Architectural Patterns

We discussed about patterns for implementing business logic in the tactical design documents, now we are going to talk about design decisions in a broader context: the different ways to orchestrate the interactions and dependencies between a system’s components.

# Business Logic vs Architectural Patterns

**The variety of concerns** **that a codebase has to take care of makes it easy for its business logic to become diffused among the different components**: that is, for some of the logic to be implemented in the user interface or database, or be duplicated in different components.

**Lacking strict organization** in implementation concerns makes the codebase hard to change**:**

* **When the business logic has to change, it may not be evident what parts of the codebase have to be affected by the change**.
* The change may have unexpected effects on seemingly unrelated parts of the system.
* Conversely, it may be easy to miss code that has to be modified. All of these issues dramatically increase the cost of maintaining the codebase.

**Architectural patterns introduce organizational principles for the different aspects of**

**a codebase and present clear boundaries between them**:

**how the business logic is wired to the system’s input, output, and other infrastructural components.**

This affects how these components interact with each other: what knowledge they share

and how the components reference each other.

Choosing the appropriate way to organize the codebase, or the correct architectural

pattern, is crucial to support implementation of the business logic in the short term

and alleviate maintenance in the long term.

Let’s explore three predominant application architecture patterns and their use cases: layered architecture, ports & adapters, and CQRS.

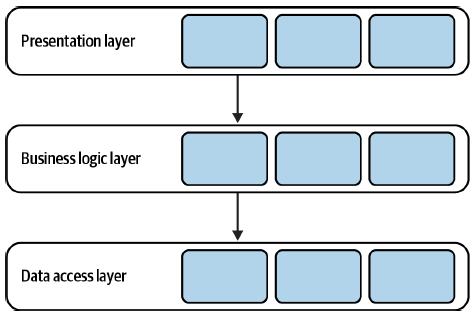
# Layered Architecture

Layered architecture is one of the most common architectural patterns. It organizes

the codebase into horizontal layers, with each layer addressing one of the following

technical concerns**: interaction with the consumers, implementing business logic, and**

**persisting the data.** You can see this represented in Figure 8-1.



In its classic form, the layered architecture consists of three layers: the presentation

layer (PL), the business logic layer (BLL), and the data access layer (DAL).

## Presentation Layer

The presentation layer, shown in Figure 8-2, implements the program’s user interface

**for interactions with its consumers**. In the pattern’s original form, this layer denotes a

graphical interface, such as a web interface or a desktop application.

In modern systems, however, the presentation layer has a broader scope: that is, **all**

**means for triggering the program’s behavior, both synchronous and asynchronous**.( and I add : the means to exposing some output??)

For example:

• Graphical user interface (GUI)

• Command-line interface (CLI)

• API for programmatic integration with other systems

• **Subscription to events in a message broker**

• **Message topics for publishing outgoing events**

**All of these are the means for the system to receive requests from the external environment and communicate the output. Strictly speaking, the presentation layer is the Program’s public interface.**

## Business Logic Layer

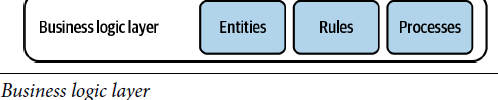
As the name suggests, this layer is responsible **for implementing and encapsulating**

**the program’s business logic.** This is the place where business decisions are implemented.

As Eric Evans says,1 this layer is the heart of software.

This layer is where the business logic patterns described in Chapters 5–7 are implemented—

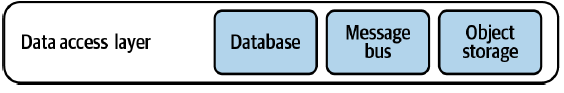
**for example,** **active records or a domain model** (see Figure 8-3).



## Data Access Layer

The data access layer **provides access to persistence mechanisms**. In the pattern’s original form, this referred to the system’s database. However, as in the case of the presentation layer, the layer’s responsibility is **broader for modern systems:**

* First, ever since the NoSQL revolution broke out, it is common for a system to work with **multiple databases**. For example, a document store can act as the operational database, a search index for dynamic queries, and an in-memory database for performance-optimized operations.
* Second, traditional databases are not the only medium for storing information. For example, **cloud-based object storage** can be used to store the system’s files, or a **message bus** can be used to orchestrate communication between the program’s different functions.( In this context, the message bus is used for the system’s internal needs. If it were exposed publicly, it would belong to the presentation layer.)
* Finally, this layer also includes integration with the various external information providers needed to implement the program’s functionality: APIs provided by external systems, or cloud vendors’ managed services, such as language translation, stock market data, and audio transcription.



## Communication between Layers

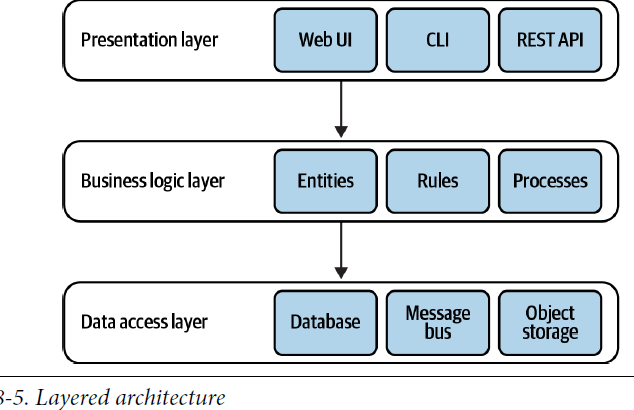
The layers are integrated in a **top-down communication model**: each layer can hold a

dependency only on the **layer directly beneath** it, as shown in Figure 8-5.

**This enforces decoupling of implementation concerns**

**and reduces the knowledge shared between the layers**.

In Figure 8-5, the presentation layer references only the businesslogic layer. It has no knowledge of the design decisions made in the data access layer.



## Variation

It’s common to see the layered architecture pattern extended with an additional layer:

**the service layer:**

*“Defines an application’s boundary with a layer of services that establishes* ***a set of available operations*** *and* ***coordinates the application’s response in each operation****.”*

*—Patterns of Enterprise Application Architecture*

The service layer acts as **an intermediary between the program’s presentation and**

**business logic layers.**

Consider the following code, without a service layer:

**namespace MvcApplication.Controllers**{

**public class UserController**: Controller{

[AcceptVerbs(HttpVerbs.Post)]

**public** ActionResult Create(ContactDetails contactDetails){

OperationResult result = **null**;

**try**{

\_db.StartTransaction();

**var** user = **new** User();

user.SetContactDetails(contactDetails)

user.Save();

\_db.Commit();

result = OperationResult.Success;

} **catch** (Exception ex) {

\_db.Rollback();

result = OperationResult.Exception(ex);

}

**return** View(result);

}}}

**The MVC controller in this example belongs to the presentation layer**. It exposes an

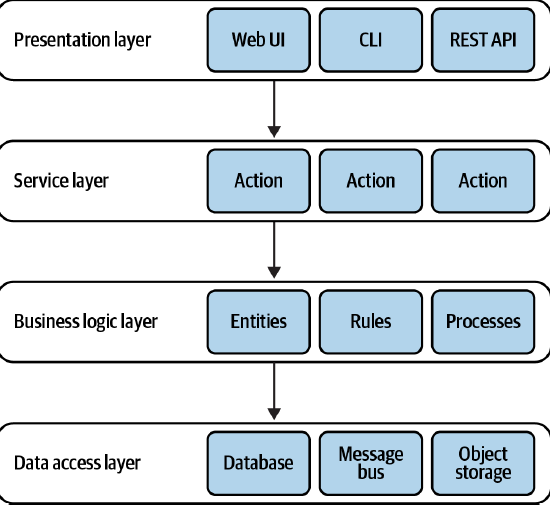
endpoint that creates a new user. The endpoint uses the User active record object to

create a new instance and save it. Moreover, it orchestrates a database transaction to

ensure that a proper response is generated in case of an error.

To further decouple the presentation layer from the underlying business logic, such

orchestration logic can be moved into a service layer, as shown in Figure 8-6.



It’s important to note that in the context of the architectural pattern, the service layer

is a logical boundary. It is not a physical service.

**The service layer acts as a facade for the business logic layer: it exposes an interface**

**that corresponds with the public interface’s methods, encapsulating the required**

**orchestration of the underlying layers**. (I add*: it acts as a façade/interface to the application’s business logic that exposes operations that the application can do*)

For example:

**interface** CampaignManagementService

{

**OperationResult** CreateCampaign(CampaignDetails details);

OperationResult Publish(CampaignId id, PublishingSchedule schedule);

OperationResult Deactivate(CampaignId id);

OperationResult AddDisplayLocation(CampaignId id, DisplayLocation newLocation);

}

**All of the preceding methods correspond to the system’s public interface**. However,

they lack presentation-related implementation details.

**The presentation layer’s responsibility becomes limited to providing the required input to the service layer and communicating its responses back to the caller.**

Let’s refactor the preceding example and extract the orchestration logic into a service layer:

**namespace ServiceLayer**{

**public class UserService**{

**public** OperationResult Create(ContactDetails contactDetails){

OperationResult result = **null**;

**try**{

\_db.StartTransaction();

**var** user = **new** User();

user.SetContactDetails(contactDetails)

user.Save();

\_db.Commit();

result = OperationResult.Success;

} **catch** (Exception ex) {

\_db.Rollback();

result = OperationResult.Exception(ex);

}

**return** result;

}}}

**namespace MvcApplication.Controllers**{

**public class UserController**: Controller{

[AcceptVerbs(HttpVerbs.Post)]

**public** ActionResult Create(ContactDetails contactDetails){

**var** result = \_userService.Create(contactDetails);

**return** View(result);

}}}

Page 124…