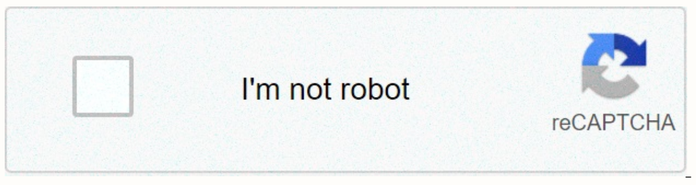

Continue

O' reilly software architecture patterns pdf

It's all too common for developers to start encoding an app without having an official architecture in place. Without a clear and well-defined architecture, most developers and architects resort to the de facto standard traditional layered architecture pattern (also known as n-level architecture), creating implicit layers by disconnecting source code modules into packages. Unfortunately, we often derive this practice from a collection of disorganized source code modules that do not have clear roles, responsibilities, and relationships with each other. It is often referred to as the big ball of mud architecture anti-pattern. Applications lacking formal architecture tend to be tightly connected, fragile, difficult to change, and without a clear vision or direction. As a result, it is very difficult to determine the architectural features of the application without fully understanding the inner workings of all components and modules of the system. The basic questions about deployment and maintenance are difficult to answer: Is the architecture scaling? What are the performance characteristics of the application? How easily does the app react to changes? What are the installation features of the application? How responsive is the architecture? Learn faster. Dig deeper. See more. Architectural patterns help you determine the basic characteristics and behavior of your application. For example, some architecture patterns can naturally drive themselves to highly scalable applications, while other architecture patterns can, of course, drive themselves into very agile applications. Knowing the characteristics, strengths, and weaknesses of each architecture pattern is necessary in order to select one that meets your unique business needs and goals. As an architect, you should always justify your architecture decisions, especially when choosing a particular architectural pattern or approach. The purpose of the report is to provide you with enough information to make and give you a good time to make this decision. The most common architecture pattern is the layered architecture pattern, also known as the n-level architecture pattern. This pattern is the de facto standard for most Java EE applications and is therefore widely known to most architects, designers and developers. The layered architecture pattern fits closely with traditional IT communication and organizational structures in most companies, making it a natural choice for most business application development activities. Components within the layered architecture design are

organized into horizontal layers, and each layer plays a specific role within the application (e.g. display logic or business logic). Although the layered architecture pattern does not specify the number and type of layers in the sample, most layered architectures are made up of four standard layers: visualization, business, persistency, and database (Figure 1-1). In some cases, business and the persistency layer is merged into a single business layer, especially if the persistency logic (e.g.SQL or HSQL) is embedded in the components of the business layer. This way, smaller applications can consist of only three layers, while larger and more complex business applications can contain five or more layers. Each layer of the layered architecture pattern has a specific role and responsibility within the application. For example, a display layer is responsible for managing all user interfaces and browser communication logic, while a business tier is responsible for implementing the specific business rules associated with the request. Each layer of the architecture forms an abstraction around the work required to fulfill a specific business request. For example, the presentation layer doesn't need to know or worry about how to get customer data; you just need to display this information on the screen in a specific format. Similarly, the business tier doesn't have to worry about how it formats customer data to display on-screen, or even where customer data comes from; you just need to get the data from the retention layer, implement business logic against the data (e.g. calculate values or aggregated data), and pass this information on to the presentation layer. Figure 1-1. Layered architecture pattern One of the powerful features of the layered architecture pattern is the separation of concerns between components. Components within a layer deal only with the logic for that layer. For example, the components of the presentation layer deal only with display logic, while the components in the business layer only deal with business logic. This type of component classification facilitates the integration of powerful roles and responsibility models into the architecture and facilitates application development, testing, regulation, and maintenance with this architecture pattern due to well-defined component interfaces and limited component scope. The following shall be replaced by the following: This is a very important concept in the layered architecture pattern. A closed layer means that when the request moves from layer to layer, it must pass directly below it to get to the next layer below it. For example, a request from the presentation layer must first pass through the business layer, then through the persistency layer, before it can finally reach the database layer. Figure 1-2. Closed layers and request access Why not allow direct access to the presentation layer or persistency layer or database layer? After all, direct database access to the tutorial layer is much faster than going through a lot of unnecessary layers just to download or save database data. The answer to this question lies in the fact that the most important concept known layers of isolation. The layers of the separation concept mean that one of the the architecture usually doesn't affect or affect components in other layers: the change is isolated from the components within that layer, and possibly another related layer (for example, a persistency layer that contains sql). If you enable direct access to the display layer for the retention layer, SQL changes within the retention layer will affect both the business layer and the display layer, creating a very tightly connected application that provides a lot of interdependence between components. This type of architecture, it will be very difficult and expensive to change. The isolation concept layers also mean that each layer is independent of other layers, so you have little or no knowledge of the inner workings of other layers in the architecture. To understand the power and importance of the concept, consider the great effort of transforming the presentation framework from JSP (Java Server Pages) to JSF (Java Server Faces). Assuming that the contracts used between the display layer and the business tier (e.g. model) remain unchanged, the business tier is not affected by the re-factoring and remains completely independent of the UI framework used by the display layer. While closed layers facilitate isolation layers and thus help isolate change within the architecture, there are times when certain layers are open. For example, suppose you want to add a shared service tier to an architecture that includes an architecture that contains common service components that are accessed by components within the business layer, such as data and string utility classes, and logging and logging classes. Creating a service tier in this case is usually a good idea because it architecturally restricts access to shared services to the business tier (not the display layer). Without a separate layer, there is no architectural variety that restricts the display layer from accessing common features, making it difficult to control access restrictions. In this example, the new service tier is probably located below the business tier, indicating that the components of the service tier are not available from the display layer. However, this is a problem, as the business layer is now needed to go through the service layer to the perseverance layer, which makes no sense at all. This is an ancient problem with layered architecture and can be solved by creating open layers within the architecture. The following shall be replaced by the following: In the following example, because the service tier is open, the business tier can now bypass it and move directly to the persistency layer, which is perfectly understandable. Figure 1-3. Open layers and request flow Taking advantage of the concept of open and closed layers helps define the relationship architecture layers and application processes, as well as providing designers and developers with the information they need to understand the different layer access restrictions within the architecture. Documenting or properly communicating which layers of architecture (and why) usually results in tightly connected and fragile architectures that are very difficult to test, maintain, and deploy. To illustrate how a layered architecture works, consider asking a business user to retrieve a person's customer information in steps 1 through 4. The black arrows indicate that the request flows to the database to retrieve customer data, and the red arrows show the response that flows back to the screen to display the data. In this example, customer information is made up of both customer data and order information (orders placed by the customer). The customer form is responsible for accepting the request and displaying customer information. You don't know where the data is, how to retrieve it, or how many database tables must be queries to retrieve the data. After the client form receives a request to retrieve customer information from a specific person, it forwards the request to the client delegated module. This module is responsible for knowing which modules in the business tier can process this request and how to get to the module and what data you need (the contract). The customer object in the business layer is responsible for aggreting all the information required for the business request (in this case, to retrieve customer data). This module calls the customer dao (data access object) module to get the persistency layer to retrieve customer data as well as order information from the order dao module. These modules turn SQL statements into the appropriate data and pass them to the client object in the business layer. As soon as the client object receives the data, it aggregates the data and forwards the data to the client delegate, which then transmits this data to the client screen to be displayed to the user. Figure 1-4. Layered architecture is an example from a technological point of view, literally dozens of ways these modules can be implemented. For example, on the Java platform, the client screen can be a (JSF) Java Server Faces screen to which the client delegates the managed bean component. The client object in the business layer can be a local spring bean or a remote EJB3 bean. The data access objects shown in the previous example are simple POJO (Plain Old Java Objects), MyBatis XML Mapper files, or even objects that contain raw JDBC calls or hibernation queries. From a Microsoft platform perspective, the client screen can be an ASP (active server pages) module that uses the .. (ActiveX data objects). A layered architecture pattern is a solid general purpose pattern, making it a good starting point for most applications, especially if you're not sure what architecture pattern is best suited for your application. However, there are a couple of things to consider from an architecture perspective when choosing this pattern. The first thing to look out for is what is called an architecture sinkhole anti-pattern. This anti-pattern describes a situation in which requests flow through multiple layers of the architecture as simple pass-through processing, with little or no logic within each layer. For example, let's say the presentation layer responds to a user's request to retrieve customer data. The presentation layer passes the request to the business layer, which simply forwards the request to the retention layer, which then makes a simple SQL call to the database layer to retrieve customer data. The data is then transferred all the way back up to the stack with no additional processing or logic to aggregate, calculate, or convert the data. Each layered architecture will have at least a few scenarios that fall into the architecture sinkhole anti-pattern. The key, however, is to analyze the percentage of requests that fall into this category. The 80-20 rule is usually a good practice to follow to determine whether you are experiencing the architecture sinkhole anti-pattern. Typically, about 20 percent of requests are simple pass-through processing and 80 percent of requests have business logic associated with the request. However, if you find that this rate is reversed, and most of the requests are simple permeable processing, you might want to consider keeping some architecture layers open, bearing in mind that it will be harder to verify change due to the lack of layer isolation. Another aspect of the layered architecture pattern is that it tends to be suitable for monolithic applications, even if the presentation layer and business layers can be installed separately into units. While this may not be a problem for some applications, it poses some potential problems in terms of installation, overall robustness and reliability, performance, and scalability. The following table describes the classification and analysis of common architecture characteristics of the layered architecture pattern. The classification of each characteristic is based on the ability to characterise a particular characteristic as a capability and what the sample is generally known for. For a side-by-side comparison of your relationship to other patterns in the report, see Summary of Sample Analysis at the end of the report. General agility rating: Low analysis: General agility is the ability to respond quickly to ever-changing environments. While the change can be isolated through isolation layers, this pattern is still cumbersome and time consuming to modify this architecture pattern because of the monolithic nature of most implementations, as well as the close interconnection of components usually found in this pattern. Simple deployment assessment: Low analysis: Depending on how you implement this pattern, deployment can cause problems, especially for larger applications. A small change in a component may require the entire application (or most of the application), leading to deployments that need to be planned, scheduled and executed outside of business hours or on weekends. Pattern is not easily suited to a continuous delivery process, further reducing the overall classification of the installation. Testability Rating: High Analysis: Because components belong to certain layers of architecture, other layers can be mocked or truncated, making this pattern relatively easy to test. Developers can mock the presentation component or screen to isolate testing within the business component, as well as the business layer, to test certain screen features. Benchmarking: Low analysis: While it's true that some layered architectures perform well, the pattern is not suitable for high-performance applications because you need to go through multiple layers of the architecture to complete a business request. Scalability rating: Low analysis: Due to the trend toward closely connected and monolithic implementations of the pattern, applications built using such an architecture pattern tend to be difficult to scale. You can scale a layered architecture by splitting layers into separate physical deployments or replicating the entire application to multiple nodes, but overall the detail is too wide, making scaling expensive. Easy development evaluation: High Analysis: Easy development gets a relatively high score, mainly because this pattern is so well known and not too complicated to implement. Because most companies develop applications by separating skill sets by layer (presentation, business, database), this pattern becomes a natural choice for most business application development. The relationship between the company's communication and organizational structure and the way the software is developing is called the Conway Act. You can Google Conway's law to get more information about this fascinating correlation. The event-driven architecture pattern is a popular distributed asynchronous architecture pattern that is used to produce highly scalable applications. It is also very adaptable and can be used for small applications and large, complex as well. The event-driven architecture consists of highly decoumed, one-purpose event processing components that receive and process events asynchronously. The event-driven architecture pattern consists of two main topologies, the mediator and the Mediator topology is usually if several steps need to be orchestrated within an event through a central broker, while the broker topology is used when you want to chain events without using a central intermediary. Since the architecture characteristics and execution strategies differ between the two topologies, it is important to understand which is most appropriate for the given situation. An intermediary topology is useful for events that take several steps and require some level of orchestration to process the event. For example, a single event to cover a stock trade may require you to first validate the trade, then check the compliance of that stock trade against the various compliance rules, assign the trade to a broker, calculate the commission, and finally place the trade with that broker. All these steps provide some level of orchestration to determine the sequence of steps and which can be done serially and in parallel. Within the broker topology, there are four main architecture components: event-by-event, event mediator, event feeds, and event processors. The event process begins with the client sending an event to an event request to transfer the event to the media. The event broadcaster receives the initial event and orchestrats the event by sending additional asynchronous events to the event channels to complete each step of the process. Event processors that monitor event channels, receive the event from the event broker, and execute a specific business logic to process the event. Figure 2-1 illustrates the overall mediator topology of the event-driven architecture pattern. Figure 2-1. Event-driven architecture topology It's common for an event-driven architecture to be anywhere from a dozen to hundreds of event look-aways. The sample does not specify the implementation of the event-by-event component; this can be a message queue, a web service endpoint, or any combination of these. Within the example, there are two types of events: an initial event and a processing event. The initial event is the original event received by the intermediary, while the processing events are those generated by the intermediary and received by the event processing components. The event broadcast component is responsible for orchestrating the steps in the initial event. For each step of the initial event, the event broadcaster sends a specific processing event to an event feed, which is then received and processed by the event processor. It is important to note that the event broker does not actually execute the business logic required to process the initial event; rather, you are familiar with the steps needed to process the initial event. Event feeds are used by the event mediating to asynchronously pass specific processing events related to each step of the initial event to the event processors. Az message lines or message topics, although topics are most widely used with the mediator topology, so that processing events can be processed by multiple event processors (each referred to as a different task based on the processing event received). The components of the event processor contain the application business logic required to process the processing event. Event processors are standalone, independent, highly independent architecture components that perform a specific task in the application or system. Although the detail of the event processing component may vary between detailed (e.g. calculation of order sales tax) and rough granular (e.g. insurance claim processing), it is important to keep in mind that in general, each event processing component must perform a single business task and not rely on other event processors to perform that task. The event broker can be implemented in a variety of ways. As an architect, you need to understand these implementation options in order for the solution selected for the event broadcaster to meet your needs and requirements. The simplest and most common implementation of event broadcast is through open source integration centers such as spring integration, Apache Camel, and Mule ESB. In these open source integration centers, event processes typically take place through Java code or DSL (domain-specific language). For more sophisticated mediation and orchestration, you can use BPEL (business process execution language) for which you can use a BPEL engine, such as open source Apache ODE. BPEL is a standard XML-like language that describes the data and steps needed to process initial events. With very large applications requiring much more sophisticated orchestration (including steps involving human interactions), you can perform the event broker using a business process manager (BPM) like jBPM. Understanding needs and aligning the right event broadcast with an implementation is essential to the success of the event-driven architecture with this topology. Using an open source integration center for very complex business process management orchestration operations is a recipe for failure, as is implementing a BPM solution to implement simple routing logic. To illustrate how the intermediary topology works, let's say you are insured by an insurance company and decide to move. In this case, the initial event can be called something like a move event. The steps for processing the move event are included in the event broadcast in steps 2 through 2. For each initial event step, the event mediate a processing event (e.g. address change, recalc quote, etc.), send the processing event to the event feed, and wait for the processing event to be processed by the appropriate event processor (e.g. offer process, etc.). This process continues until all steps in the initial event are processed. Each bar is a recalc quote and update i intermediary indicates that these steps can be run at the same time. The broker topology differs from the broker topology in that there is no central event broker; rather, message flow is chained between event processor components through a lightweight message broker (e.g. ActiveMQ, HornetQ, etc.). This topology is useful if you have a relatively simple event processing process and do not want (or do not need) central event orchestration. Broker topology has two main architecture components: a broker component and an event processor component. The broker component can be centralized or consolidated and includes all event feeds that are used within the event process. Event feeds in the broker component can be queues, message topics, or a combination of both. Figure 2-2. Mediator topology example This topology is from 2 to 3. As shown in the chart, there is no central event mediator component that controls and orchestrats the initial event; instead, each event processor component is responsible for processing an event and publishing a new event that indicates the action being taken. For example, an event processor that balances a stock portfolio can receive an initial event, a share split. Based on the initial event, the event processor can perform some portfolio rebalance and then publish a new event to the broker, the so-called rebalance portfolio, which is taken up by another event processor. Note that an event processor may publish an event but not be recorded by another event processor. This is common when you're developing an app or providing future features and extensions. Figure 2-3. Event-driven architecture broker topology To illustrate the functioning of broker topology, we will use the same example as the broker topology (an insured person moves). Since there is no central event broker to receive the initial event in the broker topology, the client process component directly accepts the event, changes the client's address, and sends out an event that says it has changed the client's address (e.g. change address event). In this example, there are two event processors who are interested in the change address event: the bid process and the claim process. The offer processor component recalculates new automatic insurance premiums based on the address change and publishes an event to other system members indicating what they've done (e.g. recalc quote event). The claims component, on the other hand, receives the same change address event, but in this case updates the pending insurance claim and publishes an event to the system as an update claim event. These new events are then taken up by other event processor components and the event chain continues through the system until the initiation event is more events. Event. 2-4. Broker topology example As shown in Figure 2-4, the broker topology is all about the chain of events to perform business function. The best way to understand the broker topology is to think of it as a relay contest. In the relay race, runners hold a baton and run a certain distance, then pass the baton to the next runner, and so on through the chain until the last runner crosses the finish line. In relay races, if a runner takes his hand off the baton, he's done with the race. This is also true of the broker's topology: if an event processor surrenders its hand from the event, it is no longer involved in processing that event. The event-driven architecture pattern is a relatively complex pattern implementation, primarily because of the distributed nature of the asynchronous. When the pattern is implemented, various distributed architecture problems, such as remote process availability, lack of response time, and intermediary reconnection logic in the event of an intermediary or intermediary failure. When selecting the architecture pattern, you must consider the lack of atomic transactions for a single business process. Because the event processor components are largely independent and distributed, it is very difficult to maintain a transactional work unit between them. For this reason, when designing your application with this pattern, you need to constantly think about which events can run independently of each other, and plan the detail of the event processors accordingly. If you find that you need to split a single work unit among event processors— that is, if you're using separate processors for something that should be an undivided transaction — this may not be the right pattern for your app. One of the most difficult aspects of the event-driven architecture pattern is the creation, maintenance, and management of event processor component contracts. Each event is usually linked to a specific contract (e.g. data values and data format to the event processor). It is extremely important to use this pattern to balance a standard data format (e.XML, JSON, Java Object, etc.) and to create a contract version policy from the start. The following table describes the classification and analysis of common architecture characteristics for the event-driven architecture pattern. The classification of each characteristic is based on the ability to characterise a particular characteristic as a capability and what the sample is generally known for. For a side-by-side comparison of your relationship to other patterns in the report, see Summary of Sample Analysis at the end of the report. General agility rating: High analysis: General agility is the ability to react quickly in an ever-changing environment. Because the event processor components are single-purpose and completely independent of other events, they are usually isolated to one or a few event processors and can be quickly put away without impacting on other components. Easy deployment assessment: High analysis: Overall, this pattern is relatively easy to deploy due to the decouned nature of the event-processor components. The broker topology is generally easier to install than the broker topology, primarily because the event broker component is somewhat closely related to the event processors: modifying an event processor component can necessitate a change in the event broker, which requires both the installed specific change. Testability Rating: Low Analysis: Although testing individual units is not too difficult, it does not require some kind of advanced test client or test tool to generate events. Testing is also complicated by the asynchronous nature of this test. Benchmarking: High analysis: While it is certainly possible to implement an event-driven architecture that is not performing well due to all the messaging infrastructure involved, usually the pattern achieves high performance through its asynchronous capabilities; in other words, the ability to perform decouned, parallel asynchronous operations exceeds the cost of queueing and dequeuing messages. Scalability rating: High analysis: Scalability is naturally available in this example through highly independent and independent event processors. Each event processor can be scaled separately for fine-grained scalability. Easy development rating: Low analysis: Development can be somewhat complicated due to the asynchronous nature of the pattern and the need for contract creation and more advanced error handling conditions within the code of unres responding processors and failed brokers. The microkernel architecture pattern (also known as the plug-in architecture pattern) is a natural pattern for product-based applications. A product-based app is an app that is packaged and downloaded in versions as a typical third-party product. However, many companies also develop and release internal business applications like software products, complete with versions, release notes, and pluggable features. They also have a natural fit to this example. The microkernel architecture pattern allows you to add additional application functionality as a plug-in for the core application, ensuring extensibility and isolation and isolation of features. The microkernel architecture pattern consists of two architecture components: a central system and a plug-in module. Application logic is shared between independent plug-ins and the core core system, providing extensibility, flexibility, and separation between application functionality and custom processing logic. Article 3-1 shall be replaced by the following: Central system of the microkernel architecture pattern contains only the minimum functionality to make the system work. Many operating systems implement the microkernel architecture pattern, hence the origin of the sample name. From a business application perspective, the core system can often be defined as general business logic sans custom code in special cases, special rules, or complex conditional processing. Figure 3-1. Microkernel architecture pattern Plug-ins are standalone, independent components that include advanced processing, additional functionality, and custom code designed to improve or expand the core system to create additional business capabilities. Typically, the plug-in must be independent of other snap-ins, but you can certainly design snap-ins that have other snap-ins present. Either way, it's important to minimize communication between plug-ins to avoid dependency issues. The central system needs to know which plug-in modules are available and how to get to them. One common way of implementing it is through some kind of plug-in registry. This registry contains information about modules in each snap-in, such as its name, data contract, and remote access protocol details (depending on how the plug-in connects to the central system). For example, a tax software snap-in for high-risk tax journaling items might have a registry entry that contains the service name (AuditChecker), the data contract (input data and output data), and the contract format (XML). WsDL (web service definition language) can be included if the plug-in is available through SOAP. Plug-in modules can be connected to the central system in a number of ways, including OSGi (open service gateway initiative), messaging, web services, or even direct point-to-point binding (i.e. object instantiation). The type of connection you use depends on the type of application you're using (small product or enterprise application) and specific needs (e.g. a single deployment or distributed deployment). The architecture pattern itself does not specify these implementation details, only that plug-in modules must remain independent of each other. Contracts between plug-in modules and the central system can range from standard contracts to individual contracts. Custom contracts are usually found in situations where the components of the plug-in are developed by a third party, where it has no control over the contract used by the plug-in. In such cases, it is common to create an adapter between the plug-in contact and the standard contract so that the central system does not need special code for each snap-in. When creating standard contracts (usually via XML or Java map) it's important to remember to create your versioning strategy from the start. Perhaps the best example of the microkernel architecture is the Eclipse IDE. Downloading the basic Eclipse product is little more than a fashionable editor. However, once you have adding plug-ins, it will be a highly customizable and useful product. Internet browsers are another common product example using the microkernel architecture: viewers and other plug-ins add additional capabilities that would not otherwise be found in the basic browser (i.e. the base system). The examples are endless for product-based software, but what about big business applications? The microkernel architecture also applies to these situations. To illustrate this, we use the example of another insurance company, but this time by processing insurance claims. Processing claims is a very complicated process. Each state has different rules and regulations on what is allowed and what is not allowed on the insurance application. For example, some states allow free windshield replacement if the windshield is damaged by a rock, while other states do not. This creates almost infinite conditions for the standard claims management process. Not surprisingly, most insurance claims applications leverage large and complex rules for engines to deal with many of these complexity. However, these rules engines can grow into a complex large ball of mud, where changing one rule affects other rules, or that a simple rule change requires a host of analysts, developers, and testers. Using the microkernel architecture pattern can solve many of these problems. The following shall be replaced by the following: This includes the basic business logic required by the insurer to process the claim, unless no individual processing is required. Each plug-in contains specific rules for that state. In this example, plug-ins can be implemented using custom source code song or separate rule engine instances. Regardless of implementation, the most important point is that state-specific rules and processing are separate from the basic claims system and can be added, removed, and changed with little or no impact on other basic systems or other plug-in modules. Figure 3-2. Microkernel architecture example A great thing about a microkernel architecture pattern is that it can be embedded or used as part of another architecture pattern. For example, if this pattern solves an issue with a specific volatile area of your application, you may not be in a state of complete architecture with this pattern. In this case, you can embed the microservices architecture pattern using a different pattern (for example, layered architecture). Similarly, the event processor components described in the previous section on event-driven architecture can be implemented using the microservices architecture. The pattern of microservices architecture provides great support for evolutionary design and incremental development. First, you can create a solid core system and the application evolves gradually, and add features without having to make significant changes to the base system. Product-based Product-based the microkernel architecture pattern is always your first choice as a novice architecture, especially those products where you will release additional features over time and want to check which users get which features. If over time you find that the pattern does not meet all requirements, you can at any time refactor your application to another architecture pattern to better meet your specific requirements. The following table describes the classification and analysis of common architecture characteristics of the microkernel architecture pattern. The classification of each characteristic is based on the ability to characterise a particular characteristic as a capability and what the sample is generally known for. For a side-by-side comparison of your relationship to other patterns in the report, see Summary of Sample Analysis at the end of the report. General agility rating: High analysis: General agility is the ability to react quickly in an ever-changing environment. The modifications are largely separable and can be implemented quickly through loosely connected plug-in modules. In general, the central system of most microkernel architectures quickly becomes stable and, as such, quite robust and requires little change over time. Easy deployment assessment: High analysis Depending on how the sample is implemented, plug-in modules can be dynamically added to the central system at runtime (e.g. at deployment), minimizing uptime during deployment. Testability Rating: High Analysis: Plug-in modules can be tested in isolation and can easily be mocked by the central system to prove or prototype a particular feature with little or no change to the central system. Benchmarking: High analysis: While the microkernel pattern is of course not suitable for high-performance applications, usually most applications made with a microkernel architecture pattern perform well as applications only include the necessary features. JBoss Application Server is a good example of this: with the plug-in architecture, you can cut the application server down to the features you need, removing expensive, unused features such as remote access, messaging and memory, caching using CPU and threads, and slowing down the application server. Scalability rating: Low analysis: Because most microkernel architecture implementations are product-based and generally smaller in size, they are implemented as a single unit and cannot be scaled. Depending on how you implement plug-ins, you can sometimes provide scalability at the plug-in functionality level, but overall this pattern is not known for producing high-scalable applications. Easy development assessment: Low analysis: The microkernel architecture requires thoughtful design and contract management, making it quite complicated to implement. Contract versioning, Plug-in records, plug-in granularity, and the wide selection available for plug-in connectivity all contribute to the complexity of implementing this pattern. The pattern of microservices architecture is rapidly gaining ground in the industry as a viable alternative to monolithic applications and service-oriented architectures. As this architecture pattern is still evolving, there is a lot of confusion in the industry about what this pattern is all about and how it is implemented. This section will report to you the most important concepts and foundational knowledge needed to understand the benefits (and trade-offs) of this important architecture pattern and to provide this appropriate pattern for your application. Regardless of the topological or implementation style you choose, there are many general basic terms for the general architecture. The first of these concepts is the concept of separately installed units. As shown in Figure 4-1, each component of the microservices architecture is deployed as a separate unit, allowing for a high degree of application and component disconnection through an efficient and simplified delivery process, greater scalability, and in-app disconnection. Perhaps the most important concept to understand this pattern is the concept of the service component. Instead of services within a microservice architecture, you might want to think of service components that can vary between a large part of a single module in detail. Service components include one or more modules (e.g. Java classes) that represent either a one-purpose function (e.g. providing weather for a particular city or city) or an independent part of a large business application (e.g. listing or automatic insurance premiums). Designing the level of detail of the appropriate service component is one of the biggest challenges in the microservices architecture. This challenge is discussed in more detail in the following service component orchestration subsection. Figure 4-1. Basic microservices architecture pattern Another key concept of the microservices architecture pattern is that it is a distributed architecture, which means that all components of the architecture are completely disconnected from another and can be accessed through some kind of remote access protocol (e.g. JMS, AMQP, REST, SOAP, RMI, etc.). The distributed nature of the architecture pattern is how it achieves some excellent scalability and deployment characteristics. One of the exciting things about the architecture of microservices is that it evolved from problems with other common architecture patterns, rather than being created as a solution waiting for a problem to arise. Of course, the architecture style of microservices has evolved from two main sources: monolithic applications developed with a pattern of layered architecture and distributed applications through the design of the service-oriented architecture. The evolutionary path from monolithic applications to the style of microservice architecture is primarily through the development of continuous delivery, the concept of a continuous deployment process from development to production, which simplifies application deployment. Monolithic applications typically consist of tightly connected components that are part of a single installable unit, making it cumbersome and cumbersome to modify, test, and deploy the application (which is why the increase in common monthly deployment cycles is typically found in most large IT stores). These factors usually lead to fragile applications that are interrupted every time something new is installed. The microservices architecture pattern solves these problems by separating the application into multiple deployable units (service components) that can be developed, tested, and deployed separately, regardless of other service components. The other evolutionary pathway that is found in the microservices architecture pattern is the service-oriented architecture pattern (SOA) implementation of the problems. While the SOA pattern is very powerful and has an unparalleled level of abstraction, heterogeneous connectivity, service orchestration, and the promise of aligning business goals with IT capabilities, it is nonetheless complicated, expensive, ubiquitous, difficult to understand and implement, and generally an exaggeration of most applications. The microservices architecture style manages this complexity by simplifying the concept of service, eliminating orchestration needs, and simplifying connectivity and access to service components. Although there are literally dozens of ways to implement the microservices architecture pattern, three main topologies stand out as the most common and popular: the API REST-based topology, application REST-based topology, and centralized messaging topology. API REST-based topology is useful for websites that offer small, standalone custom services through an API (application programming interface). This topology, which is used in the 4-2. In this topology, these detailed service components are typically available using a REST-based interface implemented through a separate deployed web-based API layer. Examples of topology include some of the one-purpose cloud-based RESTful web services found by Yahoo, Google, and Amazon. Figure 4-2 API REST-based topology The REST-based topology of the application differs from the API REST approach because client requests are received through traditional web-based or fat client business application screens, not through a simple API layer. As shown in The 4-3. It is installed as a separate Web application that remotely accesses separately deployed service components (business features) through simple REST-based interfaces. In this topology, the service components differ from those in the API-REST-based topology in that these service components tend to be larger, rougher-eyed, and represent a small portion of the entire business application rather than detailed, one-step services. This topology is common in small and medium-sized business applications that have a relatively low degree of complexity. Figure 4-3. Application REST-based topology Another common approach to the architecture of microservices is centralized message topology. This topology (shown in Figure 4-4) is similar to the previous REST-based topology, with the difference that instead of using REST for remote access, this topology uses a lightweight centralized message broker (e.g. ActiveMQ, HornetQ, etc.). It is extremely important when looking at this topology not to confuse the service-oriented architecture pattern or consider it SOA-Lite. The topology located in the lightweight message broker does not perform orchestration, transformation, or complex routing; rather, it's just an easy delivery to access remote service components. Centralized message topology is typically found in larger business applications or applications that require more sophisticated control of the transport layer between the UI and service components. The advantages of this topology over the simple REST-based topology discussed earlier are advanced queuing mechanisms, asynchronous messaging, monitoring, error handling, and improved overall load balancing and scalability. The single point of failure and architectural bottleneck problems are usually a centralized broker in relation to broker clustering and broker consolidation (splitting a single broker instance into multiple intermediary instances to divide the message throughput load into functional areas based on the system). Figure 4-4. Centralized message topology One of the main challenges of the microservices architecture pattern is to determine the appropriate level of detail for service components. If the service components are too rough-grained, you may not notice the benefits of this architecture pattern (deployment, scalability, testability, and loose connector). However, service components that are too fine-grained lead to service orchestration requirements that quickly turn the lean microservices architecture into a heavyweight service-oriented architecture, complemented by all the complexity, confusion, cost, and fluff typically found in SOA-based applications. If you find that you need to orchestrat service components from the application UI or API layer, it's likely that the service components are too Similarly, if you find that communication between service components to process a single request, it is likely that the service components are too detailed or are not properly distributed from a business point of view. Inter-service communication that can enforce unwanted interconnection between components can instead be handled through a shared database. For example, if a component that services Internet orders needs to provide customer information, it can go to the database to retrieve the necessary data, as opposed to inviting functionality within the customer service component. A shared database can handle information needs, but what about shared features? If a service component needs features in another service component, or if all service components are common, it can sometimes copy shared functionality between service components (thus violating the DRY principle: do not repeat yourself). It's a fairly common practice for most business applications to implement the microservices architecture pattern, commercializing down redundancy to a small part of business logic in order to deploy service components independent and separating them. Small utility classes can belong to this recurring code category. If you find that regardless of the level of detail of the service component, you still can't avoid orchestrating the service component, then it's a good sign that this may not be the right architecture pattern for your app. Because of the distributed nature of the sample, it is very difficult to maintain a single transactional work unit between (and) service components. Such a practice would require some kind of transaction suspension framework to roll back transactions, which gives considerable complexity to this relatively simple and elegant architectural design. The microservice architecture pattern solves many common problems found in both monolithic applications and service-oriented architectures. Because the main application components are divided into smaller, separate deployed units, applications created with the microservices architecture tend to be more robust, provide better scalability, and support continuous delivery more easily. Another advantage of this example is that it allows you to build real-time production deployments, significantly reducing the need for traditional monthly or weekend big bang production environments. Because the change is typically isolated to specific service components, only service components must be installed. If you have only one instance of a service component, you can write special code to the UI application to detect active quick installation and redirect users to an error page or queue. Alternatively, you can replace multiple instances of a service component with the real during installation, allowing continuous availability during commissioning (something that is very difficult to do with the layered architecture pattern). One final consideration to note is that because the microservices architecture pattern is a distributed architecture, it shares some of the same complex issues found in the event-driven architecture pattern, including contract creation, maintenance, and government, remote system availability, and remote access authentication and authorization. The following table describes the classification and analysis of a common architecture pattern for the microservices architecture. The classification of each characteristic is based on the ability to characterise a particular characteristic as a capability and what the sample is generally known for. For a side-by-side comparison of your relationship to other patterns in the report, see Summary of Sample Analysis at the end of the report. General agility rating: High analysis: General agility is the ability to react quickly in an ever-changing environment. Because of the concept of separate provisioned units, the change is typically isolated from each service component, which allows for quick and easy deployment. Also, applications to build with this pattern are usually very loosely connected, which also facilitates change. Simple deployment assessment: High analysis: The deployment characteristics of the microservices sample ratio are very high due to the detailed and independent nature of remote services. Services are usually installed as separate software units, which means hot installations can be performed during the day or at night at any time. The overall deployment risk is also significantly reduced because failed deployments can be restored more quickly and only affect the operations on the deployed service, resulting in continuous operations for all other operations. Testability Rating: High Analysis: By separating and segregating business functions into independent applications, the scope of testing can extend, enabling more targeted testing efforts. Regression testing of a service component is much easier and more feasible than regression testing of a complete monolithic application. Also, since the components of the service in this sample are loosely interconnected, there is much less chance from a development point of view that the change that breaks another part of the application, facilitating the test burden of having to test the entire application for a small change. Benchmarking: Low analysis: Although you can build applications from this sample that perform very well, overall, this pattern is not suitable for the distributed nature of the microservices architecture. Scalability rating: High analysis: Because the application is divided into separate deployed units, each service component is individually allowing fine-tuned fine-tuned submission of the application. For example, the supervisory area of a stock market trading application may not need to scale due to the low user volume of that feature, but the trading placement service component may need to scale, as most trading applications need to have high throughput for this feature. Easy development assessment: High Analysis: Because functionality is isolated into separate and separate service components, development becomes easier due to smaller and isolated scope. There is a much lower chance that the developer will make a change to one of the service components that would affect other service components, thereby reducing coordination between developers or development teams. Most Web-based business applications follow the same general request process: a request from the browser reaches the Web server, then the application server, and finally the database server. Although this pattern works great for a small group of users, bottlenecks begin to appear as user load increases, first in the Web server layer, then in the application server layer, and then in the database server layer. The usual response to bottlenecks based on an increase in user load is to scale out web servers. It is relatively simple and inexpensive and sometimes works to address bottleneck problems. However, in most cases, scaling out the web server layer only moves the bottleneck down to the application server. Scaling application servers can be more complex and expensive than Web servers, and usually only moves the bottleneck to the database server, which is even more difficult and expensive to scale. Even if the database is scaled to what ends up being a triangular topology, the widest part of the triangle is that of web servers (most easily scalable) and the smallest part of that database (heaviest scale). For each large amount of applications with extremely high concurrent user load, the database is usually the ultimate limiting factor in how many transactions can be processed simultaneously. While various caching technologies and database sizing products help solve these problems, the fact remains that horizontal scaling out extreme loads in a normal application is a very difficult proposition. The spatial architecture pattern is specifically designed to solve and resolve scalability and concurrity issues. It is also a useful architecture for sample applications that have

variable and unforeseen concurrent user volumes. Architectural solutions to extreme and variable scalability problems are often a better approach than scaling up a database or caching technologies are not retrofitting it to an architecture. The spatial pattern (also known as the cloud architecture pattern) minimizes the factors that limit your app's scaling. This pattern gets its name from the concept of tuple space, the idea of distributed shared memory. High scalability is achieved by removing the central database and use in-memory data grids. Application data remains in memory and replicates across all active processing units. Processing units can be dynamically started and stopped as user load increases and decreases, thereby handling variable scalability. Because there is no central database, the database bottleneck is removed, so it can be scaled almost infinitely within the application. Most of the apps that fit this example are standard websites that receive a request from the browser and perform some action. A good example of this is the bidding auction site. The site continuously receives offers from internet users through a browser request. The app would receive an offer for a specific item, record the offer with a timestamp, update the latest offer information for the item, and return the information to the browser. Within the architecture pattern, there are two primary components: a processing unit and virtualized intermediate software. Article 5-1 shall be replaced by the following: The processing unit component contains the application components (or parts of the application components). This includes web-based components as well as back-end business logic. The content of the processing unit depends on the type of application — smaller web-based applications are likely to be placed in a single processor, while larger applications can divide the functionality of the application into multiple assembling units based on the functional areas of the application. Typically, the processing unit includes application modules, an in-memory data grid, and an optional asynchronous persistent container for failover. It also includes a replication engine that virtualized intermediate software uses to replicate data changes made by one processing unit to other active processing units. Figure 5-1. Space-based architecture pattern The virtualized-middleware component handles cleaning and communication. It contains components that control different aspects of data synchronization and request handling. Virtualized intermediate software includes the messaging grid, data grid, processing grid, and installation manager. These components, which are described in detail in the next section, can be individually written or purchased as third-party products. The magic of the space-based architecture pattern lies in the virtualized middleware components and the memory data grid in each processing unit. Figure 5-2 shows the typical processing unit architecture, which includes application modules, in-memory data grid, optional asynchronous retention storage failover, and data replication engine. Virtualized intermediate software is essentially the architecture controller and handles requests, sessions, data replication, distributed request processing, and process unit deployment. Virtualized component of four main architecture components message grid, data grid, processing grid, and setup manager. Figure 5-2. Processing unit component In the case of products 5 to 3, the following shall be When a request is received in the virtualized-middleware component, the messaging grid component determines which inward processing components are available to receive the request and forwards the request to one of these processing units. The complexity of the messaging grid can range from a simple round robin algorithm to a more complex, next available algorithm that tracks which request is being processed by which processing unit. The data grid component is perhaps the most important and important element of this design. The data grid, in cooperation with the data replication engine of each processing unit, manages data replication between processing units when data updates occur. Because the messaging grid can forward the request to any available processing unit, it is essential that all processing units contain exactly the same data in the in-memory data grid. Although the 5-4th EDD is not a good way to do so, it is not the case that Figure 5-3. Messaging grid component 5-4. Data grid component In figure 5-5, the data grid component is used to create a data grid component. If a request is received that requires coordination between the processing unit types (e.g. an order processing unit and a customer processing unit), the processing network mediates and conducts the request between the two processing units. Figure 5-5. Processing grid component The Deployment Manager component handles dynamic start-up and shutdown of processing units based on load conditions. This component continuously monitors response time and user loading, and starts new processing units when the load increases and stops processing units when the load decreases. This is a critical component of achieving variable scalability needs within an application. The space-based architecture pattern is a complex and expensive pattern to perform. This is a good architecture choice for smaller web-based applications with variable loads (e.g. social media sites, bidding and auction sites). However, it is not suitable for large traditional relational database applications with large amounts of operational data. Although the spatial architecture pattern does not require a centralized data store, one is typically included to perform initial in-memory data grid loading and asynchronously retain data updates performed by processing units. It is also common practice to create separate partitions that and widely used transactional data from non-active data in order to reduce the memory footprint of the in-memory data grid in each processing unit. It is important to note that while the alternative name for this model is cloud architecture, processing units (as well as virtualized middleware) should not reside in cloud-based services or PaaS (as a platform service). It is just as easy to reside on local servers, which is one of the reasons i prefer the name space-based architecture. For product implementation, you can implement many of the architectural components of the sample through third-party products, such as GemFire, JavaSpaces, GigaSpaces, IBM Object Grid, nCache, and Oracle Coherence. Since the pattern implementation varies greatly in cost and capabilities (especially data replication time) as an architect, you must first determine what your specific goals and needs are before making any product choices. The following table describes the classification and analysis of common architecture characteristics for the spatial architecture pattern. The classification of each characteristic is based on the ability to characterise a particular characteristic as a capability and what the sample is generally known for. For a side-by-side comparison of your relationship to other patterns in the report, see Summary of Sample Analysis at the end of the report. General agility rating: High analysis: General agility is the ability to react quickly in an ever-changing environment. Because processing units (deployed instances of the application) can be folded up and down quickly, applications respond well to changes in user load (environmental changes). The architectures created with this pattern generally respond well to encoding changes due to the small application size and dynamic nature of the pattern. Easy deployment assessment: High analytics: Although spatial architectures tend not to be independent or distributed, they're dynamic, and sophisticated cloud-based tools allow applications to easily push to servers, simplifying deployment. Testability Rating: Low Analysis: Achieving very high user loads in a test environment is expensive and time-consuming, making it difficult to test the scalability aspects of your app. Benchmarking: High analytics: High performance can be achieved through in-memory data access and through analyting caching mechanisms. Scalability rating: High analysis: High scalability results from the fact that there is little or no dependency on a central database, so it essentially removes this restrictive bottleneck from the scalability equation. Easy development assessment: Low analysis: Sophisticated cache and in-memory data grid products make this pattern relatively complex to develop, because the knowledge of the tools and products used to create this type of architecture. In addition, particular attention should be paid to the development of these types of architectures, so that nothing in the source code affects performance and scalability. Scalability.