The Addison-Wesley Signature Series

A Vaughn Vernon Signature Book

Continuous Architecture in Practice

Software Architecture in the Age of Agility and DevOps

Murat Erder
Pierre Pureur
Eoin Woods

Foreword by Kurt Bittner

Continuous Architecture Principles

Principle 1: Architect products; evolve from projects to products.

Principle 2: Focus on quality attributes, not on functional requirements.

Principle 3: Delay design decisions until they are absolutely necessary.

Principle 4: Architect for change—leverage the "power of small."

Principle 5: Architect for build, test, deploy, and operate.

Principle 6: Model the organization of your teams after the design of the system you are working on.

This becomes more relevant for larger systems or systems of systems. Neglecting proper data ownership can result in different components interpreting the data in different ways—in the worst case, resulting in inconsistency in business values.

Let us go back to the components we have in our TFX system and think about them from a data-first perspective. It is important to understand which component is going to **master** which set of data. One simple but effective architectural practice is to create a table of the key data entities and map them to services, as shown in Table 3.3. If you want, you can detail further with a traditional CRUD (create, read, update, delete) view. However, we believe this table is sufficient for managing dependencies.

Looking at a system from this data-first perspective enables us to see which components master a lot of data and which are consumers of data. As a result, we clearly understand data dependencies of the overall system. In the simple example in Table 3.3, it is obvious that the Counterparty Manager and Contract Manager master most of the major data elements; the others are predominantly consumers. The Document Manager and Payment Service are the components that master only a few elements and have limited dependencies on other components. By managing data dependencies, the team ensures loosely coupled components and evolvability of the system, which supports applying principle 4, *Architect for change—leverage the "power of small."* In our case study, the team has taken the approach of one component mastering a data class.

Once a decision is made that a data entity is owned by a single component, we must decide how the data should be shared with other components. It is important that a data entity is always interpreted consistently across the entire system. The safe way to accomplish this is to share data by reference, that is, by passing an identifier that uniquely identifies the data element. Any component that needs further detail regarding a data entity can always ask the data source for additional attributes that it might need.

Table 3.3 *TFX Data Ownership*

Main Data Entities	Document Manager	Contract Manager	Counterparty Manager	Fees and Commissions Manager	Payment Service
L/C terms		Master	Consume	Consume	
L/C document	Master	Consume			
Good		Master	Consume		
Buyer/seller		Consume	Master	Consume	
Bank		Consume	Master		Consume
Payment			Consume		Master
Fees and commissions		Consume		Master	

Whenever you have a system with distributed data, you also have to look out for **race conditions**, where the data can be inconsistent due to multiple updates to the same data set. Passing data by value increases the possibility of this happening. For example, let us assume that a component (e.g., the UI Service) calls the Fees and Commissions Manager to calculate fees and commissions for a certain seller. This requires data about both the seller and the contract. If only the relevant identifiers for the seller and contract are passed, the Fees and Commissions Manager can then call the Contract Manager and Counterparty Manager to get other attributes required for the calculation. This guarantees that the data about the seller and contract is always represented consistently and isolates us from any data model changes in the Contract Manager and Counterparty Manager. If additional data attributes (e.g., contract value, settlement date) were added to the original call, it could create a data-consistency risk because the details of the contract or seller could have been modified by other services, particularly if the UI Service had cached the values for a period of time. We can assume this is a minor risk given the nature of the TFX business process, but it is an important consideration in distributed systems.

As can be seen from this example, referring to data entities by reference creates additional communication between components. This is a significant tradeoff that can add extra read workload on the databases, which can be crippling to a system if not managed effectively. Another way to look at it is to say that we have a tradeoff between modifiability and performance.

Let us assume that the TFX team runs performance tests and discovers that calls from the Fees and Commissions Manager cause extra load on the Counterparty Manager. They can then decide that the attributes for the seller required by the Fees and Commission Manager are limited and are already present in the UI. As discussed previously, the risk of inconsistency (i.e., the attributes being changed between the call to the Fees and Commission Manager) is low. Consequently, they can decide to pass those attributes directly to the Fees and Commission Manager.

In this example, to meet performance requirements, the TFX team decides not to strictly adhere to passing information only by reference. Instead, they pass all the attributes required (i.e., passed data) by value. However, this is done and documented as an explicit decision.

In this example, we also mentioned that the risk of inconsistency between updates is low. Managing multiple updates to the same dataset from different components in a distributed system can get complex. For example, how should the team manage a scenario where two different components want to update the same contract in the Contract Manager? This can naturally happen in the TFX case study if we assume the importer and exporter are updating attributes of the contract at the same time. If both UIs were accessing the same database, traditional database techniques such as locks could be used to avoid a conflict. However, in the case of TFX, the database is not shared—the UI Service for both importer and exporter can retrieve the same

version of the contract from the Contract Manager and independently try to submit updates via an API call. To protect against inconsistent updates, the team would need additional logic both in the Contract Manager and the UI Service. For example, the Contract Manager could check the version of the contract that an update request is made against. If the version in the database is more recent, then the data must be returned to the UI Service with the latest information provided, and a resubmission must be requested. The UI Service will also need additional logic to handle this condition. Distributed systems give us flexibility and resilience, but we also need to make our components deal with unexpected events. We cover resiliency in Chapter 7, "Resilience as an Architectural Concern."

Before leaving the topic of data ownership, we briefly touch on the topic of **metadata**, which, as every technologist knows, is data about data. It basically means that you have business-specific data, such as attributes associated with a contract. Then you have data describing either the attributes or the entire contract, such as when it was created, last updated, version, and component (or person) who updated it. Managing metadata has become increasingly important, particularly for big data systems and artificial intelligence. There are three main reasons for this growing importance.

First, large data analytics systems contain data from multiple sources in varying formats. If you have sufficient metadata, you can discover, integrate, and analyze such data sources in an efficient manner.

Second is the need to track data lineage and provenance. For any output generated from the system, you should be able to clearly identify the journey of the data from creation to the output. For example, in the TFX system, let us assume the team has implemented an attribute in the Analytics Manager that calculates and tells the average payment amount for a particular seller against multiple L/Cs. They should be able to clearly trace how that attribute was calculated and all data attributes from each service (e.g., Counterparty Manager, Contract Manager) that were required as an input to the calculation. This sounds easy for simple data flows, but it can get complex in large data analytics environments and systems of systems. Metadata is what makes addressing such a challenge much easier.

Finally, as highlighted by principle 5, *Architect for build, test, deploy, and operate*, Continuous Architecture emphasizes the importance of automating not only software development but also data management processes, such as data pipelines. Metadata is a strong enabler of such capabilities.

Data Integration

The previous section presented a data-centric view and touched briefly on data sharing. Let us now look at data integration in more detail. The topic of data integration covers a wide area, including batch integration, traditional extract–transform–load (ETL), messaging, streaming, data pipelines, and APIs. Providing a detailed overview

of integration styles is beyond the scope of the book. At a high level, there are two reasons for data integration:

- Integrating data between components to facilitate a business process, such as how data is shared between different services in the TFX system. This is where we would find messaging, APIs, remote procedure calls, and sometimes file-based integration. We could also architect our system by utilizing streaming technologies, which would be a different approach to data integration.
- Moving data from multiple sources into a single environment to facilitate additional capabilities such as monitoring and analytics. This is where we traditionally used ETL components, and these days, we consider other solutions such as data pipelines.

This section focuses on APIs to provide some examples of how to think about the first type of integration. The reason for this section is to provide a glimpse into the larger world of integration and to emphasize the importance of thinking about integration from a data-centric perspective. We focused on APIs mainly because of their dominance in the industry as an integration style. The same approach can apply easily to messaging. As we explain in Chapter 5, the team can architect the TFX system to switch between synchronous and asynchronous communications based on the tradeoffs they want to make.

Let us start with a brief look into the Semantic Web and resources. The Web is the most successful distributed system that is scalable and resilient. It is built on distributing data via a common protocol, the Hypertext Transfer Protocol (HTTP). The Semantic Web (sometimes referred to as *linked data*) is a method of publishing structured data so that it can be interlinked and become more useful through semantic queries. It builds on standard Web technologies such as HTTP, the Resource Description Framework (RDF), and URIs, but rather than using them to serve Web pages for human readers, it extends them to share information in a way that can be read automatically by computers.[18] Tim Berners-Lee[19] defined four principles of the Semantic Web:

1. Use URIs (Universal Resource Identifiers) as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs so that they can discover more things.

18. https://en.wikipedia.org/wiki/Linked_data

19. Tim Berners-Lee, *Linked Data* (last updated 2009), https://www.w3.org/DesignIssues/LinkedData.html

The Semantic Web approach has not become as widely used in the industry as originally anticipated. However, the concepts behind the approach are very powerful and provide insights into how to deal with data integration. At the core of data distribution is the concept of a *resource*. A resource is a data structure that you can expect to be returned by a Web endpoint. Representational state transfer (REST) is the most prevalent integration approach that relies on the concept of a resource. The huge success of REST (compared to former integration approaches such as Simple Object Access Protocol [SOAP]) is partially because it is an architectural pattern rather than a tightly specified standard or a specific set of technologies. The initial elements of the REST architectural pattern were first introduced by Roy Fielding in 2000,[20] and it primarily consists of a set of architectural constraints that guide its users to use mature and well-understood Web technologies (particularly HTTP) in a specific way to achieve loosely coupled interoperability. Two key benefits REST brought to integration was to change the focus on interfaces from a verb-centric view to a noun-centric view and to eliminate complex, centrally managed middleware such as enterprise service buses (ESBs). In essence, rather than defining a very large set of verbs that could be requested (e.g., get_customer, update_account), a set of very basic verbs (HTTP methods, e.g., GET, PUT, POST, DELETE) are executed on a noun—which is a resource such as Account. However, it is important to note that there are several different ways in which you can model and build a successful REST API, which goes beyond the scope of this book.

Although RESTful APIs are widely used, they are not the only approach you can take. Two examples of other approaches are:

- GraphQL[21] enables clients to ask for a specific set of data in one call rather than in multiple calls, as required in REST.
- gRPC[22] is a lightweight, efficient manner to conduct remote procedure calls via defined contracts.

For the TFX system, the team decides that they will expose a set of REST APIs externally to enable partners and clients to access their information. Exposing your data via APIs is an extremely common mechanism used by platform providers not only for technical integration but also to support different business models that support revenue generation, known as the *API economy*. The choice of REST is driven mainly by its wide adoption in the industry.

20. Fielding R. Architectural styles and the design of network-based software architectures. University of California, Irvine, Dissertation. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

21. https://graphql.org

22. https://grpc.io

However, as mentioned earlier, this does not mean that every API has to be RESTful. For example, RESTful APIs can create a chatty interface between mobile components and services that provide the data. This is where an approach such as GraphQL can provide value. However, the team does not see a need to use GraphQL in TFX because its architecture includes the UI Service. The objective of the UI Service is to ensure that there are no chatty interactions by creating an API that is tailored to the UI needs.

The topic of integration in general is incredibly wide, and it is not a core focus of this book. It is important that when you think about your data architecture, you focus not only on how you store and manage your data but also on how you integrate it.

Data (Schema) Evolution

The way we represent and share data evolves along with our software product. How we manage this evolution of our data is commonly referred to as *schema evolution*. Schema evolution can be considered from two perspectives: within a component (intercomponent) and between components (intracomponent).

Within a component, the schema is how the database represents data to the application code. When you implement a database, you always make decisions around entities, fields, and data types, that is, you define a schema (formally or informally). SQL databases have strictly defined schemas, whereas NoSQL databases do not impose as much rigor up front. However, even when you implement a NoSQL database, you still make modeling decisions on how your entities are represented and their relationships—resulting in a schema even if the database has no manifestation of these decisions.

With time, you realize that you need to enhance your initial understanding of the data, introducing new entities, attributes, and relationships. Within a component, managing how the application code deals with the changes in the underlying data structures is what schema evolution is about. Backward compatibility is a commonly used tactic in this context and basically means that older application code can still read the evolving data schema. However, there is a tradeoff in increased complexity of application code.

Architecting for build, test, deploy, and operate (principle 5) helps in dealing with this challenge. Wherever possible, we should treat the database schema as code. We want to version and test it, just as we would do with any other software artifact. Extending this concept to datasets used in testing is also extremely beneficial. We discuss this subject in more detail when we cover artificial intelligence and machine learning in Chapter 8, "Software Architecture and Emerging Technologies."

Between components, schema evolution focuses on data defined in interfaces between two components. For most applications, this is an interface definition