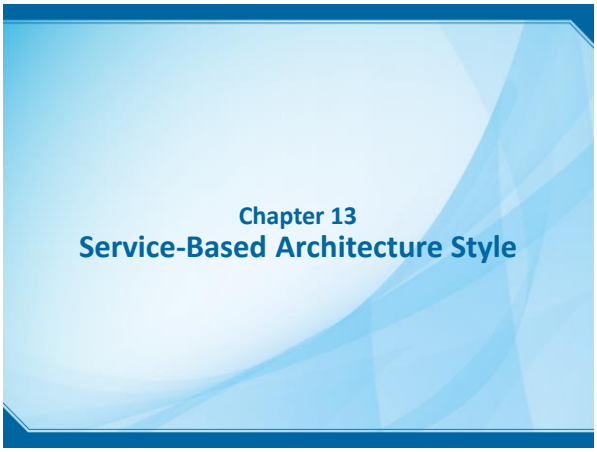

1




2

## Service-Based Architecture Style

*Introduction*

- Neal Ford argued that organizations transition more easily from a monolithic architecture to a service-based architecture than to a microservices architecture.
- Microservice architectures have many benefits, and Ford recommended them for *greenfield projects*. But for organizations which already have a monolithic architecture there are many hurdles to overcome in switching to microservices: breaking apart the code, likely breaking apart a monolithic database, adopting DevOps practices, monitoring, dealing with the network, distributed transactions, etc. Switching to a service-based architecture doesn't require nearly as much change
- A service-based architecture provides more delivery speed than a monolith or Service-Oriented Architecture (SOA) by breaking the code apart in the domain-centric way advocated by microservice and Domain-Driven Design (DDD) proponents. SOA advocates breaking the architecture apart into layers rather than by domain.



3

## Service-Based Architecture Style

*Topology*

- The basic topology of service-based architecture follows a distributed macro layered structure consisting of a separately deployed user interface, separately deployed remote coarse-grained services, and a monolithic database.
- Services within this architecture style are typically coarse-grained "portions of an application" (usually called domain services) that are independent and separately deployed.
- Because the services typically share a single monolithic database, the number of services within an application context generally range between 4 and 12 services, with the average being about 7 services.

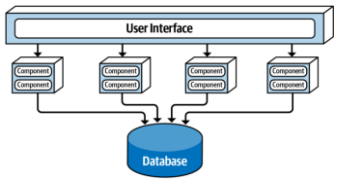



*Figure 13-1. Basic topology of the service-based architecture style*



4

## Service-Based Architecture Style

*Features*

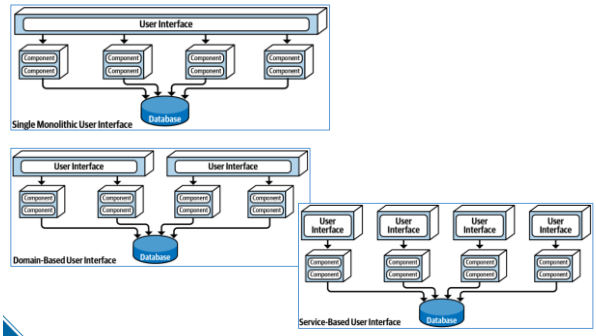
- In most cases there is only a single instance of each domain service within a service-based architecture. However, based on scalability, fault tolerance, and throughput needs, multiple instances of a domain service can certainly exist.
- Services are accessed remotely from a user interface using a remote access protocol.
  - While REST is typically used to access services from the user interface, messaging, remote procedure call (RPC), or even SOAP could be used as well.
  - While an API layer consisting of a proxy or gateway can be used to access services from the user interface (or other external requests), in most cases the user interface accesses the services directly using a service locator pattern embedded within the user interface, API gateway, or proxy.
- One important aspect of service-based architecture is that it typically uses a centrally shared database. This allows services to leverage SQL queries and joins in the same way a traditional monolithic layered architecture would.



5

## Service-Based Architecture Style

*User interface variants*



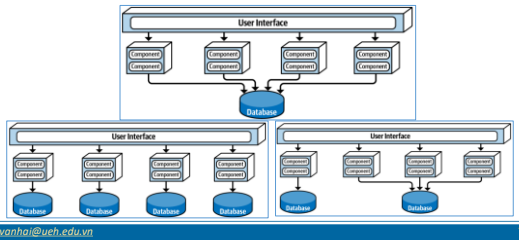



6

## Service-Based Architecture Style

*Database variants*

- Opportunities may exist to break apart a single monolithic database into separate databases, even going as far as domain-scoped databases matching each domain service (similar to microservices).
- In these cases, it is important to make sure the data in each separate database is not needed by another domain service. This avoids interservice communication between domain services (something to definitely avoid with service-based architecture) and also the duplication of data between databases






7

## Service-Based Architecture Style

*Adding API*

- It is also possible to add an API layer consisting of a reverse proxy or gateway between the user interface and services.
- This is a good practice when exposing domain service functionality to external systems or when consolidating shared cross-cutting concerns and moving them outside of the user interface (such as metrics, security, auditing requirements, and service discovery).

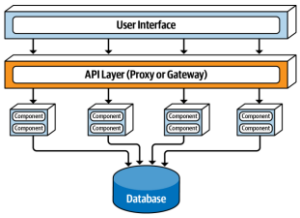



*Figure 13-4. Adding an API layer between the user interface and domain services*



8

## Service-Based Architecture Style

*Service Design and Granularity*

- Because domain services in a service-based architecture are generally coarse-grained, each domain service is typically designed using a layered architecture style consisting of an API facade layer, a business layer, and a persistence layer.
- Another popular design approach is to domain partition each domain service using sub-domains similar to the modular monolith architecture style.
- Regardless of the service design, a domain service must contain some sort of API access facade that the user interface interacts with to execute some sort of business functionality. The API access facade typically takes on the responsibility of orchestrating the business request from the user interface.

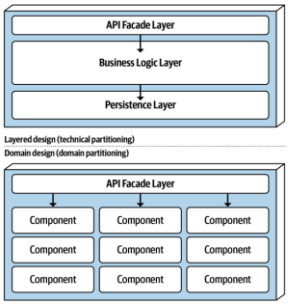



*Figure 13-5. Domain service design variants*



9

## Service-Based Architecture Style

*Database Partitioning*

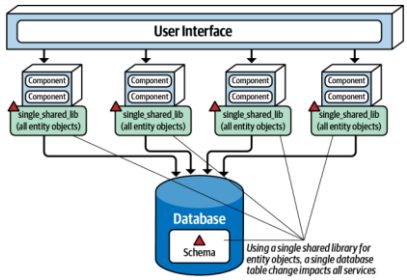



*Figure 13-6. Using a single shared library for database entity objects*



10

## Service-Based Architecture Style

*Database Partitioning*

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

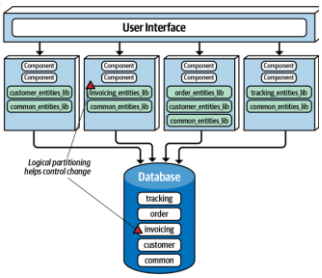



*Figure 13-7. Using multiple shared libraries for database entity objects*



11

## Service-Based Architecture Style

*Example Architecture*

- To illustrate the flexibility and power of the service-based architecture style, consider the real-world example of an electronic recycling system used to recycle old electronic devices (such as an iPhone or Galaxy cell phone). The processing flow of recycling old electronic devices works as follows: first, the customer asks the company (via a website or kiosk) how much money they can get for the old electronic device (called quoting). If satisfied, the customer will send the electronic device to the recycling company, which in turn will receive the physical device (called receiving). Once received, the recycling company will then assess the device to determine if the device is in good working condition or not (called assessment). If the device is in good working condition, the company will send the customer the money promised for the device (called accounting). Through this process, the customer can go to the website at any time to check on the status of the item (called item status). Based on the assessment, the device is then recycled by either safely destroying it or reselling it (called recycling). Finally, the company periodically runs ad hoc and scheduled financial and operational reports based on recycling activity (called reporting).

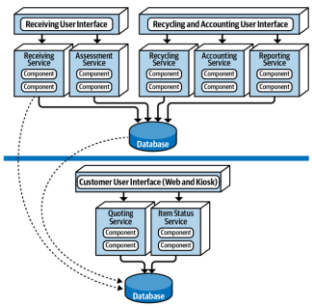



*Figure 13-8. Electronics recycling example using service-based architecture*



12

## Service-Based Architecture Style

*Architecture Characteristics Ratings*

- Service-based architecture is a domain-partitioned architecture, meaning that the structure is driven by the domain rather than a technical consideration (such as presentation logic or persistence logic).
- Although service-based architecture doesn't contain any five-star ratings, it nevertheless rates high (four stars) in many important and vital areas.
- Breaking apart an application into separately deployed domain services using this architecture style allows for faster change (agility), better test coverage due to the limited scope of the domain (testability), and the ability for more frequent deployments carrying less risk than a large monolith (deployability).
- These three characteristics lead to better time-to-market, allowing an organization to deliver new features and bug fixes at a relatively high rate.

| Architecture characteristic | Star rating |
|---|---|
| Partitioning type | Domain |
| Number of quanta | 1 to many |
| Deployability | ★★★★ |
| Elasticity | ★★ |
| Evolutionary | ★★★ |
| Fault tolerance | ★★★★ |
| Modularity | ★★★★ |
| Overall cost | ★★★★ |
| Performance | ★★★ |
| Reliability | ★★★★ |
| Scalability | ★★★ |
| Simplicity | ★★★ |
| Testability | ★★★★ |



13

## Chapter 14
# Event-Driven Architecture Style

15

## Event-Driven Architecture Style

*The request-based model*

- Most applications follow what is called a request-based model.
  - Requests made to the system to perform some sort of action are send to a request orchestrator.
  - The request orchestrator is typically a user interface, but it can also be implemented through an API layer or enterprise service bus.
  - The role of the request orchestrator is to deterministically and synchronously direct the request to various request processors. The request processors handle the request, either retrieving or updating information in a database.

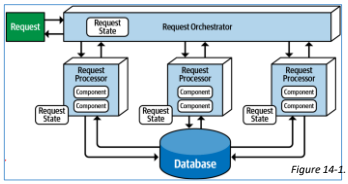



*Figure 14-1. Request-based model*



16

## Event-Driven Architecture Style

*Introduction*

- An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website.
- Event-driven architecture is made up of decoupled event processing components that asynchronously receive and process events. It can be used as a standalone architecture style or embedded within other architecture styles (such as an eventdriven microservices architecture).
- An event-based model, on the other hand, reacts to a particular situation and takes action based on that event.
  - An example of an event-based model is submitting a bid for a particular item within an online auction. Submitting the bid is not a request made to the system, but rather an event that happens after the current asking price is announced. The system must respond to this event by comparing the bid to others received at the same time to determine who is the current highest bidder.



17

## Event-Driven Architecture Style

*Topology*

- There are two primary topologies within event-driven architecture:
  - the mediator topology and
  - the broker topology.
- The mediator topology is commonly used when you require control over the workflow of an event process, whereas the broker topology is used when you require a high degree of responsiveness and dynamic control over the processing of an event.
- Because the architecture characteristics and implementation strategies differ between these two topologies, it is important to understand each one to know which is best suited for a particular situation



18

## Event-Driven Architecture Style

*Broker Topology*

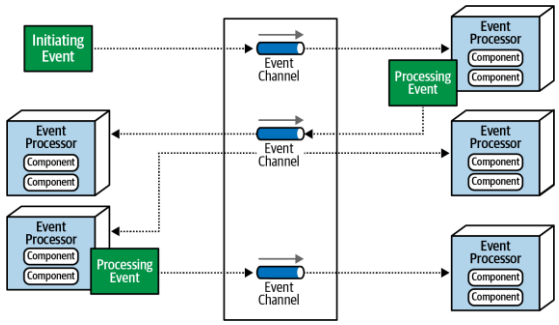



*Figure 14-2. Broker topology*



19

## Event-Driven Architecture Style

*Broker Topology*

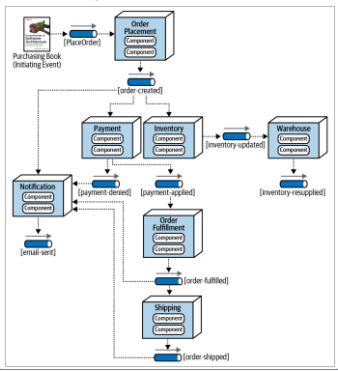
- There is no central event-mediator. Rather, the message flow is distributed across the event processor components in a chain-like broadcasting fashion through a lightweight message broker (such as RabbitMQ, ActiveMQ, HornetQ, and so on).
- This topology is useful when you have a relatively simple event processing flow and you do not need central event orchestration and coordination.
- The event broker component is usually federated (meaning multiple domain-based clustered instances), where each federated broker contains all of the event channels used within the event flow for that particular domain. Because of the decoupled asynchronous fire-and-forget broadcasting nature of the broker topology, topics (or topic exchanges in the case of AMQP) are usually used in the broker topology using a publish-and-subscribe messaging model.



20

## Event-Driven Architecture Style

*Broker Topology - example*

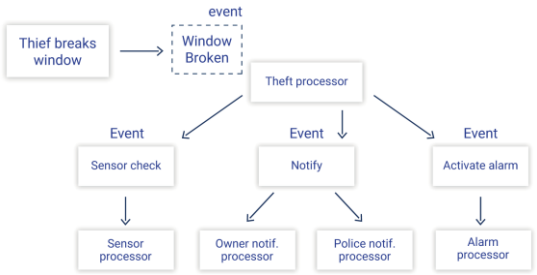





21

## Event-Driven Architecture Style

*Broker Topology - example (2)*






22

## Event-Driven Architecture Style

*Broker Topology – Trade-off*

| Advantages | Disadvantages |
|---|---|
| Highly decoupled event processors | Workflow control |
| High scalability | Error handling |
| High responsiveness | Recoverability |
| High performance | Restart capabilities |
| High fault tolerance | Data inconsistency |



23

## Event-Driven Architecture Style

*Mediator Topology*

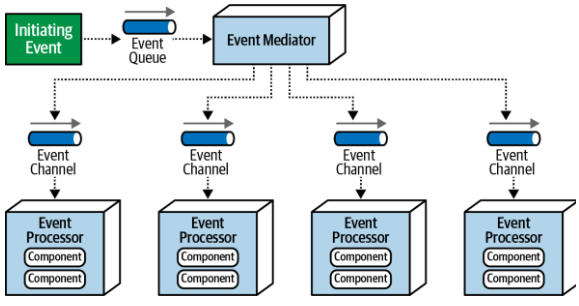



*Figure 14-5. Mediator topology*



24

## Event-Driven Architecture Style

*Mediator Topology*

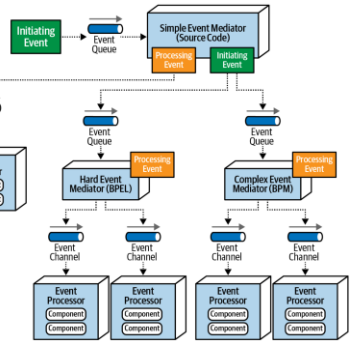
- In most implementations of the mediator topology, there are multiple mediators, usually associated with a particular domain or grouping of events. This reduces the single point of failure issue associated with this topology and also increases overall throughput and performance.
  - For example, there might be a customer mediator that handles all customer-related events (such as new customer registration and profile update), and another mediator that handles order-related activities (such as adding an item to a shopping cart and checking out).
- The event mediator can be implemented in a variety of ways, depending on the nature and complexity of the events it is processing.
  - For example, for events requiring simple error handling and orchestration, a mediator such as Apache Camel, Mule ESB, or Spring Integration will usually suffice.
  - Message flows and message routes within these types of mediators are typically custom written in programming code (such as Java or C#) to control the workflow of the event processing.



25

## Event-Driven Architecture Style

*Mediator Topology*




BPEL - Business Process Execution Language
BPM - Business Process Management
BPMN - Business Process Model and Notation

https://bpmn.io/
https://www.jbpm.org/

*Figure 14-6. Delegating the event to the appropriate type of event mediator*



26

## Event-Driven Architecture Style

*Mediator Topology – example*

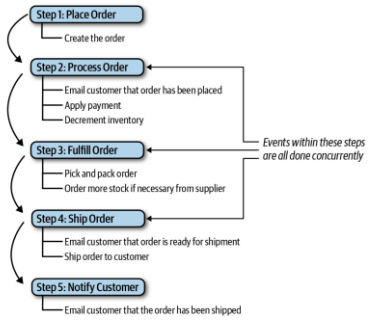


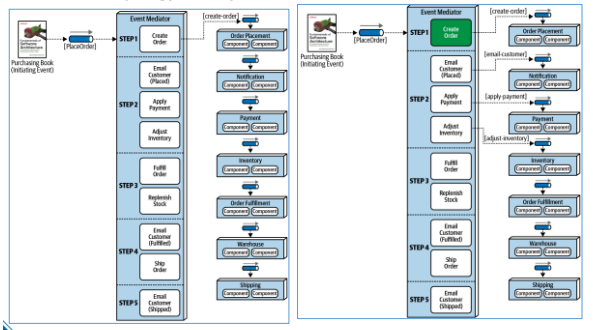

*Figure 14-7. Mediator steps for placing an order*



27

## Event-Driven Architecture Style

*Mediator Topology - example*

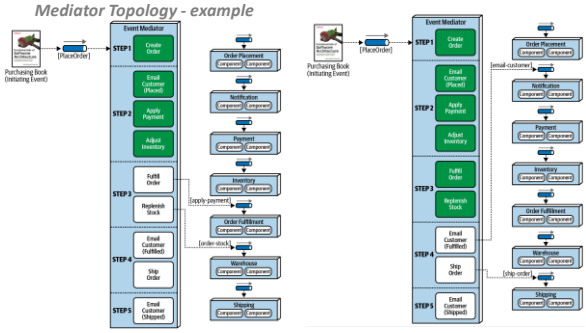




28

## Event-Driven Architecture Style

*Mediator Topology - example*

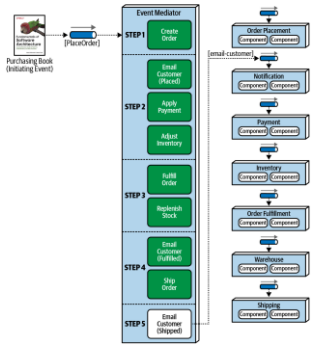




29

## Event-Driven Architecture Style

*Mediator Topology - example*






30

## Event-Driven Architecture Style

*Mediator Topology – Trade-off*

| Advantages | Disadvantages |
|---|---|
| Workflow control | More coupling of event processors |
| Error handling | Lower scalability |
| Recoverability | Lower performance |
| Restart capabilities | Lower fault tolerance |
| Better data consistency | Modeling complex workflows |



31

## Event-Driven Architecture Style

*Architecture Characteristics Ratings*

| Architecture characteristic | Star rating |
|---|---|
| Partitioning type | Technical |
| Number of quanta | 1 to many |
| Deployability | ★★★ |
| Elasticity | ★★★ |
| Evolutionary | ★★★★★ |
| Fault tolerance | ★★★★★ |
| Modularity | ★★★★ |
| Overall cost | ★★★ |
| Performance | ★★★★★ |
| Reliability | ★★★ |
| Scalability | ★★★★★ |
| Simplicity | ★ |
| Testability | ★★ |



40

## Chapter 15
# Space-Based Architecture Style

41

## Space-Based Architecture Style

*Introduction*

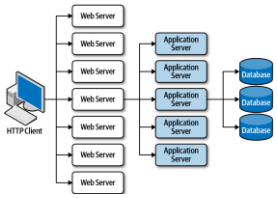



*Figure 15-1. Scalability limits within a traditional web-based topology*

- The space-based architecture style is specifically designed to address problems involving high scalability, elasticity, and high concurrency issues. It is also a useful architecture style for applications that have variable and unpredictable concurrent user volumes. Solving the extreme and variable scalability issue architecturally is often a better approach than trying to scale out a database or retrofit caching technologies into a non-scalable architecture.



42

## Space-Based Architecture Style

*Topology*

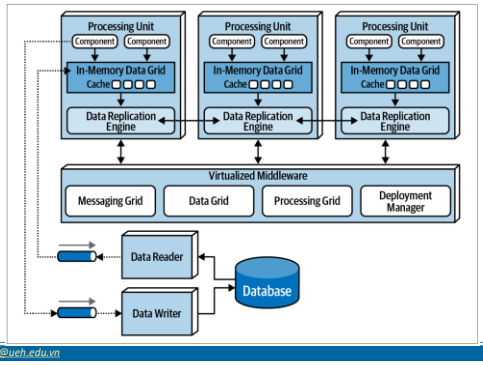
- Space-based architecture gets its name from the concept of tuple space, the technique of using multiple parallel processors communicating through shared memory.
- High scalability, high elasticity, and high performance are achieved by removing the central database as a synchronous constraint in the system and instead leveraging replicated in-memory data grids.
- Application data is kept in-memory and replicated among all the active processing units.
  - When a processing unit updates data, it asynchronously sends that data to the database, usually via messaging with persistent queues.
  - Processing units start up and shut down dynamically as user load increases and decreases, thereby addressing variable scalability.
- Because there is no central database involved in the standard transactional processing of the application, the database bottleneck is removed, thus providing near-infinite scalability within the application.



43

## Space-Based Architecture Style

*General Topology*






44

## Space-Based Architecture Style

*Topology - Processing Unit*

- The processing unit contains the application logic (or portions of the application logic). This usually includes web-based components as well as backend business logic. The contents of the processing unit vary based on the type of application.
- The contents of the processing unit vary based on the type of application. Smaller web-based applications would likely be deployed into a single processing unit, whereas larger applications may split the application functionality into multiple processing units based on the functional areas of the application.
- The processing unit can also contain small, single-purpose services (as with microservices).
- In addition to the application logic, the processing unit also contains an inmemory data grid and replication engine usually implemented through such products as Hazelcast, Apache Ignite, and Oracle Coherence.

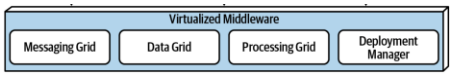


45

## Space-Based Architecture Style

*Topology - Virtualized Middleware*

- The virtualized middleware handles the infrastructure concerns within the architecture that control various aspects of data synchronization and request handling.
- The components that make up the virtualized middleware include:
  - messaging grid,
  - data grid,
  - processing grid, and
  - deployment manager.






46

## Space-Based Architecture Style

*Topology - Virtualized Middleware: Messaging Grid*

- The messaging grid manages input request and session state.
- When a request comes into the virtualized middleware, the messaging grid component determines which active processing components are available to receive the request and forwards the request to one of those processing units.
- The complexity of the messaging grid can range from a simple round-robin algorithm to a more complex next-available algorithm that keeps track of which request is being processed by which processing unit.
- This component is usually implemented using a typical web server with load-balancing capabilities (such as HA Proxy and Nginx).

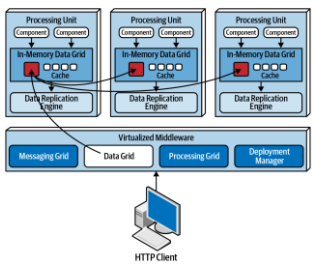


47

## Space-Based Architecture Style

*Topology - Virtualized Middleware: Data Grid*

- The data grid component is perhaps the most important and crucial component in this architecture style. In most modern implementations the data grid is implemented solely within the processing units as a replicated cache.
- However, for those replicated caching implementations that require an external controller, or when using a distributed cache, this functionality would reside in both the processing units as well as in the data grid component within the virtualized middleware.
- Since the messaging grid can forward a request to any of the processing units available, it is essential that each processing unit contains exactly the same data in its in-memory data grid.






48

## Space-Based Architecture Style

*Topology - Virtualized Middleware*

- Processing grid
  - The processing grid is an optional component within the virtualized middleware that manages orchestrated request processing when there are multiple processing units involved in a single business request.
  - If a request comes in that requires coordination between processing unit types (e.g., an order processing unit and a payment processing unit), it is the processing grid that mediates and orchestrates the request between those two processing units.
- Deployment manager
  - The deployment manager component manages the dynamic startup and shutdown of processing unit instances based on load conditions. This component continually monitors response times and user loads, starts up new processing units when load increases, and shuts down processing units when the load decreases. It is a critical component to achieving variable scalability (elasticity) needs within an application.



49

## Space-Based Architecture Style

*Topology - Data Pumps*

- A data pump is a way of sending data to another processor which then updates data in a database.
- Data pumps are a necessary component within space-based architecture, as processing units do not directly read from and write to a database.
- Data pumps within a space-based architecture are always asynchronous, providing eventual consistency with the in-memory cache and the database. When a processing unit instance receives a request and updates its cache, that processing unit becomes the owner of the update and is therefore responsible for sending that update through the data pump so that the database can be updated eventually.
- Data pumps are usually implemented using messaging.
- In most cases there are multiple data pumps, each one usually dedicated to a particular domain or subdomain.
- Data pumps usually have associated contracts, including an action associated with the contract data (add, delete, or update). The contract can be a JSON schema, XML schema, an object, or even a value-driven message (map message containing name-value pairs).

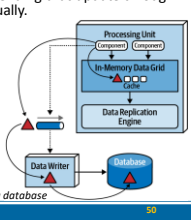



*Figure 15-7. Data pump used to send data to a database*



50

## Space-Based Architecture Style

*Topology - Data Writers*

- The data writer component accepts messages from a data pump and updates the database with the information contained in the message of the data pump.
- Data writers can be implemented as services, applications, or data hubs (such as Ab Initio). The granularity of the data writers can vary based on the scope of the data pumps and processing units.
- Types:
  - A domain-based data writer contains all of the necessary database logic to handle all the updates within a particular domain (such as customer), regardless of the number of data pumps it is accepting.
  - Alternatively, each class of processing unit can have its own dedicated data writer component. In this model the data writer is dedicated to each corresponding data pump and contains only the database processing logic for that particular processing unit (such as Wallet). While this model tends to produce too many data writer components, it does provide better scalability and agility due to the alignment of processing unit, data pump, and data writer



51

## Space-Based Architecture Style

*Topology - Data Writers*

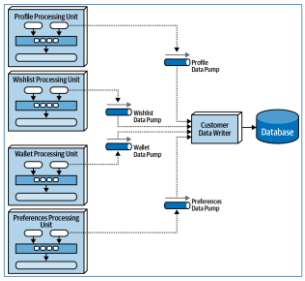



*Figure 15-8. Domain-based data writer*

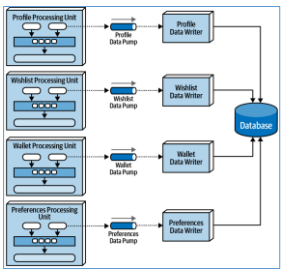



*Figure 15-9. Dedicated data writers for each data pump*



52

## Space-Based Architecture Style

*Topology - Data Readers*

- Whereas data writers take on the responsibility for updating the database, data readers take on the responsibility for reading data from the database and sending it to the processing units via a reverse data pump.
- Data readers are only invoked under one of three situations: a crash of all processing unit instances of the same named cache, a redeployment of all processing units within the same named cache, or retrieving archive data not contained in the replicated cache.

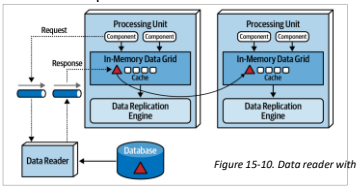



*Figure 15-10. Data reader with reverse data pump*



53

## Space-Based Architecture Style

*Implementation Examples - Concert Ticketing System*

- Concert ticketing systems have a unique problem domain in that concurrent user volume is relatively low until a popular concert is announced. Once concert tickets go on sale, user volumes usually spike from several hundred concurrent users to several thousand (possibly in the tens of thousands, depending on the concert), all trying to
- acquire a ticket for the concert (hopefully, good seats!). Tickets usually sell out in a matter of minutes, requiring the kind of architecture characteristics supported by space-based architecture.
- There are many challenges associated with this sort of system. First, there are only a certain number of tickets available, regardless of the seating preferences. Seating availability must continually be updated and made available as fast as possible given the high number of concurrent requests. Also, assuming assigned seats are an option, seating availability must also be updated as fast as possible. Continually accessing a central database synchronously for this sort of system would likely not work—it would be very difficult for a typical database to handle tens of thousands of concurrent requests through standard database transactions at this level of scale and update frequency.
- Space-based architecture would be a good fit for a concert ticketing system due to the high elasticity requirements required of this type of application. An instantaneous increase in the number of concurrent users wanting to purchase concert tickets would be immediately recognized by the deployment manager, which in turn would start up a large number of processing units to handle the large volume of requests. Optimally, the deployment manager would be configured to start up the necessary number of processing units shortly before the tickets went on sale, therefore having those instances on standby right before the significant increase in user load.



58

## Space-Based Architecture Style

*Implementation Examples - Online Auction System*

- Online auction systems (bidding on items within an auction) share the same sort of characteristics as the online concert ticketing systems described previously—both require high levels of performance and elasticity, and both have unpredictable spikes in user and request load. When an auction starts, there is no way of determining how many people will be joining the auction, and of those people, how many concurrent bids will occur for each asking price.
- Space-based architecture is well suited for this type of problem domain in that multiple processing units can be started as the load increases; and as the auction winds down, unused processing units could be destroyed. Individual processing units can be devoted to each auction, ensuring consistency with bidding data. Also, due to the asynchronous nature of the data pumps, bidding data can be sent to other processing (such as bid history, bid analytics, and auditing) without much latency, therefore increasing the overall performance of the bidding process.



59

## Space-Based Architecture Style

*Architecture Characteristics Ratings*

| Architecture characteristic | Star rating |
|---|---|
| Partitioning type | Domain and technical |
| Number of quanta | 1 to many |
| Deployability | ★★★ |
| Elasticity | ★★★★★ |
| Evolutionary | ★★★ |
| Fault tolerance | ★★★ |
| Modularity | ★★★ |
| Overall cost | ★★ |
| Performance | ★★★★★ |
| Reliability | ★★★★ |
| Scalability | ★★★★★ |
| Simplicity | ★ |
| Testability | ★ |



60




61