##### A definition of software architecture

Graphical user interface, text, application

Description automatically generated

That’s obviously a quite abstract definition. But its essence is that an application’s architecture is its decomposition into parts (the elements) and the relationships (the relations) between those parts. Decomposition is important for a couple of reasons:

* It facilitates the division of labor and knowledge. It enables multiple people (or multiple teams) with possibly specialized knowledge to work productively together on an application.
* It defines how the software elements interact.

##### The 4+1 view model of software architecture

##### More concretely, an application’s architecture can be viewed from multiple perspectives, in the same way that a building’s architecture can be viewed from structural, plumbing, electrical, and other perspectives.

##### The 4+1 view model describes an application’s architecture using four views, along with scenarios that show how the elements within each view collaborate to handle requests.

##### Diagram Description automatically generated

* **Logical view—** The software elements that are created by developers. In object-oriented languages, these elements are classes and packages. The relations between them are the relationships between classes and packages, including inheritance, associations, and depends-on.
* ***Implementation view*—** The output of the build system. This view consists of modules, which represent packaged code, and components, which are executable or deployable units consisting of one or more modules. In Java, a module is a JAR file, and a component is typically a WAR file or an executable JAR file. The relations between them include dependency relationships between modules and composition relationships between components and modules.
* ***Process view*—** The components at runtime. Each element is a process, and the relations between processes represent interprocess communication.
* ***Deployment*—** How the processes are mapped to machines. The elements in this view consist of (physical or virtual) machines and the processes. The relations between machines represent networking. This view also describes the relationship between processes and machines.

In addition to these four views, there are the scenarios—the +1 in the 4+1 model—that animate views. Each scenario describes how the various architectural components within a particular view collaborate in order to handle a request. A scenario in the logical view, for example, shows how the classes collaborate. Similarly, a scenario in the process view shows how the processes collaborate.

##### Why architecture matters

An application has two categories of requirements. The first category includes the *functional* requirements, which define what the application must do. They’re usually in the form of use cases or user stories. Architecture has very little to do with the functional requirements. You can implement functional requirements with almost any architecture, even a big ball of mud.

Architecture is important because it enables an application to satisfy the second category of requirements: its *quality of service* requirements. These are also known as *quality attributes* and are the so-called *-ilities*. The quality of service requirements define the runtime qualities such as scalability and reliability. They also define development time qualities including maintainability, testability, and deployability. The architecture you choose for your application determines how well it meets these quality requirements.

#### **Overview of architectural styles**

In the physical world, a building’s architecture often follows a particular style, such as Victorian, American Craftsman, or Art Deco. Each style is a package of design decisions that constrains a building’s features and building materials.

Text

Description automatically generated

A particular architectural style provides a limited palette of elements (components) and relations (connectors) from which you can define a view of your application’s architecture. An application typically uses a combination of architectural styles.

##### The layered architectural style

The classic example of an architectural style is the layered architecture. A *layered architecture* organizes software elements into layers. Each layer has a well-defined set of responsibilities. A layered architecture also constraints the dependencies between the layers. A layer can only depend on either the layer immediately below it (if strict layering) or any of the layers below it.

The popular three-tier architecture is the layered architecture applied to the logical view. It organizes the application’s classes into the following tiers or layers:

* ***Presentation layer*—** Contains code that implements the user interface or external APIs
* ***Business logic layer*—** Contains the business logic
* ***Persistence layer*—** Implements the logic of interacting with the database

The layered architecture is a great example of an architectural style, but it does have some significant drawbacks:

* ***Single presentation layer*—** It doesn’t represent the fact that an application is likely to be invoked by more than just a single system.
* ***Single persistence layer*—** It doesn’t represent the fact that an application is likely to interact with more than just a single database.
* ***Defines the business logic layer as depending on the persistence layer*—** In theory, this dependency prevents you from testing the business logic without the database.

Also, the layered architecture misrepresents the dependencies in a well-designed application. The business logic typically defines an interface or a repository of interfaces that define data access methods. The persistence tier defines DAO classes that implement the repository interfaces. In other words, the dependencies are the reverse of what’s depicted by a layered architecture.

##### About the hexagonal architecture style

*Hexagonal architecture* is an alternative to the layered architectural style.

The hexagonal architecture style organizes the logical view in a way that places the business logic at the center. Instead of the presentation layer, the application has one or more *inbound adapters* that handle requests from the outside by invoking the business logic. Similarly, instead of a data persistence tier, the application has one or more *outbound adapters* that are invoked by the business logic and invoke external applications. A key characteristic and benefit of this architecture is that the business logic doesn’t depend on the adapters. Instead, they depend upon it.

##### An example of a hexagonal architecture, which consists of the business logic and one or more adapters that communicate with external systems. The business logic has one or more ports. Inbound adapters, which handled requests from external systems, invoke an inbound port. An outbound adapter implements an outbound port, and invokes an external system.

##### Diagram Description automatically generated

The business logic has one or more ports. A *port* defines a set of operations and is how the business logic interacts with what’s outside of it. In Java, for example, a port is often a Java interface. There are two kinds of ports: inbound and outbound ports. An inbound port is an API exposed by the business logic, which enables it to be invoked by external applications. An example of an inbound port is a service interface, which defines a service’s public methods. An outbound port is how the business logic invokes external systems. An example of an output port is a repository interface, which defines a collection of data access operations.

An outbound adapter implements an outbound port and handles requests from the business logic by invoking an external application or service. An example of an outbound adapter is a *data access object* (DAO) class that implements operations for accessing a database. Another example would be a proxy class that invokes a remote service. Outbound adapters can also publish events.

An important benefit of the hexagonal architectural style is that it decouples the business logic from the presentation and data access logic in the adapters. The business logic doesn’t depend on either the presentation logic or the data access logic. Because of this decoupling, it’s much easier to test the business logic in isolation. Another benefit is that it more accurately reflects the architecture of a modern application. The business logic can be invoked via multiple adapters, each of which implements a particular API or UI. The business logic can also invoke multiple adapters, each one of which invokes a different external system. Hexagonal architecture is a great way to describe the architecture of each service in a microservice architecture.

#### **The microservice architecture is an architectural style**

I’ve discussed the 4+1 view model and architectural styles, so I can now define monolithic and microservice architecture. They’re both architectural styles. Monolithic architecture is an architectural style that structures the implementation view as a single component: a single executable or WAR file. This definition says nothing about the other views. A monolithic application can, for example, have a logical view that’s organized along the lines of a hexagonal architecture.

**Pattern: Monolithic architecture**

Structure the application as a single executable/deployable component. See <http://microservices.io/patterns/monolithic.html>.

**Pattern: Microservice architecture**

Structure the application as a collection of loosely coupled, independently deployable services. See <http://microservices.io/patterns/microservices.html>.

##### What is a service?

A *service* is a standalone, independently deployable software component that implements some useful functionality.

##### A service has an API that encapsulates the implementation. The API defines operations, which are invoked by clients. There are two types of operations: commands update data, and queries retrieve data. When its data changes, a service publishes events that clients can subscribe to

##### Diagram Description automatically generated

A service has an API that provides its clients access to its functionality. There are two types of operations: commands and queries.

The API consists of commands, queries, and events. A command, such as createOrder(), performs actions and updates data. A query, such as findOrderById(), retrieves data. A service also publishes events, such as OrderCreated, which are consumed by its clients.

A service’s API encapsulates its internal implementation. Unlike in a monolith, a developer can’t write code that bypasses its API. As a result, the microservice architecture enforces the application’s modularity.

Each service in a microservice architecture has its own architecture and, potentially, technology stack. But a typical service has a hexagonal architecture. Its API is implemented by adapters that interact with the service’s business logic. The operations adapter invokes the business logic, and the events adapter publishes events emitted by the business logic.

**What is loose coupling?**

An important characteristic of the microservice architecture is that the services are loosely coupled.

All interaction with a service happens via its API, which encapsulates its implementation details. This enables the implementation of the service to change without impacting its clients. Loosely coupled services are key to improving an application’s development time attributes, including its maintainability and testability. They are much easier to understand, change, and test.

The requirement for services to be loosely coupled and to collaborate only via APIs prohibits services from communicating via a database. You must treat a service’s persistent data like the fields of a class and keep them private. Keeping the data private enables a developer to change their service’s database schema without having to spend time coordinating with developers working on other services. Not sharing database tables also improves runtime isolation. It ensures, for example, that one service can’t hold database locks that block another service. Later on, though, you’ll learn that one downside of not sharing databases is that maintaining data consistency and querying across services are more complex.

##### The role of shared libraries

Developers often package functionality in a library (module) so that it can be reused by multiple applications without duplicating code. After all, where would we be today without Maven or npm repositories? You might be tempted to also use shared libraries in microservice architecture. On the surface, it looks like a good way to reduce code duplication in your services. But you need to ensure that you don’t accidentally introduce coupling between your services.

Imagine, for example, that multiple services need to update the Order business object. One approach is to package that functionality as a library that’s used by multiple services. On one hand, using a library eliminates code duplication. On the other hand, consider what happens when the requirements change in a way that affects the Order business object. You would need to simultaneously rebuild and redeploy those services. A much better approach would be to implement functionality that’s likely to change, such as Order management, as a service.

You should strive to use libraries for functionality that’s unlikely to change. For example, in a typical application it makes no sense for every service to implement a generic Money class. Instead, you should create a library that’s used by the services.

##### The size of a service is mostly unimportant

One problem with the term microservice is that the first thing you hear is micro. This suggests that a service should be very small. This is also true of other size-based terms such as miniservice or nanoservice. In reality, size isn’t a useful metric.

A much better goal is to define a well-designed service to be a service capable of being developed by a small team with minimal lead time and with minimal collaboration with other teams. In theory, a team might only be responsible for a single service, so that service is by no means *micro*. Conversely, if a service requires a large team or takes a long time to test, it probably makes sense to split the team and the service. Or if you constantly need to change a service because of changes to other services or if it’s triggering changes in other services, that’s a sign that it’s not loosely coupled. You might even have built a distributed monolith.

The microservice architecture structures an application as a set of small, loosely coupled services. As a result, it improves the development time attributes—maintainability, testability, deployability, and so on—and enables an organization to develop better software faster. It also improves an application’s scalability, although that’s not the main goal. To develop a microservice architecture for your application, you need to identify the services and determine how they collaborate.