Les modèles d'architecture logicielle compréhension et comment les utiliser

Mark Richards T

Bien que l'éditeur et l'auteur ont utilisé des efforts de bonne foi pour s'assurer que les renseignements et les instructions contenues dans ce travail sont exacts, l'éditeur et l'auteur déclinent toute responsabilité pour les erreurs ou omissions, sans limitation responsabilité pour dommages résultant de l'utilisation de ce travail.

L'utilisation de l'information et les instructions contenues dans ce travail est à vos propres risques. Si l'un des exemples de code ou d'autres technologies cette œuvre contient ou décrit est l'objet de licences open source ou les droits de propriété intellectuelle des tiers, il est de votre responsabilité de veiller à ce que votre utilisation de celui-ci est conforme à ces licences

Table des matières

Introduction

Dans plusieurs cas les développeurs commencent à coder une application sans aucun modèle d'architecture formelle en place. En l'absence d'un modèle d'architecture claire et bien définie la plupart des développeurs auront recours au traditionnel modèle d'architecture en couches(n-tiers), créant des couches implicites en séparant les modules de code dans des paquets. Malheureusement, quoi souvent, les résultats de cette pratique est une collection de code en module non organisé qui manquent de rôles, de responsabilités et de relations claires entre eux. Ceci est communément appelé l'anti-pattern "grosse boule de boue".

Les applications dépourvues d'architecture formelle sont généralement étroitement couplées, fragiles, difficiles à changer, et sans une vision claire ou une direction précise. En conséquence, il est très difficile de déterminer les caractéristiques architecturales de l'application sans comprendre le fonctionnement de chaque composant et module dans le système.

Les questions de base sur le déploiement et la maintenance sont difficiles à répondre. L'architecture évolue-t-elle? Quelles sont les caractéristiques de performance de l'application? Avec quelle facilité l'application va-t-elle répondre au changement? Quelles sont les caractéristiques de déploiement de l’application? Quelle est la réactivité de l'architecture?

Les modèles d'architecture aident à définir les caractéristiques de base et le comportement d'une application. Par exemple, certains modèles d'architecture se prêtent naturellement à des applications hautement évolutives, tandis que d'autres modèles d'architecture se prêtent naturellement vers des applications très agiles. Connaître la caractéristique les tics, les forces et les faiblesses de chaque modèle d'architecture sont nécessaires afin de choisir celui qui répond à vos besoins et objectifs. En tant qu'architecte, vous devez toujours justifier vos décisions d'architecture, particulièrement quand il s'agit de choisir un modèle ou une approche.

Dans ce document nous allons étudier cinq modèles d'architecture en élaborant leurs forces et leurs faiblesses. Ce qui vous permettra d'être en mesure de bien choisir celui qui s'apparente le mieux aux systèmes informatiques que vous mettrez en place.

CHAPITRE 1

Architecture en couches

Le modèle le plus courant est celui de l'architecture en couches, aussi connu sous le nom d'architecture n-tiers. Ce modèle est le standard de facto pour la plupart des applications Java EE et est donc largement connu de la plupart des architectes, designers, et développeurs. Le modèle d'architecture en couches correspond étroitement aux traditionnelles communications IT et aux structures organisationnelles de la plupart des entreprises, ce qui en fait un choix naturel pour la plupart des applications commerciales.

Description du modèle en couches

Les composants du modèle d'architecture en couches sont organisés en couches horizontales, chaque couche jouant un rôle spécifique dans l'application (par exemple, la logique de présentation ou la logique du domaine d'affaires). Bien que le modèle d'architecture en couches ne spécifie pas le nombre et type de couches qui doivent exister, la plupart des méthodes de travail se composent de quatre couches standard:

Présentation,
Domaine d'affaires
Persistance
Base de données

Dans certains cas, la couche du domaine d'affaires et la couche de persistance sont combinées en une couche unique particulièrement lorsque la logique de persistance (par exemple, SQL ou HSQL) est intégrée dans les composants de la couche de la couche du domaine d'affaires. Ainsi, les plus petites applications peuvent avoir seulement trois couches, alors que les plus grandes et plus complexes peuvent contenir cinq couches, voire même plus.

Chaque couche de ce modèle d'architecture a un rôle spécifique et une responsabilité dans l'application. Par exemple, la couche de présentation est responsable de la gestion de toute la logique de communication de l'interface utilisateur, alors que la couche du domaine d'affaires est responsable de l'exécution de règles spécifiques associées au domaine de l'application. Chaque couche de l'architecture forme une abstraction autour du travail qui doit être fait pour satisfaire un besoin particulier. Par exemple, la couche de présentation n'a pas besoin de savoir

ou de s'inquiéter de la façon d'obtenir les données pour l'interface usager; il a seulement besoin d'afficher cette information sur un écran au format particulier. De même, la couche du domaine d'affaires ne doit pas être préoccupée par la façon de la mise en forme des données pour l'affichage sur un écran ou même de la provenance des données. Il suffit d'obtenir les données de la couche de persistance et d'exécuter la logique du domaine d'affaires et de transmettre les résultats à la couche de présentation.
Une des caractéristiques puissantes du modèle d'architecture en couches est la séparation des préoccupations entre les composants. Les composants dans une couche spécifique ne traitent que de la logique qui appartient à cette couche. Par exemple, les composants de la couche de présentation ne traitent que de la logique de présentation, tandis que les composants résidant dans la couche du domaine d'affaires traite uniquement avec la logique métier. Ce type de classification des composants facilite la mise en place du modèle des rôles et responsabilités et par le fait même, rend également facile à développer, tester, et maintenir une application. En utilisant ce modèle d'architecture, les composants sont bien définis et on limite ainsi la portée des interfaces de ces composants.

Concepts clés

Une couche peut-être caractérisée comme étant ouverte ou fermée. Par défaut, chacune des couches de cette architecture est marquée comme étant fermée (voir Figure 1.1). C'est un concept très important dans ce modèle d'architecture. Une couche fermée signifie que comme une demande se déplace de couche en couche, elle doit passer par la couche juste en dessous pour arriver à la couche suivante. Par exemple, une demande ayant pour origine la couche de présentation doit d'abord passer par la couche du domaine d'affaires puis par la couche de persistance avant d'arriver finalement à la couche de base de données.

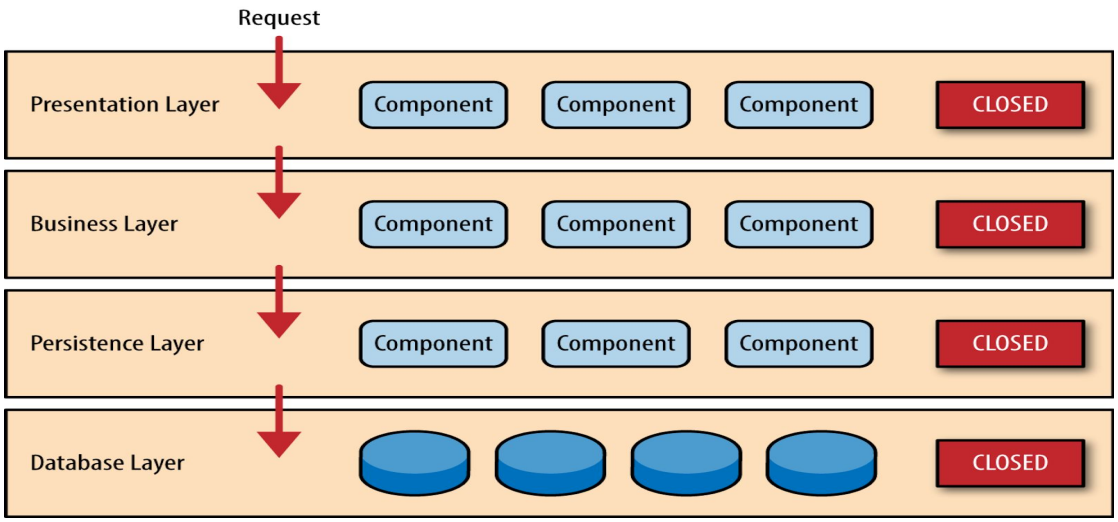

Figure 1.1

Pourquoi ne pas autoriser l'accès direct de la couche de présentation vers la couche de persistance ou la couche de base de données? Après tout, accéder directement à la base de données à partir de la couche de présentation n'est-il pas beaucoup plus rapide que de passer par un tas de couches "inutiles" juste pour récupérer ou enregistrer des informations? La réponse à cette question réside dans un concept clé connu comme couche d'isolation. Le concept d'isolation signifie que les changements effectués dans une des couches de l'architecture n'ont généralement pas d'impact et n'affecte pas les composants des autres couches. Autrement dit, le changement est isolé pour les composants dans une couche donnée, et éventuellement pour une autre couche associée (comme la couche persistance et couche contenant le SQL). Si vous autorisez la couche de présentation d'avoir directement accès à la couche de persistance, les modifications apportées au SQL dans la couche de persistance affecteraient à la fois la couche de métier et la couche de présentation, produisant ainsi une application très étroitement couplée avec beaucoup d'interdépendances entre les composants. Ce type d'architecture devient alors très difficile et coûteux à modifier.

Le concept de couches d'isolation signifie également que chaque couche est indépendante des autres couches, ayant ainsi peu ou pas de connaissance du fonctionnement interne des autres couches de l'architecture est nécessaire pour travailler à l'intérieur d'une couche. Alors que les couches fermées facilitent les couches d'isolation et donc aident à isoler le changement dans l'architecture, il y a des moments où ça peut devenir censé pour certaines couches d'être ouverte. Par exemple, supposons que vous vouliez ajouter une couche de services partagés à une architecture contenant des composants de service accessibles par des composants au sein de couche de données (par exemple, classes d'utilitaires de données et de chaînes ou audit et journalisation de classes). Créer une couche de services est généralement une bonne idée dans ce cas parce du point de vue de l'architecture, il restreint l'accès aux services partagés à la couche de gestion (et non la couche de présentation). Sans une séparation des couches, il n'y a rien au niveau de l'architecture qui limite la couche de présentation d'accéder à ces services communs, ce qui rend difficile la gestion de cette restriction d'accès.

Dans cet exemple, la nouvelle couche de services devrait être placée en dessous de la couche du domaine d'affaires. Et ce pour empêcher les composants de la couche de services d'être accessibles par la couche de présentation. Cependant, cela pose un problème en ce sens que la couche du domaine d'affaires doit maintenant passer par la couche de services pour accéder à la couche de persistance, ce qui n'a pas de sens en soi. C'est un problème séculaire avec l'architecture en couches. Il est résolu en créant des couches ouvertes au sein de l'architecture. Comme illustré dans la Figure 1-2, la couche de services dans ce cas est marquée comme ouverte, ce qui signifie que les demandes sont autorisées à contourner cette couche ouverte et aller directement à la couche suivante. Dans l'exemple suivant, puisque la couche de services est ouverte, la couche du domaine d'affaires est désormais autorisée à la contourner et allez directement à la couche de persistance, ce qui est parfaitement logique.

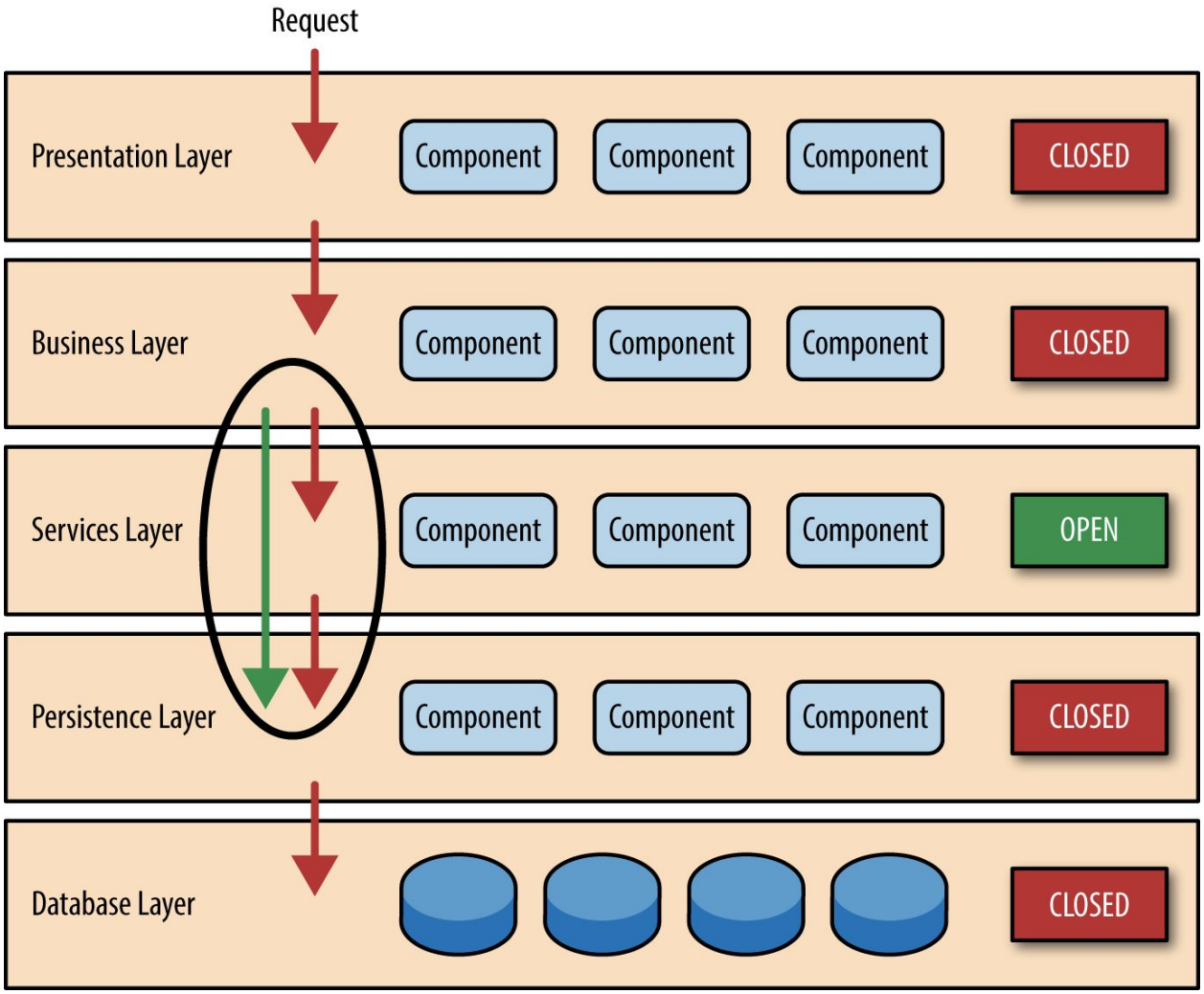

Figure 1.2

Tirer parti du concept de couches ouvertes et fermées aide à définir la relation entre les couches d'architecture et les flux de demandes et aussi permet de fournir aux concepteurs et aux développeurs les informations nécessaires pour comprendre les différentes restrictions d'accès à la couche dans l'architecture. Le problème de ne pas bien documenter ou de ne pas bien communiquer quelles couches dans l'architecture sont ouvertes et fermées (et pourquoi) se traduit généralement par des architectures fortement couplées et cassantes qui deviennent très difficiles à tester, à maintenir et à déployer.

Exemple de l’architecture en couche

Pour illustrer le fonctionnement de l'architecture en couches, considérez une requête d'un utilisateur qui veut récupérer des informations sur une commande effectuée par un client, tel qu’illustré dans la Figure 1-3. Les flèches noires montrent la requête qui descend vers la base de données pour récupérer les données du client, et les flèches rouges indiquent la réponse qui revient à l'écran pour afficher les données. Dans cet exemple, l'information se compose à la fois des données du client et des données de la commande (commandes passées par le client).

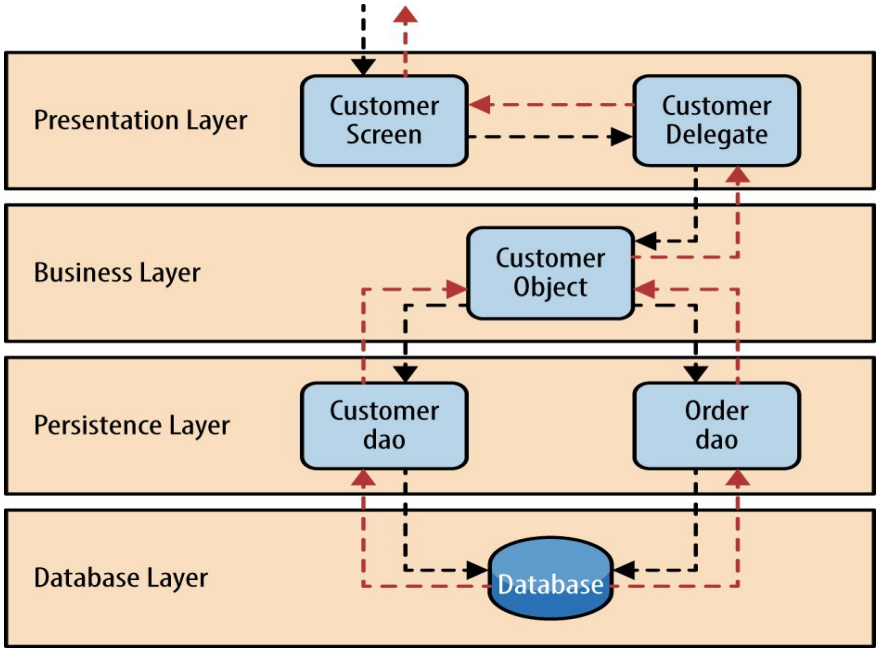

Figure 1-3

L'interface usager est responsable de l'acceptation de la demande et de l'affichage des informations. Il ne sait pas où sont les données, comment elles sont récupérées ou combien de tables de base de données doivent être interrogées pour obtenir les données. Une fois que l'interface usager reçoit la requête d’un utilisateur pour obtenir des informations sur un client particulier, il transmet cette demande à l’objet client mandaté (delegate). Cet objet est chargé de savoir quels modules de la couche du domaine d’affaires peuvent traiter cette demande, comment accéder à ce module et quelles sont les données dont il a besoin (le contrat). L'objet client dans la couche du domaine d’affaires est responsable de l'agrégation de toutes les informations nécessaires à la requête de l’utilisateur (dans ce cas, pour obtenir des informations sur la commande d’un client). Ce module appelle ensuite le module client dao (data access object) dans la couche de persistance pour obtenir les données du client, ainsi que le module dao de commande pour obtenir les informations de la commande. Ces modules exécutent à

leur tour des instructions SQL pour extraire les données correspondantes et les transmettre à l'objet client dans la couche du domaine d'affaires. Une fois que l'objet client reçoit les données, il regroupe les données et transmet ces informations au délégué du client, qui les transmet ensuite à l'interface usager pour être présenté à l'utilisateur.

D'un point de vue technologique, il existe littéralement des dizaines de façons de mettre en œuvre ces modules. Par exemple, la plate-forme Javascript, pour l'interface utilisateur, ou un client de type (JSF) Java Server Faces couplées à l'objet client mandaté (delegate) en tant que composant de bean géré. L'objet client dans la couche du domaine d'affaires peut être un bean Spring local ou un bean EJB3 distant. Les objets d'accès aux données illustrés dans l'exemple précédent peuvent être implémentés comme de simples objets POJO (Plain Old Java Objects), des fichiers MyBatis XML Mapper ou même des objets encapsulant des appels JDBC bruts ou des requêtes Hibernate. Du point de vue de la plate-forme Microsoft, l'interface utilisateur peut être un module ASP (pages serveur actives) utilisant le framework .NET pour accéder aux modules C # dans la couche du domaine d'affaires, les modules d'accès aux données client et commande étant ADO (ActiveX Objets de données).

Considérations

Le modèle d'architecture en couches est un modèle général solide, ce qui en fait un bon point de départ pour la plupart des applications, en particulier lorsque vous ne savez pas quel modèle d'architecture convient le mieux à votre application. Cependant, il y a un certain nombre de choses à considérer du point de vue de l'architecture lors du choix de ce modèle.

La première chose à surveiller est ce qui est connu sous le nom de l'anti-pattern du gouffre. Cet anti-pattern décrit la situation où les demandes traversent plusieurs couches de l'architecture sous la forme d'un simple traitement transversal avec peu ou pas de logique dans chaque couche. Par exemple, supposons que la couche de présentation répond à une demande de l'utilisateur pour récupérer les données du client. La couche de présentation transmet la requête à la couche du domaine d'affaires, qui transmet simplement la requête à la couche de persistance, qui effectue ensuite un appel SQL simple à la couche de base de données pour extraire les données client. Les données sont ensuite transmises à la pile sans aucun traitement ou logique supplémentaire pour agréger, calculer ou transformer les données.

Chaque architecture en couches aura au moins quelques scénarios qui tombent dans l'anti-pattern du gouffre. La clé, cependant, est d'analyser le pourcentage de demandes qui entrent dans cette catégorie. La règle des 80-20 est généralement une bonne pratique à suivre pour déterminer si vous rencontrez l'anti-modèle de gouffre. Il est typique d'avoir environ 20% des demandes en tant que traitement de transfert simple et 80% des demandes ayant une logique du domaine d'affaire associée à la requête. Cependant, si vous constatez que ce rapport est inversé et que la majorité de vos demandes sont un simple traitement de transfert,

vous pouvez envisager d'ouvrir certaines couches d'architecture, en gardant à l'esprit qu'il sera plus difficile de contrôler le changement. En raison du manque d'isolation des couches.
Une autre considération avec le modèle d'architecture en couches est qu'il tend à se prêter à des applications monolithiques, même si vous divisez la couche de présentation et les couches de gestion en unités déployables distinctes. Bien que cela ne soit pas un problème pour certaines applications, cela pose des problèmes potentiels en termes de déploiement, de robustesse générale et de fiabilité, de performance et d'évolutivité.

Analyse de cette architecture

Le tableau suivant contient une évaluation et une analyse des caractéristiques d'architecture communes pour le modèle d'architecture en couches. L'évaluation pour chaque caractéristique est basée sur la tendance naturelle de la caractéristique en tant que capacité basée sur une implémentation typique du modèle, ainsi que sur la nature généralement reconnue de ce modèle. Pour une comparaison côte à côte de la façon dont ce modèle se rapporte à d'autres modèles dans ce rapport, veuillez vous référer à l'Annexe A à la fin de ce rapport.

Agilité globale : Faible

L'agilité globale est la capacité à répondre rapidement à un environnement en constante évolution. Bien que le changement puisse être isolé grâce aux couches d'isolation de ce modèle, il est encore fastidieux et difficile d'apporter des modifications à ce modèle d'architecture en raison de la nature monolithique de la plupart des implémentations ainsi que du couplage étroit des composants du modèle.

Facilité de déploiement : Faible

Selon la manière dont vous implémentez ce modèle, le déploiement peut devenir un problème, en particulier pour les applications plus volumineuses. Une petite modification apportée à un composant peut nécessiter un redéploiement de l'ensemble de l'application (ou d'une grande partie de l'application), ce qui entraîne des déploiements qui doivent être planifiés et exécutés en dehors des heures de bureau ou les week-ends. En tant que tel, ce modèle ne se prête pas facilement à un “pipeline” de livraison continu, réduisant davantage la note globale pour le déploiement.

Testabilité : Élevée

Comme les composants appartiennent à des couches spécifiques de l'architecture, d'autres couches peuvent être raillées ou tronquées, ce qui rend ce modèle relativement facile à tester. Un développeur peut simuler un composant ou un écran de présentation pour isoler les tests

dans un composant du domaine d'affaire. Également on peut simuler la couche du domaine d'affaires pour tester certaines fonctionnalités de l'écran.

Performance : Faible

S'il est vrai que certaines architectures en couches peuvent bien fonctionner, le modèle ne se prête pas à des applications de hautes performances. La raison principale est l'inefficacité engendrée par le fait de devoir passer par plusieurs couches de l'architecture pour répondre aux demandes du domaine d'affairs.

Évolutivité : Faible

En raison de la tendance vers des implémentations étroitement couplées et monolithiques de ce modèle, les applications construites en utilisant ce modèle d'architecture sont généralement difficiles à faire évoluer. Vous pouvez faire évoluer une architecture en couches en divisant les couches en déploiements physiques distincts ou en répliquant l'intégralité de l'application sur plusieurs nœuds, mais dans l'ensemble la granularité est trop large, ce qui complique son évolutivité.

Facilité de développement : Élevée

La facilité de développement obtient une note relativement élevée, principalement parce que ce modèle est si bien connu et n'est pas trop complexe à mettre en œuvre. Comme la plupart des entreprises développent des applications en séparant les ensembles de compétences par couche (présentation, entreprise, base de données), ce modèle devient un choix naturel pour la plupart des développements d'applications d'affaires. Le lien entre la structure de communication et d'organisation d'une entreprise et la façon dont elle développe le logiciel est décrit comme la loi de Conway [1].

[1] *Loi de Conway : Les organisations qui conçoivent les systèmes sont contraintes de produire des modèles qui sont des copies de leur propre structure de communication (1967)*

Chapitre 2

Architecture pilotée par évènement (Event-Driven)

Le modèle d'architecture piloté par événement est un modèle d'architecture asynchrone distribuée populaire utilisé pour produire des applications hautement évolutives. Il est également très adaptable et peut être utilisé pour de petites applications ainsi que pour des applications complexes et de grande taille. L'architecture pilotée par événement est constituée de composants de traitement d'événements à usage unique hautement découplés qui reçoivent et traitent les événements de manière asynchrone. Le modèle d'architecture piloté par les événements se compose de deux topologies principales, le médiateur et le courtier. La topologie du médiateur est généralement utilisée lorsque vous devez orchestrer plusieurs étapes au sein d'un événement via un médiateur central, tandis que la topologie du courtier est utilisée lorsque vous voulez chaîner des événements sans recourir à un médiateur central. Parce que les caractéristiques d'architecture et les stratégies de mise en œuvre diffèrent entre ces deux topologies, il est important de comprendre chacune d'entre elles pour savoir quelle est la mieux adaptée à votre situation particulière.

Topologie : Médiateur

Le médiateur est utile pour les événements comportant plusieurs étapes et nécessitant un certain niveau d'orchestration pour traiter l'événement. Par exemple, un seul événement pour effectuer une transaction boursière peut vous obliger à valider d'abord la transaction, puis vérifier la conformité de cette transaction par rapport à diverses règles de conformité, affecter la transaction à un courtier, calculer la commission et enfin effectuer la transaction avec ce courtier. Toutes ces étapes nécessitent un certain niveau d'orchestration pour en déterminer l'ordre et aussi déterminer celles qui peuvent être effectuées en série ou en parallèle.
Il existe quatre principaux types de composants d'architecture dans la topologie du médiateur: les files d'attente d'événements, un médiateur d'événements, les canaux d'événements et les processeurs d'événements. Le flux d'événements commence avec un client envoyant un

événement à une file d'attente d'événements, qui est utilisée pour transporter l'événement vers le médiateur d'événements. Le médiateur d'événements reçoit l'événement initial et orchestre cet événement en envoyant des événements asynchrones supplémentaires aux canaux d'événements pour exécuter chaque étape du processus. Les processus événementiels, qui écoutent sur les canaux d'événements, reçoivent l'événement du médiateur d'événement et exécutent une logique du domaine d'affaires spécifique pour traiter l'événement. La Figure 2-1 illustre la topologie générale du médiateur du modèle d'architecture piloté par les événements.

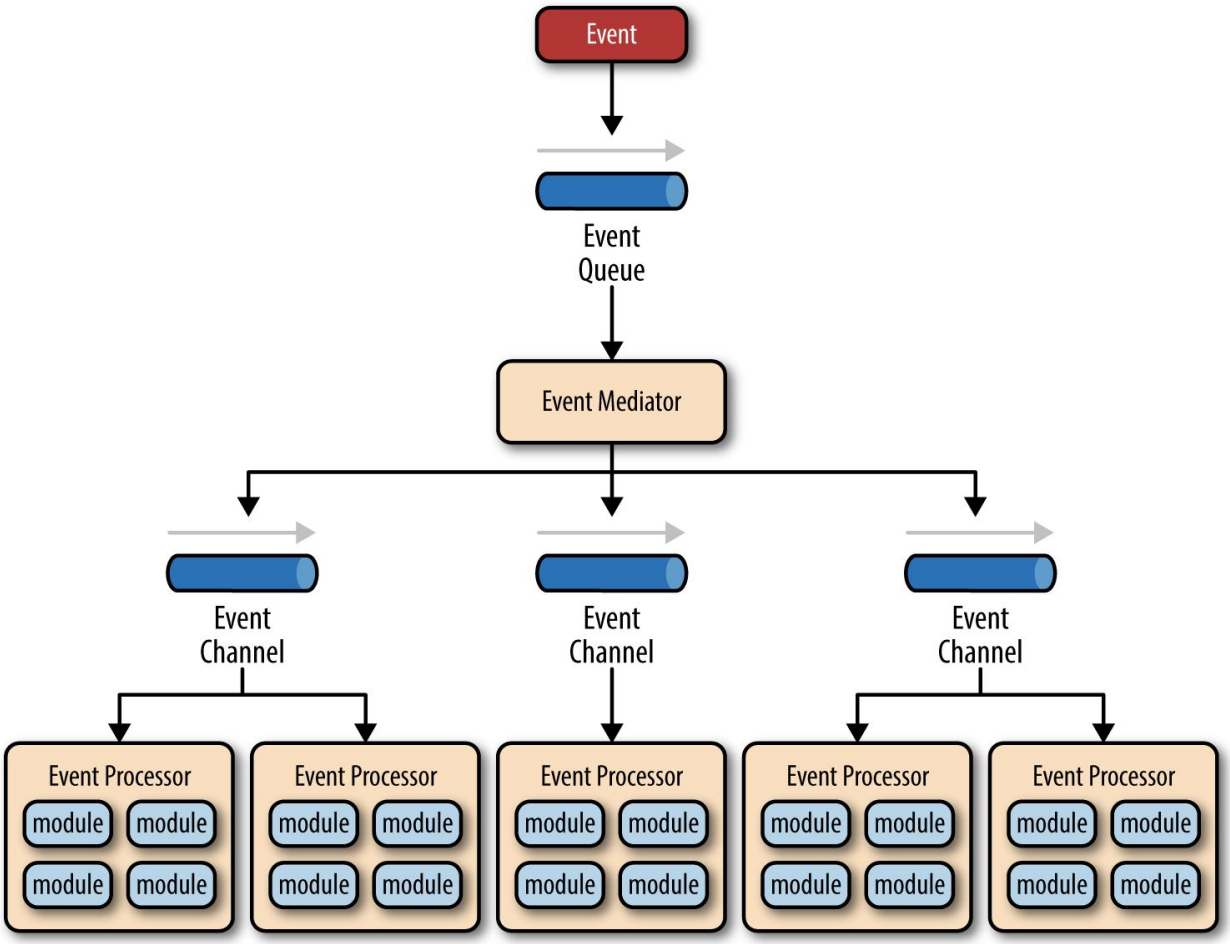

Figure 2-1 Topologie du médiateur

Il est courant d'avoir entré plusieurs centaines d'éléments dans une file d'attente d'événements dans une architecture événementielle. Le modèle ne spécifie pas l'implémentation du composant de file d'attente d'événements; il peut s'agir d'une file d'attente de messages, d'un point de terminaison de service Web ou de toute combinaison de ceux-ci.
Il existe deux types d'événements dans ce modèle: un événement initial et un événement de traitement. L'événement initial est l'événement d'origine reçu par le médiateur, tandis que les événements de traitement sont ceux qui sont générés par le médiateur et reçus par les

composants de traitement d'événements.
Le composant événement-médiateur est responsable de l'orchestration des étapes contenues dans l'événement initial. Pour chaque étape de l'événement initial, le médiateur d'événements envoie un événement de traitement spécifique à un canal d'événement, qui est ensuite reçu et traité par le processeur d'événements. Il est important de noter que le médiateur d'événements n'effectue pas la logique du domaine d'affaire nécessaire au traitement de l'événement initial. il connaît plutôt les étapes nécessaires pour traiter l'événement initial.

Les canaux d'événement sont utilisés par le médiateur d'événements pour transmettre de manière asynchrone des événements de traitement spécifiques, liés à chaque étape de l'événement initial, aux processeurs d'événements. Les canaux d'événements peuvent être des files d'attente de messages ou des sujets de message. Cependant les sujets de message sont les plus utilisés avec la topologie du médiateur quand on veut que le traitement soit effectué sur plusieurs processeurs d'événements (chacun effectuant une tâche différente selon l'événement de traitement reçu).

Les composants du processeur d'événements contiennent la logique du domaine d'affaire de l'application nécessaire pour traiter l'événement de traitement. Les processeurs d'événements sont des composants d'architecture indépendants, fortement découplés, qui exécutent une tâche spécifique dans l'application ou dans le système. Bien que la granularité de la composante du traitement des événements puisse varier de fine (p. Ex., calculer la taxe de vente sur une commande) à grossière (par exemple, traiter une réclamation d'assurance), il est important de garder à l'esprit que le composant du processeur d'événements doit effectuer une seule tâche du domaine d'affaires et ne pas compter sur d'autres processeurs d'événements pour accomplir cette tâche spécifique.

Le médiateur de l'événement peut être mis en œuvre de diverses manières. En tant qu'architecte, vous devez comprendre chacune de ces options d'implémentation pour vous assurer que la solution que vous choisissez pour le média d'événement correspond à vos besoins et exigences.

La mise en œuvre la plus simple et la plus fréquente du médiateur événementiel se fait par le biais de concentrateurs d'intégration Open Source tels que Spring Integration, Apache Camel ou Mule ESB. Les flux d'événements dans ces concentrateurs d'intégration open source sont généralement implémentés via du code Java ou un langage DSL (langage spécifique au domaine). Pour une médiation et une orchestration plus sophistiquées, vous pouvez utiliser BPEL (langage d'exécution de processus d'affaires) couplé à un moteur BPEL tel que l'ODE Apache open source. BPEL est un langage standard de type XML qui décrit les données et les étapes requises pour le traitement d'un événement initial. Pour les très grandes applications nécessitant une orchestration beaucoup plus sophistiquée (y compris les étapes impliquant des interactions humaines), vous pouvez implémenter le médiateur d'événements à l'aide d'un gestionnaire de processus d'affaires (BPM) tel que jBPM.

Comprendre vos besoins et les associer à l'implémentation correcte du médiateur d'événements est essentiel au succès de toute architecture pilotée par les événements utilisant cette topologie. L'utilisation d'un concentrateur d'intégration open source pour effectuer une orchestration de gestion de processus d’affaires très complexe est une recette pour l'échec, tout comme l'implémentation d'une solution BPM pour exécuter une logique de routage simple. Pour illustrer le fonctionnement de la topologie du médiateur, supposons que vous soyez assuré auprès d'une compagnie d'assurance et que vous décidiez de déménager. Dans ce cas, l'événement initial peut être appelé quelque chose comme un événement de relocalisation. Les étapes impliquées dans le traitement d'un événement de relocalisation sont contenues dans le médiateur d'événements, comme illustré à la Figure 2-2. Pour chaque étape d'événement initial, le médiateur d'événement créé un évènement de traitement (par exemple, changement d'adresse, recalcul, etc.), envoie cet événement de traitement au canal d'événement et attend que l'événement de traitement soit traité par le processeur d'événements correspondant. , processus client, processus de devis, etc.). Ce processus se poursuit jusqu'à ce que toutes les étapes de l'événement initial aient été traitées. La barre unique au-dessus de “recal quote” et “update claim” indique que ces étapes peuvent être exécutées simultanément.

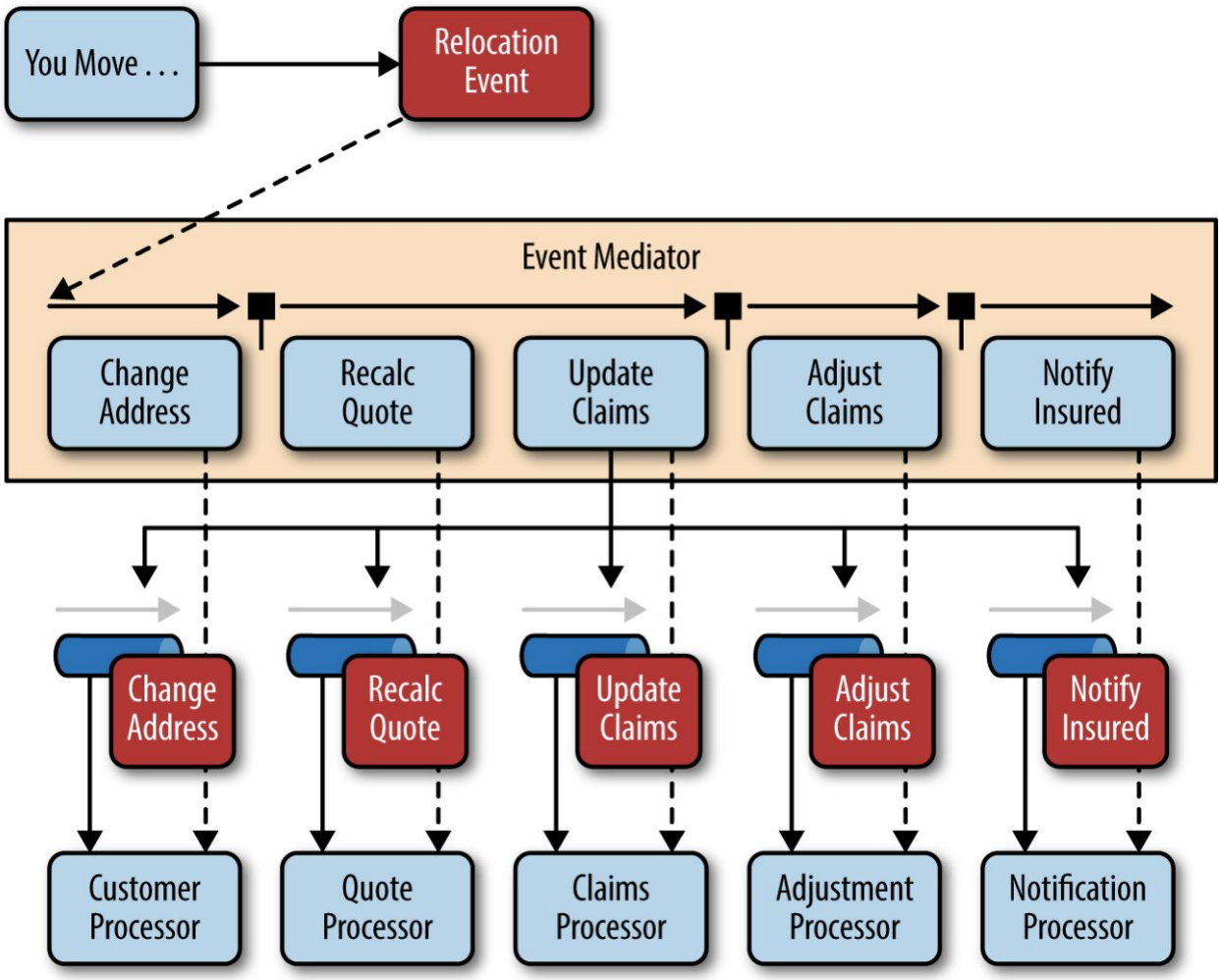

Figure 2-2 Exemple de la topologie du médiateur

Topologie : Courtier

La topologie du courtier diffère de la topologie du médiateur en ce sens qu'il n'y a pas de médiateur central d'événements; à la place, le flux de messages est réparti à travers les composants du processeur d'événement d'une manière en chaîne par l'intermédiaire d'un courtier de messages léger (par exemple, ActiveMQ, HornetQ, etc.). Cette topologie est utile lorsque vous avez un flux de traitement d'événements relativement simple et que vous ne voulez pas (ou n'avez pas besoin) d'orchestration d'événements centralisée.
Il existe deux principaux types de composants d'architecture dans la topologie du courtier: un composant de courtier et un composant de processeur d'événements. Le composant courtier peut être centralisé ou fédéré et contient tous les canaux d'événement utilisés dans le flux d'événements.

Les canaux d'événements contenus dans le composant courtier peuvent être des files d'attente de messages, des sujets de message ou une combinaison des deux.

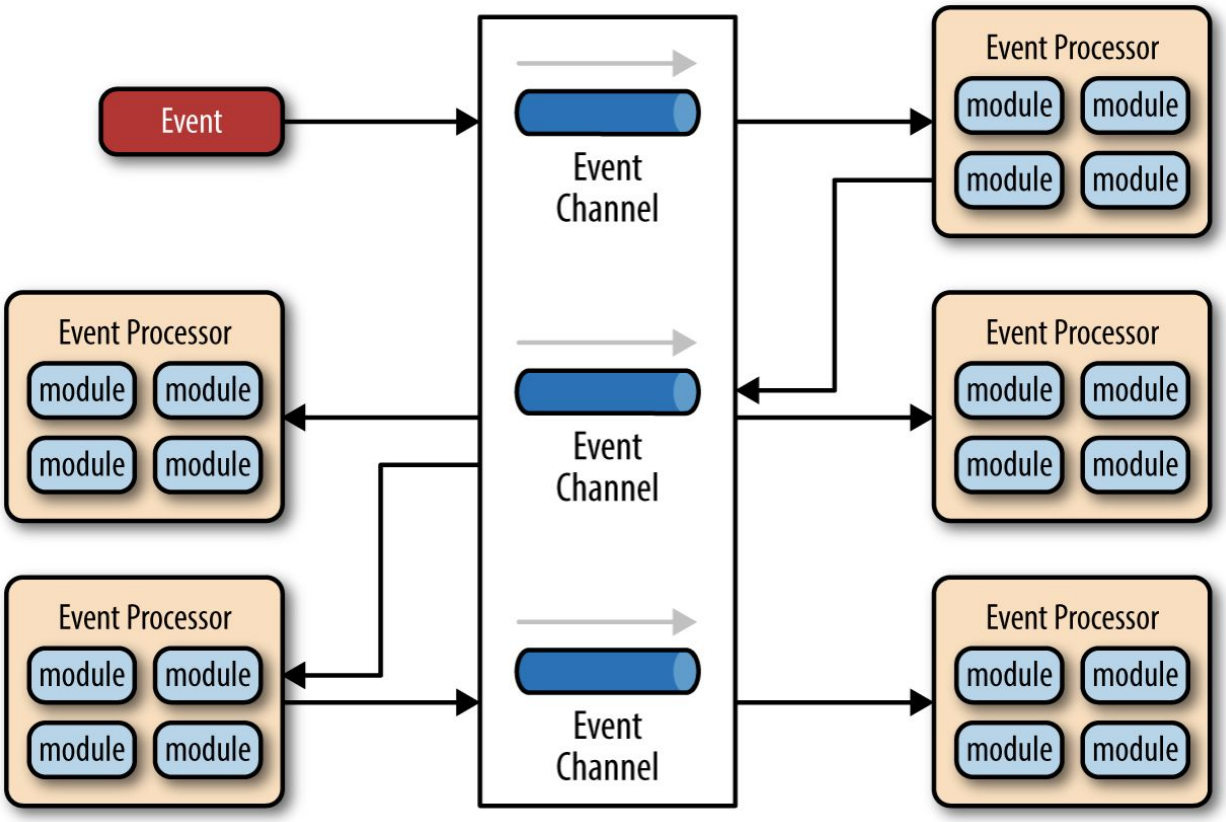

Figure 2-3 Topologie du courtier

Cette topologie est illustrée à la figure 2-3. Comme vous pouvez le voir sur le diagramme, il n'y a pas de composant central d'événement-médiateur contrôlant et orchestrant l'événement initial; à la place, chaque composant de traitement d'événement est responsable du traitement d'un événement et de la publication d'un nouvel événement indiquant l'action qu'il vient d'effectuer. Par exemple, un processeur d'événements qui équilibre un portefeuille d'actions peut recevoir un événement initial appelé division d'actions. Sur la base de cet événement initial, le processeur d'événements peut effectuer un rééquilibrage de portefeuille, puis publier un nouvel événement dans le portefeuille de rééquilibrage appelé Broker, qui sera ensuite récupéré par un autre processeur d'événements. Notez qu'il peut y avoir des moments où un événement est publié par un processeur d'événements, mais n’est pas capturé par un autre processeur d'événements. Ceci est courant lorsque vous développez une application ou que vous prévoyez de futures fonctionnalités et extensions.

Pour illustrer le fonctionnement de la topologie du courtier, nous utiliserons le même exemple que dans la topologie du médiateur (une personne assurée déménage). Comme il n'y a pas de médiateur central pour recevoir l'événement initial dans la topologie du courtier, le composant processus client reçoit directement l'événement, modifie l'adresse du client et envoie un événement indiquant qu'il a changé l'adresse d'un client (par exemple, changement d'adresse) . Dans cet exemple, deux processeurs d'événements sont intéressés par l'événement changement d'adresse : le processus de devis et le processus de réclamation. Le composant processeur de devis recalcule les nouveaux tarifs d'assurance auto basés sur le changement d'adresse et publie un événement dans le reste du système indiquant ce qu'il a fait (par exemple, recalculer l'événement de devis). D'autre part, le composant de traitement des demandes reçoit le même événement changement d'adresse, mais dans ce cas, il met à jour une demande d'assurance en cours et publie un événement dans le système en tant qu'événement de demande de mise à jour. Ces nouveaux événements sont ensuite captés par d'autres composants du processeur d'événements, et la chaîne d'événements continue à travers le système jusqu'à ce qu'il n'y ait plus d'événements publiés pour cet événement initiateur particulier.

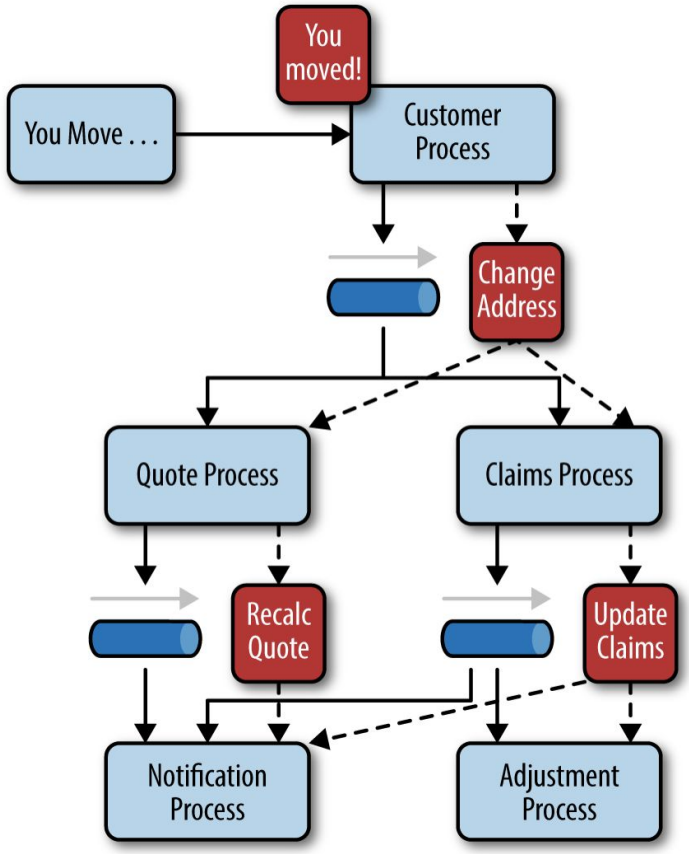

Figure 2-4 Exemple de la topologie du courtier

Comme vous pouvez le voir à la Figure 2-4, la topologie du courtier concerne uniquement le chaînage des événements pour exécuter une fonction du domaine d'affaires. La meilleure façon de comprendre la topologie du courtier est de le considérer comme une course à relais. Lors d'une course à relais, les coureurs tiennent un témoin et courent sur une certaine distance, puis relaie le témoin au coureur suivant, et ainsi de suite jusqu'à ce que le dernier coureur franchisse la ligne d'arrivée. Dans les courses de relais, une fois qu'un coureur passe le témoin, il a terminé sa course. Cela est également vrai avec la topologie du courtier: une fois qu'un événement est passé, celui-ci envoie le traitement à un autre évènement et il n'est plus impliqué dans le traitement de ce cas spécifique.

Considérations

Le modèle d'architecture piloté par événement est un modèle relativement complexe à mettre en œuvre, principalement en raison de sa nature distribuée asynchrone. Lors de la mise en œuvre de ce modèle, vous devez résoudre divers problèmes d'architecture distribués, tels que la disponibilité des processus distants, le manque de réactivité et la logique de reconnexion du courtier en cas de défaillance du courtier ou du médiateur.

Une considération à prendre en compte lors du choix de ce modèle d'architecture est le manque de transactions atomiques pour un processus d'affaire unique. Les composants du processeur d'événements étant fortement découplés et distribués, il est très difficile de maintenir une unité de travail transactionnelle entre eux. Pour cette raison, lorsque vous concevez votre application à l'aide de ce modèle, vous devez continuellement penser aux événements qui peuvent et ne peuvent pas fonctionner indépendamment et planifier, en conséquence, la granularité de vos processeurs d'événements. Si vous trouvez que vous devez diviser une seule unité de travail entre processeurs d'événements, c'est-à-dire si vous utilisez des processeurs distincts pour quelque chose qui devrait être une transaction indivise, ce n'est probablement pas le bon modèle pour votre application.

L'un des aspects les plus difficiles de la structure de l'architecture événementielle est peut-être la création, la maintenance et la gouvernance des contrats de composants de processeurs d'événements. Chaque événement a généralement un contrat spécifique associé (par exemple, les valeurs de données et le format de données transmis au processeur d'événements). Il est d'une importance vitale lors de l'utilisation de ce modèle de régler sur un format de données standard (par exemple, XML, JSON, objet Java, etc.) et d'établir dès le départ une politique de gestion des versions de contrat.

Analyse de cette architecture

Le tableau suivant contient une évaluation et une analyse des caractéristiques d'architecture communes pour le modèle d'architecture piloté par évènements. L'évaluation pour chaque caractéristique est basée sur la tendance naturelle de la caractéristique en tant que capacité basée sur une implémentation typique du modèle, ainsi que sur la nature généralement reconnue de ce modèle. Pour une comparaison côte à côte de la façon dont ce modèle se rapporte à d'autres modèles dans ce rapport, veuillez vous référer à l'Annexe A à la fin de ce rapport.

Agilité globale : Élevée

L'agilité globale est la capacité à répondre rapidement à un environnement en constante évolution. Comme les composants du processeur d'événements sont à usage unique et complètement découplé des autres composants du processeur d'événements, les modifications sont généralement associées à un ou plusieurs processeurs d'événements et peuvent être effectuées rapidement sans impact sur les autres composants.

Facilité de déploiement : Élevée

Dans l'ensemble, ce modèle est relativement facile à déployer en raison de la nature découplée des composants du processeur d'événements. La topologie du courtier a tendance à être plus facile à déployer que la topologie du médiateur, principalement parce que le composant médiateur d'événement est étroitement couplé aux processeurs d'événements: un changement dans un composant processeur d'événements peut également nécessiter un changement du médiateur d'événement et requiert un redéploiement pour tout changement donné.

Testabilité : Faible

Bien que les tests unitaires individuels ne soient pas trop compliqués, ils nécessitent une sorte de client de test spécialisé ou un outil de test pour générer des événements. Les tests sont également compliqués par la nature asynchrone de ce modèle.

Performance : Élevée

Bien qu'il soit certainement possible de mettre en œuvre une architecture pilotée par les événements qui ne fonctionne pas bien en raison de toute l'infrastructure de messagerie impliquée, en général, le modèle atteint des performances élevées grâce à ses capacités asynchrones; en d'autres termes, la capacité à effectuer des opérations asynchrones parallèles découplées l'emporte sur le coût de mise en file d'attente et de suppression des messages

Évolutivité : Élevée

L'évolutivité est naturellement réalisée dans ce modèle grâce à des processeurs d'événements

hautement indépendants et découplés. Chaque processeur d'événements peut être mis à l'échelle séparément, ce qui permet une évolutivité fine.

Facilité de développement : Faible

Le développement peut être assez compliqué en raison de la nature asynchrone du modèle ainsi que de la création de contrat et du besoin de conditions de gestion des erreurs plus avancées dans le code pour les processeurs d'événements non réactifs et les courtiers défaillants.

Chapitre 3

Architecture micro-kernel (plug-in)

Le modèle d'architecture de micro-kernel (parfois appelé modèle d'architecture de plug-in) est un modèle naturel pour la mise en œuvre d'applications basées sur des produits. Une application basée sur un produit est une application qui est empaquetée et disponible au téléchargement dans les versions en tant que produit tiers standard. Toutefois, de nombreuses entreprises développent et commercialisent également leurs applications internes, telles que les produits logiciels, ainsi que les versions, les notes de publication et les fonctionnalités connectables. Ce sont également un ajustement naturel pour ce modèle. Le modèle d'architecture de micro-kernel vous permet d'ajouter des fonctionnalités d'application supplémentaires en tant que plug-ins à l'application de base, offrant une extensibilité ainsi qu'une séparation et une isolation des fonctions.

Description du modèle d'architecture micro-kernel

Le modèle d'architecture de micro-kernel comprend deux types de composants d'architecture: un système de base et des modules enfichables. La logique d'application est divisée entre des modules de plug-in indépendants et le système de base, offrant extensibilité, flexibilité et isolation des fonctionnalités de l'application et de la logique de traitement personnalisée. La figure 3-1 illustre le modèle d'architecture de base du micro-kernel.

Le système de base du modèle d'architecture de micro-kernel ne contient traditionnellement que la fonctionnalité minimale requise pour rendre le système opérationnel. De nombreux systèmes d'exploitation implémentent le modèle d'architecture du micro-kernel, d'où l'origine du nom de ce modèle. Du point de vue d'une application, le système principal est souvent défini comme la logique du domaine d'affaire générale sans code personnalisé pour des cas particuliers, des règles spéciales ou un traitement conditionnel complexe.

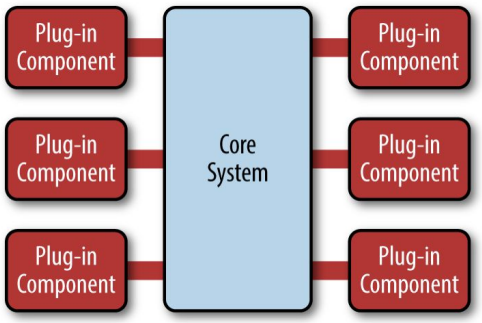

Figure 3-1 modèle d'architecture du micro-kernel

Les modules de plug-in sont des composants indépendants et autonomes qui contiennent un traitement spécialisé, des fonctionnalités supplémentaires et un code personnalisé destinés à améliorer ou à étendre le système principal pour produire des fonctionnalités supplémentaires. Généralement, les modules de plug-in doivent être indépendants des autres modules de plug-in, mais vous pouvez certainement concevoir des plug-ins nécessitant la présence d'autres plug-ins. Quoi qu'il en soit, il est important de réduire au minimum la communication entre les plug-ins pour éviter les problèmes de dépendance.
Le système de base doit savoir quels modules de plug-in sont disponibles et comment y accéder. Une manière courante de mettre en œuvre ceci est à travers une sorte de registre de plug-in.

Ce registre contient des informations sur chaque module de plug-in, y compris des éléments tels que son nom, son contrat de données et les détails du protocole d'accès distant (en fonction de la manière dont le plug-in est connecté au système principal). Par exemple, un plug-in pour logiciel d'impôt qui signale les éléments d'audit fiscal à risque élevé peut avoir une entrée de registre contenant le nom du service (AuditChecker), le contrat de données (données d'entrée et de sortie) et le contrat format (XML). Il peut également contenir un WSDL (Web Services Definition Language) si le plug-in est accessible via SOAP.
Les modules enfichables peuvent être connectés au système central de diverses manières, y compris OSGi (initiative de passerelle de service ouvert), la messagerie, les services Web ou même la liaison point à point directe (c'est-à-dire l'instanciation d'objet). Le type de connexion que vous utilisez dépend du type d'application que vous créez (produit de petite taille ou grande application commerciale) et de vos besoins spécifiques (par exemple, déploiement unique ou déploiement distribué). Le modèle d'architecture lui-même ne spécifie aucun de ces détails d'implémentation, mais seulement que les modules de plug-in doivent rester indépendants les

uns des autres.
Les contrats entre les modules de plug-in et le système de base peuvent varier de contrats standard à des contrats personnalisés. Les contrats personnalisés se trouvent généralement dans des situations où des composants de plug-in sont développés par un tiers dans lequel vous n'avez aucun contrôle sur le contrat utilisé par le plug-in. Dans de tels cas, il est courant de créer un adaptateur entre le contact du plug-in et votre contrat standard afin que le système principal n'ait pas besoin de code spécialisé pour chaque plug-in. Lors de la création de contrats standard (généralement implémentés via XML ou une liste de clé valeurs en Java), il est important de ne pas oublier de créer une stratégie de version dès le départ.

Exemple de l’architecture micro-kernel

Le meilleur exemple de l'architecture du micro-kernel est peut-être l'IDE Eclipse. Le téléchargement du produit de base Eclipse vous fournit un peu plus qu'un éditeur de fantaisie. Cependant, une fois que vous commencez à ajouter des plug-ins, il devient un produit hautement personnalisable et utile. Les navigateurs Internet sont un autre exemple de produit courant utilisant l'architecture de micro-kernel: les visionneuses et autres plug-ins ajoutent des capacités supplémentaires qui ne sont pas trouvées dans le navigateur de base (c'est-à-dire le système principal).

Les exemples sont infinis pour les logiciels basés sur les produits, mais qu'en est-il des applications de grande entreprise? L'architecture du micronoyau s'applique également à ces situations. Pour illustrer ce point, prenons l'exemple d'une autre compagnie d'assurance, qui fait le traitement des réclamations d'assurance.

Le traitement des réclamations est un processus très compliqué. Chaque Province ou État a des règles et règlements différents pour ce qui est autorisé, ou pas, dans une réclamation d'assurance. Par exemple, certaines provinces autorisent le remplacement gratuit du pare-brise si votre pare-brise est endommagé par une roche. Tandis que d’autres provinces ne le permettent pas. Cela crée un ensemble presque infini de conditions pour un processus de réclamation standard.

Sans surprise, la plupart des applications de réclamations d'assurance exploitent des moteurs de règles complexes pour gérer ce genre de complexité. Cependant, ces moteurs de règles peuvent se transformer en une grosse boule de boue où la modification d'une règle impacte d'autres règles, ou si un simple changement de règle nécessite une armée d'analystes, de développeurs et de testeurs. L'utilisation du modèle d'architecture micro-kernel peut résoudre plusieurs de ces problèmes.

La pile de dossiers que vous voyez dans la Figure 3-2 représente le système de base pour le traitement des réclamations. Il contient la logique d’affaires de base requise par la compagnie d'assurance pour traiter une réclamation, sauf en cas de traitement personnalisé. Chaque

module de plug-in contient les règles spécifiques pour cet état. Dans cet exemple, les modules de plug-in peuvent être implémentés à l'aide de code source personnalisé ou d'instances de moteur de règles distinctes. Indépendamment de la mise en œuvre, le point clé est que les règles spécifiques à l'état et le traitement sont distincts du système de revendications principal et peuvent être ajoutés, supprimés et modifiés avec peu ou pas d'effet sur le reste du système principal ou des autres modules enfichables .

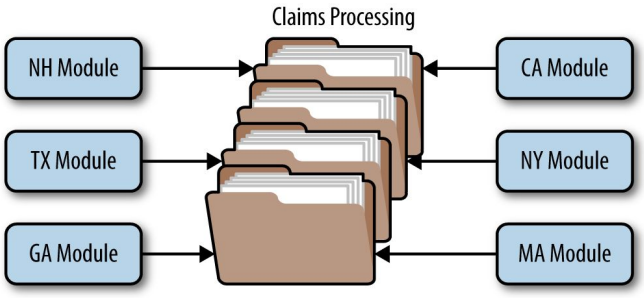

Figure 3-2 Exemple de l’architecture du Micro-kernel

Considérations

Une chose importante à propos du modèle d'architecture micro-kernel est qu'il peut être intégré ou utilisé dans le cadre d'un autre modèle d'architecture. Par exemple, si ce modèle résout un problème particulier que vous avez avec une zone volatile spécifique de l'application, vous pouvez constater que vous ne pouvez pas implémenter l'architecture entière en utilisant ce modèle. Dans ce cas, vous pouvez incorporer le modèle d'architecture de micro-kernal dans un autre modèle que vous utilisez (par exemple, une architecture en couches). De même, les composants du processeur d'événements décrits dans la section précédente sur l'architecture pilotée par les événements pourraient être implémentés en utilisant le modèle d'architecture de micro-kernel.
Le modèle d'architecture de micro-kernel fournit un excellent support pour la conception évolutive et le développement incrémental. Vous pouvez d'abord créer un système de base solide et, au fur et à mesure que l'application évolue, ajouter des fonctionnalités sans avoir à apporter de modifications importantes au système principal.

Pour les applications basées sur des produits, l'architecture micro-kernel doit toujours être votre premier choix en tant qu'architecture de départ, en particulier pour les produits pour lesquels vous allez proposer des fonctionnalités supplémentaires au fil du temps et qui veulent savoir quelles fonctionnalités les utilisateurs recherchent. Si vous trouvez au fil du temps que le modèle ne répond pas à toutes vos exigences, vous pouvez toujours refactoriser votre application à un autre modèle d'architecture mieux adapté à vos besoins spécifiques.

Analyse de cette architecture

Le tableau suivant contient une évaluation et une analyse des caractéristiques d'architecture communes pour le modèle d'architecture de micro-kernel. L'évaluation pour chaque caractéristique est basée sur la tendance naturelle de cette caractéristique en tant que capacité basée sur une implémentation typique du modèle, ainsi que sur la nature généralement reconnue de ce modèle. Pour une comparaison côte à côte de la façon dont ce modèle se rapporte à d'autres modèles dans ce rapport, veuillez vous référer à l'Annexe A à la fin de ce rapport.

Agilité globale : Élevée

L'agilité globale est la capacité à répondre rapidement à un environnement en constante évolution. Les modifications peuvent en grande partie être isolées et mises en œuvre rapidement grâce à des modules enfichables à couplage lâche. En général, le système de base de la plupart des architectures de micro-kernel tend à devenir rapidement stable et, en tant que tel, est assez robuste et nécessite peu de changements dans le temps.

Facilité de déploiement : Élevée

Selon la manière dont le modèle est implémenté, les modules de plug-in peuvent être ajoutés dynamiquement au système de base lors de l'exécution (par exemple, déploiement à chaud), ce qui minimise les temps d'arrêt pendant le déploiement.

Testabilité : Élevée

Les modules enfichables peuvent être testés isolément et peuvent être facilement simulés par le système principal pour démontrer une caractéristique particulière avec peu ou pas de changement dans le système central.

Performance : Élevée

Bien que le modèle de micro-kernel ne se prête pas naturellement aux applications de haute performance, la plupart des applications construites à l'aide du modèle d'architecture micro-kernel fonctionnent bien parce que vous pouvez personnaliser et rationaliser les applications pour inclure uniquement celles dont vous avez besoin. JBoss Application Server en est un bon exemple: avec son architecture de plug-in, vous pouvez réduire le serveur d'applications aux seules fonctionnalités dont vous avez besoin, en supprimant les

fonctionnalités coûteuses non utilisées telles que l'accès distant, la messagerie et la mise en cache consommant de la mémoire.

Évolutivité : Faible

Comme la plupart des implémentations d'architecture à micro-kernel sont basées sur des produits et sont généralement de plus petite taille, elles sont implémentées en tant qu'unité distincte et ne sont donc pas très évolutives. Selon la manière dont vous implémentez les modules de plug-in, vous pouvez parfois fournir une évolutivité au niveau de la fonctionnalité du plug-in, mais globalement ce modèle n'est pas connu pour produire des applications hautement évolutives

Facilité de développement : Faible

L'architecture du micro-kernel nécessite une conception réfléchie et une gouvernance des contrats, ce qui la rend plutôt complexe à mettre en œuvre. La gestion des versions des contrats, les registres de plug-ins internes, la granularité des plug-ins et les nombreux choix disponibles pour la connectivité des plug-ins contribuent tous à augmenter complexité de la mise en œuvre de ce modèle.

Chapitre 4

Modèle d’architecture des microservices

Le modèle d'architecture des microservices gagne rapidement du terrain dans l'industrie en tant qu'alternative viable aux applications monolithiques et aux architectures orientées service. Parce que ce modèle d'architecture est en constante évolution, il y a beaucoup de confusion dans l'industrie quant à ce que ce modèle et comment il est mis en œuvre. Cette section du rapport vous fournira les concepts clés et les connaissances de base nécessaires pour comprendre les avantages (et les compromis) de cet important modèle d'architecture et de déterminer s'il est le bon modèle pour votre application.

Description du modèle d'architecture microservices

Quel que soit la topologie ou le style d'implémentation que vous avez choisi, plusieurs concepts de base communs s'appliquent au modèle d'architecture général. Le premier de ces concepts est la notion d'unités déployées séparément. Comme l'illustre la Figure 4-1, chaque composant de l'architecture de microservices est déployé en tant qu'unité distincte, facilitant le déploiement grâce à un pipeline de distribution efficace et simplifié, une évolutivité accrue et un haut degré de découplage des applications et des composants au sein de votre application.

Le concept le plus important à comprendre avec ce modèle est peut-être la notion de composante de service. Plutôt que de penser aux services au sein d'une architecture de microservices, il est préférable de penser aux composants de service, qui peuvent varier en granularité d'un seul module à une grande partie de l'application. Les composants de service contiennent un ou plusieurs modules (par exemple, des classes Java) qui représentent soit une fonction à usage unique (par exemple, fournir la météo pour une ville donnée) ou une partie indépendante d'une grande application (par ex. détermination des taux d'assurance auto). Concevoir le bon niveau de granularité de composant de service est l'un des plus grands défis dans une architecture de microservices. Ce problème est abordé plus en détail dans la sous-section suivante relative à l'orchestration des services.

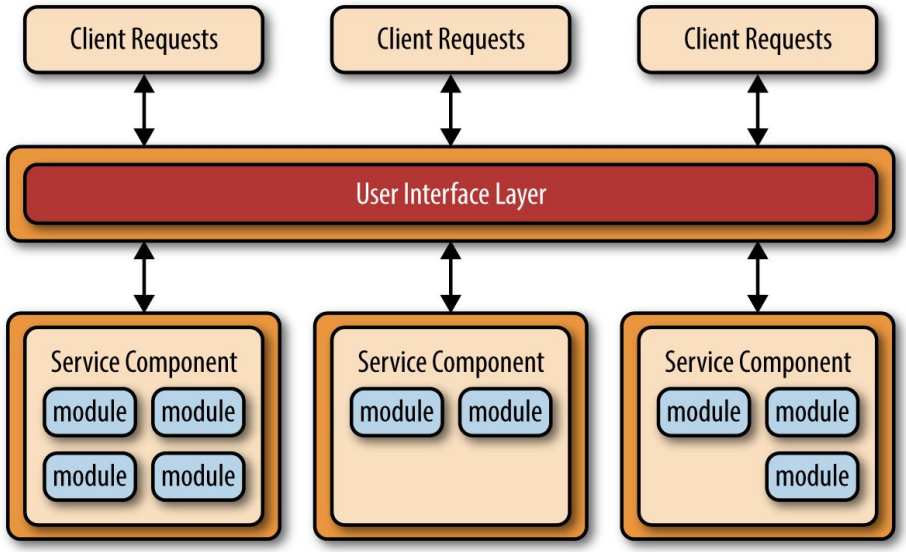

Figure 4-1 Base de l’architecture des microservices

Un autre concept clé dans le modèle d'architecture de microservices est qu'il s'agit d'une architecture distribuée, ce qui signifie que tous les composants de l'architecture sont totalement découplés et accessibles via un protocole d'accès distant (par exemple : REST, SOAP, RMI, etc.). La nature distribuée de ce modèle d'architecture est la manière dont il atteint certaines de ses caractéristiques d'évolutivité et de déploiement supérieures.

L'un des aspects passionnants de l'architecture des microservices est qu'elle a évolué à partir de problèmes associés à d'autres modèles d'architecture courants, plutôt que d'être créée en tant que solution en attente d'un problème. Le style d'architecture des microservices a naturellement évolué à partir de deux sources principales: les applications monolithiques développées en utilisant le modèle d'architecture en couches et les applications distribuées développées à travers le modèle d'architecture orientée service. Le chemin évolutif des applications monolithiques vers un style d'architecture de micro-serveurs a été principalement motivé par le développement de la livraison en continu.

Les applications monolithiques consistent généralement en des composants étroitement couplés qui font partie d'une seule unité déployable, ce qui la rend lourde et difficile à modifier, à tester et à déployer (d'où l'augmentation des cycles de «déploiement mensuel» courants dans la plupart des grandes entreprises). Ces facteurs conduisent généralement à des applications fragiles qui se brisent à chaque fois que quelque chose de nouveau est déployé. Le modèle d'architecture des microservices résout ces problèmes en séparant l'application en plusieurs unités déployables (composants de service) qui peuvent être développées, testées et déployées individuellement indépendamment des autres composants de service.

L'autre chemin d'évolution qui conduit au modèle d'architecture des microservices provient des

problèmes rencontrés avec les applications implémentant le modèle d'architecture orientée service (SOA service oriented architecture pattern). Même si le modèle SOA est très puissant et offre des niveaux d'abstraction, une connectivité hétérogène, une orchestration de service et la promesse d'aligner les objectifs du domaine d'affaire sur les capacités informatiques, il est néanmoins complexe, coûteux, omniprésent, difficile à comprendre et à implémenter, et est généralement exagéré pour la plupart des applications. Le style d'architecture des microservices répond à cette complexité en simplifiant la notion de service, en éliminant les besoins d'orchestration et en simplifiant la connectivité et l'accès aux composants de service.

Topologie du modèle

Bien qu'il existe littéralement des dizaines de façons de mettre en œuvre un modèle d'architecture de microservices, les trois topologies principales sont devenues les plus populaires: la topologie basée sur l'API REST, la topologie basée sur l'APPLICATION REST et la topologie de messagerie centralisée.

La topologie basée sur l'API REST est utile pour les sites Web qui exposent de petits services individuels autonomes via une sorte d'API (application programming interface). Cette topologie, illustrée à la Figure 4-2, est constituée de composants de service très précis (d'où le nom de microservices) qui contiennent un ou deux modules qui exécutent des fonctions spécifiques du domaine d'affaires indépendamment du reste des services. Dans cette topologie, ces composants de service à granularité fine sont généralement accessibles à l'aide d'une interface REST implémentée via une couche API Web déployée séparément. Des exemples de cette topologie incluent certains des services Web RESTful utilisé entre autres par les entreprises Yahoo, Google et Amazon.

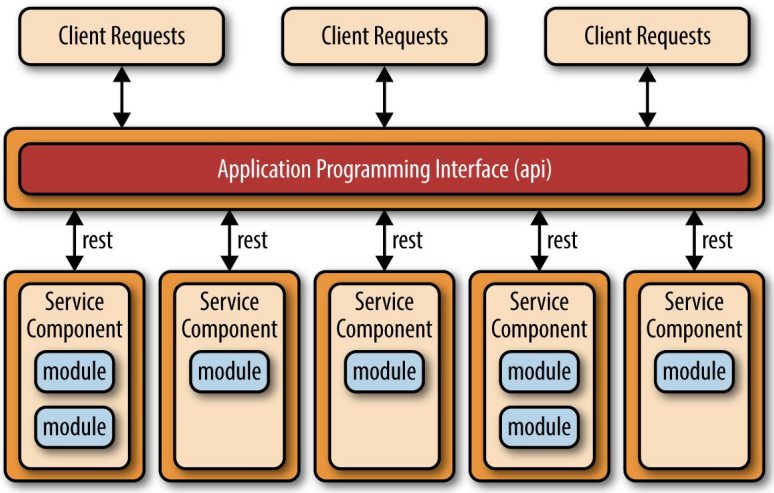

Figure 4-2 Topologie de l'API-REST

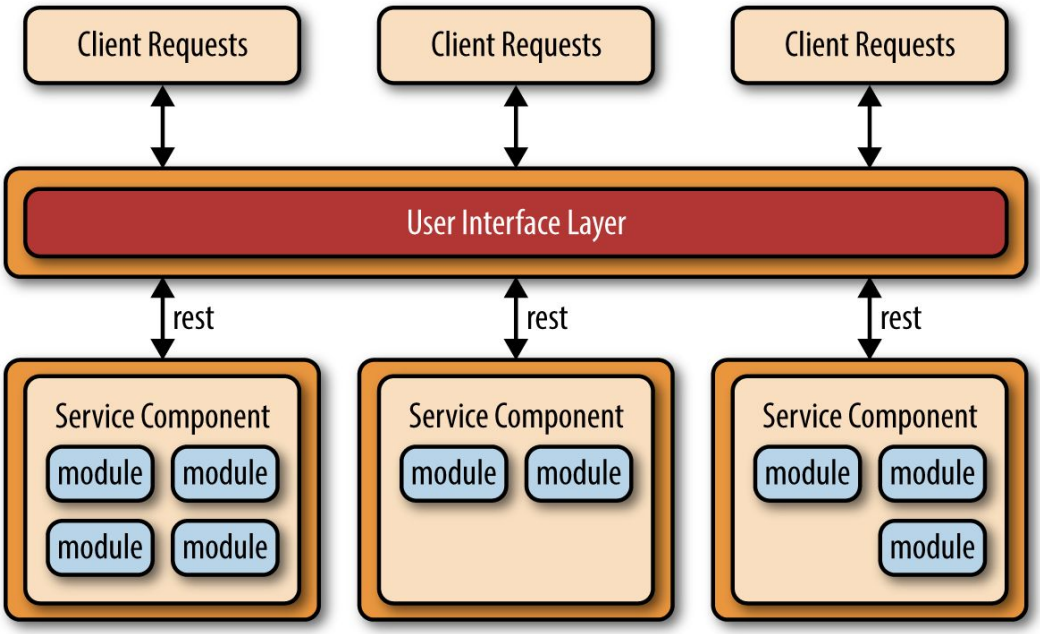

Figure 4-3 Topologie de l'APPLICATION-REST

La topologie APPLICATION-REST diffère de l'approche basée sur l'API-REST en ce sens que les demandes des clients sont reçues via des interfaces usager d'une d'application traditionnelle basées sur le Web ou de gros clients plutôt que via une couche API simple. Comme illustré à la Figure 4-3, la couche d'interface utilisateur de l'application est déployée en tant qu'application Web distincte qui accède à distance aux composants de service déployés séparément via de simples interfaces basées sur REST. Les composants de service de cette topologie diffèrent de ceux de la topologie API-REST en ce sens que ces composants de service ont tendance à être plus grands, plus grossiers et représentent une petite partie de l'application plutôt que des composants unique représentant des actions précises. Cette topologie est courante pour les applications professionnelles de petite et moyenne taille présentant un degré de complexité relativement faible.

Une autre approche courante dans le modèle d'architecture de microservices est la topologie de messagerie centralisée. Cette topologie (illustrée dans la Figure 4-4) est similaire à la topologie basée sur REST de l'application précédente, sauf qu'au lieu d'utiliser REST pour l'accès distant, cette topologie utilise un courtier de messages centralisé léger (par exemple, ActiveMQ, HornetQ, etc.). Il est d'une importance capitale de ne pas confondre cette topologie avec le modèle d'architecture orientée services ou de le considérer comme "SOA-Lite". Le courtier de messages léger trouvé dans cette topologie n'effectue aucune orchestration, transformation ou routage complexe. Il s'agit simplement d'un transport léger pour accéder aux composants de service à distance.
La topologie de messagerie centralisée se trouve généralement dans les applications ou applications de grande taille nécessitant un contrôle plus sophistiqué de la couche de transport

entre l'interface utilisateur et les composants de service. Les avantages de cette topologie par rapport à la topologie REST décrite précédemment sont les mécanismes de mise en file d'attente avancés, la messagerie asynchrone, la surveillance, la gestion des erreurs et une meilleure répartition et équilibrage de la charge globale. Le point de défaillance unique et les problèmes de goulots d'étranglement architecturaux généralement associés à un courtier centralisé sont résolus par le regroupement de courtiers et la fédération de courtiers (division d'une seule instance de courtier en plusieurs courtiers pour répartir la charge de traitement des messages selon les domaines fonctionnels du système).

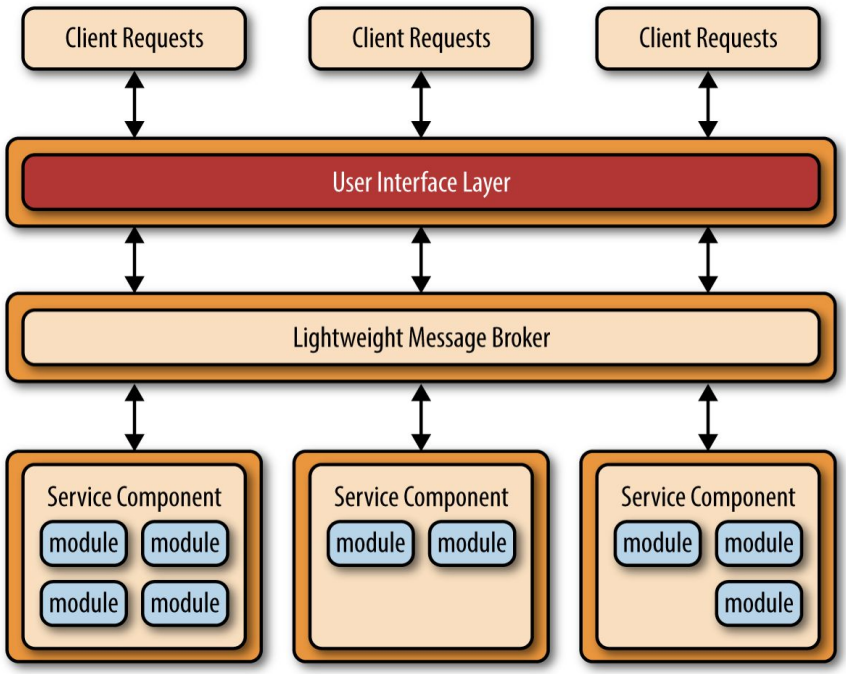

Figure 4-3 Topologie de messagerie centralisée.

Éviter les dépendances et l'orchestration

L'un des principaux défis du modèle d'architecture de microservices est de déterminer le niveau de granularité correct pour les composants de service. Si les composants du service sont trop grossiers, vous ne réaliserez peut-être pas les avantages de ce modèle d'architecture (déploiement, évolutivité, testabilité et couplage faible). En contre parti, les composants de service trop sophistiqués conduiront à des exigences d'orchestration, ce qui transformera rapidement votre architecture de microservices allégée en une architecture orientée service lourde, complète avec toute la complexité, la confusion, la dépense et le flou typique. trouvé avec les applications SOA.

Si vous constatez que vous devez orchestrer vos composants de service à partir de l'interface utilisateur ou de la couche API de l'application, alors les composants de votre service sont probablement trop fins. De même, si vous constatez que vous devez effectuer une communication inter-service entre des composants de service pour traiter une seule requête, il est probable que la granularité de vos composants de service soit trop fine ou que vos composants ne soient pas définis correctement du point de vue de la fonctionnalité de l'application.

La communication interservices, qui pourrait forcer des couplages indésirables entre composants, peut être gérée à la place via une base de données partagée. Par exemple, si un composant de service transmettant des commandes Internet a besoin d'informations client, il peut aller dans la base de données pour récupérer les données nécessaires plutôt que d'invoquer des fonctionnalités dans le composant de service client.

La base de données partagée peut gérer les besoins d'informations, mais qu'en est-il des fonctionnalités partagées? Si un composant de service a besoin d'une fonctionnalité contenue dans un autre composant de service ou commune à tous les composants de service, vous pouvez parfois copier la fonctionnalité partagée entre les composants de service (violant ainsi le principe DRY don't repeat yourself - évitez les duplications). Il s'agit d'une pratique assez courante dans la plupart des applications qui implémentent le modèle d'architecture microservices. On accepte la redondance de petites portions répétées de la logique du domaine d'affaire pour préserver l'indépendance des composants de service et séparer leur déploiement.

Les petites classes d'utilité pourraient tomber dans cette catégorie de code répété.
Si vous constatez que, quel que soit le niveau de granularité des composants de service, vous ne pouvez toujours pas éviter l'orchestration de composants de service, cela indique que ce n'est peut-être pas le bon modèle d'architecture pour votre application. En raison de la nature distribuée de ce modèle, il est très difficile de maintenir une unité de travail transactionnelle unique entre les composants de service. Une telle pratique nécessiterait une sorte de cadre de compensation des transactions pour les transactions de rétrocession, ce qui ajoute une complexité considérable à ce modèle d'architecture relativement simple et élégant.

Considérations

Le modèle d'architecture de microservices résout un bon nombre des problèmes courants rencontrés dans les applications monolithiques ainsi que dans les architectures orientées service. Comme les principaux composants d'application sont divisés en unités plus petites et déployées séparément, les applications construites à l'aide du modèle d'architecture de microservices sont généralement plus robustes, offrent une meilleure évolutivité et peuvent plus facilement prendre en charge une distribution continue.

Un autre avantage de ce modèle est qu'il offre la possibilité d'effectuer des déploiements de

production en temps réel, ce qui réduit considérablement le besoin de déploiements de production «big bang» mensuels ou de week-end traditionnel. Comme le changement est généralement isolé pour des composants de service spécifiques, seuls les composants de service qui changent doivent être déployés. Si vous n'avez qu'une seule instance d'un composant de service, vous pouvez écrire du code spécialisé dans l'application d'interface utilisateur pour détecter un déploiement actif et rediriger les utilisateurs vers une page d'erreur ou une page en attente. Vous pouvez également permuter plusieurs instances d'un composant de service pendant un déploiement en temps réel, ce qui permet une disponibilité continue pendant les cycles de déploiement (ce qui est très difficile à faire avec le modèle d'architecture en couches).

Une dernière considération à prendre en compte est que le modèle d'architecture des microservices étant une architecture distribuée, il partage certains des mêmes problèmes complexes que l'architecture événementielle, y compris la création de contrats, la maintenance et la gouvernance, la disponibilité du système à distance ainsi que l'authentification et l'autorisation d'accès à distance

Analyse de cette architecture

Le tableau suivant contient une évaluation et une analyse des caractéristiques d'architecture communes pour le modèle d'architecture des microservices. La notation de chaque caractéristique est basée sur la tendance naturelle pour cette caractéristique en tant que capacité basée sur une implémentation typique du modèle, ainsi que sur la nature généralement connue du modèle. Pour une comparaison côte à côte de la façon dont ce modèle se rapporte à d'autres modèles dans ce rapport, veuillez vous référer à l'Annexe A à la fin de ce rapport.

Agilité globale : Élevée

L'agilité globale est la capacité à répondre rapidement à un environnement en constante évolution. En raison de la notion d'unités déployées séparément, le changement est généralement isolé des composants de service individuels, ce qui permet un déploiement rapide et facile. En outre, les applications construites à l'aide de ce modèle ont tendance à être faiblement couplées, ce qui facilite également les changements.

Facilité de déploiement : Élevée

Les caractéristiques de déploiement et l'aisance de configuration des microservices sont très élevées en raison de la nature fine et indépendante des services distants. Les services sont généralement déployés en tant qu'unités distinctes de logiciels, ce qui permet d'effectuer des «déploiements à chaud» à tout moment de la journée ou de la nuit. Le risque global de déploiement est également réduit de manière significative, car les déploiements en panne peuvent être restaurés plus rapidement et n'impactent que les opérations sur le service déployé, ce qui entraîne la poursuite des opérations pour sur tous les autres services.

Testabilité : Élevée

En raison de la séparation et de l'isolement des fonctionnalités dans des applications indépendantes, les tests peuvent être étendus, ce qui permet des efforts de test plus ciblés. Les tests de régression pour un composant de service particulier sont beaucoup plus faciles et plus faisables que les tests de régression pour une application monolithique entière. De plus, étant donné que les composants de service dans ce modèle sont faiblement couplés, il y a beaucoup moins de chance de développement qu'une modification qui casse une autre partie de l'application, ce qui allège le test de devoir tester l'ensemble de l'application pour une modification mineure.

Performance : Faible

Bien que vous puissiez créer des applications implémentées à partir de ce modèle qui fonctionnent très bien, dans l'ensemble ce modèle ne se prête pas naturellement à des applications hautes performances en raison de la nature distribuée du modèle d'architecture de microservices.

Évolutivité : Élevée

Comme l'application est divisée en unités déployées séparément, chaque composant de service peut être mis à l'échelle individuellement, ce qui permet une mise à l'échelle précise de l'application. Par exemple, la zone d'administration d'une application boursière peut ne pas avoir besoin d'être redimensionnée en raison des faibles volumes d'utilisateurs pour cette fonctionnalité, mais le composant de service de placement peut avoir besoin d'évoluer en raison du débit élevé requis par la plupart des applications de trading.

Facilité de développement : Élevée

Comme la fonctionnalité est isolée en composants de service séparés et distincts, le développement devient plus facile en raison de la portée plus petite et isolée. Il y a beaucoup moins de chances qu'un développeur apporte une modification à un composant de service qui affecterait d'autres composants de service, réduisant ainsi la coordination nécessaire entre les développeurs ou les équipes de développement.

Chapitre 5

Modèle d'architecture en nuage

La plupart des applications basées sur le Web suivent en général le même flux de demandes : une requête provenant d'un navigateur rencontre le serveur Web, puis un serveur d'applications, puis finalement le serveur de base de données. Bien que ce modèle fonctionne bien pour un petit nombre d'utilisateurs, les goulets d'étranglement apparaissent lorsque la charge de l'utilisateur augmente, d'abord sur la couche serveur Web, puis sur la couche serveur d'application et enfin sur la couche serveur de base de données. La réponse habituelle aux goulets d'étranglement basés sur une augmentation de la charge de l'utilisateur est d'étendre les serveurs Web. Ceci est relativement facile et peu coûteux, et suffit parfois pour résoudre les problèmes de goulot d'étranglement. Cependant, dans la plupart des cas où la charge des utilisateurs est élevée, la mise à l'échelle de la couche du serveur Web déplace simplement le goulot d'étranglement vers le serveur d'applications. La mise à l'échelle des serveurs d'applications peut être plus complexe et plus coûteuse que les serveurs Web et ne fait généralement que déplacer le goulot d'étranglement vers le serveur de base de données, ce qui est encore plus difficile et coûteux à mettre à l'échelle. Même si vous pouvez dimensionner la base de données, vous obtenez une topologie en forme de triangle, la partie la plus large du triangle étant les serveurs Web (la plus facile à mettre à l'échelle) et la plus petite étant la base de données.
Dans toute application à volume élevé avec une charge d'utilisation simultanée extrêmement importante, la base de données sera généralement le facteur limitant au final le nombre de transactions que vous pouvez traiter simultanément. Bien que diverses technologies de mise en cache et produits de mise à l'échelle de base de données aident à résoudre ces problèmes, il n'en demeure pas moins que l'extension d'une application normale pour des charges extrêmes est une proposition très difficile.

Le modèle d'architecture en nuage est spécifiquement conçu pour traiter et résoudre les problèmes d'évolutivité et de simultanéité. C'est également un modèle d'architecture utile pour les applications qui ont des volumes d'utilisateurs simultanés variables et imprévisibles. Résoudre le problème de grande extensibilité et d’extensibilité variable sur le plan architectural est souvent une meilleure approche que d'essayer d'étendre une base de données ou de moderniser des technologies de mise en cache dans une architecture non évolutive.

Description du modèle d'architecture en nuage

Le modèle d'architecture en nuage minimise les facteurs qui limitent la charge de transaction à traiter de l'application. Ce modèle est basé sur l'idée de la mémoire partagée et distribuée. L'évolutivité élevée est obtenue en supprimant la contrainte de base de données centrale et en utilisant des grilles de données en mémoire répliquées à la place. Les données d'application sont conservées en mémoire et répliquées parmi toutes les unités de traitement actives. Les unités de traitement peuvent être démarrées et arrêtées dynamiquement au fur et à mesure que la charge de l'utilisateur augmente et diminue, abordant ainsi l'évolutivité des variables. Parce qu'il n'y a pas de base de données centrale, le goulot d'étranglement de la base de données est supprimé, offrant une évolutivité quasi infinie au sein de l'application.
La plupart des applications qui correspondent à ce modèle sont des sites Web standard qui reçoivent une requête d'un navigateur et exécutent une sorte d'action. Un site d'enchères est un bon exemple. Le site reçoit continuellement des offres d'internautes via une requête de navigateur. L'application recevrait une enchère pour un élément particulier, enregistrerait cette enchère avec un horodatage et mettrait à jour les dernières informations sur l'enchère pour l'élément, puis renverrait l'information au navigateur.

Il y a deux composants principaux dans ce modèle d'architecture: une unité de traitement et un intergiciel virtualisé. La figure 5-1 illustre le modèle d'architecture en nuage et ses principaux composants.

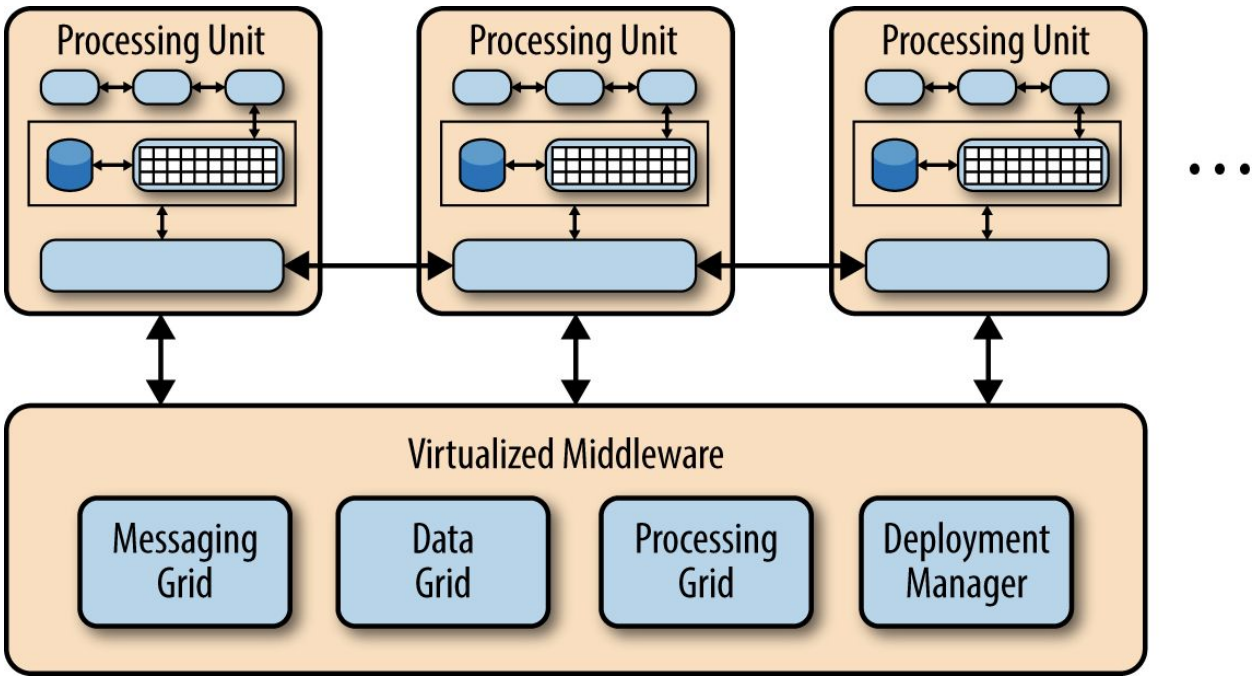

Figure 5-1 Le modèle d'architecture en nuage

Le composant unité de traitement contient les composants de l'application (ou des parties des composants de l'application). Cela inclut les composants Web ainsi que la logique d’affaires. Le contenu de l'unité de traitement varie en fonction du type d'application: des applications Web plus petites seront probablement déployées dans une unité de traitement unique, alors que des

applications plus importantes peuvent diviser la fonctionnalité en plusieurs unités de traitement basées sur les domaines fonctionnels de l'application. L'unité de traitement contient généralement les modules d'application, ainsi qu'une grille de données en mémoire et une mémoire persistante asynchrone facultative pour le basculement. Il contient également un moteur de réplication utilisé par le middleware virtualisé pour répliquer les modifications de données effectuées par une unité de traitement vers d'autres unités de traitement actives.

Le composant middleware virtualisé gère le nettoyage et les communications. Il contient des composants qui contrôlent divers aspects de la synchronisation des données et de la gestion des requêtes. Le middleware virtualisé comprend la grille de messagerie, la grille de données, la grille de traitement et le gestionnaire de déploiement. Ces composants, décrits en détail dans la section suivante, peuvent être personnalisés ou achetés en tant que produits externes.

Dynamique du modèle

La magie du modèle d'architecture en nuage réside dans les composants de middleware virtualisés et de la grille de données en mémoire contenue dans chaque unité de traitement. La Figure 5-2 illustre l'architecture d'unité de traitement type contenant les modules d'application, la grille de données in-memory, la mémoire de persistance asynchrone optionnelle pour le basculement et le moteur de réplication de données.
Le middleware virtualisé est essentiellement le contrôleur de l'architecture et gère les demandes, les sessions, la réplication des données, le traitement des demandes distribuées et le déploiement des unités de processus. Le middleware virtualisé comprend quatre composants d'architecture principaux: la grille de messagerie, la grille de données, la grille de traitement et le gestionnaire de déploiement.

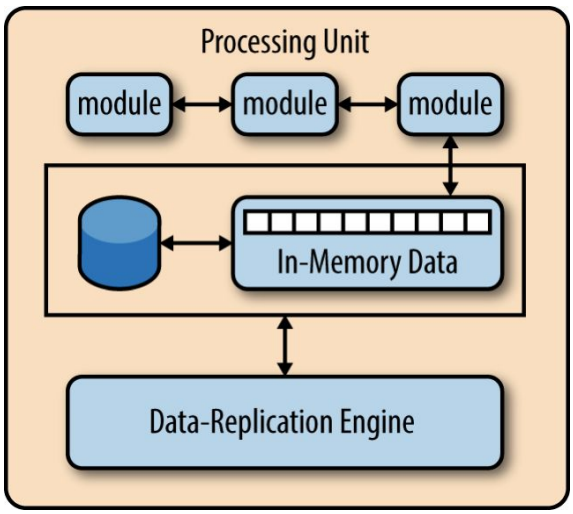

Figure 5-2 Le composant processeur

Grille de messagerie

La grille de messagerie, illustrée à la Figure 5-3, gère les demandes d'entrée et les informations de session. Lorsqu'une requête arrive dans le composant du middleware virtualisé , le composant de messagerie-grille détermine quels composants de traitement actifs sont disponibles pour recevoir la requête et transmet la demande à l'une de ces unités de traitement. La complexité de la grille de messagerie peut aller d'un simple algorithme à tour de rôle à un algorithme suivant, plus complexe, qui permet de suivre la demande traitée par l'unité de traitement.

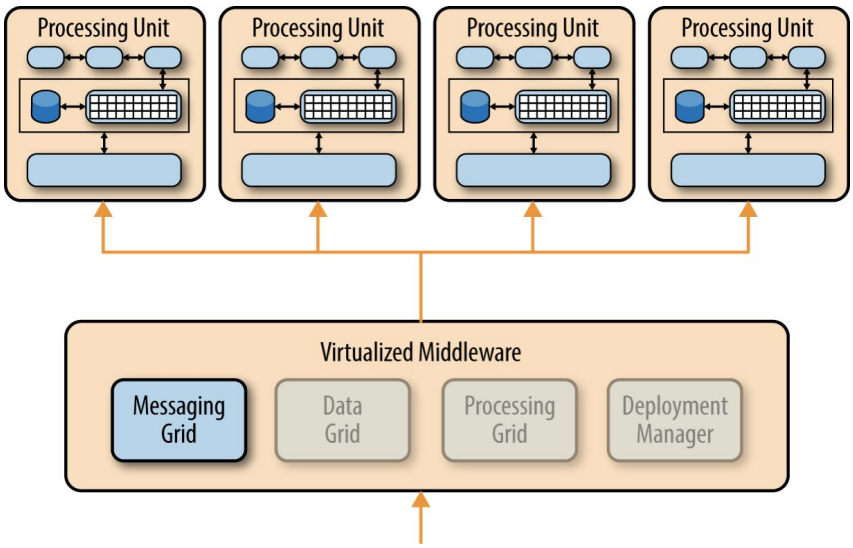

Figure 5-3 Grille de messagerie

Grille de données

La composante de grille de données est peut-être l'élément le plus important et le plus crucial de ce modèle. La grille de données interagit avec le moteur de réplication de données dans chaque unité de traitement pour gérer la réplication de données entre les unités de traitement lorsque des mises à jour de données se produisent. Comme la grille de messagerie peut transmettre une requête à l'une quelconque des unités de traitement disponibles, il est essentiel que chaque unité de traitement contienne exactement les mêmes données dans sa grille de données en mémoire. Bien que la Figure 5-4 montre une réplication de données synchrone entre les unités de traitement, en réalité, cela se fait en parallèle de manière asynchrone et très rapide, complétant parfois la synchronisation des données en quelques microsecondes (un millionième de seconde).

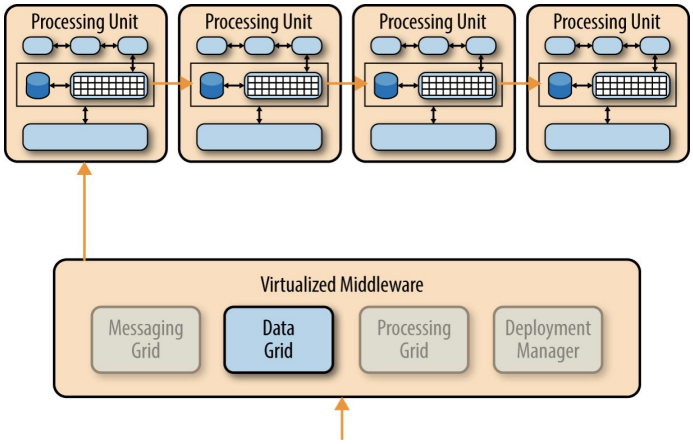

Figure 5-4 Grille de données

Grille de traitement

La grille de traitement, illustrée à la Figure 5-5, est un composant optionnel du middleware virtualisé qui gère le traitement des demandes distribuées lorsqu'il y a plusieurs unités de traitement ou chacune traite une partie de l'application. Si une requête qui nécessite une coordination entre les types d'unités de traitement (par exemple, une unité de traitement des commandes et une unité de traitement client) est nécessaire, c'est le traitement grille qui médiatise et orchestre la requête entre ces deux unités de traitement.

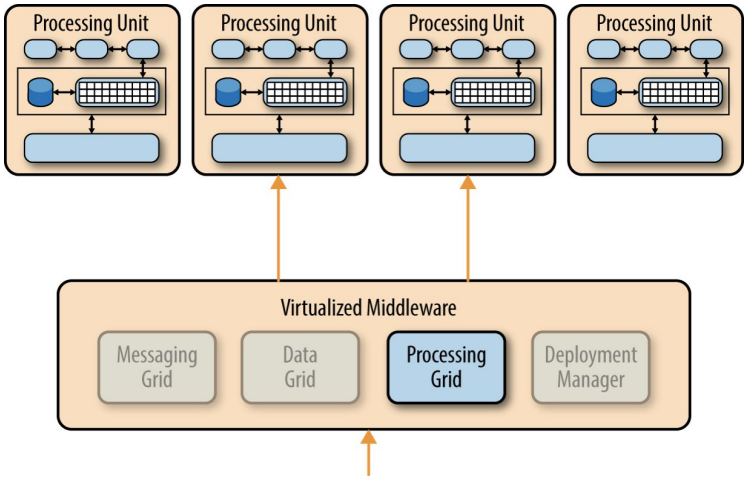

Figure 5-5 Grille de traitement

Gestionnaire de déploiement

Le composant de gestionnaire de déploiement gère le démarrage et l'arrêt dynamiques des unités de traitement en fonction des conditions de charge. Ce composant surveille en

permanence les temps de réponse et les charges utilisateur, et démarre de nouvelles unités de traitement lorsque la charge augmente, et arrête les unités de traitement lorsque la charge diminue. C'est un composant essentiel pour atteindre les besoins d'évolutivité variable au sein d'une application.

Considérations

Le modèle d'architecture en nuage est un modèle complexe et coûteux à mettre en œuvre. C'est un bon choix d'architecture pour les petites applications Web à charge variable (par exemple, les sites de médias sociaux ou les sites d'enchères). Cependant, il n'est pas bien adapté aux applications de bases de données relationnelles à grande échelle traditionnelles avec de grandes quantités de données opérationnelles.

Bien que le modèle d'architecture en nuage ne nécessite pas de banque de données centralisée, il est généralement inclus pour effectuer la charge initiale de la grille de données en mémoire et maintenir de façon asynchrone les mises à jour de données effectuées par les unités de traitement. C'est aussi une pratique courante de créer des partitions séparées provenant de données inactives, afin de réduire l'empreinte mémoire de la grille de données dans chaque unité de traitement.

Il est important de noter que même si le nom de ce modèle est l'architecture en nuage, les unités de traitement (ainsi que le middleware virtualisé) n'ont pas besoin de résider sur des services hébergés dans le nuage. Il peut tout aussi bien résider sur des serveurs locaux.

Du point de vue de l'implémentation du produit, vous pouvez implémenter de nombreux composants d'architecture dans ce modèle via des produits tiers tels que GemFire, JavaSpaces, GigaSpaces, IBM Object Grid, nCache et Oracle Coherence. Comme la mise en œuvre de ce modèle varie énormément en termes de coûts et de capacités (en particulier les temps de réplication des données), en tant qu'architecte, vous devez d'abord établir quels sont vos objectifs et besoins spécifiques avant de sélectionner un produit.

Analyse de cette architecture

Le tableau suivant contient une évaluation et une analyse des caractéristiques d'architecture communes pour le modèle d'architecture en nuage. L'évaluation pour chaque caractéristique est basée sur la tendance naturelle de cette caractéristique en tant que capacité basée sur une implémentation typique du modèle, ainsi que sur la nature généralement reconnue de ce modèle. Pour une comparaison côte à côte de la façon dont ce modèle se rapporte à d'autres modèles dans ce rapport, veuillez vous référer à l'Annexe A à la fin de ce rapport.

Agilité globale : Élevée

L'agilité globale est la capacité à répondre rapidement à un environnement en constante évolution. Étant donné que les unités de traitement (instances déployées de l'application) peuvent être déplacées rapidement, les applications répondent bien aux modifications liées à une augmentation ou à une diminution de la charge utilisateur (modifications de l'environnement). Les architectures créées à l'aide de ce modèle répondent généralement bien aux changements de codage en raison de la petite taille de l'application et de la nature dynamique du modèle.

Facilité de déploiement : Élevée

Bien que les architectures en nuage ne soient généralement pas découplées et distribuées, elles sont dynamiques et des outils “cloud” sophistiqués permettent aux applications d'être facilement «poussées» vers les serveurs, ce qui simplifie le déploiement.

Testabilité : Faible

La simulation de charges d'utilisateurs très élevées dans un environnement de test est à la fois coûteuse et longue, ce qui rend difficile le test des aspects d'évolutivité de l'application.

Performance : Élevée

Des performances élevées sont atteintes grâce aux mécanismes d'accès et de mise en cache des données en mémoire intégrés dans ce modèle.

Évolutivité : Élevée

L'évolutivité élevée vient du fait qu'il y a peu ou pas de dépendance vis-à-vis d'une base de données centralisée, supprimant ainsi essentiellement ce goulot d'étranglement limitant celle-ci.

Facilité de développement : Faible

La mise en cache sophistiquée et les produits de grille de données en mémoire rendent ce modèle relativement complexe à développer, principalement en raison du manque de familiarité avec les outils et les produits utilisés pour créer ce type d'architecture. En outre, un soin particulier doit être pris lors du développement de ces types d'architectures pour s'assurer que rien dans le code source n'affecte les performances et l'évolutivité.

ANNEXE A

Résumé de l'analyse des modèles

La figure A-1 résume la qualification de l'analyse pour chacun des modèles d'architecture décrits dans ce rapport. Ce résumé vous aidera à déterminer quel modèle pourrait être le mieux adapté à votre situation. Par exemple, si votre préoccupation architecturale principale est l'évolutivité, vous pouvez regarder à travers ce graphique et voir que le modèle piloté par événement, le modèle de microservices et le modèle en nuage sont probablement de bons choix de modèle d'architecture. De même, si vous choisissez le modèle d'architecture en couches pour votre application, vous pouvez vous reporter au graphique pour constater que le déploiement, les performances et l'évolutivité peuvent être des zones à risque dans votre architecture.

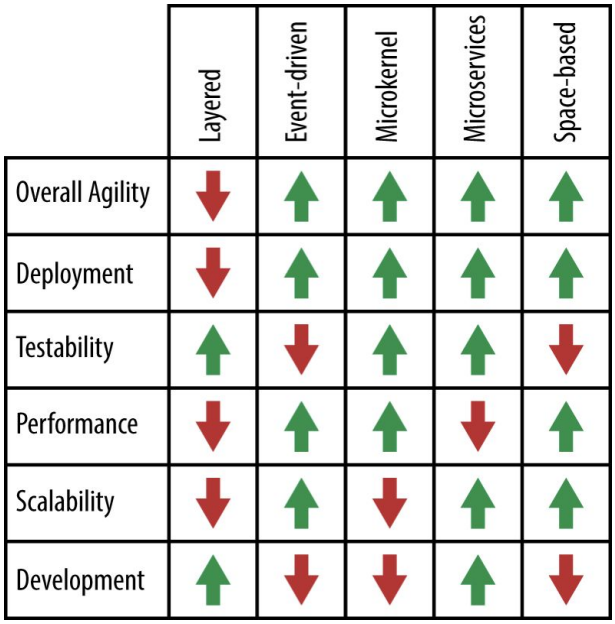

Figure A-1 Sommaire de l’analyse des modèles

Bien que ce tableau vous aidera à choisir le bon modèle, il y a beaucoup plus à prendre en considération lors du choix d'un modèle d'architecture. Vous devez analyser tous les aspects de votre environnement, y compris le support de l'infrastructure, les compétences du développeur, le budget du projet, les délais du projet et la taille de l'application (pour n'en nommer que quelques-uns). Choisir le bon modèle d'architecture est essentiel, car une fois qu'une architecture est en place, il est très difficile (et coûteux) de le changer.

À propos de l'auteur

Mark Richards est un architecte logiciel expérimenté et impliqué dans l'architecture, la conception et la mise en œuvre d'architectures de microservices, d'architectures orientées services et de systèmes distribués dans J2EE et d'autres technologies. Il travaille dans l'industrie du logiciel depuis 1983 et possède une vaste expérience et expertise dans l'application, l'intégration et l'architecture d'entreprise. Mark a été président du New England Java Users Group de 1999 à 2003. Il est l'auteur de nombreux ouvrages et vidéos techniques, dont Software Architecture Fundamentals (O'Reilly video), Enterprise Messaging (O'Reilly video), Java Message Service, 2nd Edition (O'Reilly), and a contributing author to 97 Things Every Software Architect Should Know (O'Reilly). Mark est titulaire d'une maîtrise en informatique et de nombreuses certifications d'architecte et de développeur d'IBM, Sun, The Open Group et BEA. Il est conférencier régulier à la série de symposium No Fluff Just Stuff et a pris la parole dans plus de 100 conférences et groupes d'utilisateurs du monde entier sur divers sujets techniques liés à l'entreprise. Quand il ne travaille pas, Mark peut généralement être vu dans des sentiers de randonnée dans les Montagnes Blanches ou le long du sentier des Appalaches.

Traduction libre : Bernard Beaulieu