or providing some user action, the view will send this information to a controller.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Controller

As users navigate a web application, requests are routed to the appropriate controller based on routing configuration. A controller acts as an intermediary between the model and the view.

A controller executes application logic to select the appropriate view and sends it the information that it needs to render the user interface. Views notify controllers of user actions so that the controller can respond to them. Controllers will update the model based on user actions.

Advantages of the MVC pattern

Using the MVC pattern allows for a separation of concerns. By separating the presentation from the data, it makes it easier to change one of them without affecting the other. It also makes each part easier to test. However, it is difficult to achieve a complete separation. For example, adding a new field to the application will require a change in both the data and the presentation.

The MVC pattern makes presentation objects more reusable. Separating the user interface from the data allows UI components to be reused. It also means that a model can be reused with more than one view.

The separation of the presentation from the business logic and data allows developers to specialize in either frontend or backend development. This can also speed up the development process as some tasks can take place in parallel. For example, one developer can work on the user interface while another works on the business logic.

Disadvantages of the MVC pattern

If your development team is not set up so that developers are focused on either frontend or backend development, this does require developers to be skilled in both areas (full-stack developers). This will require developers who are skilled in multiple technologies.

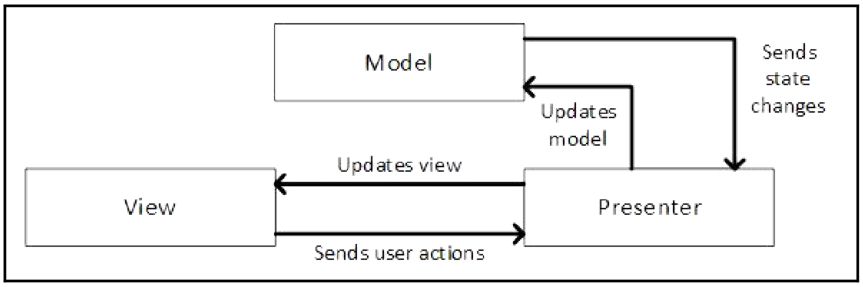
If a model is very active and is providing notifications directly to views, frequent changes to a model may result in excessive updates to views.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

The Model-View-Presenter pattern

The **Model-View-Presenter** (**MVP**) pattern is a variation on the MVC pattern. Like the MVC pattern, it provides a separation between UI logic and business logic. However, the presenter takes the place of the controller in the MVP pattern.

Each view in the MVP pattern typically has a corresponding interface (view interface). Presenters are coupled with the view interfaces. As compared with the MVC pattern, the view is more loosely coupled to the model because the two do not interact with each other directly:



Both web and desktop applications can use the MVP pattern. The main components of this pattern are the **Model**, **View**, and **Presenter**.

Model

As was the case with the MVC pattern, the model represents the business model and the data. It interacts with the database to retrieve and update data. The model receives messages from the presenter for updates and reports state changes back to the presenter.

Models in the MVP pattern do not interact directly with views and only interact with the presenter.

View

The view is responsible for displaying the user interface and data. Each view in the MVP pattern implements an interface (view interface). As the user interacts with the view, the view will send messages to the presenter to act on the events and data.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Ingeno, J. (2018). Software architect's handbook : Become a successful software architect by implementing effective architecture concepts. Packt Publishing, Limited.
Created from senecac on 2022-09-01 14:55:49.

Presenters are loosely coupled with views through the view interface. Views are more passive in the MVP model and rely on the presenter to provide information on what to display.

Presenter

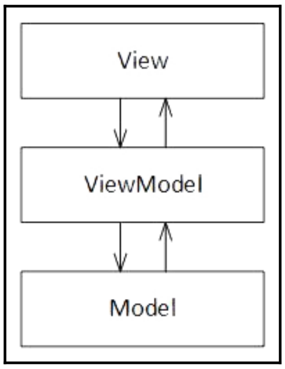
The presenter is the intermediary between the model and the view. It interacts with both of them. Each view has a presenter and the view notifies the presenter of user actions. The presenter updates the model and receives state changes from the model.

A presenter will receive data from the model and format it for the view to display, taking an active role in presentation logic. Presenters encapsulate presentation logic and views play a more passive role.

Unlike the MVC pattern, where a controller can interact with multiple views, in the MVP pattern, each presenter typically handles one, and only one, view.

The Model-View-ViewModel pattern

The **Model-View-ViewModel** (**MVVM**) pattern is another software architecture pattern and it shares similarities with MVC and MVP in that they all provide a SoC. Partitioning the various responsibilities makes an application easier to maintain, extend, and test. The MVVM pattern separates the UI from the rest of the application:



Copyright © 2018. Packt Publishing, Limited. All rights reserved.

There is typically a significant amount of interaction between views and ViewModels, facilitated by data binding. The MVVM pattern works well for rich desktop applications, although it can be used for other types of application, such as web and mobile applications. An example of a framework that can be used to build MVVM applications is **Windows Presentation Foundation** (**WPF**).

The main components of MVVM are the **Model**, **View**, and **ViewModel**. Let's take a look at each of these in more detail.

Model

The model in the MVVM pattern plays a similar role as in MVC and MVP. It represents the business domain object and the data. The model uses the database to retrieve and update data.

In MVVM applications, there may be direct binding with model properties. As a result, models commonly raise property changed notifications.

View

The view is responsible for the user interface. It is the part of the application that is visible to users. In the MVVM pattern, the view is active. Unlike a passive role where the view is completely manipulated by a controller or a presenter, and does not have knowledge of the model, in MVVM views are aware of the model and ViewModel.

While views handle their own events, they do not maintain state. They must relay user actions to the ViewModel, which can be done through a mechanism such as data binding or commands. A goal with the MVVM pattern is to minimize the amount of code in views.

ViewModel

The ViewModel in the MVVM pattern is similar to the controller and presenter objects that we covered with the MVC and MVP patterns in that they coordinate between the view and the model.

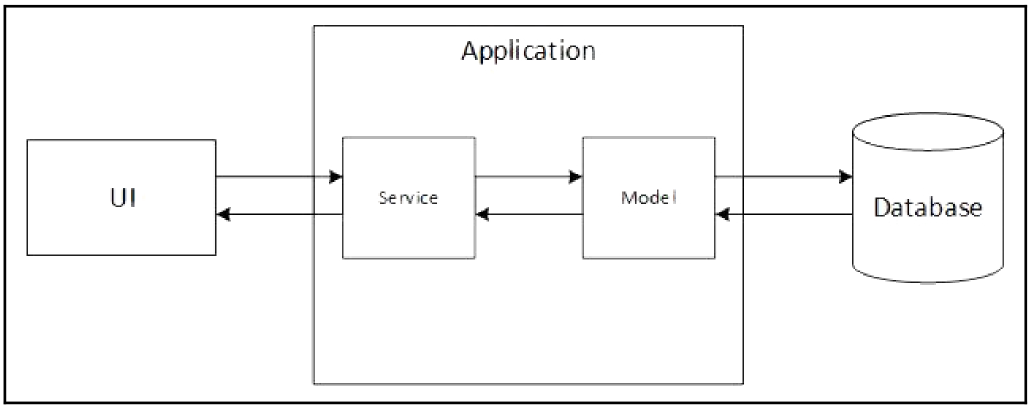
ViewModels provide data to views for display and manipulation, and also contain interaction logic to communicate with views and models. ViewModels must be capable of handling user actions and data input sent from views. It is the ViewModel that contains navigation logic to handle moving to a different view.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Views and ViewModels communicate through multiple methods, such as data binding, commands, method calls, properties, and events.

The Command Query Responsibility Segregation pattern

Command Query Responsibility Segregation (**CQRS**) is a pattern in which the model that is used to read information is separated from the model that is used to update information. In a more traditional architecture, a single object model is used for both reading and updating data:



Compromises become necessary in order to use a single object model as domain classes are required to serve all purposes. The same representation of an entity must support all of the **create, read, update, and delete** (**CRUD**) operations, making them larger than they need to be in all circumstances.

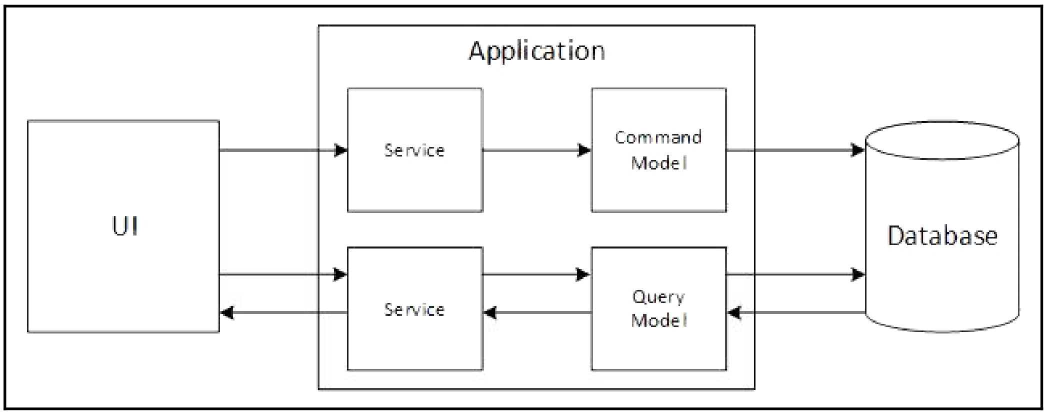
They contain all of the properties the object will need for various scenarios. If the class is more than just a **data transfer object** (**DTO**), it may also contain methods for behavior. With this approach, classes are not ideal for all of the situations in which they need to be used as there is often a mismatch between what is required by the read and write representations of the data. This can also make managing security and authorization more complex as each class is used for both read and write operations.

In a collaborative domain, multiple operations may be taking place in parallel on the same set of data. There is a risk of data contention if records are locked, or update conflicts due to concurrent updates. Workloads between read and write tasks differ, which means that they also have different performance and scalability requirements.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

The query model and the command model

One way to overcome the challenges of using a single object model for both queries and commands is to separate the two. This pattern, known as CQRS, results in two separate models. The **query model** is responsible for reads and the **command model** is responsible for updates:



Query objects only return data and do not alter state, while command objects alter state and do not return data. Another way of looking at this concept is that asking a question (a query) should not alter the answer. In order to perform an action that will mutate state, a command is used.

When the system needs to read data, it goes through the query model, and when the system needs to update data, it goes through the command model. As part of processing a command, the system may need to read data, but beyond what is necessary for completing a command, the reading of data should go through the query model.

Although not required, if CQRS is taken to the next level, the query and command models can be made to utilize separate databases. This gives each of the two models its own schema, which can be optimized for its specific usage. If separate databases are used, the two databases must be kept in sync. One way to do that is through the use of events.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Using event-sourcing with CQRS

Although CQRS can be used without events, they do complement each other so it is common for systems that use CQRS to leverage the use of events. Events are one way to effectively communicate state changes so that the query model can stay up to date as the command model updates data.

As we saw in the *Event-sourcing* section earlier in this chapter, event-sourcing involves persisting events that take place in a system such that the event store can serve as the record of truth. When the command model alters the state of the system, events can be raised so that the query model can be kept in sync.

Keeping the query and command models in sync is necessary when the query and command models use separate data stores. In addition, a data store for the query model may contain denormalized data that has been optimized for particular queries. Having an event store allows us to replay past events to recreate the system's current state, which can be useful for updating the denormalized data in the query model's data store.

Advantages of CQRS

CQRS is well suited to complex domains and it provides a separation of concerns that helps to minimize and manage the complexity. Separating a system into query and command models makes it more maintainable, extensible, and flexible. Development teams can also be organized so that one team focuses on the query model while another focuses on the command model.

Segregating the responsibility between commands and queries can help to improve performance, scalability, and security. Performance can be improved by optimizing the schema specifically for each model. The schema for the query model can be optimized for queries, while the schema for the command model can be optimized for updates. Data in the query model's data store can be denormalized in order to increase the performance of queries that the application needs to execute.

Workloads between read and write operations will differ and using CQRS allows you to scale each of them independently of the others. Security is improved with CQRS because it makes it easier than with a single object model to ensure that only the right classes can update data.

Security can be easier to implement and test when using CQRS because each class is either used for reads or writes, but not both. This reduces the chance of inadvertently exposing data and operations that should not be available to a particular user in a certain context.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Disadvantages of CQRS

For systems that simply need basic CRUD operations, implementing a CQRS system may introduce unnecessary complexity. CQRS systems have a higher degree of complexity, especially when combined with event-sourcing. For this reason, it is important to understand that CQRS is not applicable to all situations. Software architects should be aware that CQRS does not have to be applied to the entirety of a software system. It can be applied to just some subsystems of a larger architecture where it will be of the greatest benefit.

While using different data stores for query and command models can improve performance and increase security, you do have to consider that when you perform read operations, you may be reading data that is stale.

If separate databases are used in a CQRS system, the database used for reads must be kept up to date with the database used for writes. The software system will follow an *eventual consistency* model where if no new updates are made to a given item, eventually all access to that item will acquire the latest data.

Whether the system is using event-sourcing or some other mechanism to keep the two in sync, there will be some time delay, even if it is a small one, before they are made consistent. This means that any data that is read could potentially be stale if the latest updates to it have not been applied yet. This is in contrast with a *strong consistency* model, in which all data changes are atomic and a transaction is not allowed to complete until all of the changes have been completed successfully or, in the case of a failure, everything has been undone.

Service-oriented architecture

Service-oriented architecture (**SOA**) is an architectural pattern for developing software systems by creating loosely coupled, interoperable services that work together to automate business processes. A service is a part of a software application that performs a specific task, providing functionality to other parts of the same software application or to other software applications. Some examples of service consumers include web applications, mobile applications, desktop applications, and other services.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

SOA achieves a *SoC*, which is a design principle that separates a software system into parts, with each part addressing a distinct concern. We discussed *SoC* in `Chapter 6`, *Software Development Principles and Practices*. A key aspect of SOA is that it decomposes application logic into smaller units that can be reused and distributed. By decomposing a large problem into smaller, more manageable concerns satisfied by services, complexity is reduced and the quality of the software is improved.

Each service in a SOA encapsulates a certain piece of logic. This logic may be responsible for a very specific task, a business process, or a subprocess. Services can vary in size and one service can be composed of multiple other services to accomplish its task.

What makes SOA different from other distributed solutions?

Distributing application logic and separating it into smaller, more manageable units is not what makes SOA different from previous approaches to distributed computing. Naturally, you might think the biggest difference is the use of web services, but keep in mind that SOA does not require web services, although they happen to be a perfect technology to implement with SOA. What really sets SOA apart from a traditional distributed architecture is not the use of web services, but how its core components are designed.

Although SOA shares similarities with earlier distributed solutions, it is much more than just another attempt to create reusable software. The differences undoubtedly provide significant new value to organizations when implemented properly. Many benefits can be realized from a properly designed SOA.

Benefits of using a SOA

There are a number of benefits from using a SOA, including:

- Increases alignment between business and technology
- Promotes federation within an organization
- Allows for vendor diversity
- Increases intrinsic interoperability
- Works well with agile development methodologies

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Increases alignment between business and technology

A SOA leads to increased alignment between business and technology. The fulfillment of business requirements needs business logic and business processes to be accurately represented in technology solutions. Business logic, in the form of business entities and business processes, exists in the form of physical services with an SOA.

This alignment of business and technology promotes organizational agility. Change is something that practically all organizations must face, and it exists due to a variety of factors, such as market forces, technology changes, new business opportunities, and corporate mergers. Regardless of the cause of change, SOA provides organizational agility through service abstraction and loose coupling between business and application logic. When changes are required, they can more easily be made so that the business remains in alignment with the technology.

Promotes federation within an organization

A SOA promotes federation. Federation in an organization is an environment in which the software applications and resources work together while simultaneously maintaining their autonomy. Federation gives organizations the freedom not to be required to replace all of their existing systems that must work together. As long as there is a common, open, and standardized framework, legacy and non-legacy applications can work together. Organizations have the flexibility to choose whether they want to replace certain systems, allowing them to use a phased approach to migration.

Allows for vendor diversity

Another advantage of SOA is vendor diversity. In addition to allowing organizations with potentially disparate vendors to work together, an organization can use different vendors internally to achieve best-of-breed solutions.

While it is not a goal of SOA to increase vendor diversity, it provides the option of vendor diversity when there is an advantage to introducing new technologies.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Increases intrinsic interoperability

SOA provides increased intrinsic interoperability for an organization. It allows for the sharing of data and the reuse of logic. Different services can be assembled together to help automate a variety of business processes. It can allow an existing software system to integrate with others through web services. Greater interoperability can lead to the realization of other strategic goals.

Works well with agile development methodologies

Another benefit of SOA is that it lends itself well to agile software development methodologies. The fact that complex software systems are broken down into services with small, manageable units of logic fits well with an iterative process and how tasks are allocated to resources.

You may also find that having developers take on tasks is easier with SOA because each task can be made to be manageable in size and more easily understood. Although this is beneficial for any developer, it is particularly helpful for junior developers or those who are new to a project and may not have as much experience with the functionality and business domain.

Cost-benefit analysis of SOA

As a software architect, if you are considering SOA for an application, you will need to explain the reasons why you are considering it. Adopting an SOA comes at some cost, but there are points you can make to justify the costs and ways that you can alleviate them.

The cost of implementing a SOA may outweigh the benefits for some organizations, so each case must be considered separately. While it may not be appropriate for some organizations to implement an SOA, for others a properly designed SOA will bring many benefits, including a positive return on investment.

Adopting SOA can be a gradual, evolutionary process. Because a contemporary SOA promotes federation, creating an SOA does not have to be an all-or-nothing process. An organization does not have to replace all existing systems at once. Legacy logic can be encapsulated and can work with new application logic. As a result, the adoption of SOA and its related costs can be spread out over time.

Adopting SOA leads to reduced integration expenses. A loosely coupled SOA should reduce complexity and therefore will reduce the cost to integrate and manage such systems. Loosely coupled services are more flexible and can be used in more situations.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

SOA can increase asset reuse. It is common for each application to be built in isolation, leading to higher development costs and greater maintenance costs over time. However, by creating business processes by reusing existing services, costs and time to market can be reduced.

SOA increases business agility. Change is something that all organizations must face. Regardless of the cause of the change, by using loosely coupled services, organizational agility is increased and both the time and cost to adapt to change are reduced.

Adopting SOA reduces business risk and exposure. A properly designed SOA facilitates the control of business processes, allows for the implementation of security and privacy policies, and provides audit trails for data, which can all reduce risk. It can also help with regulatory compliance. The penalties for non-compliance can be significant, and SOA can provide organizations with increased business visibility that reduces the risk of changing regulations.

Challenges with SOA

Although adopting an SOA does provide many benefits, it also introduces some new complexities and challenges. SOA solutions may allow organizations to do more, including automating more of their business processes. This can cause enterprise architectures to grow larger in scope and functionality as compared to legacy systems. Taking on a larger scope of functionality will add complexity to a software system.

In an SOA, new layers may be added to software architectures, providing more areas where failure can occur and making it more difficult to pinpoint those failures. In addition, as increasing numbers of services are created, deploying new services and new versions of existing services must be managed carefully so that troubleshooting can be effective when an error occurs with a specific transaction.

Another challenge with successful SOA adoption is related to people and not technology. SOA is a mature architectural style that has been around a long time. The technology exists to allow organizations to automate a variety of complex business processes. However, people can be a challenge to SOA-adoption because there are still technical and business professionals who are not familiar with SOA and really do not know what it means. People can also be naturally resistant to change, and if your organization is not already using SOA, change will be necessary. In order for a SOA to be successful, there has to be buy-in from the people in the organization. They have to be committed and this comes down to the culture of the team and the people on it, including managers, software architects, developers, and business analysts.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Key principles for service orientation

Service-oriented solutions are designed so that they adhere to certain key principles. They are:

- Standardized service contract
- Service loose coupling
- Service abstraction
- Service reusability
- Service autonomy
- Service statelessness
- Service discoverability
- Service composability

These principles are detailed in Thomas Erl's book, *Service-Oriented Architecture, Second Edition*. Service-orientation principles are applied to the service-oriented analysis and design phases of the SOA delivery life cycle.

Standardized service contract

Each service should have a standardized service contract, consisting of a technical interface and service description. Even though we want services to be independent, they have to adhere to a common agreement so that units of logic can maintain a certain level of standardization.

In order to have standardized service contracts, all service contracts within a particular service inventory should follow a set of design standards. Standardization enables interoperability and allows the purpose of services to be more easily understood.

Service loose coupling

Services should be loosely coupled and independent of each other. Service contracts should be designed to have independence from service consumers and from their implementations.

Loosely coupled services can be modified faster and easier. Decoupling service contracts from their implementations allows service contracts to be modified with minimal impact to service consumers and service implementations. By minimizing the dependencies between services, each service can change and evolve independently while minimizing the effects of those changes on other services.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Ingeno, J. (2018). Software architect's handbook : Become a successful software architect by implementing effective architecture concepts. Packt Publishing, Limited.
Created from senecac on 2022-09-01 14:55:49.

Service abstraction

Service contracts should only contain information that it is necessary to reveal, and service implementations should also hide their details. Any information that is not essential to effectively use the service can be abstracted out.

Design decisions, such as the technology used for a service, can be abstracted away. This follows the information hiding principle that was covered in `Chapter 6`, *Software Development Principles and Practices*. If a design decision needs to be changed later, the goal is that it can be made with minimal impact.

Service reusability

Services should be designed with reusability in mind, with their service logic being independent of a particular technology or business process. When services can be reused for different purposes, software development teams experience increased productivity, leading to savings in both costs and time.

Service reusability increases organizational agility because organizations can use existing services to respond to new business automation needs. Existing services can be composed together to create solutions for new problems or to take advantage of a new opportunity. Reusable services can accelerate development and may allow a feature or product to reach the market faster. In some cases, this can be critical for a project.

Decomposing tasks into more services for reusability requires more analysis and potentially introduces more complexity. However, when reusable services are designed correctly, there can be significant long-term cost savings. If a need arises that is satisfied by an existing service, resources do not have to be devoted to working on it.

Service reuse leads to higher quality software because existing services have already been tested. They may already be in production and, if there were any defects with the service, they may have already been exposed and corrected.

Service autonomy

Services should be designed to be autonomous, with more independence from their runtime environments. The design should seek to provide services with increased control over their runtime environments.

When services can operate with less dependence on resources in their runtime environments that they cannot control, this leads to better performance and increased reliability of those services at runtime.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Service statelessness

Service designs should strive to minimize the amount of state management that takes place, and separate state data from services.

Services can reduce their resource consumption if they do not manage state when it is unnecessary, which will allow them to handle more requests reliably. Having statelessness in services improves service scalability and improves the reusability of services.

Service discoverability

Services need to be discoverable. By including consistent and meaningful metadata with a service, the purpose of the service and the functionality it provides can be communicated. Service developers are required to provide this metadata.

Services should be discoverable by humans who are searching manually as well as software applications searching programmatically. Services must be aware of each other for them to interact.

Service composability

Services should be designed so that they are composable. This is the ability to use a service in any number of other services, and those services may themselves be composed of other services.

Some other service-orientation principles facilitate service composability. Service composition is heavily related to service reusability. The ability to create solutions by composing existing services provides organizations with one of the most important SOA benefits: organizational agility.

SOA delivery strategies

There are three main SOA delivery strategies: top-down, bottom-up, and agile. A delivery strategy is needed to coordinate the delivery of application, business, and process services.

The three SOA delivery strategies mirror the main approaches to software architecture design that were covered in `Chapter 5`, *Designing Software Architectures*.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

The top-down strategy

The top-down strategy begins with analysis. It centers on the organization's business logic and requires that business processes become service-oriented. The top-down approach, when done properly, results in a high quality SOA. Each service is thoroughly analyzed, and as a result, reusability is maximized.

The downside is that this approach requires many resources, in terms of time and money. There is substantial pre-work that must take place with the top-down strategy. If an organization has the time and money to invest in a project, then this may be an effective approach.

It should be noted that, because analysis occurs at the beginning, it could be quite some time before any results are realized. This may or may not be acceptable for a given project. In order to meaningfully perform the service-oriented analysis and service-oriented design stages of the SOA life cycle, the top-down strategy has to be used at least to some extent.

The bottom-up strategy

The bottom-up approach, in contrast, begins with the web services themselves. They are created on an *as-needed* basis. Web services are designed and deployed based on immediate needs.

Integration with an existing system is a common motivation for using the bottom-up strategy. Organizations want to add web services to an existing application environment to allow for integration with a legacy system. A wrapper service is created to expose logic in an existing system.

Although this approach is common in the industry, it is not a valid approach to achieving SOA. In order to create a valid SOA later, a lot of effort and refactoring will probably be required. Web services created with this approach may not be *enterprise ready*. They are created to serve some need, so if you are not careful they will not take into consideration the enterprise as a whole.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

The agile strategy

The third approach is an agile strategy, which is sometimes referred to as a meet-in-the-middle approach. It is a compromise between the top-down and bottom-up approaches. In this approach, analysis can occur concurrently with design and development. As soon as enough analysis has been completed, design and development can begin. While such efforts are underway, analysis continues with other functionality. This approach pairs well with an iterative, agile software development methodology.

This is sort of a *best-of-both-worlds* approach in that a proper design can be completed that will yield all of the service-oriented qualities. This approach can fulfill immediate needs while maintaining service-oriented qualities of the architecture.

However, as more analysis is finished, this approach may require completed services to be revisited. Services can become misaligned after ongoing analysis, requiring them to be refactored.

Service-oriented analysis

Service-oriented analysis is a stage in the SOA project life cycle and is used to decide what services should be built and what logic should be encapsulated by each service. The analysis is an iterative process that takes place once for each business process.

When a team is committed to building a SOA, it should perform some form of analysis specific to service-orientation and beyond standard analysis. One way that organizations can improve service modeling is how they go about incorporating service-oriented analysis and design into their software development process. Each organization has its own software development methodology and should determine how best to include service modeling into their own process.

Service-Oriented Architecture, Second Edition, by Thomas Erl, details three steps to service-oriented analysis: defining business automation requirements, identifying existing automation systems, and modeling candidate services.

Defining business automation requirements

The first step in service-oriented analysis is to define the business automation requirements for the business process being analyzed in the current iteration. Requirements can be gathered using the organization's normal method of eliciting and capturing requirements.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

With those requirements, the business process we want to automate can be documented at a high level. The details of the business process are used when we model candidate services.

Identifying existing automation systems

Once the requirements have been established for the current iteration, the next step in service-oriented analysis involves identifying what parts, if any, of the business process logic are already automated.

Taking into consideration existing systems that may already automate all or part of any of the business processes allows us to determine what parts of the business processes still need to be automated. This information serves as an input when we model candidate services.

Modeling candidate services

The final step, modeling candidate services, consists of identifying service operation candidates and grouping them into candidate services. It is important to note that these candidate operations and services are abstract and a logical model. During design, other factors, such as constraints and limitations, will be considered. The final concrete design may differ from the service candidates.

Modeling candidate services should be a collaborative process between technical and business resources. Business analysts and domain experts can use their business knowledge to help the technical team define service candidates.

Service layers and service models

Enterprise logic consists of both business and application logic. Business logic is an implementation of the business requirements and includes an organization's business processes. These requirements include things such as constraints, dependencies, pre-conditions, and post-conditions.

Application logic is the implementation of business logic in a technology solution. Application logic might be implemented in a purchased solution, a custom developed solution, or some combination of the two. The development team works to design and develop the application logic. Topics such as performance requirements, security constraints, and vendor dependencies are considered in the technical solution.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

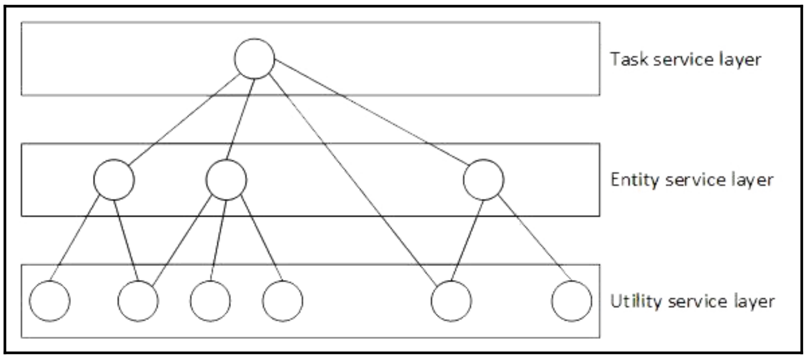
Service-orientation is related to business and application logic because a SOA is a way to represent, execute, and share that logic. Service-orientation principles can be applied to both business and application logic.

The role of services is to realize the concepts and principles introduced by service-orientation. A service layer in a software architecture is typically placed between the business and application layers. This allows services to represent business logic and abstract application logic. Just as different applications within an organization's application layer can be implemented in different technologies, services within the service layers can also be implemented in different technologies.

Abstraction is one of the important characteristics of an SOA and it enables other key characteristics, such as organizational agility. Abstraction is critical because abstracting business and application logic allows for a service-oriented solution with loosely coupled services. Achieving the appropriate level of abstraction is not a trivial task, but it can be accomplished with a dedicated team. By creating layers of abstraction, or service layers, the team can figure out how services should represent application and business logic, and how to best promote agility.

During service modeling, it becomes apparent that there are some common types of services. These types are **service models**, which can be used to classify candidate services. Those candidate services can then be grouped together based on their service model into a *service layer*.

The three common service models (and layers) are task service, entity service, and utility service:



Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Task service

This type of service has a **non-agnostic functional context**, which means that it contains business process logic and was created for a specific business task or process. Task services do not have a great deal of reuse potential.

Task services typically compose multiple services in their logic and are sometimes referred to as *task-centric business services* or *business process services*.

If a task service has extensive orchestration logic or is hosted within an orchestration platform, it is sometimes referred to as an *orchestrated task service*. Orchestration logic automates the arrangement, coordination, and management of multiple services to accomplish a task.

Entity service

This service model has an **agnostic functional context**, meaning its logic is not bound to a single business process and is reusable. Entity services are business-centric services that are associated with one or more business entities. They are sometimes referred to as *entity-centric business services* or *business entity services*.

Business entities come from the business domain, and involving people on the team who thoroughly understand the business domain and business processes will be beneficial in performing the analysis needed to determine the appropriate entity services. Entity services may compose utility services in their logic.

Utility service

Utility services, like entity services, have an agnostic functional context. They contain multi-purpose logic and are highly reusable. The difference between entity services and utility services is that utility services are not associated with a business entity or business logic.

Utility services are concerned with technology-related functions, such as a software system's cross-cutting concerns. Examples include logging, caching, notifications, authentication, and authorization.

Service-oriented design

The service-oriented design phase begins once the analysis is complete. Having a thorough understanding of the requirements and using the service models from the analysis stage will allow for the creation of a proper service design.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Ingeno, J. (2018). Software architect's handbook : Become a successful software architect by implementing effective architecture concepts. Packt Publishing, Limited.
Created from senecac on 2022-09-01 14:55:49.

The service-oriented design phase uses the logical candidate services that were derived during service-oriented analysis and creates the physical service designs. Before designing the implementation of the services, the first step is to design the physical service interfaces. We need to make decisions regarding the service interfaces based on the service candidates, the requirements that need to be met, as well as the organization and industry standards that will be required to have a standardized service contract. Once the service contracts have been established, the logic and implementation of the service can be designed.

Designing the service interfaces and implementing them are two distinct steps. We should fully focus on the service contracts first, independent of their implementations. Some teams design the two concurrently or skip to the development stage and just let the service interface *emerge* from the implemented service.

However, software architects should devote time to considering the service contracts before the implementations. Not only do we need to ensure that the service contracts satisfy the requirements, but they need to follow the key principles of service orientation, which include the fact that they should be loosely coupled from their implementations. Only after the service contracts have been established do we need to consider the design of the implementations.

Service interface design

One of the main goals of service-oriented design is to derive the physical service interface definitions based on the service candidates that were determined during service-oriented analysis. Service interface design is significant because the design phase is the first time that real technology is identified.

If you recall the key principles for service orientation, they need to be applied to the design of the service interfaces. Service contracts need to be standardized with each other and within a service inventory. They should be loosely coupled from their implementations, with design decisions abstracted out so that the interfaces only contain what is essential for service consumers.

Service interface design identifies internal and external exposure of the services. For example, an order posting service may need to be used externally as well as internally. This is an example of a service that may need more than one interface published for the same service. Each interface to the service may expose different operations and will require different levels of security and authentication. The differences must be determined and each interface must be designed prior to the design of the implementations.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

In addition to developers, a service interface plays an important role in testing and quality assurance. Testers need the service interface to design their tests. Once the service interface is known, a test harness can be created that is capable of invoking the service that needs to be tested. Services need to be tested independently of other services as well as within the services that will be consuming it.

Service interface granularity

Making decisions about interface granularity can be very important in service-oriented design. Granularity can have a significant impact on performance and other concerns. Usually, a service interface contains more than one operation, and the operations of a service should be semantically related.

Fine-grained service operations offer greater flexibility to service consumers but lead to more network overhead, which could reduce performance. The more coarse-grained service operations we have, the less flexible they are, although they do reduce network overhead and could therefore improve performance.

Software architects should seek to find the right balance between the number of services and the number of operations in each service. You do not want to group too many operations into a single service, even if they are semantically related, because it makes the service too bulky and hard to understand. It may also increase the number of service versions that will need to be released going forward, as parts of the service need to be modified. However, if your service interface is too fine-grained, you may end up with an unnecessarily large number of service interfaces.

Service registries

A **service registry** contains information about the available services and is a key component of SOA governance. It helps to make systems interoperable and facilitates the discovery of web services. Although some organizations may find that a service registry is not needed, many SOA implementations can benefit from having a service registry. As an organization begins to publish and use more and more web services, some of which may be outside the organization, the need for a centralized registry becomes more apparent.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

There are numerous benefits to using a service registry. By promoting the discovery of web services, organizations can facilitate reuse and avoid building multiple web services that perform similar tasks. Developers can programmatically query a service registry to discover web services that already exist that can satisfy their needs. Similar to the benefits that are derived from any type of reused code, quality is improved and there is an increased level of dependability from reusing web services. Reused web services have already been tested and are usually already being successfully used in another part of the system or in another system.

Service registries can either be private or public. As the name implies, public registries can include any organization. This even includes organizations that do not have any web services to offer. Private registries are restricted to those services that the organization develops itself or services that it has leased or purchased.

The benefits of public service registries include being able to find the right businesses and services for a particular need. It can also lead to new customers or allow more access to current customers. It could allow an organization to expand their offerings and extend their market reach. Service registries can be a useful and powerful tool for finding available web services since they can be searched manually by people or programmatically through a standardized API by an application. However, because of these capabilities, organizations should take the time to decide whether to use a public or private registry and what services they want to register into them.

One of the challenges of implementing a truly useful and reliable registry service is the administration of the registry. This includes keeping it up to date by adding new services, removing obsolete services, and updating versions, service descriptions, and web service locations.

Service descriptions

In order for services to interact with each other, they must be aware of each other. **Service descriptions** serve the important purpose of providing this awareness. They provide information about the available services so that potential consumers can decide whether a particular service will satisfy their needs.

Service descriptions help to foster loose coupling, an important principle of SOAs. Dependencies between services are minimized as services can work together simply through the awareness they have of each other through their service descriptions.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Any service that wants to act as an ultimate receiver must have service description documents. Service descriptions typically have both abstract and concrete information. The abstract part details the service interface without getting into the details of the specific technologies being used. The beauty of the abstraction is that the integrity of the service description is maintained even if the details of the technical implementation are changed in the future. The abstract description typically includes a high-level overview of the service interface, including what operations it can perform. Input and output messages of the operations are also detailed.

The concrete part of the service description provides details about the physical transport protocol that is connected to the web service interface. This specific transport and location information includes the binding (requirements for the service to establish a connection or for a connection to be established with a service), port (physical address of the web service), and service (a group of related endpoints) so that the web service can be used.

Possible challenges to developing service descriptions include the following:

- Decomposing web services properly based on business needs
- Determining the exact purpose and responsibilities of a particular service
- Deciding on the operations that a web service will need to provide
- Properly communicating a service's interface in the abstract part of the service description so that potential service consumers can make an informed decision based on their needs

Structuring namespaces

A **namespace** is a unique **Uniform Resource Locator** (**URI**). Namespaces are used to group related services and elements together and to differentiate between different ones that share the same name. It is important for software architects to put thought into namespaces. By providing a unique namespace, even if your organization uses services from another one, your elements will be guaranteed to be unique. Even if two organizations have a service with the same name, they will be differentiated by their namespace.

In addition to providing unique names, namespaces are used to logically organize various services and elements. An appropriate namespace should provide meaning to the service or element so that someone who is looking at it can gain an understanding of the service. Namespaces make it easier to name new services as well as to find existing ones.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

In order to select good namespaces, we have to consider how namespaces are structured. A company's domain name is an important part of a namespace and because domain names are unique, they are commonly part of a namespace. Typically, the role follows the domain name in a namespace. This will allow differentiation between schema (for example, message types) and interfaces (for example, web services).

A business area typically follows the role in the structure of a namespace. This is where domain experts and business analysts can assist software architects in coming up with a business structure that makes sense. Another part of a namespace that is common is some form of version or date. This allows differentiation between multiple versions of the same service or element. Versioning is another important use of namespaces.

Orchestration and choreography

Service orchestration and **service choreography** serve important roles in SOAs, as they are approaches to assembling multiple services so that they can work together.

Orchestration represents business process logic in a standardized way using services. It automates the execution of a workflow by coordinating and managing different services. In service orchestration, there is a centralized process containing fixed logic. An orchestrator controls the process by deciding which services to invoke and when to invoke them. Orchestration is analogous to the conductor of an orchestra, who unifies and directs individual performers to create an overall performance.

Interoperability for an organization is promoted in solutions using orchestration because of the integration endpoints that are introduced in processes. In a SOA, orchestrations themselves are services. This promotes federation because multiple business processes, potentially from different applications, can be merged together.

Choreography is another form of service composition. Choreographies define message exchanges and can involve multiple participants, each of which may assume multiple roles.

In contrast with orchestration, there is no centralized process, or orchestrator, that is controlling it. With choreography, there is an agreed upon set of coordinated interactions that specify the conditions in which data will be exchanged. Each service in a choreography acts autonomously to execute its part based on the conditions that were established and the actions of the other participants.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.

Both orchestration and choreography can be used for business-process logic owned by a single organization (intra-organization) and collaboration between multiple organizations (inter-organization). However, orchestration is less likely to be used when there are multiple organizations involved because you would need to own and operate the orchestration. Choreography allows for collaboration without having a single organization control the whole process.

Summary

Software architects should be familiar with software architecture patterns, as they are a powerful tool when designing a software architecture. Architecture patterns provide a proven solution to recurring problems for a given context.

Leveraging architecture patterns gives the software architect a high-level structure of the software system, and provides a grouping of design decisions that have been repeated and used successfully. Using them reduces complexity by placing constraints on the design and allows us to anticipate the qualities that the software system will exhibit once it is implemented.

In this chapter, you learned about some of the common software architecture patterns available, including layered architecture, EDA, MVC, MVP, MVVM, CQRS, and SOA.

The focus of the next chapter is on some of the relatively newer software architecture patterns and paradigms. These include microservice architecture, serverless architecture, and cloud-native applications. As cloud deployment of software applications becomes the dominant trend, these concepts become crucial for any software architect to understand.

Copyright © 2018. Packt Publishing, Limited. All rights reserved.