THE STANDARD

HASSAN HABIB

The Ultimate Guide to Building Enterprise-Level Systems from Idea to Product

The Standard

Hassan Habib

Free to read online at hassanhabib.com/books/the-standard

Thank you, Lord for all you taught me. Forgive me if I forget anything

Contents

0 **Introduction**

0.0 The Theory

0.1 Purposing, Modeling, and Simulation

0.2 Principles

0.3 Testimonials

1 **Brokers**

1.8 Standardized Provider Abstraction Libraries (SPAL)

2 **Services**

2.1 Foundation Services (Broker-Neighboring)

2.2 Processing Services (Higher-Order Business Logic)

2.3 Orchestration Services (Complex Higher Order Logic)

2.4 Aggregation Services (The Knot)

3 **Exposers**

3.1 Communication Protocols

3.1.1 RESTful APIs

3.2 User Interfaces

3.2.1 Web Applications

0

Introduction

This is The Standard. A collection of decades of experience in the engineering industry. I authored it to help you navigate the vast ocean of knowledge. The Standard is not perfect and never will be, and it reflects the ongoing evolution of the engineering industry. While one person may write it, it is the collection of thoughts from hundreds of engineers I've had the honor to interact with and learn from throughout my life.

The Standard holds hundreds of years of collective experiences from many different engineers. As I have traveled the world and worked in various industries, I've had the chance to work with many kinds of engineers - some of them were mad scientists who would fixate on minor details of every routine. Others have been business engineers who cared more about the results than the means to get to these results. In addition to others, I've learned from them what makes a simple engineering guide that can light the way for all other engineers to be inspired by it and follow it. Therefore, I have made this Standard, hoping it will be a compass for engineers to find the best way to engineer solutions that will hopefully change the world.

This Standard requests engineers worldwide to read through it and extract their experiences and knowledge to enrich an engineering Standard worthy of the software industry. Today, we know the origins of man and all the animals on earth. We know how hot boiling water is and how long a yard is. Our ships' masters know the precise measurements of latitude and longitude. Yet, we have neither a chart nor a compass to guide us through the vast sea of code. The time has come to accord our great craft with the same dignity and respect as the other standards defined by science.

The Standard has immense value for those still finding their way in this industry. Or even those who have lost their way. And the Standard can guide them towards a better future. More importantly, the Standard is written for everyone equally to

inspire every engineer or engineer-to-be to look forward to focusing on what matters the most about engineering- its purpose, not its technicalities. When engineers have any form of Standard, I have observed that they start focusing more on what can be accomplished in our world today. When engineers follow some form of Standard, their energy and focus become more on what to achieve rather than how. I collected and then authored this Standard, hoping it will eliminate confusion and enable engineers to focus on what matters most—use technology for higher purposes and establish its equivalent goals. The art and science of designing software have come a long way and have proven to be one of the most powerful tools a person could have today. It deserves a proper introduction, and how we educate youth about it matters.

Essentially, The Standard is my interpretation of SOLID principles and many other practices and patterns that continue to enrich our designs and development to achieve solid systems. The Standard aims to help every engineer find guidance in their day-to-day work. But more importantly, the Standard can ensure that every engineer has the guidance required when they need to build robust systems that can land on the moon, solve the most complex problems, and ensure the survival of humankind and its evolution.

The Standard is intentionally technology-agnostic. Its principles apply to any programming language, and its tri-nature foundation can guide any development or design decisions beyond software. The Standard shall not be tied to any particular technology, nor shall it be a limitation to those who want to follow it, regardless of their language of preference. I will be using C# on the .NET framework only to materialize and realize the concepts of this Standard. However, knowing that at the early stages of forming this Standard, I was heavily using Scala as a programming language.

But what's more important about The Standard? The Standard is the option to set a measure for expertise, influence, and knowledge depth before making any decisions. It is also meant to play the role of inspiration for generations of engineers to come to either follow it, improve on it or come up with their own. The alternative is to build software without standards, which is subject to chaos and injustice when investing the best time into the best efforts. Our industry today is in

chaos in terms of standardization. Unqualified individuals may have or take leadership positions, influencing those much more qualified to make unfortunate decisions.

The Standard is also my labor of love for the rest of the world. It is driven by and written with a passion for enhancing the engineering experience and producing efficient, rugged, configurable, pluggable, and reliable systems that can withstand any challenges or changes that occur almost daily in our industry.

[*] Introduction to The Standard

[*] Questions about The Standard

0.0

The Theory

0.0.0 Introduction

When designing any system, it is of utmost importance for designers to back up their design with a particular theory. Theories play a massive role in ensuring their design's purposes, models, and simulations are cohesive and extensible within a specific domain.

No matter how chaotic it may seem, any system is influenced by at least one theory created by the designer or inherited from previous designers or their methods. Regardless of what or who the influencer may be, the designer needs to understand the theory they follow fully. Otherwise, it will negatively impact their future decisions regarding extending their design to keep up with a forever-changing and expanding universe.

Early on, I realized that the simpler any theory is, the easier it becomes for other designers to adapt and extend its reach beyond the original designer's dreams. A universe built on simpler patterns can make it much easier for those who marvel at its beauty to understand and appreciate it more than those who give in to the fact that it's complex beyond their comprehension.

A theory about the universe could make life much more purposeful and enriched with tales about survival, evolution, and fulfillment.

0.0.1 Finding Answers

Early on in my life, I struggled with schooling. Nothing that was taught to me made any sense to me. Everyone at school seemed more concerned with memorizing and regurgitating what they've memorized during their exam than truly understanding what was taught, questioning its origins, and validating its purposes.

I realized at an earlier age that I needed some magical equation to help me distinguish between what's true and what's not, right and wrong, what is driven by a purpose, and what's an imitation for those with an actual purpose.

I was named all kinds of names during my schooling years. But I didn't mind much of that because my heart, mind, and body were fully invested in finding the answer to everything. So, I started my search to develop a theory that could explain the world to me.

When looking for answers, keeping your heart and mind open to all options is essential. Don't let any social or traditional structures limit your mind from seeking the truth about the universe and embracing the answers from everywhere.

After years of searching, I settled on a theory that made it simple for a simple person like myself to understand everything. I called it The Tri-Nature of Everything.

0.0.2 Tri-Nature

The Tri-Nature theory states that everything in this world comprises three main categories: dependencies, purposes, and exposures. Each component is crucial to its system's survival, evolution, and fulfillment.

Let's talk about these components here.

0.0.2.0 Purpose

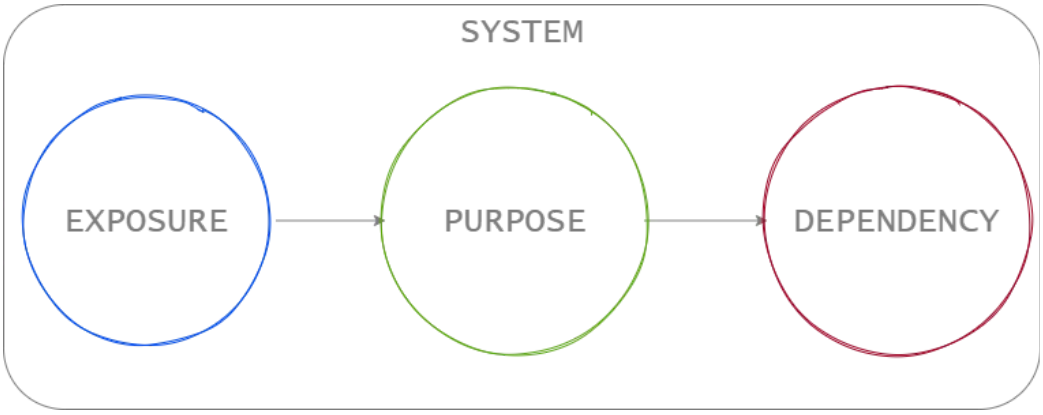
Everything around us has a purpose. It was created and designed with a specific reason in the mind of its creator. We design cars to take us from point A to point B. We design cups for drinking, plates for eating, and shoes for walking. Everything has a core purpose that governs its design and legitimizes its existence.

0.0.2.1 Dependency

However, every system must have a dependency in one form or another. For instance, we, as biological systems, rely on food and water to survive. Cars rely on oil or electricity. Computer systems rely on power and electricity, and so on. Regardless of its impact and importance, every method must have a dependency, whether small or big.

0.0.2.2 Exposure

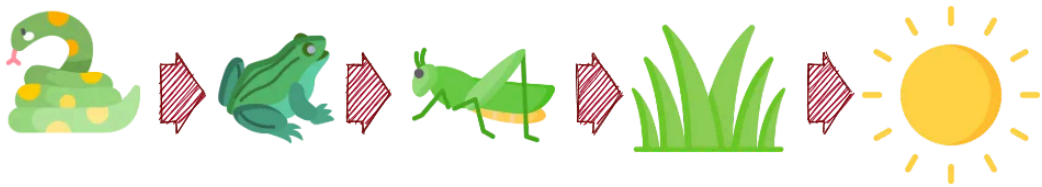
Every system must expose itself to allow other systems to integrate and consume its capabilities. However, it must reveal itself somehow to become a dependency for different systems to rely on it. For instance, power outlets are an exposure layer for power sources to allow other systems to plug in and consume their services. Gas stations are exposure layers for underground oil tanks to store that oil.

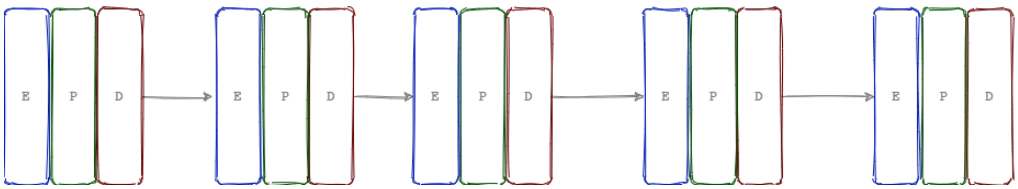


0.0.3 Everything is Connected

In the larger scheme of things, all systems are connected. A simple example of this is the food chain in nature. The sun is a dependency for the grass to grow; grasshoppers are grass consumers, while frogs feed on grasshoppers, snakes feed off of frogs, and so on.

Every food chain member is a system with dependencies, purposes, and exposure.



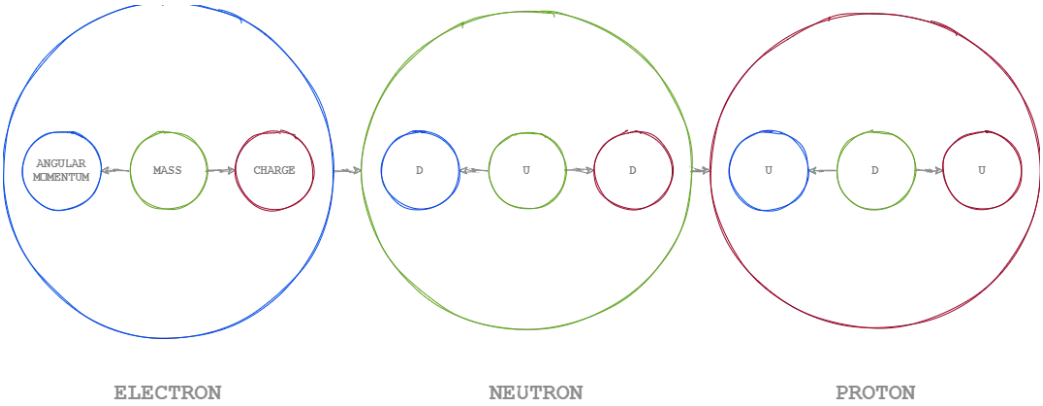


Since computer systems are nothing but a reflection of our reality, these systems integrations represent a chain of infinite dependencies where each one of these systems relies on one or more systems to fulfill its purpose. A simple mobile application could rely on a backend system to persist its data. However, the backend system relies on a cloud-based system to store the data. And the cloud-based system relies on a file system to perform basic persistence operations and so on.

0.0.4 Fractal Pattern

The Tri-Nature pattern of Things could also be perceived at the smallest and largest scales of any system. Every system is infinitely comprised of three components, each of which has three components, and so on. That's what we call a fractal pattern.

For instance, the smallest known component in the universe is the quarks within a neutron within an atom. These quarks are three components: two down quarks and one up quark. But if you zoom out slightly, you will see that the more extensive system where these quarks reside also comprises three components: electrons, protons, and neutrons.



If we zoom far out from the sub-atomic level to the solar system, the pattern continues to repeat at a massive scale. Our solar system is comprised of the sun, planets, and moons. They fall within the dependency purposing and exposure patterns as the components in the sub-atomic level as follows:

And if we zoom further out at scale, we find that galaxies are made of dust, gas, and dark matter.

The Tri-Nature pattern continues to repeat itself in every aspect of our lives. Every component in our universe, from the smallest sub-atomic parts to the scale of galaxies and solar systems, follows the same rule.

0.0.5 Systems Design & Architecture

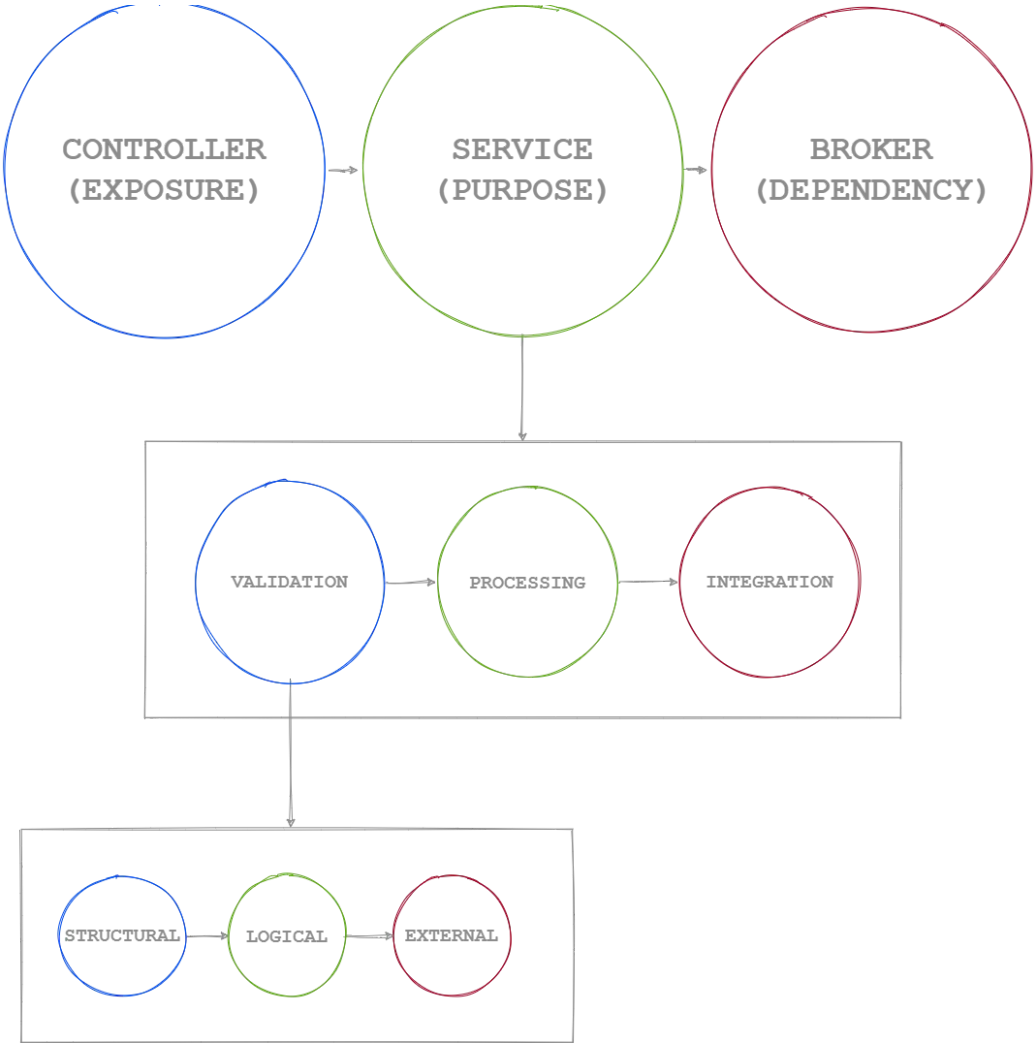
It is now evident that we can follow a theory to design systems! We can now develop every component in our software according to The Tri-Nature of Everything. The rules and guidelines that govern software design according to the theory are called the Standard. It refers to the universal standard in designing systems in every matter.

The Standard dictates at the low-level architecture that every system out there should be comprised of brokers (dependencies) and services (purposes), as well as exposers (exposures).

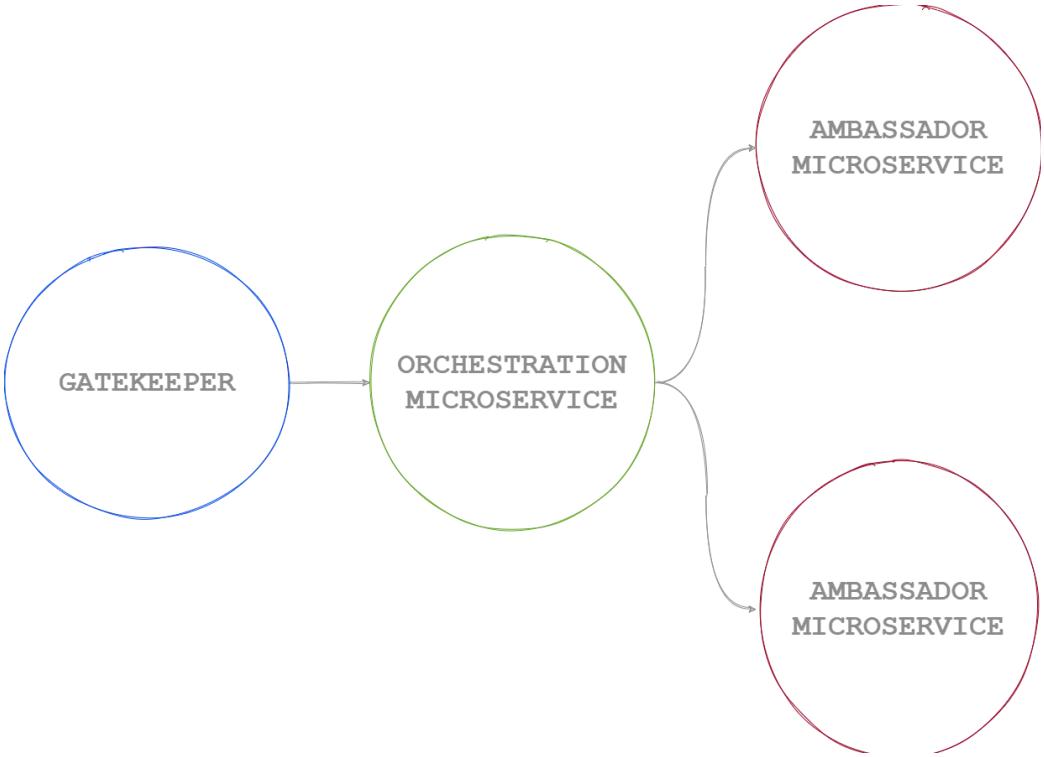
For instance, when designing a simple RESTful API, we may need to integrate with a database system, validate incoming data based on specific business rules, and expose these capabilities to the outside world so that the API consumers can integrate with it.

According to The Standard, that system would look like this:

The same pattern would repeat itself when digging deeper into these components. For instance, a service is comprised of validation components, processing components, and integration components. And then, if we zoom in a bit further, these same validation components are comprised of three more refined components: structural, logical, and external. The pattern continues to go on and on to the lowest level of our design, as shown here:



The same pattern also applies to larger systems if we zoom out of the one system realm into distributed modern systems such as microservice architectures - the same pattern should apply as follows:



In a distributed system, some services act as ambassadors to external or local resources, equivalent to a broker component at the service level. However, a purpose-driven component must come into play to orchestrate business flows by combining one or many primitive resource-consumption operations from these ambassador services. The final part is the exposure layer, a thin gatekeeper layer that becomes the first point of contact between the outside world and your microservice architecture.

The same pattern of Tri-Nature will continue to repeat itself across several systems, may it be large across multiple organizations or small within one single service.

Also check my recorded sessions:

- Introduction to Systems Design & Architecture
- Designing Real-Life Reverse Factoring System by The Standard

0.0.6 Conclusion

In conclusion, The Tri-Nature of Everything is the theory that powers up The Standard. The Tri-Nature theory heavily influences every single aspect of the rules and guidelines of The Standard. But it's important to understand that the theory goes beyond designing some software system. It can apply to management styles, writing books, making meals, establishing relationships, and every other aspect of our lives, which goes beyond the purpose of The Standard here.

After so many years of research and experimentation with the Tri-Nature theory, it is evident now that it works! It helps simplify some of the most complex systems out there. It plays well with our intuition as human beings. It makes it even simpler for automatons in the future to expedite our development processes of software and hardware and everything else in between.

Finally, The Standard is an ongoing journey of questioning The Tri-Nature theory. The further we go into uncharted waters in business domains, the more we discover some new territories where my theory still stands. Even for the most chaotic systems out there, the theory applies in specific ways, even if the components of said systems don't entirely adhere to The Standard form of distinction.

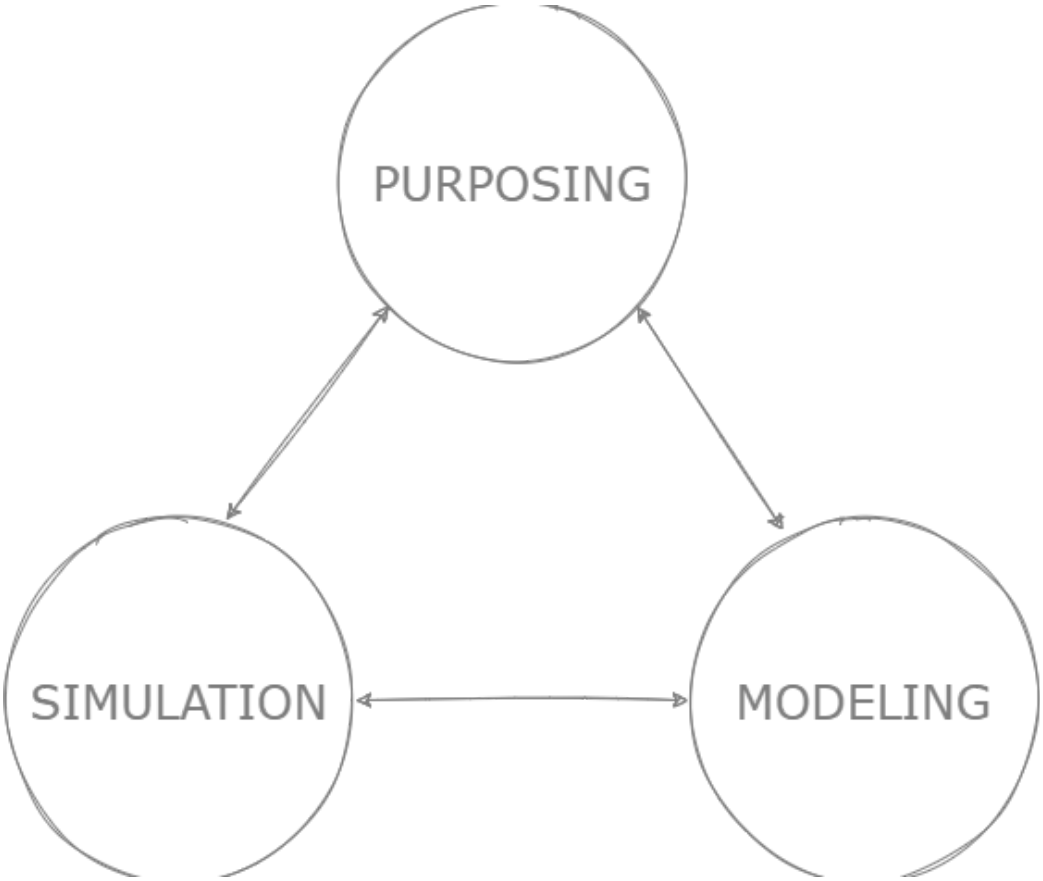
0.1

Purposing, Modeling, and Simulation

0.1.0 Introduction

The Standard defines the software engineering process in three main categories: Purposing, Modeling, and Simulation. Each aspect is crucial in guiding engineering efforts toward a successful solution and fulfilling a purpose.

The order in which these aspects are followed is also intentional. A purpose must exist to shape the modeling process, and one can't simulate interactions without models. But while that order at the initiation of the engineering process is crucial, it's important to understand that the process is selectively iterative. A change in the purpose may reflect a change in the simulation. Still, not necessarily the modeling, and a change in the models may not necessarily require changing the purpose or the simulation.



0.1.1 Purposing

The purposing process is our ability to find out why we need a solution. For instance, if we have an issue with knowing how many items are on the shelf in a grocery store, we deem the manual counting process inefficient, and a system needs to be implemented to ensure we have the proper count of items.

Reasoning relies heavily on our ability to observe problems and then articulate a problem to devise a solution that addresses the given problem. Purposing, therefore, is to find a reason to take action.

So, we have the observation, the articulation of the reasoning (the problem), and the intent for a solution. All of these aspects constitute the Purposing portion of engineering software.

0.1.1.0 Observation

We live in a world full of observables. Our inspiration is triggered by our ambition to achieve more. Our dreams reveal blockers in our way that we need to solve to continue our journey and fulfill our dreams. From the moment a young student uses a calculator to solve a complex equation to the moment that same student becomes an astronaut, calculating the trajectory of satellites orbiting our planet.

Observation is our ability to detect an issue that's blocking a goal from being achieved. Issues can be as simple as having the proper count of items on a grocery store shelf or as complex as understanding why we can't capture images of planets millions of light-years away. Engineers would describe these as observable problems.

The greater the purpose, the more complex a problem will be. But starting with more minor purposes is a way to train our minds to tackle bigger ones—step by step, one problem at a time.

0.1.1.1 Articulation

Describing the observable is an art in and of itself because describing a problem well is halfway to its solution. The clearer the articulation of the problem, the more likely it is to be understood by others, helping us to solve that very same problem.

Articulation is only sometimes with words. It's also with figures and shapes. It is not an accident that some of the most advanced ancient cultures have used figures and shapes to describe their times and history. Figures are a universal language, understood and interpreted by anyone who can relate to them much faster than learning a spoken language. A figure or shape might be the most optimum way to illustrate an idea, as its pictures are worth thousands of words.

Articulation requires a passion for solving the issue, whether written, spoken, or illustrated. A passionate mind conveys the hidden message of the criticality of the problem to be solved. Articulating a problem is a big part of selling a solution. Our ability to convey an idea to other engineers and those investing in and using this solution is one of the most critical aspects of engineering software.

0.1.1.2 Solutioning

A part of the purpose is the way to fulfill it. In the engineering industry, fulfilling the goals can be done by more than just any- means. Software fails worldwide because the solutions aspect was overlooked as a trivial part of the purpose. You may have heard of engineers up against a deadline who decide to cut corners to achieve a goal. In our Standard, this is a violation. A solution *must* not simply reach a goal but must be a purpose in and of itself to aid ambient architectural issues such as optimization, readability, configurability, and longevity. Solutioning is part of the purpose of software craftsmanship.

0.1.2 Modeling

Modeling is the second most crucial aspect of software engineering. We can extract models from the actors in any problem, whether these actors are living beings, objects, or others. We extract only the attributes relevant to the problem we are trying to solve and discard everything else. For instance, when trying to count the items on a grocery store shelf, we would need a model for these items.

A more straightforward example would be detecting perishable items in a grocery store. The only attribute we are concerned with here is the expiration date on the item. Everything else, including the label, color, weight, or any other details, is outside the scope of the modeling process and the solution.

Modeling, then, can only exist with a purpose. The purpose defines the scope or the framework of which the modeling should occur. Modeling without a purpose leaves the door open for attracting an infinite number of attributes every single element in the observable universe may have.

The relationship between the purposing and modeling attributes is proportional. The more complex the purpose is, the more likely the modeling process will require more attributes from the real world to model in the solution.

We express our models in programming languages as a `class`. The aforementioned perishable items problem above can be represented as follows:

```
public class Item
{
```

```
        public DateTimeOffset ExpirationDate {get; set;}
}
```

The name of the `class` represents the overall *type* of the item. Since all items have the same attribute of `ExpirationDate`, the name shall stay as generic as possible.

Now, imagine if our purpose grew more complex. Let's assume the new problem is identifying the more expensive perishable items so the store can put them up front for selling before the less costly items. In this case, our model would require a new attribute such as `Price` so a computer program or a solution can determine which is more valuable. This is what our new model would look like:

```
public class Item
{
        public double Price {get; set;}
        public DateTimeOffset ExpirationDate {get; set;}
}
```

0.1.2.0 Model Types

Models govern the entire process of simulating a problem (and its solution). Models break into three main categories: Data Carriers, Operational, and Configurations. Let's discuss those types in the following sections:

0.1.2.0.0 Data Carrier Models

Data carrier models have one primary purpose: to carry data points across systems. They can vary based on the type of data they carry. Some carry other models to represent a complex system, while others represent references to the original data points they represent.

Data carrier models in a relational fashion can be broken into three different categories. These categories make the priority development, design, and engineering areas much clearer. For instance, we can only start developing secondary/supporting models if our primary models are in place first. Let's talk about these categories in detail:

0.1.2.0.0.0 Primary Models

Primary models are the pillars of every system. Any given system can only proceed in design and engineering with a clear definition and materializing of these primary models. For instance, if we are building a schooling system, models like `Student`, `Teacher`, and `Course` are considered Primary models.

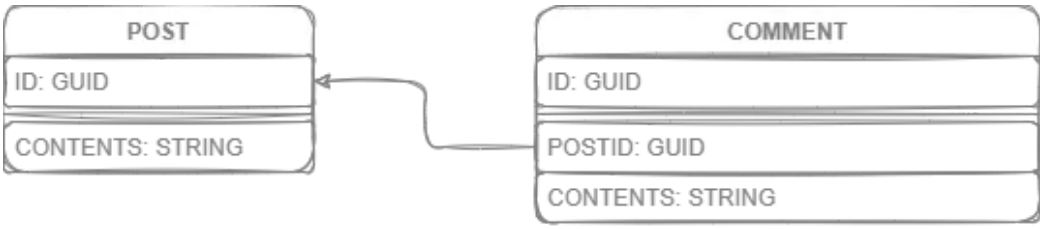
Primary relational storage schema models do not contain foreign keys or references to any other physical model. We call these models Primary because they are self-sufficient. They don't rely physically on some other model to exist. This means that a given primary model like `Student` may still exist in a schooling system, regardless of whether a `Teacher` record exists or not. This is called physical dependency.

Primary models, however, may rely conceptually or logically on other models. For instance, a `Student` model has a logical relationship to a `Teacher`, simply because there can never be a student without a teacher and vice versa. A `Student` model also has a conceptual relationship with its host and neighboring hosting services. For instance, there's a conceptual relationship between a `Student` model and a `Notification` model regarding business flow. Any student in any school conceptually relies on notifications to attend classes and complete assignments or other events.

0.1.2.0.0.1 Secondary Models

On the other hand, *Secondary* models have a hard dependency on Primary models. In a relational database model, secondary models usually have foreign keys referencing another model in the overall database schema. But even in non-relational storage systems, secondary models can be represented as nested entities within a given larger entity or have a loose reference to another entity.

Let's talk about some examples of secondary models. A `Comment` model in a social media platform cannot exist without a `Post` model. You cannot comment on something that doesn't exist. In a relational database, the comments model would look something like this:



In the example above, a secondary model `Comment` has a foreign key `PostId` referencing the primary key `Id` in a `Post` model. In a non-relational system, secondary models can easily be identified as nested objects within a given entity. Here's an example:

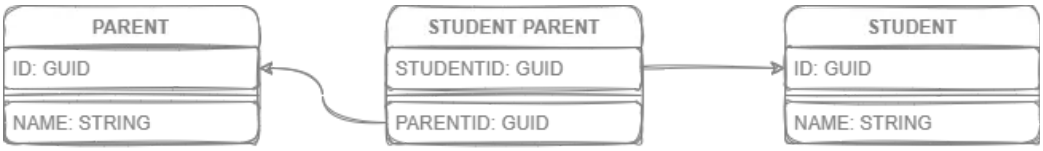
```
{
        "id": "some-id",
        "content": "some post",
        "comments": [
                {
                        "id": "comment-id",
                        "content": "some comment"
                }
        ]
}
```

Secondary models may generally have logical and conceptual relations to other models within their host, neighboring, or external systems. However, their chances of having these conceptual relations are much less than those of Primary models.

0.1.2.0.0.2 Relational Models

Relational models are connectors between two Primary models. Their main responsibility is to materialize a many-to-many relationship between two entities. For instance, a `Student` may have multiple teachers, and a `Teacher` may have multiple students. In this case, we need a relational model to act as an intermediary model.

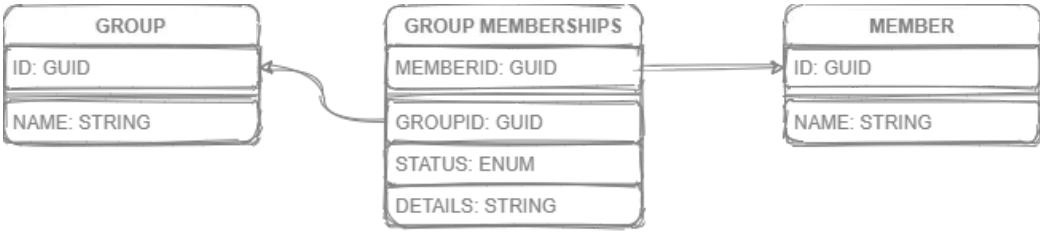
Relational models are not supposed to have any details. They only contain references to other models, which is their primary key. A composite key that aggregates two or more foreign keys within it. Let's take a look at an example:



0.1.2.0.0.3 Hybrid Models

There's a situation where a model connects multiple entities but also carries its data. I highly advise against following that path to maintain purity in your system design and control the complexity of your models. However, this approach is sometimes a necessary option to proceed with a specific implementation or business flow. In this case, we can propose a hybrid model that can carry particular details about the relationship between two independent entities.

A hybrid model can describe the detachment between two entities in a many-to-many relationship in a soft-delete scenario. Here's an example of a hybrid model that can occur in reality. Let's assume a group member does not want to be a part of a particular group anymore. We consider their group membership as `Deactivated` with a reason attached without actually deleting the record. Here's what it would that look like:



Hybrid models combine secondary models in the way they reference Primary models. They implement a relational nature in allowing multiple entities to relate to each other without exclusivity. In a non-relational data model, the referencing integrity may become looser, given the linear nature of that schema.

0.1.2.0.1 Operational Models

Operational models mainly target the simulation aspect of any software system. Think about all the primitive, complex, and exposure operations a simple scenario could require for a successful simulation to be implemented. Let's assume we are trying to solve a problem where we can simplify student registrations in some schools. The registration process will require some simulation to add these students' information into a computerized system.

Operational models will handle the entire process's exposure, processing, and integration by offering services that offer APIs/UIs to enter, post, add, and insert/persist students' information into some schooling systems.

The Standard focuses heavily on operational models because they represent the core of any system in terms of business flows. Operational models are also where most development and design resources go in any software development effort. Operational models can be broken into three main categories: Integration, Processing, and Exposure.

Let's talk about the operational models here.

0.1.2.0.1.0 Integration Models (Brokers)

Integration operational models' primary responsibility is to connect any existing system with external resources, which can be localized to the system's environment, like reading the current date or time, or remote, like calling an external API or persisting data in some database.

We call these integration models Brokers. They play a liaison role between processing operational models and external systems. Here's an example:

```
public partial class ApiBroker
{
        public async ValueTask<Student> PostStudentAsync(Student student) =>
                this.apiBroker.PostAsync<Student>(student, url);
}
```

The integration model above offers the capability to call an external API while abstracting the configuration details away from the processing operational models.

Like any other operational model type, they don't hold data but instead, use private class members and constants to share internal data across their public and private methods. The `ApiBroker` here as a model represents a simulation of integration with an external system.

In upcoming chapters, we will discuss Brokers extensively to clarify the rules and guidelines for developing brokers with external resources or systems.

0.1.2.0.1.1 Processing Models (Services)

Processing models are the holders of all business-specific simulations, such as student registrations, requesting a new library card, or retrieving student information based on specific criteria. Processing models can be either primitive/foundational, high-order/processing, or advanced/orchestrators.

Processing models, in general, either rely on integration models or self-relying like computational processing services or rely on each other.

Here's an example of a simple foundational/primitive service:

```
public partial class StudentService : IStudentService
{
        private readonly IStorageBroker storageBroker;
        ...

        public async ValueTask<Student> AddStudentAsync(Student student) =>
                await this.storageBroker.InsertStudentAsync(student);
}
```

A higher-order service would do/look as follows:

```
public partial class StudentProcessingService : IStudentProcessingService
{
        private readonly IStudentService studentService;
        ...

        public async ValueTask<Student> UpsertStudentAsync(Student student)
        {
                ....

                Student maybeStudent = await this.studentService
                        .RetrieveStudentByIdAsync(student.Id);

                return maybeStudent switch
                {
                        null => await this.studentService.AddStudentAsync(student),
```

```
                        _ => await this.studentService.ModifyStudentAsync(student)
                }
        }
}
```

More advanced orchestration-type services would combine multiple processing or foundational services as follows:

```
public partial class StudentOrchestrationService : IStudentOrchestrationService
{
        private readonly IStudentProcessingService studentProcessingService;
        private readonly IStudentLibraryCardProcessingService
studentLibraryCardProcessingService;
        ...

        public async ValueTask<Student> RegisterStudentAsync(Student student)
        {
                ....
                Student upsertedStudent = await this.studentProcessingService
                        .UpsertStudentAsync(student);

                ...

                await
this.studentLibraryCardProcessingService.AddStudentLibraryCardAsync(studentLibraryC
ard);
        }
}
```

In general, operational models are only concerned with the nature of simulation or processing of specific data carrier models; they are not concerned with holding data or retaining a status. In general, operational models are stateless in that they don't retain any details that went through them other than delegating logging for observability and monitoring purposes.

0.1.2.0.1.2 Exposure Models (Exposers)

Exposure models handle the HMI in all scenarios where humans and systems interact. They could be simple RESTful APIs and SDKs or just UIs like in web, mobile, or desktop applications, including command-line-based systems/terminals.

Exposure operational models are like the integration models; they allow the outside world to interact with your system. They sit on the other end of any system and are responsible for routing every request, communication, or call to the proper

operational models. Exposure models never communicate directly with integration models and don't have any configuration other than their dependencies injected through their constructors.

Exposure models may have their language in terms of operations; for instance, an integration model might use a language like `InsertStudent`, while an exposure model for an API endpoint would use a language like `PostStudent` to express the same operation in an exposure context.

Here's an example of exposure models:

```
public class StudentsController
{
        private readonly IStudentOrchestrationService studentOrchestrationService;

        [HttpPost]
        public async ValueTask<ActionResult<Student>> PostStudentAsync(Student
student)
        {
                Student registeredStudent = await this.studentOrchestrationService
                        .RegisterStudentAsync(student);

                return Ok(registeredStudent);
        }
}
```

The above model exposes an API endpoint for RESTful communication to allow students to be registered into a schooling system. We will further discuss the types of exposure models based on the context and the systems they are implemented within.

0.1.2.0.2 Configuration Models

The last type of model in any system is the configuration model. It can represent the entry point into a system, register dependencies for any system, or act as middleware to route URLs into their respective functions within an exposure model.

Configuration models usually appear at the beginning of a system's launch, handling incoming and outgoing communications or underlying system operations like memory caching, thread management, etc.

In a simple API application, you may see models that look like this:

```
public class Startup
{
        public void ConfigureServices(IServices services)
        {
                services.AddTransient<IStorageBroker, StorageBroker>();
                services.AddOAuth();
        }
}
```

As you can see from the code snippet above, the configuration model `Startup` offers capabilities to handle dependency injection-based registration of contracts to their concrete implementations. They may handle adding security or setting up a middleware pipeline. Configuration models are technology-specific. They may differ from a Play framework in Scala to a Spring or Flex in Python or Java. We will outline high-level rules according to The Standard for configuration models, but we will not dive deeper into the details of implementing any of them.

0.1.3 Simulation

The simulation aspect of software engineering is our ability to resemble the interactions to and from the models. For instance, in the grocery store example, a simulation would be the act of *selling* the item. Selling the item requires multiple modifications to the item in terms of deducting the count of the available items and reordering the items left based on the most valuable available item.

We can describe the simulation process as illustrating the relationships between models, which are programmed as `functions`, `methods`, or `routines`; these terms all mean the same thing. If we have a software service that is responsible for item sales, a simulation process will look like this:

```
public class SaleService
{
        public void Sell(Item item) => Items.Remove(item);
}
```

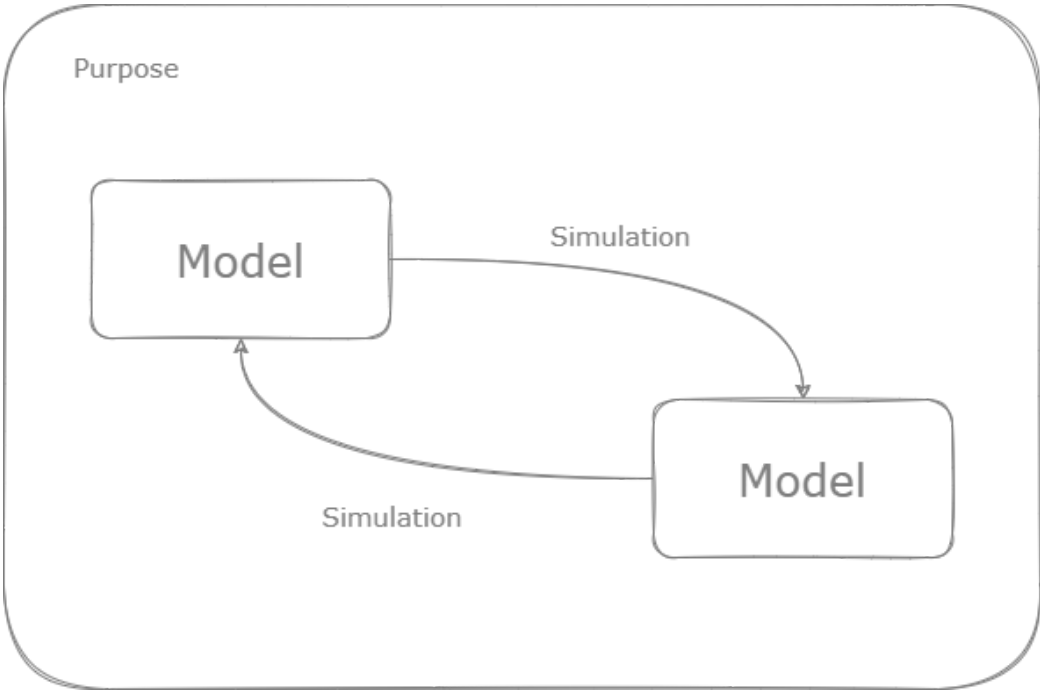
In the example above, we have a model called `SaleService` that offers functionality to simulate the sales process in the real world on a model of an item. And this is how you describe everything in object-oriented programming. Everything is an object (from a model), and these objects interact with each other (simulation).

Object interaction, in general, can be observed in three different types. A model is taking an action on another model. For instance, the `SaleService` is executing an action of `Sell` on an `Item` model. That's a model interacting with another model. In the very same example, a simulation could be something happening to the model from another model, such as the `Item` in the example above. The last type of simulation is a model interacting with itself, such as models that self-dispose once their purpose is achieved, as they are no longer needed, so they self-destruct.

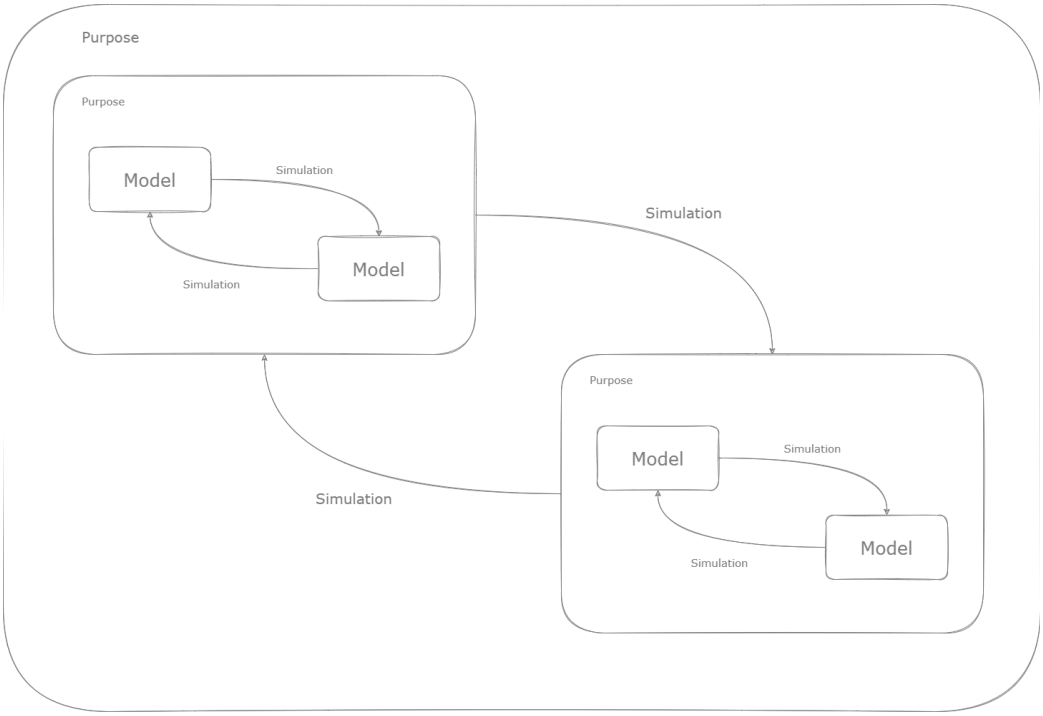
The simulation process is the third and last aspect of software engineering. We will explore it deeply when we discuss brokers, services, and exposers to illustrate how industrial software's modeling and simulation process happens.

0.1.4 Summary

If we consider *purposing* to be the domain or the framework in which models interact, then the following illustration should simplify and convey the picture a bit clearer:



It's important to understand that computer software can serve multiple purposes. Computer software can interact with other software that shares common purposes. The purpose of the software becomes the model, and the integrations become the simulations in that aspect. Here's a 10,000 feet example:



The complexity of any large system can be broken into smaller problems if the single-purpose or single-responsibility aspect is enforced for each subsystem. Modern software architectures call this granularity and modularization, which we will discuss briefly throughout the architecture aspect of The Standard.

[*] Purposing, Modeling & Simulation (Part 1)

0.2

Principles

In this chapter, we will explore the principles of The Standard. These principles apply to all components in a Standard-compliant system, whether these components are brokers, services, or exposers.

0.2.0 People-First

The main idea of this principle is to build and design Standard-compliant systems with people in mind, not just the people who will utilize the system but also the people who will be maintaining and evolving it.

A system must honor simplicity over complexity to follow the people-first principle. Simplicity leads to rewritability. It also leads to designing monolithic systems built with a modular mindset to allow a true fractality in the overall pattern of the system.

The Standard also enforces the principles of measuring advanced engineering concepts against the understanding of mainstream engineers. New engineers in the industry are the leaders of tomorrow. If they are not buying in on any system, they'll eventually give up and rewrite it repeatedly.

0.2.0.0 Simplicity

Code written according to The Standard has to be simple. There are measures to ensure this simplicity takes place; these measures are as follows:

0.2.0.0.0 Excessive Inheritance

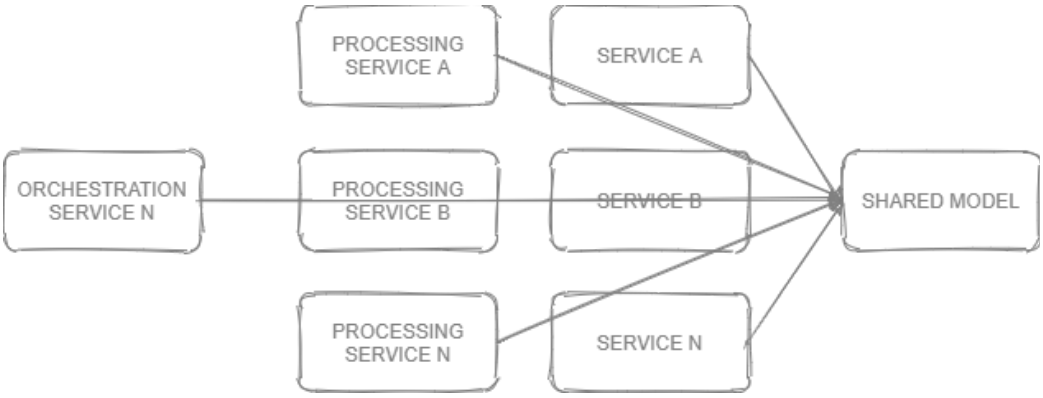
Any software written according to The Standard shall not have more than one level of inheritance. Over one level of inheritance will be considered excessive and prohibited except in versioning cases for the vertical scaling of flows. Excessive inheritance has proven to be a source of confusion and difficulty in terms of readability and maintainability over the years.

0.2.0.0.1 Entanglement

0.2.0.0.1.0 Horizontal Entanglement

Building “common” components in every system that promises to simplify development processes is another prohibited practice in Standard-compliant systems. This practice manifests itself in components with names like `Utils`, `Commons`, or `Helpers`. These terminologies and what they imply in terms of false promised simplifications are not allowed. Any system built according to The Standard should comprise Brokers, Services, or Exposers, nothing more or less.

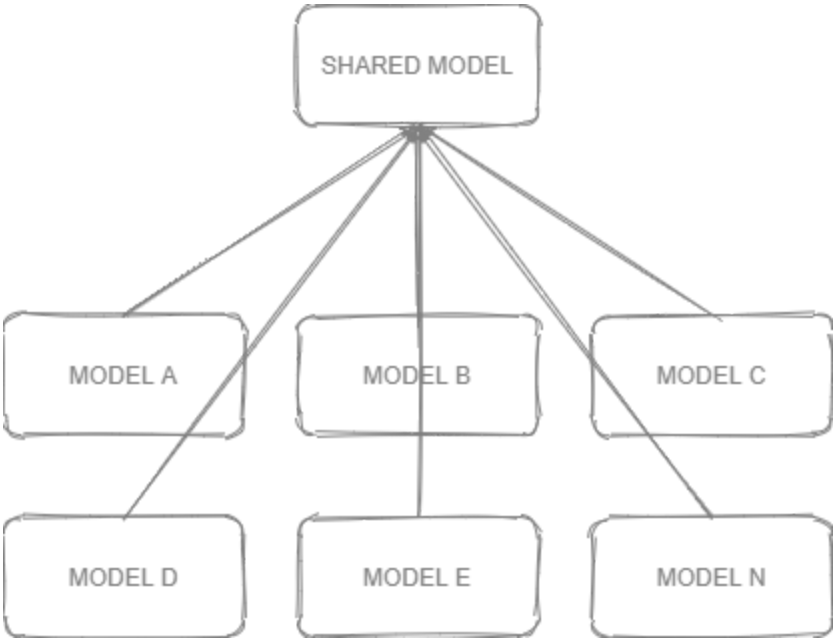
Another example of horizontal entanglements is shared models across multiple independent flows - sharing exceptions, validation rules, or any other form of entanglement across multiple flows.



0.2.0.0.1.1 Vertical Entanglement

This principle also applies to scenarios where base components are used. Unless these base components are native or external, they will not be allowed in a Standard-compliant system. Local base components create a vertical level of entanglement that harms the maintainability and readability of code. Vertical entanglements are just as harmful as `Commons` components, creating single points of failure across any system.

Entanglements (vertical or horizontal) also prevent engineers in any system (especially newcomers) from fully understanding the system's depth and fully owning its functionality. They also deter engineers from having the opportunity to build end-to-end flows when half of the functionality is componentized for the sake of development expedition and simplicity.



0.2.0.0.2 Autonomous Components

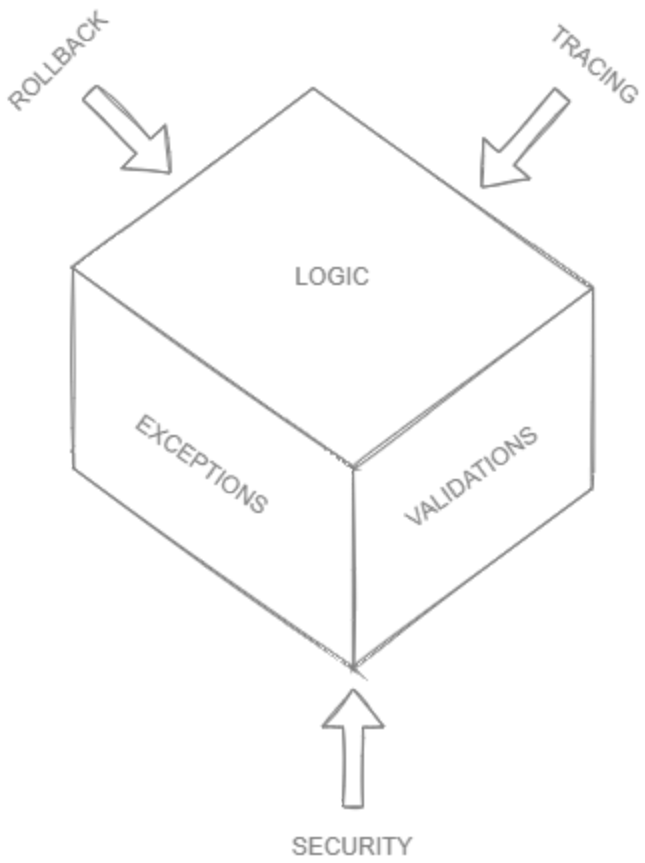
Every component in every system should be self-sufficient. Every component implements its validations, tooling, and utilities in one of its dimensions with no hard dependency on any other external components except through dependency injection. This principle favors duplication over presumed simplification via code entanglement.

Autonomous components will open up the opportunity for every engineer on every team to fully own every dependency and tool their component may need to fulfill its purpose. This may cause some code duplication to open an equal opportunity for every developer to learn how to build and evolve a component fully.

0.2.0.0.2.0 No Magic

Autonomous components put all their routines up front, in plain sight of the engineer. No hidden routines, shared libraries, or magical extensions that require chasing references once an inevitable split of the giant monolith begins to occur.

We will treat Objects like they are in nature: multi-dimensional components self-containerized like atoms in nature. These components perform their validations, exception handling, tracing, security, localization, and everything else.



Components built according to The Standard strictly adhere to the idea of *What You See Is What You Get* (WYSIWYG) - everything concerning the components will be on the component itself.

0.2.1 Rewritability

Every system should be developed with the rewritability principle in mind. This principle dictates that our assumptions in the systems we develop have the potential to be reexamined and probably reconsidered. Every system should be easily rewritable as a measure of adherence to forever growing and changing business requirements.

Rewritable code is easy to understand, modify, and fully rewrite. It is extremely modular and autonomous, encouraging engineers to evolve it with the least effort and risk possible.

Rewritable code doesn't play tricks on the reader. It should be plug-and-play—fork, clone, build, and run all its tests successfully with no issues. There are no hidden dependencies, injected routines at runtime, or unknown prerequisites.

0.2.2 Mono-Micro

Build monolithic systems with a modular mindset, with every flow fully independent from other flows. For instance, we may build a monolithic system with a microservice mindset, meaning that any flow can be extracted from the system and turned into its microservice or lambda with the least effort possible.

This principle goes hand in hand with the concept of autonomous components at a higher level, where flows are also autonomous from their neighboring flow and their hosting system.

0.2.3 Level 0

Code must be understandable by an entry-level individual in the engineering craft. Since the majority of engineers in our industry will always be new to the craft, our code base continues to live based on its ease of understanding by most engineers in the industry.

Level 0 engineers are our measure of success. Their ability to understand our code guarantees that this code will continue to live and evolve with the next generation of engineers.

This principle also mandates that every engineer in the industry closely examine their code and pair with juniors in the field to see if they meet this principle.

0.2.4 Open Code

As a principle, open code dictates that everything written according to The Standard should be commonly available to the public. This applies to internal architectural practices, trial libraries, and any other form of module development that doesn't allow every engineer to learn and evolve any given system. Developing internal tools that are not accessible shall inevitably harm the engineering experience for those who are trying to evolve these very tools.

The principle also recognizes that under extreme circumstances, such as when safety or security are threatened or when one is under some contractual obligation, one cannot make code, tooling, platforms, and patterns available to the public. However, The Standard does not permit making the source proprietary solely for personal or organizational gain.

0.2.5 Airplane Mode (Cloud-Foreign)

The Standard enforces the idea of airplane mode. Where engineers can set up their entire infrastructure on their local machine without needing or having any network connection, this principle goes heavily against principles such as Cloud-Native applications, where a given system cannot stand and run without cloud infrastructure.

The Standard also encourages its adopters to develop the proper tooling to bridge the gap between cloud infrastructural components and local components such as queues, event hubs, and other tools to make it easily testable and modifiable.

0.2.6 No Toasters

The Standard shall be taught man to man, not machine to man. No stylecops or analyzers should be implemented as the primary mechanism to force compliance. It must be driven by passion in the heart and conviction in the mind, not coercion.

The Standard is best carried through person-to-person teaching, fostering an engineering culture of open discussion, shared understanding, and deeply held conviction.

AI-assisted coding (including “vibe coding”) is welcome, provided a human remains actively engaged in the process and retains full responsibility for the final code produced. AI may serve as a tool for acceleration, exploration, and even code review, but it must never replace human judgment.

The final authority for approval, acceptance, and merge decisions must always rest with a human.

0.2.7 Pass Forward

The Standard shall be taught at no cost as it arrived to you at no cost. It should also be passed forward to the next engineer at no cost, regardless of their financial, social, or educational status. The Standard is pure knowledge given by the selfless to the selfless. There shall be no profiteering off of it; neither shall it be a reason for someone to belittle others or make them feel less. Teaching The Standard for profit violates it and denies the beneficiary (the violator) any further guidance from The Author.

0.2.8 All-In/All-Out

The Standard must be fully embraced or entirely rejected. Any system incorporating only some aspects of The Standard will not be recognized as a Standardized system. Any system that continues to adhere to previous versions of The Standard will be obligated to elevate its standards to reclaim its status of standardization.

0.2.9 Readability over Optimization

Readability is more important than optimization. If an optimum software isn’t readable then it’s not truly optimum, and it’s not truly standardized. When in doubt, The Standard honors readability over optimization.

0.2.10 Last Day

Every day can be the last day on any given project. Therefore, every effort—whether it be design, development, documentation, or test automation—must be brought to a good stopping point by the end of each engineering day. Engineers adhering to The Standard must ensure their work is in a state that can be seamlessly picked up by another engineer on the next engineering day. This practice ensures project continuity and, more importantly, accommodates any potential unforeseen circumstances.

0.3

Testimonials

This chapter is about testimonials from engineers all over the world. They have all used The Standard in their daily projects and benefited from it. Here's their testimonials:

"The Standard helped me think out of the box while designing any system."

Awais Shabir
Lead Software Engineer
Cloud Solution, Saudi Arabia

"The Standard provides a focus on a common architecture and approach for teams that I lead."

Dennis Landi
Technical Architect
Allied Data, Inc., USA

"The Standard gives the full potential to realize the impact of true software engineer can give to themself and also to the society in terms of knowledge, delivering quality and maintainable software."

Shri Humrudha Jagathisun
Senior Software Engineer
Microsoft, USA

“The Standard provides Better separation of concerns. Unification of project templates between different teams.”

Raúl Lorenzo Boulllosa
Software Engineer
Altia Consultores, Spain

“The Standard help me in writing more and more clean code, writing better Test-Driven Solutions and my team starts working in more coherence”

Mabrouk Mahdhi
Sr. Software Engineer
Messer SE & Co. KGaA, Germany

“The Standard helped with consistent implementation of software that could be understood by all levels of experience. Proper adoption of TDD. Simplification. Set a standard that is known and used in code reviews to ensure code quality.”

Christo du Toit
Senior Digital Developer
National Health Services, England

“It helps in easy design, consistent and easy to read code bases, easy to organize tasks, and high code quality.”

Kailu Hu
Software Engineer
Microsoft, USA

“The Standard helped with Increased speed in developing excellent software.”

Yusuf Corr
Software Engineer
Microsoft, USA

"The Standard helped the team in implementing the code quality standards reducing many problems and the risk of project failures. It also helps in the development of software programs that are less complex and thereby reduce errors."

Marthin Thomas
Software Developer
Alpha Events and Marketing, Namibia

"The standard learnings helped me get my first full time position as a web developer"

Florian Renard
Web Developer
HR Team, France

"Better internal and cross team collaboration. Less meetings and churn with conversations like "tabs vs spaces". Common set of concepts and buzzwords to facilitate communication and collaboration. PRs, continuous integration/feedback and engineering quality become automatic. Easily one of the best ways to remove the roadblocks that inhibit new teams."

Josh McCall
Software Engineer
Microsoft, USA

"The breakdown of existing N-Tier architectures in to more granular pieces make for a higher quality of tested on inception code."

Paul Ward
Head of Technology
Corporate LinX, England

"The Standard is a wonderful book. It helps me to re-think the software design, and project refactoring."

Sam Xu
Senior Software Engineer
Microsoft, China

"The Standard takes out the ambiguity that comes with writing code. It takes into consideration many edge cases and threats which go unseen so many times in SDL. It is developer friendly and gives the flexibility of anyone working on any service/product if it uses The Standard. My team can take vacation and the work will still go on as if that same engineer is writing the code."

Geetika Jain
Software Engineer
Microsoft, India

"The Standard, according to me should be every engineer's toolkit. It unravels some common assumptions, unknowns and silent doubts that would come to every engineer's mind but end up staying inside unanswered. This living document should be adopted broadly worldwide to an extent that it becomes the common language for software development making it easier to people to collaborate and bring qualitatively engineered products to life."

Bhavana Konchada
Senior Software Engineer
Microsoft, India

"Introducing Standard has organized my code base. Now it is more readable for me and everyone who is new. I have also learned new things that i was not aware of."

Vishwanadha Goli
Software Engineer
Microsoft, India

"Provided a framework for engineers to grow within. Inspired engineers to think more on design and architecture within the source code, not just the application resource topology. Taught engineers to think of the developer experience (onboarding, readability, maintainability) beyond their immediate project."

Sean Hobbs
Software Engineer II
Microsoft, USA

"The Standard helped me transition from intimidated to empowered, as I sought out to create a system strong enough to support my goals. Envisioning the product and the UI was the easy part - knowing how to tie everything together in a way that would allow the product to continue evolving seemed impossible. Hassan's blueprint has given me clarity, motivated me to use best practices, and enabled me to create software that will scale appropriately."

Danielle Scott
Software Engineer
Microsoft, USA

"Relaunching my career as a software engineer having left it for almost 15 years, The Standard helps me to quickly regain confidence and skills in coding and software design. The Standard removes ambiguity and applies SOLID principles elegantly. It also speaks to me on many levels, personally and professionally, through its observation of the world and everything in it."

Christopher Tang
Software Developer
Axios IT Pty Ltd, Australia

"The-Standard gave me hope, enriched my knowledge and enabled me build highly-profitable apps. If you think this is the next tech-book on how to write "good" software, you are utterly mistaken. Personally, The-Standard taught me the craft of software engineering, like topics ranging from effective team management to proper commit history."

Elbek Normurodov
Staff Software Engineer
Piorsoft, LLC, Uzbekistan

"It allowed me to introduce structure within my hobby projects. It lends itself to a test driven approach which has helped plug gaps in projects at work. Though the terminology used has taken time to getting used to, it allows an easy way to communicate new ideas between each other."

Callum Marshall
Full Stack Developr
Corporate LinX, UK

"The Standard provided me with a unified understanding of software development principles and best practices in an easy and approachable way. Hassan has managed to distil years of practical wisdom into a simple-to-understand manual

that is easy to grasp and follow. I am finally clear about where different pieces of code need to go and why. I have began applying it to one proprietary project for a client of the company I work at. I cannot over-emphasize how much I appreciate the concept of brokers-services-exposers — an unforgettable and powerful way to think about software. Thank you Hassan!"

Junaid Bedford
Software Developer
iOCO Digital, South Africa

" I am always looking for best practices in my job. Standard will unify my professional dictionary not only in communication but also to prioritize everyday tasks, optimize cost of projects and knowledge flow in whole organization."

Marek Słowikowski
IT Architect
MSUI, Poland

"The Standard completely changed my perspective on thinking about how enterprise level robust systems are developed. - Being part of a really good community created around Standard helps to discuss and use the best/state-of-the-art engineering practices in the open-source project (and possibly my team projects) which follows Standard guidelines. - The Standard also helped me to get much deeper understanding on the design pattern used while developing a software. It also helped me to effectively participate more in my team design discussion sessions."

Shivang Soni
Software Developer
Microsoft, India

“With a cumulative of over 42 years of software development, The Standard principles people first attitude has shed a bright light on the software engineering industry. The extremly easy approach to these concepts help not just programming newcomers, but as well can greatly beneifit us old timers of the software world. When I first put a MS-Dos 5.25in. floppy in a PC with no hard drives, I never in a million years thought anything can be easier to follow.” Hassan Habib and the Standard Community are the epitome of People First always someone somewhere to help or listen

Gregory Hays
Business Application Development
Southern California, USA

“The Standard provides a solid framework that bridges the knowledge gap between engineers at different experience levels. From my experience, writing software the right way—prioritizing scalability and maintainability—is one of the biggest perks of following The Standard.”

Abdulsamad Osunlana
Software Engineer II
Aspen Publishing, USA

“The Standard taught me how to write clean, organized code and design clear architectures. But it’s more than just software—it’s about people. It pushes you to write code so intuitive that even a beginner can understand it, blending technical excellence with human-centric thinking.”

Zafar Urakov
Software Engineer II
Aspen Publishing, USA

1

Brokers

1.0 Introduction

Brokers are a liaison between business logic and the outside world. They are wrappers around external libraries, resources, services, or APIs to satisfy a local interface for the business to interact with them without having to be tightly coupled with any particular resources or external library implementation.

Brokers, in general, are meant to be disposable and replaceable - they are built with the understanding that technology evolves and changes all the time. Therefore, at some point in the lifecycle of a given application, it will be replaced with a recent technology that gets the job done faster.

However, brokers also ensure that your business is pluggable by abstracting away any specific external resource dependencies from what your software is trying to accomplish.

For instance, you have an API built to consume and serve data from a SQL server. At some point, you decided that a better, more economical option for your API is to rely on a NoSQL technology instead. Having a broker to remove the dependency on SQL will make it much easier to integrate with NoSql with the least time and cost humanly possible.

1.1 On The Map

In any application, mobile, desktop, web, or simply an API, brokers usually reside at the “tail” of any app because they are the last point of contact between our custom code and the outside world.

Whether the outside world is simply local storage in memory or an independent system that resides behind an API, they all have to reside behind Brokers in any application.

In the following low-level architecture for a given API - Brokers reside between our business logic and the external resource:

1.2 Characteristics

There are a few simple rules that govern the implementation of any broker - these rules are:

1.2.0 Implements a Local Interface

Brokers must satisfy a local contract and implement a local interface to allow decoupling between their implementation and the services that consume them.

For instance, given that we have a local contract, `IStorageBroker` that requires an implementation for any given CRUD operation for a local model `Student` - the contract operation would be as follows:

```
public partial interface IStorageBroker
{
    ValueTask<IQueryable<Student>> SelectAllStudentsAsync();
}
```

An implementation for a storage broker would be as follows:

```
public partial class StorageBroker
{
    private DbSet<Student> Students { get; set; }

    public async ValueTask<IQueryable<Student>> SelectAllStudentsAsync() =>
        await SelectAllAsync<Student>();
}
```

A local contract implementation can be replaced at any point, from utilizing the Entity Framework as shown in the previous example to using a completely different technology like Dapper or an entirely different infrastructure like an Oracle or PostgreSQL database.

1.2.1 No Flow Control

Brokers should not have any form of flow control, such as if statements, while loops, or switch cases. That's because flow-control code is considered business logic, and it fits better in the services layer, where business logic should reside, not the brokers.

For instance, a broker method that retrieves a list of students from a database would look something like this:

```
public async ValueTask<IQueryable<Student>> SelectAllStudentsAsync() =>
    await SelectAllAsync<Student>();
```

A simple function that calls the native EntityFramework `DbSet<T>` and returns a local model like `Student`.

1.2.2 No Exception Handling

Exception handling is a form of flow control. Brokers are not supposed to handle exceptions but rather let them propagate to the broker-neighboring services, where they can be properly mapped and localized.

1.2.3 Own Their Configurations

Brokers are also required to handle their configurations - they may have a dependency injection from a configuration object to retrieve and set up the configurations for whichever external resource they are integrating.

For instance, connection strings in database communications are required to be retrieved and passed into the database client to establish a successful connection, as follows:

```
public partial class StorageBroker : EFxceptionsContext, IStorageBroker
{
    private readonly IConfiguration configuration;
```

```
    public StorageBroker(IConfiguration configuration)
    {
        this.configuration = configuration;
        this.Database.Migrate();
    }

    ...
    ...

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
        string connectionString =
this.configuration.GetConnectionString("DefaultConnection");
        optionsBuilder.UseSqlServer(connectionString);
    }
}
```

1.2.4 Natives from Primitives

Brokers may construct an external model object based on primitive types passed by broker-neighboring services.

For instance, in an e-mail notification broker, input parameters for a `.Send(...)` function require the basic input parameters such as the subject, content, or address. Here's an example:

```
    public async ValueTask SendMailAsync(List<string> recipients, string subject,
string content)
    {
        Message message = BuildMessage(recipients, ccRecipients, subject, content);
        await SendEmailMessageAsync(message);
    }
```

The primitive input parameters will ensure no strong dependencies between the broker-neighboring services and the external models. Even in situations where the broker is simply a point of integration between your application and an external RESTful API, it's very highly recommended that you build your native models to reflect the same JSON object sent or returned from the API instead of relying on NuGet libraries, DLLs or shared projects to achieve the same goal.

1.2.5 Naming Conventions

The contracts for brokers shall remain as generic as possible to indicate their overall functionality; for instance, we say `IStorageBroker` instead of `ISqlStorageBroker` to indicate a particular technology or infrastructure.

With a single storage broker, it might be more convenient to maintain the same name as the contract. However, in the case of concrete implementations of brokers, it all depends on how many brokers you have that provide similar functionality. In our case, we have a concrete implementation of `IStorageBroker`, so the name would be `StorageBroker`.

However, if your application supports multiple queues, storage, or e-mail service providers, you might need to start specifying the overall target of the component; for instance, an `IQueueBroker` would have multiple implementations such as `NotificationQueueBroker` and `OrdersQueueBroker`.

However, if the concrete implementations target the same model and business logic, a diversion to the technology might be more beneficial. In this case, for instance, `IStorageBroker`, two different concrete implementations could be `SqlStorageBroker` and `MongoStorageBroker`. This case is typical in situations where the intention is to reduce production infrastructure costs.

1.2.6 Language

Brokers speak the language of the technologies they support. For instance, in a storage broker, we say `SelectById` to match the SQL `Select` statement; in a queue broker, we say `Enqueue` to match the language.

If a broker supports an API endpoint, it shall follow the RESTFul semantics, such as `POST`, `GET`, or `PUT`. Here's an example:

```
public async ValueTask<Student> PostStudentAsync(Student student) =>
    await this.PostAsync(RelativeUrl, student);
```

1.2.7 Up & Sideways

Brokers cannot call other brokers because they are the first point of abstraction and require no additional abstractions or dependencies other than a configuration access model.

Brokers also can't have services as dependencies, as the flow in any given system shall come from the services to the brokers and not the other way around.

For instance, even when a microservice has to subscribe to a queue, brokers will pass forward a listener method to process incoming events but not call the services that provide the processing logic.

The general rule here is that only services can call brokers, while brokers can only call external native dependencies.

1.2.8 One Resource, One Broker

Each Broker must represent exactly one resource. For example, if a system integrates with a SQL database, there should be a single Broker responsible for holding the configuration and managing all interactions with that database. The reverse is also true: A Broker must never handle more than one resource and a single resource must only be managed by one Broker.

1.3 Organization

Brokers supporting multiple entities, such as Storage brokers, should leverage partial classes to break down the responsibilities per entity.

For instance, if we have a storage broker that provides all CRUD operations for both `Student` and `Teacher` models, then the organization of the files should be as follows:

- IStorageBroker.cs
  - IStorageBroker.Students.cs
  - IStorageBroker.Teachers.cs
- StorageBroker.cs
  - StorageBroker.Students.cs

- StorageBroker.Teachers.cs

The primary purpose of this particular organization, leveraging partial classes, is to separate the concern for each entity to a finer level, which should make the maintainability of the software much higher.

Broker file and folder naming conventions strictly focus on the plurality of the entities they support and the singularity of the overall resource supported.

For instance, we say `IStorageBroker.Students.cs`. We also say `IEmailBroker` or `IQueueBroker.Notifications.cs` - singular for the resource and plural entities.

The same concept applies to the folders or namespaces containing these brokers.

For instance, we say:

```
namespace OtripleS.Web.Api.Brokers.Storages
{
    ...
}
```

And we say:

```
namespace OtripleS.Web.Api.Brokers.Queues
{
    ...
}
```

1.4 Broker Types

In most applications built today, some common Brokers are usually needed to get an enterprise application up and running - some of these are Storage, Time, APIs, Logging, and Queues.

Some brokers interact with existing system resources, such as *time*, to allow broker-neighboring services to treat time as a dependency and control how a particular service would behave based on the value of *time* at any point in the past, present, or future.

1.4.0 Entity Brokers

Entity brokers provide integration points with the system's external resources to fulfill business requirements.

For instance, entity brokers integrate with storage, providing capabilities to store or retrieve records from a database.

Entity brokers are also like queue brokers, providing a point of integration to push messages to a queue for other services to consume and process to fulfill their business logic.

Broker-neighboring services can only call entity brokers because they require a level of validation on the data they receive or provide before proceeding.

1.4.1 Support Brokers

Support brokers are general-purpose brokers. They provide the functionality to support services, but they have no characteristics that make them different from any other system.

An excellent example of a support broker is the `DateTimeBroker`, a broker specifically designed to abstract away the business layer's strong dependency on the system date time.

Time brokers don't target any specific entity; they are almost the same across many systems.

Another example of support brokers is the `LoggingBroker` - they provide data to logging and monitoring systems to enable the system's engineers to visualize the overall flow of data across the system and be notified in case any issues occur.

Support Brokers may be called across the entire business layer: foundation, processing, orchestration, coordination, management, or aggregation services, unlike Entity Brokers. Logging brokers are required as a supporting component in the system to provide all the capabilities needed for services to log their errors, calculate a date, or perform any other supporting functionality.

You can find real-world examples of brokers in the OtripleS project here.

1.5 Implementation

Here's a real-life implementation of an entire storage broker for all CRUD operations for `Student` entity:

1.5.1 Asynchronization Abstraction

In order to abstract at an operational level all example implementations in this section will utilize the ValueTask type to provide an asynchronization abstraction. This approach encourages a consistent handling of asynchronous operations across all method implementations, promoting a clean and efficient abstraction that aligns with Standard principles.

For IStorageBroker.cs:

```
namespace OtripleS.Web.Api.Brokers.Storages
{
    public partial interface IStorageBroker
    {
    }
}
```

For StorageBroker.cs:

```
using System;
using System.Linq;
using System.Threading.Tasks;
using EFxceptions.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using OtripleS.Web.Api.Models.Users;

namespace OtripleS.Web.Api.Brokers.Storages
{
    public partial class StorageBroker : EFxceptionsIdentityContext<User, Role,
Guid>, IStorageBroker
    {
        private readonly IConfiguration configuration;

        public StorageBroker(IConfiguration configuration)
        {
            this.configuration = configuration;
            this.Database.Migrate();
        }
```

```
        private async ValueTask<T> InsertAsync<T>(T @object)
        {
            this.Entry(@object).State = EntityState.Added;
            await this.SaveChangesAsync();

            return @object;
        }

        private async ValueTask<IQueryable<T>> SelectAllAsync<T>() where T : class
=> this.Set<T>();

        private async ValueTask<T> SelectAsync<T>(params object[] @objectIds) where
T : class =>
            await this.FindAsync<T>(objectIds);

        private async ValueTask<T> UpdateAsync<T>(T @object)
        {
            this.Entry(@object).State = EntityState.Modified;
            await this.SaveChangesAsync();

            return @object;
        }

        private async ValueTask<T> DeleteAsync<T>(T @object)
        {
            this.Entry(@object).State = EntityState.Deleted;
            await this.SaveChangesAsync();

            return @object;
        }

        ...

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {

optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
            string connectionString =
this.configuration.GetConnectionString("DefaultConnection");
            optionsBuilder.UseSqlServer(connectionString);
        }
    }
}
```

For IStorageBroker.Students.cs:

```
using system;
using system.Linq;
using system.Threading.Tasks;
using OtripleS.Web.Api.Models.Students;
```

```
namespace OtripleS.Web.Api.Brokers.Storages
{
    public partial interface IStorageBroker
    {
        ValueTask<Student> InsertStudentAsync(Student student);
        ValueTask<IQueryable<Student>> SelectAllStudentsAsync();
        ValueTask<Student> SelectStudentByIdAsync(Guid studentId);
        ValueTask<Student> UpdateStudentAsync(Student student);
        ValueTask<Student> DeleteStudentAsync(Student student);
    }
}
```

For StorageBroker.Students.cs:

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using OtripleS.Web.Api.Models.Students;

namespace OtripleS.Web.Api.Brokers.Storages
{
    public partial class StorageBroker
    {
        private DbSet<Student> Students { get; set; }

        public async ValueTask<Student> InsertStudentAsync(Student student) =>
            await InsertAsync(student);

        public async ValueTask<IQueryable<Student>> SelectAllStudentsAsync() =>
            await SelectAllAsync<Student>();

        public async ValueTask<Student> SelectStudentByIdAsync(Guid studentId) =>
            await SelectAsync<Student>(studentId);

        public async ValueTask<Student> UpdateStudentAsync(Student student) =>
            await UpdateAsync(student);

        public async ValueTask<Student> DeleteStudentAsync(Student student) =>
            await DeleteAsync(student);
    }
}
```

Support Brokers:

For IDateTimeBroker.cs:

```
public interface IDateTimeBroker
{
```

```
    ValueTask<DateTimeOffset> GetCurrentDateTimeOffsetAsync();
}
```

For DateTimeBroker.cs:

```
public class DateTimeBroker : IDateTimeBroker
{
    public async ValueTask<DateTimeOffset> GetCurrentDateTimeOffsetAsync() =>
        DateTimeOffset.UtcNow;
}
```

For ILoggingBroker.cs:

```
public interface ILoggingBroker
{
    ValueTask LogInformationAsync(string message);
    ValueTask LogTraceAsync(string message);
    ValueTask LogDebugAsync(string message);
    ValueTask LogWarningAsync(string message);
    ValueTask LogErrorAsync(Exception exception);
    ValueTask LogCriticalAsync(Exception exception);
}
```

For LoggingBroker.cs:

```
public class LoggingBroker : ILoggingBroker
{
    private readonly ILogger<LoggingBroker> logger;

    public LoggingBroker(ILogger<LoggingBroker> logger) =>
        this.logger = logger;

    public async ValueTask LogInformationAsync(string message) =>
        this.logger.LogInformation(message);

    public async ValueTask LogTraceAsync(string message) =>
        this.logger.LogTrace(message);

    public async ValueTask LogDebugAsync(string message) =>
        this.logger.LogDebug(message);

    public async ValueTask LogWarningAsync(string message) =>
        this.logger.LogWarning(message);

    public async ValueTask LogErrorAsync(Exception exception) =>
        this.logger.LogError(exception.Message, exception);

    public async ValueTask LogCriticalAsync(Exception exception) =>
        this.logger.LogCritical(exception, exception.Message);
}
```

1.6 Summary

Brokers are the first layer of abstraction between your business logic and the outside world. But they are not the only layer of abstraction because a few native models will still leak through your brokers to your broker-neighboring services. It is natural to avoid doing any mappings outside the realm of logic, in our case, the foundation services.

For instance, in a storage broker, regardless of the ORM used, some native exceptions from your ORM (EntityFramework, for example) will occur, such as `DbUpdateException` or `SqlException`. In that case, we need another layer of abstraction to act as a mapper between these exceptions and our core logic to convert them into local exception models.

This responsibility lies in the hands of the broker-neighboring services. I also call them foundation services; these services are the last point of abstraction before your core logic, which consists of local models and contracts.

1.7 FAQs

Over time, some common questions arose from the engineers I worked with throughout my career. Since some of these questions reoccurred on several occasions, it might be helpful to aggregate them here so everyone can learn about other perspectives on brokers.

1.7.0 Is the Brokers Pattern the same as the Repository Pattern?

From an operational standpoint, brokers seem to be more generic than repositories.

Repositories usually target storage-like operations, mainly towards databases. However, brokers can be an integration point for any external dependency, such as e-mail services, queues, and other APIs.

A more similar pattern for brokers is the Unit of Work pattern. It mainly focuses on the overall operation without having to tie the definition or the name with any particular operation.

In general, all these patterns try to implement the same SOLID principles: separation of concern, dependency injection, and single responsibility.

But because SOLID are principles and not exact guidelines, it's expected to see all different implementations and patterns to achieve that principle.

1.7.1 Why can't the brokers implement a contract for methods that return an interface instead of a concrete model?

That would be an ideal situation, but that would also require brokers to do a conversion or mapping between the native models returned from the external resource SDKs or APIs and the internal model that adheres to the local contract.

Doing that on the broker level will require introducing business logic into that realm, which is outside the purpose of that component.

We define business logic code as any intended sequential, selective, or iteration code. Brokers are not unit-tested because they lack business logic. They may be part of an acceptance or integration test but certainly not part of unit-level tests simply because they do not contain business logic.

1.7.2 If brokers were a layer of abstraction from the business logic, why do we allow external exceptions to leak through them onto the services layer?

Brokers are only the *first* layer of abstraction, but not the only one. Broker-neighboring services are responsible for converting the native exceptions occurring in a broker into a more local exception model that can be handled and processed internally within the business logic realm.

Business logic emerges in the processing, orchestration, coordination, and aggregation layers, where all the exceptions, returned models, and operations are local to the system.

1.7.3 Why do we use partial classes for brokers who handle multiple entities?

Since brokers must own their configurations, it makes more sense to partialize when possible to avoid reconfiguring every storage broker for each entity.

Partial classes are a feature in the C# language, but it should be possible to implement the same behavior through inheritance in other programming languages.

1.7.4 Are brokers the same as providers (Provider Pattern)?

No. Providers blur the line between services (business logic) and brokers (integration layer). Brokers target particular disposable components within the system, but providers include more than just that.

1.7.5 Why is ValueTask preferred over Task in the implementations outlined in this section?

ValueTask is a struct that is used to represent a task that returns a value. It is a value-based representation of a task that can be used to avoid allocations in the case where a task completes synchronously.

In order to abstract at an operational level, all example implementations in this section utilize the ValueTask type to provide an asynchronization abstraction. This approach encourages a consistent handling of asynchronous operations across all method implementations, promoting a clean and efficient abstraction that aligns with the Standard principles. By using ValueTask, we reduce memory allocations in scenarios where the result is often available immediately, thereby enhancing performance while maintaining code clarity.

1.7.6 Why does or how can I suppress warnings thrown in my code files?

Warnings are thrown by the compiler to alert you of potential issues in your code. It is important to address these warnings to ensure that your code is clean and free of potential bugs. Engineers and developers can resolve warnings by following the suggestions provided by the compiler. This may involve making changes to the code, refactoring, or updating dependencies.

It is important to regularly review and address warnings in your code to maintain code quality and ensure that your application runs smoothly.

Here is one way to resolve warnings in your code: To suppress a warning given by the IDE or compiler, you can configure your .csproj file using the <NoWarn> element. This approach allows you to disable specific warnings globally across your project.

For example, adding <NoWarn>CS1998</NoWarn> to a <PropertyGroup> in your .csproj file will suppress the warning CS1998, which indicates an asynchronous method lacking await operators. If you need to suppress multiple warnings, you can list them separated by commas. This method provides a clean and consistent way to manage warnings without scattering #pragma warning directives throughout your code.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

        <PropertyGroup>
                <TargetFramework>net8.0</TargetFramework>
                <Nullable>disable</Nullable>
                <ImplicitUsings>disable</ImplicitUsings>
        <WarningLevel>CS1998</WarningLevel>
        </PropertyGroup>
    ...
```

1.8 Standardized Provider Abstraction Libraries (SPAL)

When brokers rely on external libraries or routines, The Standard mandates the use of Standardized Provider Abstraction Libraries. These libraries decouple brokers from any specific external provider, ensuring extensibility, distributability, and local mockability (Cloud-Foreign).

For full details, see 1.8 Broker Provider Abstraction.

1.9 Versioning

1.9.0 Versioning Strategy

When a model is upgraded name — `EventAddress` becomes `EventAddressV1`, then `EventAddressV2`. Each version lives in its own dedicated partial class file (e.g., `StorageBroker.EventAddressV1s.cs`, `StorageBroker.EventAddressV2s.cs`),

keeping concerns cleanly separated and leaving the original unversioned file untouched for backward compatibility.

Each version of the model has its own set of operations, following the same naming conventions but with the version suffix included. This means each version is also persisted in its own table.

This allows older consumers of `EventAddress` or `EventAddressV1` to not be broken when a new version is introduced. Newer versions can also introduce additional operations — for example, both `V1` and `V2` expose a `DeleteAsync` method that does not exist on the original `EventAddresses` partial, reflecting evolving capabilities without retroactively modifying the legacy surface.

1.9.0.0 Special Considerations

The following concerns must be addressed before a new broker version is introduced.

- **Data Migration**: When data must be moved from one version to another, a migration strategy must be defined before the new version is introduced. Migration scripts or data transformation tooling must carry all records from the old version to the new while preserving every relationship and constraint.
- **Data Retention and Archiving**: When older versions of data must be retained for historical or auditing purposes, those records must be treated as read-only. An archiving strategy must be established so that older records remain retrievable without risk of accidental modification.
- **Deprecation**: When a version is ready to be retired, all consumers must be notified before removal begins. The older version must remain accessible until a defined end-of-life date, giving consumers sufficient time to migrate to the current version.
- **Schema Naming Conflicts**: When multiple versions of a model coexist in the same database, all table names, index names, and constraint names must include the version suffix to prevent naming collisions. This applies equally to Entity Framework fluent configurations and any raw migration scripts.

1.9.1 Pluralization

The plural `s` is always appended So the pattern is:

```
[Operation] + [ModelName] + [Version] + s + Async
```

Model	Method
`EventAddress`	`SelectAllEventAddressesAsync`
`EventAddressV1`	`SelectAllEventAddressV1sAsync`
`EventAddressV2`	`SelectAllEventAddressV2sAsync`

This means the plural `s` binds to the version tag, producing `V2s` rather than the potentially more "natural" but non-standard `sV2`. The same convention applies consistently to the `DbSet` property names (`EventAddressV2s`) and the file names (`StorageBroker.EventAddressV2s.cs`), keeping naming uniform across the entire broker layer.

1.9.2 File Name Conventions

File names follow a structured pattern

```
[I]StorageBroker . [ModelName][Vn][s] . [Configurations] . cs
```

- The optional `I` prefix indicates an interface file.
- `[ModelName]` is the entity name as-is.
- `[Vn]` is the version suffix (`V1` , `V2` , etc.), omitted for the original unversioned model.
- `[s]` is appended only when the model name is **singular** — if the name already ends in `s` (e.g., `Archives`), no extra `s` is added.
- `.Configurations` is an optional segment used for Entity Framework fluent configuration partial files.

Examples

File	Breakdown
`IStorageBroker.EventAddressV2s.cs`	Interface·singular model `EventAddress`·version `V2`·plural `s`

File	Breakdown
`StorageBroker.EventAddressV2s.cs`	Implementation · singular model `EventAddress` · version `V2` · plural `s`
`StorageBroker.EventArchivesV2.Configurations.cs`	Implementation · already-plural model `EventArchives` · version `V2` · no extra `s` · EF config

---

[*] Implementing Abstract Components (Brokers)

[*] Implementing Abstract Components (Part 2)

[*] Generating Model Migrations w/ EntityFramework

1.8

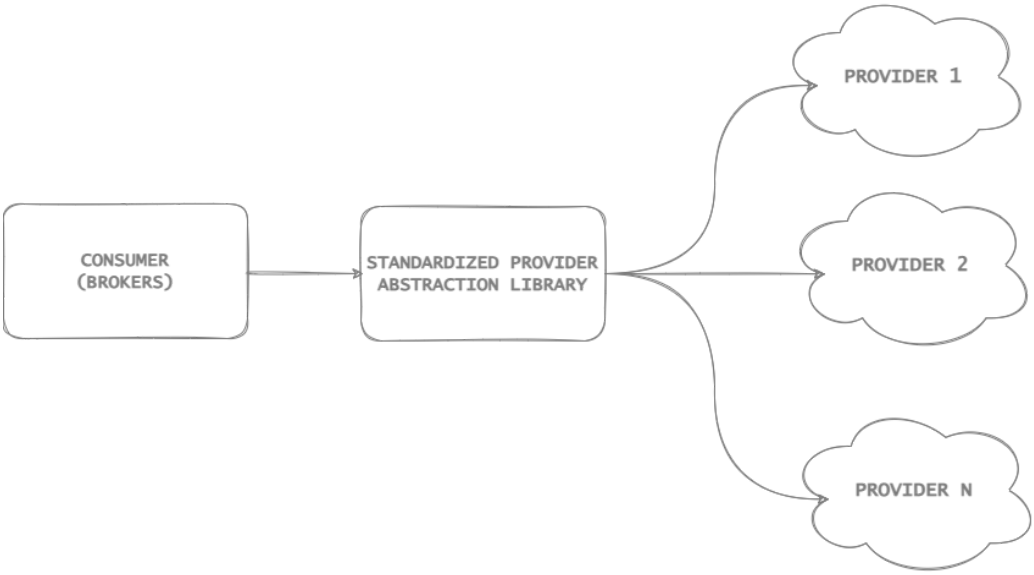
Standardized Provider Abstraction Libraries (SPAL)

This particular chapter will discuss the providers brokers may rely on to execute a certain operation. Most Brokers rely on external libraries or routines not owned by the engineers developing the system. These external libraries may or may not follow best practices regarding abstraction and testing.

The Standard mandates relying on standardized providers for systems design and development. These standardized providers must have the following characteristics:

1.8.0 Extensibility

Standardized provider abstraction libraries must be extensible to support more external providers. For example, a library that supports communicating with a database to SQL Server must be extensible enough to support communicating with MongoDB, MariaDB, or any other providers without any additional costs from the consumer of these libraries.



1.8.0.0 Configurability

For Standardized libraries to be usable with several providers, it must allow engineers to configure them to target a particular provider, local or remote. For instance, provider abstraction libraries can be configured in the following fashion:

```
public class StorageBroker
{
    private readonly IProviderAbstraction providerAbstraction;

    public StorageBroker()
    {
        this.providerAbstraction = new ProviderAbstraction();

        this.providerAbstraction.Configure(provider =>
            provider.UseSqlServer(connectionString: ....));
    }
}
```

Another example of using a different provider would be:

```
public class StorageBroker
{
    private readonly IProviderAbstraction providerAbstraction;

    public StorageBroker()
    {
        this.providerAbstraction = new ProviderAbstraction();
```

```
        this.providerAbstraction.Configure(provider =>
            provider.UseMariaDb(connectionString: ....));
    }
}
```

1.8.1 Distributability

Abstraction libraries must allow several engineers to publish their extensions of the library. The library does not need to have implementations of all providers in one binary. Instead, these libraries must provide an interface or a contract that all other extensions must implement to support a specific provider.

For instance, Let's assume we have the core standardized contract `Standard.Storages.Core`. We may publish a library called `Standard.Storages.Sql`. Anyone else can also publish `Standard.Storages.MongoDb` to support the same interface. An interface would look something like this:

```
public interface IStorageProvider
{
    ValueTask<T> Insert(T @object);
    ValueTask<IQueryable<T>> SelectAll();
    ...
    ...
    void Configure(Options options);
}
```

This contract's capabilities must be the bare minimum any provider can provide. However, the additional options in the provider extension may expose more capabilities that may or may not exist in other libraries.

1.8.2 External Mockability (Cloud-Foreign)

Standardized provider libraries must allow communications with mocked local phantom APIs. For instance, if a system requires communication with a queue or an event bus in the cloud, the provider library abstracts that technology must allow a local connection for Acceptance Testing and Airplane-Mode runs, which we discussed earlier as Cloud Foriegn Principle.

External Mockability may rely on other external libraries that implement patterns such as PACT to create phantom or fake external API instances running on the local machine or network. Here's an example:

```
public class EventBroker
{
    private readonly IEventAbstractProvider eventAbstractProvider;

    public EventBroker(IConfiguration configuration)
    {
        this.eventAbstractProvider =
            new EventAbstractProvider(
                configuration.Connection,
                configuration.TargetServerType);
    }
}
```

In the above snippet, the `TargetServerType` can be either `Remote` or `Local` as mandatory options, but the engineers developing the library may add other options if they so choose.

Abstract provider libraries must mimic the exact behavior of their providers. For instance, in a queue-listening scenario, these libraries must expose an API that supports eventing for incoming messages, even if they are local and not from an external service such as the cloud.

It is also acceptable to have the option to support local intranet networks and governed networks that are not connected to the public internet in specific scenarios using these very same libraries.

Standardized provider abstraction libraries are subsystems that must have their own Brokers, Services, and Exposure layers according to The Standard. These libraries will further simplify the development of customer-facing systems with well-defined exceptions to handle expectations and simpler modifications since they are open-source and Standard-Compliant.

1.8.3 Local to Global

Engineers may develop their local provider abstraction libraries using the same solution, assuming they need help finding an existing effort to support their needs. By doing so, these engineers are encouraged to open-source and publish that

abstraction project to support other engineers in The Standard Community who may have the exact needs.

This practice encourages engineers everywhere to create a collective effort and hive-mind patterns to solve a problem once and share it with the rest of The Standard Engineering Community for further enhancement and support.

Standard-compliant edge (customer-facing) systems should no longer add any non-standard libraries to their APIs, Desktop or Web Applications, or any other systems.

2

Services

2.0 Introduction

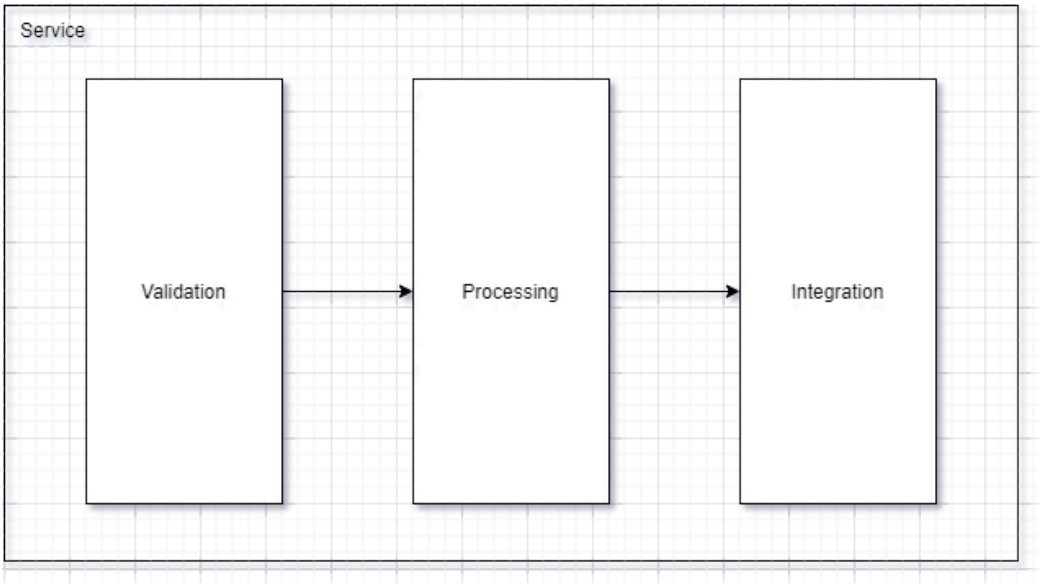
Services, in general, are the containers of all the business logic in software—they are the core component of any system and the main component that makes one system different from another.

Our main goal with services is to keep them agnostic from specific technologies or external dependencies.

Any business layer is more compliant with The Standard if it can plug into other dependencies and exposure technologies with the least integration effort.

2.0.0 Services Operations

When we say business logic, we mainly refer to three main categories of operations: validation, processing, and integration.



Let's talk about these categories.

2.0.0.0 Validations

Validations ensure that incoming or outgoing data match a particular set of rules, such as structural, logical, or external validations, in that exact order of priority. We will discuss this in detail in the upcoming sections.

2.0.0.1 Processing

Processing mainly focuses on flow control, mapping, and computation to satisfy a business need—the processing operations distinguish one service from another and, in general, one piece of software from another.

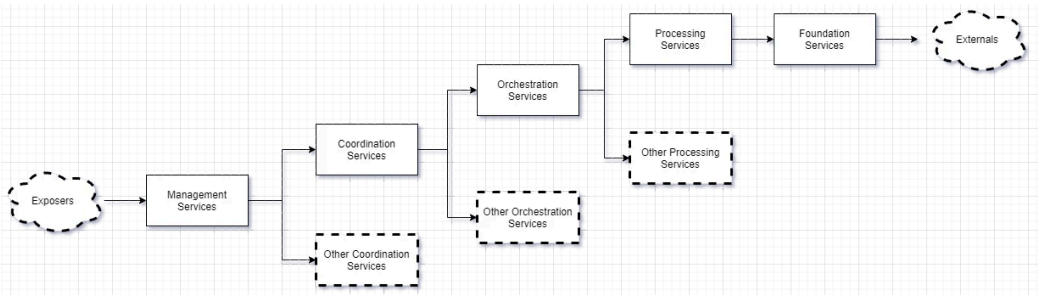
2.0.0.2 Integration

Finally, the integration process focuses on retrieving or pushing data from or to any integrated system dependencies.

We will discuss these aspects in detail in the upcoming chapter. The main thing to understand about services is that their design is to be pluggable and configurable, allowing them to easily integrate with any technology from a dependency standpoint and easily plug into any exposure functionality from an API perspective.

2.0.1 Services Types

Services are classified into several types based on their disposition in any given architecture. They fall into three main categories: validators, orchestrators, and aggregators.



2.0.1.0 Validators

Validator services are mainly broker-neighboring services or foundation services.

These services' primary responsibility is to add a validation layer on top of the existing primitive operations, such as the CRUD operations, to ensure incoming and outgoing data is validated structurally, logically, and externally before sending the data in or out of the system.

2.0.1.1 Orchestrators

Orchestrator services are the core of the business logic layer. They can be processors, orchestrators, coordinators, or management services, depending on the type of their dependencies.

Orchestrator services mainly focus on combining multiple primitive operations or multiple high-order business logic operations to achieve an even higher goal.

Orchestrator services are: The decision-makers within any architecture. The owners of the flow control in any system. The main component that makes one application or software different from the other.

We intentionally design Orchestrator services to be longer-lived than any other type of service in the system.

2.0.1.2 Aggregators

The primary responsibility of the aggregator services is to tie the outcome of multiple processing, orchestration, coordination, or management services to expose one single API for any given API controller or UI component to interact with the rest of the system.

Aggregators are the gatekeepers of the business logic layer. They ensure the data exposure components (like API controllers) interact with only one point of contact to interact with the rest of the system.

Aggregators, in general, don't care about the order in which they call the operations that are attached to them. Still, it is sometimes necessary to execute a particular operation, such as creating a student record before assigning a library card.

In the following chapters, we will discuss each type of these services.

2.0.2 Overall Rules

Several rules govern the overall architecture and design of services in any system.

These rules ensure the system's overall readability, maintainability, and configurability - in that particular order.

2.0.2.0 Do or Delegate

Every service should either do or delegate the work, but not both.

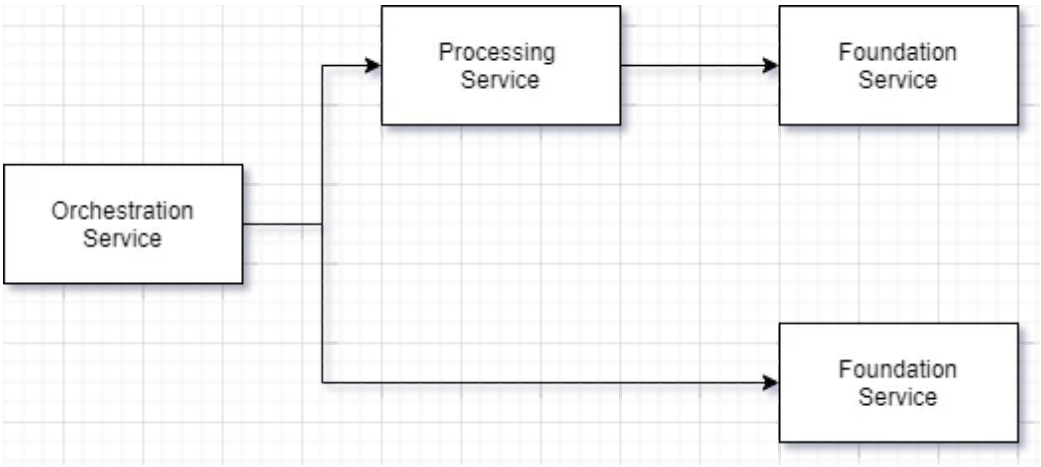
For instance, a processing service should delegate the work of persisting data to a foundation service rather than try to do that work itself.

2.0.2.1 Two-Three (Florance Pattern)

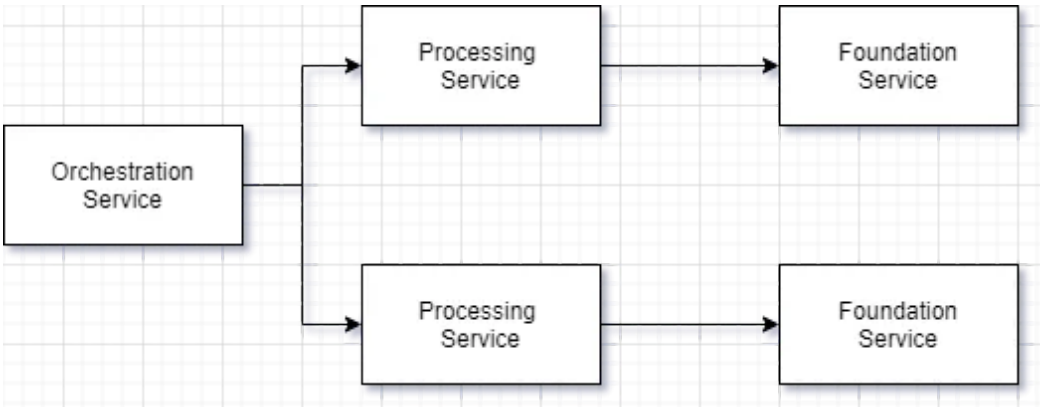
For Orchestrator services, the dependencies of services (not brokers) should be limited to two or three, not one, four, or more.

Suppose an Orchestrator depends only on one service. In that case, it violates the definition of orchestration, which is the combination of multiple operations from different sources to achieve a higher order of business logic.

This pattern violates Florance Pattern



This pattern follows the symmetry of the Florance Pattern



The Florance pattern also ensures the balance and symmetry of the overall architecture.

For instance, you can't orchestrate between a foundation and a processing service. This causes an imbalance in your architecture and difficulty when trying to combine one unified statement with the language each service speaks based on its level and type.

The aggregators are the only types of services allowed to violate this rule, where the combination and the order of services or their calls don't have any real impact.

We will discuss the Florance pattern in detail in the upcoming sections of The Standard.

2.0.2.2 Single Exposure Point

API controllers, UI components, or any other form of system data exposure should have one single point of contact with the business logic layer.

For instance, an API endpoint that offers endpoints for persisting and retrieving student data should not have multiple integrations with multiple services but one service that provides all these features.

Sometimes, a single orchestration, coordination, or management service does not offer everything related to a particular entity. An aggregator service combines all these features into one service that is ready to be integrated with exposure technology.

2.0.2.3 Same or Primitives I/O Model

All services must maintain a single contract regarding their return and input types, except if they are primitives.

For instance, a service that provides operations for an entity type `Student` - should not return from any of its methods from any other entity type.

You may return an aggregation of the same entity, whether it's custom or native, such as `List<Student>` or `AggregatedStudents` models, or a primitive type like getting students count, or a boolean indicating whether a student exists or not but not any other non-primitive or non-aggregating contract.

A similar rule applies for input parameters - any service may receive an input parameter of the same contract, a virtual aggregation contract, or a primitive type but not any other contract.

This rule focuses the responsibility of a service on a single entity and all its related operations.

When a service returns a different contract, it violates its naming convention like a `StudentOrchestrationService` returning `List<Teacher>` - and it starts falling into the trap of being called by other services from entirely different data pipelines.

If primitive input parameters belong to a different entity model that is not necessarily a reference to the primary entity, it begs the question of orchestrating between two processing or foundation services to maintain a unified model without breaking the pure-contracting rule.

Suppose an orchestration service requires a combination of multiple different contracts. In that case, a new unified virtual model indicates the need for a new unique contract for the orchestration service, with mappings implemented underneath on the concrete level of that service to maintain compatibility and integration safety.

2.0.2.4 Every Service for Itself

Every service is responsible for validating its inputs and outputs. Do not rely on services upstream or downstream to validate your data.

This is a defensive programming mechanism to ensure that if implementations are swapped behind contracts, the responsibility of any given service is not affected if downstream or upstream services decide to pass on their validations for any reason.

Within any monolithic, microservice, or serverless architecture-based system, every service is designed to split off from the system at some point and become the last point of contact before integrating with some external resource broker.

For instance, in the following architecture, services map parts of an input `Student` model into a `LibraryCard` model. Here's a visual of the models:

Student

```
public class Student
{
    public Guid Id {get; set;}
    public string Name {get; set;}
}
```

LibraryCard

```
public class LibraryCard
{
    public Guid Id {get; set;}
    public Guid StudentId {get; set;}
}
```

Now, assume that our orchestrator service `StudentOrchestrationService` is ensuring every new student that gets registered will need to have a library card, so our logic may look as follows:

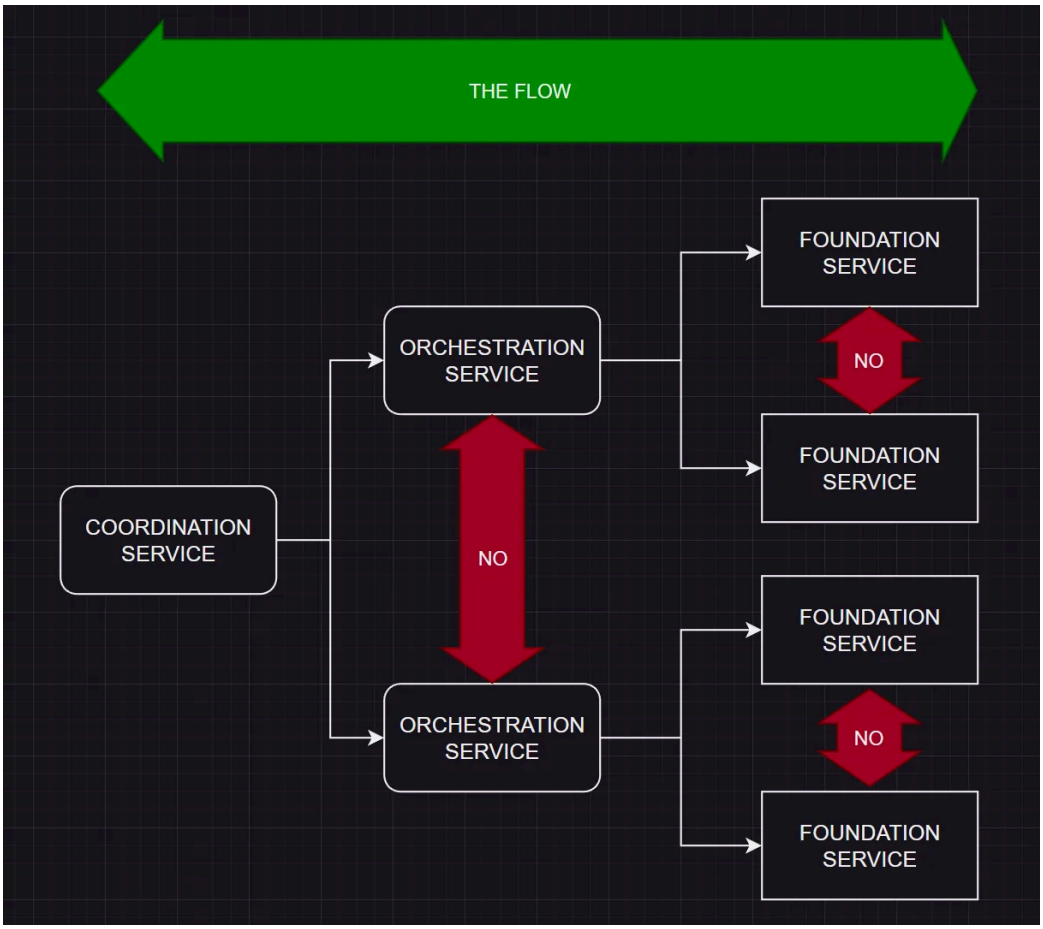
```
public async ValueTask<Student> RegisterStudentAsync(Student student)
{
    Student registeredStudent =
        await this.studentProcessingService.RegisterStudentAsync(student);

    await AssignStudentLibraryCardAsync(student);

    return registeredStudent;
}

private async ValueTask<LibraryCard> AssignStudentLibraryCardAsync(Student student)
{
    LibraryCard studentLibraryCard = MapToLibraryCard(student);
```

```
    return await
this.libraryCardProcessingService.AddLibraryCardAsync(studentLibraryCard);
}

private LibraryCard MapToLibraryCard(Student student)
{
    return new LibraryCard
    {
        Id = Guid.NewGuid(),
        StudentId = student.Id
    };
}
```

As you can see above, a valid student id is required to map to a `LibraryCard successfully`. Since the mapping is the orchestrator's responsibility, we must ensure that the input student and its id are in good shape before proceeding with the orchestration process.

2.0.2.5 Flow Forward

Services cannot call services at the same level. For instance, Foundation Services cannot call other Foundation Services, and Orchestration Services cannot call other Orchestration Services from the same level. This principle is called a Flow-Forward - as the illustration shows:



2.0.2.5.0 For APIs

Due to fractality, The same rule applies to methods within these services. Public APIs cannot call public APIs. Here's an example:

```
public async ValueTask<Student> RetrieveStudentByIdAsync(Guid studentId)
{
    ...

    return await this.storageBroker.SelectStudentByIdAsync(studentId);
}

public async ValueTask<Student> ModifyStudentAsync(Student student)
{
    ...

    Student maybeStudent =
        await this.storageBroker.SelectStudentByIdAsync(studentId);
```

```
    ...
    ...
}
```

In the Foundation Service example above, we cannot call `RetriveStudentByIdAsync` in a `public` method from another `public` method such as `ModifyStudentAsync`. You will see that both methods call the exact same method from a lower dependency, like a `StorageBroker`, fully independent of one another.

While this may seem redundant, the reason for this is that `public` APIs, contracts, or otherwise, are destined to be deprecated at some point in their lifetime. They may also be changed completely from an implementation standpoint. If a `public` API depended on another `public` API at the same level, the deprecation of one will cause a cascading effect on all others. That's a symptom of Chaotic design, which The Standard strongly prohibits.

2.0.3 Versioning

File versioning governs how changes to models and services are managed and communicated in the codebase.

2.0.3.0 Model Versioning

When a model's structure is changed (including adding, removing, or modifying properties), a new version of the model must be created. The version number must be incremented, and both the model file and its path must reflect the new version. This preserves backward compatibility and allows consumers to upgrade at their discretion.

New model versions must be created alongside the previous version and organised in a versioned subfolder.

Example:

Existing (V0):

```
Models/Foundations/Students/Student.cs
```

New extended model (V1) - nested under the previous version:

```
Models/Foundations/Students/V1/StudentV1.cs
```

2.0.3.1 Service Versioning

The same principle applies to services. When a service's contract changes due to a model update, increment the model version in the service filename and folder path:

```
Services/Foundations/Students/V1/StudentV1Service.cs
```

Exceptions related to the new model version should follow the same nesting convention and reside under the corresponding versioned model path:

```
Models/Foundations/Students/V1/Exceptions/
```

2.0.3.2 Service Behavior Versioning

The filename of the service always reflects the model version and not the behavior version:

```
Services/Foundations/Students/V1/StudentV1Service.cs
```

Every time there is a behavior change, increment the method behavior version in the method name by creating a new method:

We will follow this convention for all behaviour changes:

```
[Operation] + [OperationBehaviorVersion] + [Resource] + [ResourceContractVersion] +
[Optional QueryDescriptor] + [AsyncSuffix]
```

e.g.

```
Retrieve + V1 + Student + V1 + ById + Async
```

(V0 is implied and does not need to be explicitly stated in [OperationBehaviorVersion] OR [ResourceContractVersion])

Example of method behavior versioning for a model version at V0:

```
Services/Foundations/Students/StudentService.cs
```

```
ValueTask<Student> RetrieveStudentByIdAsync(Guid studentId)
ValueTask<Student> RetrieveV1StudentByIdAsync(Guid studentId)
ValueTask<Student> RetrieveV2StudentByIdAsync(Guid studentId)
```

Example of method behavior versioning for a model version greater than V0:

```
Services/Foundations/Students/V1/StudentV1Service.cs
```

```
ValueTask<StudentV1> RetrieveStudentV1ByIdAsync(Guid studentId)
ValueTask<StudentV1> RetrieveV1StudentV1ByIdAsync(Guid studentId)
ValueTask<StudentV1> RetrieveV2StudentV1ByIdAsync(Guid studentId)
```

2.0.3.3 Model Upgrade and Behavior Version Reset

Whenever the model version is incremented, the service filename and folder path must reflect the new model version, and the behavior version resets to V0:

```
Services/Foundations/Students/V2/StudentV2Service.cs
```

The service method behavior version resets to V0 whenever a new model version is introduced. Any further behavior changes for that model version will result in an increment of the method behavior version (V1, V2, etc.), while the model version remains constant.

Example:

Before model upgrade (model V1 with multiple behavior versions):

```
internal interface IStudentV1Service
{
    ValueTask<StudentV1> RetrieveStudentV1ByIdAsync(Guid studentId)
    ValueTask<StudentV1> RetrieveV1StudentV1ByIdAsync(Guid studentId)
    ValueTask<StudentV1> RetrieveV2StudentV1ByIdAsync(Guid studentId)
}
```

After model upgrade (model V2 with only the latest behavior version):

```
internal interface IStudentV2Service
{
    ValueTask<StudentV2> RetrieveStudentV2ByIdAsync(Guid studentId);
}
```

The new V2 model service only exposes the latest behavior under the implied V0 naming convention. The V2 in this instance refers to the model version since it is appended to the resource. The behavior version resets on a model upgrade and since the model is now at the implied V0 we do not have to indicate it after the operation.

2.0.3.4 Naming Conventions

The naming convention for versioned files

- A versioned model is named `{Entity}V{n}.cs`
- A versioned service named after the model version `{Entity}V{n}Service.cs`
  - Any behaviour versioning is reflected in the method name, immediately after the operation. When the behaviour version is omitted, it is implied to be V0. The version token immediately after the operation belongs to the behaviour, while the version token appended to the resource name belongs to the model — the two are tracked independently:
    - V0 (implied) → `ValueTask<EventAddressV1>` `AddEventAddressV1Async(EventAddressV1 eventAddress);`
    - V1 → `ValueTask<EventAddressV1>` `AddV1EventAddressV1Async(EventAddressV1 eventAddress);`
    - V2 → `ValueTask<EventAddressV1>` `AddV2EventAddressV1Async(EventAddressV1 eventAddress);`
- When the model version changes, the service filename increments to reflect the new model version, and the behavior version resets to V0. Following the formula from `2.0.3.2`, the implied V0 behaviour version is omitted:
  - Before (model V1): `{Entity}V1Service.cs`
  - After (model V2): `{Entity}V2Service.cs`
  - The method naming resets to the implied V0 convention:
    - V0 (implied, reset) → `ValueTask<EventAddressV2>` `AddEventAddressV2Async(EventAddressV2 eventAddress);`

This ensures that the model version, behaviour version, and file structure are all independently readable and unambiguous at every level.

2.0.4 Class Visibility and Exposure Control

Classes must be declared with the lowest possible visibility required for their intended use. By default, types should be internal unless there is a clear and deliberate need to expose them outside the assembly.

This principle reduces the exposure surface area of a library, protects internal implementation details, and prevents unintended coupling by consumers.

A smaller public surface:

- Improves maintainability by allowing internal refactoring without breaking consumers
- Enforces clear architectural boundaries
- Limits misuse of internal logic that was never designed for external consumption

2.0.4.0 Exception Visibility Rules

Exception types follow a stricter, intent-driven visibility model:

- **Localization Exceptions** — `public` These represent well-defined, domain-specific failures that are meaningful to consumers. They are expected to propagate beyond the library boundary and must therefore be accessible.
- **Categorization Exceptions** — `internal` These are used strictly for internal classification and logging. They must never escape the library boundary and should be stripped or translated at each layer. Keeping them internal ensures they cannot leak into external contracts.
- **Exposure Exceptions** — `public` These represent failures that are intentionally exposed to consumers (e.g., validation or dependency failures). They form part of the contract and must be accessible outside the library.

Example

Public localization exceptions - meaningful to consumers

```
public class NotFoundStudentException : Xeption
{
    public NotFoundStudentException(string message)
        : base(message)
    { }
}

public class InvalidStudentException : Xeption
{
    public InvalidStudentException(string message)
        : base(message)
    { }
```

```
}

public class FailedStudentServiceException : Xeption
{
    public FailedStudentServiceException(string message, Exception innerException)
        : base(message, innerException)
    { }
}
```

Internal categorization exceptions - used only within the library

```
internal class StudentValidationException : Xeption
{
    public StudentValidationException(string message, Xeption innerException)
        : base(message, innerException)
    { }
}

internal class StudentDependencyValidationException : Xeption
{
    public StudentDependencyValidationException(string message, Xeption
innerException)
        : base(message, innerException)
    { }
}

internal class StudentDependencyException : Xeption
{
    public StudentDependencyException(string message, Xeption innerException)
        : base(message, innerException)
    { }
}

internal class StudentServiceException : Xeption
{
    public StudentServiceException(string message, Xeption innerException)
        : base(message, innerException)
    { }
}

internal class StudentService : IStudentService
{
    // Internal logic hidden from consumers
}
```

Public contract for student operations

```
public interface IStudentClient
{
```

```
    ValueTask<Student> GetStudentByIdAsync(Guid studentId);
}
```

Internal implementation of the public contract, with all dependencies and logic hidden

```
internal class StudentClient : IStudentClient
{
    private readonly IStudentService studentService;

    public StudentClient(IStudentService studentService) =>
        this.studentService = studentService;

    public ValueTask<Student> GetStudentByIdAsync(Guid studentId) =>
        this.studentService.RetrieveStudentByIdAsync(studentId);
}
```

Public university client exposing only the public contract, with all internal details hidden

```
public interface IUniversityClient
{
    IStudentClient Students { get; }
}

public class UniversityClient : IUniversityClient
{
    public UniversityClient(UniversityClientConfigurations configurations)
    {
        IServiceProvider serviceProvider = RegisterServices(configurations);
        InitializeClients(serviceProvider);
    }

    public IStudentClient Students { get; private set; }

    private void InitializeClients(IServiceProvider serviceProvider)
    {
        this.Students = serviceProvider.GetRequiredService<IStudentClient>();
    }

    private static IServiceProvider RegisterServices(
        UniversityClientConfigurations configurations)
    {
        var services = new ServiceCollection()
            // Internal dependencies
            .AddTransient<IStorageBroker, StorageBroker>()
            .AddTransient<IDateTimeBroker, DateTimeBroker>()
            .AddTransient<IStudentService, StudentService>()
```

```
            // Internal client mapped to public interface
            .AddTransient<IStudentClient, StudentClient>()

            // Configurations
            .AddSingleton(configurations);

        return services.BuildServiceProvider();
    }
}
```

Key Takeaways The consumer only interacts with UniversityClient and interfaces StudentClient and all supporting services are hidden (internal) Dependency injection is fully encapsulated within the library The library retains full control over implementation while exposing a stable contract

Guiding Principle

If a type is not explicitly part of the public contract, it must remain internal.

Visibility is not just an access modifier—it is a design decision that defines the boundary between what a library promises and what it is free to change.

2.1

Foundation Services (Broker-Neighboring)

2.1.0 Introduction

Foundation services are the first point of contact between your business logic and the brokers.

In general, the broker-neighboring services are a hybrid of business logic and an abstraction layer for the processing operations where the higher-order business logic happens, which we will talk about further when we start exploring the processing services in the next section.

Broker-neighboring services main responsibility is to ensure the incoming and outgoing data through the system is validated and vetted structurally, logically and externally.

You can also think of broker-neighboring services as a layer of validation on top of the primitive operations the brokers already offer.

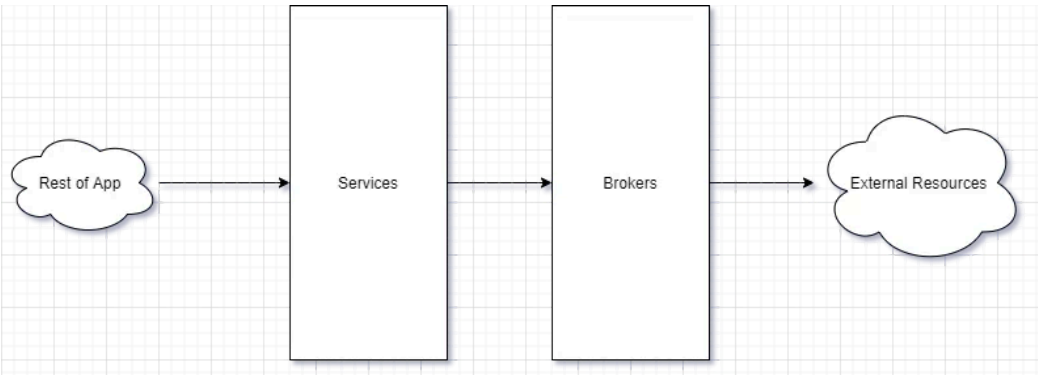
For instance, if a storage broker is offering `InsertStudentAsync(Student student)` as a method, then the broker-neighboring service will offer something as follows:

```
public async ValueTask<Student> AddStudentAsync(Student student)
{
        await ValidateStudentOnAddAsync(student);

        return await this.storageBroker.InsertStudentAsync(student);
}
```

This makes broker-neighboring services nothing more than an extra layer of validation on top of the existing primitive operations brokers already offer.

2.1.1 On The Map

The broker-neighboring services reside between your brokers and the rest of your application, on the left side higher-order business logic processing services, orchestration, coordination, aggregation or management services may live, or just simply a controller, a UI component or any other data exposure technology.



2.1.2 Characteristics

Foundation or Broker-Neighboring services in general have very specific characteristics that strictly govern their development and integration.

Foundation services in general focus more on validations than anything else - simply because that's their purpose, to ensure all incoming and outgoing data through the system is in a good state for the system to process it safely without any issues.

Here's the characteristics and rules that govern broker-neighboring services:

2.1.2.0 Pure-Primitive

Broker-neighboring services are not allowed to combine multiple primitive operations to achieve a higher-order business logic operation.

For instance, broker-neighboring services cannot offer an *upsert* function, to combine a `Select` operations with an `Update` or `Insert` operations based on the outcome to ensure an entity exists and is up to date in any storage.

But they offer a validation and exception handling (and mapping) wrapper around the dependency calls, here's an example:

```
public ValueTask<Student> AddStudentAsync(Student student) =>
TryCatch(async () =>
{
        await ValidateStudentOnAddAsync(student);

        return await this.storageBroker.InsertStudentAsync(student);
});
```

In the above method, you can see `ValidateStudentOnAdd` function call preceded by a `TryCatch` block. The `TryCatch` block is what I call Exception Noise Cancellation pattern, which we will discuss soon in this very section.

But the validation function ensures each and every property in the incoming data is validated before passing it forward to the primitive broker operation, which is the `InsertStudentAsync` in this very instance.

2.1.2.1 Single Entity Integration

Services strongly ensure the single responsibility principle is implemented by not integrating with any other entity brokers except for the one that it supports.

This rule doesn't necessarily apply to support brokers like `DateTimeBroker` or `LoggingBroker` since they don't specifically target any particular business entity and they are almost generic across the entire system.

For instance, a `StudentService` may integrate with a `StorageBroker` as long as it only targets the functionality offered by the partial class of the `StorageBroker.Students.cs` file when exists.

Foundation services should not integrate with more than one entity broker of any kind simply because it will increase the complexity of validation and orchestration of which goes beyond the main purposes the service which is just simply validation.

We push this responsibility further to the orchestration-type services to handle.

2.1.2.2 Business Language

Broker-neighboring services speak primitive business language for their operations. For instance, while a Broker may provide a method with the name `InsertStudentAsync` - the equivalant of that on the service layer would be `AddStudentAsync`.

In general, most of the CRUD operations shall be converted from a storage language to a business language, and the same goes for non-storage operations such as Queues, for instance we say `PostQueueMessage` but on the business layer we shall say `EnqueueMessage`.

Since the CRUD operations are the most common ones in every system, our mapping to these CRUD operations would be as follows:

Brokers	Services
Insert	Add
Select	Retrieve
Update	Modify
Delete	Remove

As we move forward towards higher-order business logic services, the language of the methods being used will lean more towards a business language rather than a technology language as we will see in the upcoming sections.

2.1.3 Responsibilities

Broker-neighboring services play three very important roles in any system. The first role is to abstract away native broker operations from the rest of the system. Irregardless of whether a broker is a communication between a local or external storage or an API - broker-neighboring services will always have the same contract/verbiage to expose to upper stream services such as processing, orchestration or simply exposers like controllers or UI components. The second and most important role is to offer a layer of validation on top of the existing primitive operations a broker already offers to ensure incoming and outgoing data is valid to be processed or persisted by the system. The third role is to play the role

of a mapper of all other native models and contracts that may be needed to completed any given operation while interfacing with a broker. Foundation services are the last point of abstraction between the core business logic of any system and the rest of the world, let's discuss these roles in detail.

2.1.3.0 Abstraction

The first and most important responsibility for foundation/broker-neighboring services is to ensure a level of abstraction exists between the brokers and the rest of your system. This abstraction is necessary to ensure the pure business logic layer in any system is verbally and functionally agnostic to whichever dependencies the system is relying on to communicate with the outside world.

Let's visualize a concrete example of the above principle. Let's assume we have a `StudentProcessingService` which implements an `UpsertStudentAsync` functionality. Somewhere in that implementation there will be a dependency on `AddStudentAsync` which is exposed and implemented by some `StudentService` as a foundation service. Take a look at this snippet:

```
public async ValueTask<Student> UpsertStudentAsync(Student student)
{
        ...

        return await this.studentService.AddStudentAsync(student);
}
```

The contract between a processing or an orchestration service and a foundation service will always be the same irregardless of what type of implementation or what type of brokers the foundation service is using. For example, `AddStudentAsync` could be a call to a database or an API endpoint or simply putting a message on a queue. It doesn't impact in any way, shape or form the upstream processing service implementation. Here's an example of three different implementations of a foundation service that wouldn't change anything in the implementation of it's upstream services:

With a storage broker:

```
public async ValueTask<Student> AddStudentAsync(Student student)
{
        ...
```

```
        return await this.storageBroker.InsertStudentAsync(student);
}
```

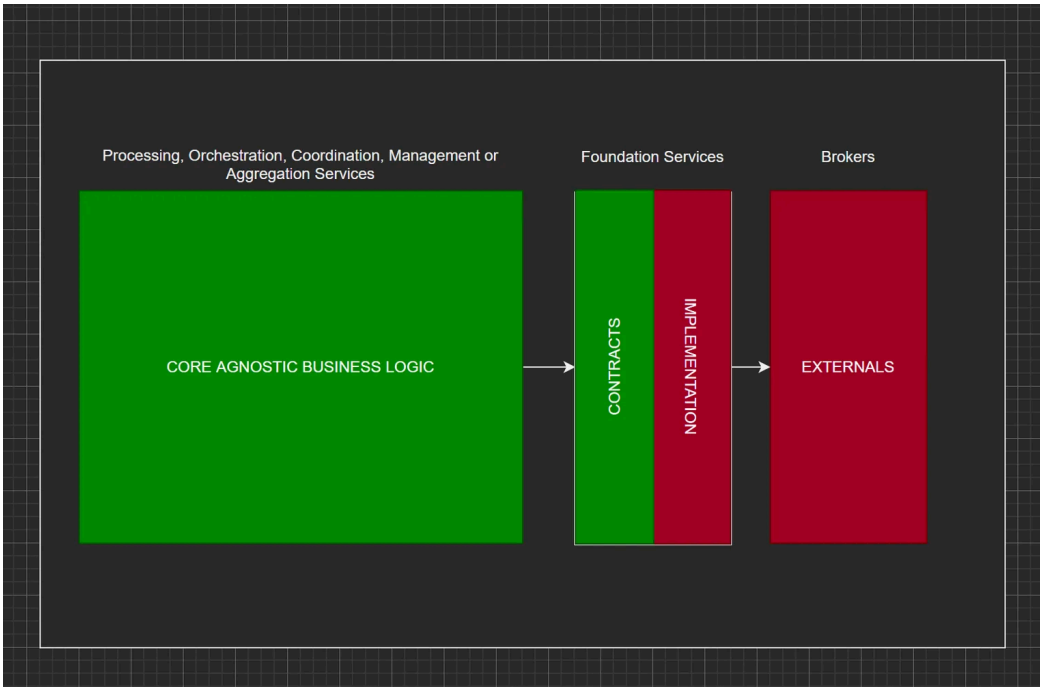
Or with a queue broker:

```
public async ValueTask<Student> AddStudentAsync(Student student)
{
        ...

        return await this.queueBroker.EnqueueStudentAsync(student);
}
```

or with an API broker:

```
public async ValueTask<Student> AddStudentAsync(Student student)
{
        ...

        return await this.apiBroker.PostStudentAsync(student);
}
```

here's a visualization of that concept:



In all of these above cases, the underlying implementation may change, but the exposed contract will always stay the same for the rest of the system. We will discuss in later chapters how the core, agnostic and abstract business logic of your system starts with Processing services and ends with Management or Aggregation services.

2.1.3.0.1 Implementation

Let's talk about a real-life example of implementing a simple `Add` function in a foundation service. This is also called our Happy Path test. It is generally the starting point of most CRUD type foundation services. Let's assume we have the following contract for our `StudentService`:

```
public IStudentService
{
        ValueTask<Student> AddStudentAsync(Student student);
}
```

For starters, let's go ahead and write a failing test for our service as follows:

```
private async Task ShouldAddStudentAsync()
{
        // given
        Student randomStudent = CreateRandomStudent();
        Student inputStudent = randomStudent;
        Student storageStudent = inputStudent;
        Student expectedStudent = storageStudent.DeepClone();

        this.storageBrokerMock.Setup(broker =>
                broker.InsertStudentAsync(inputStudent))
                        .ReturnsAsync(storageStudent);

        // when
        Student actualStudent =
                await this.studentService.AddStudentAsync(inputStudent);

        // then
        actualStudent.Should().BeEquivalentTo(expectedStudent);

        this.storageBroker.Verify(broker =>
                broker.InsertStudentAsync(inputStudent),
                        Times.Once);

        this.storageBrokerMock.VerifyNoOtherCalls();
        this.loggingBrokerMock.VerifyNoOtherCalls();
}
```

In the above test, we defined four variables with the same value. Each variable contains a name that best fits the context it will be used in. For instance, `inputStudent` best fits in the input parameter position, while `storageStudent` best fits to what gets returned from the storage broker after a student is persisted sucessfully.

We also randomize our inputs and outputs across all tests to ensure that the test is targeting a certain functional behavior. There are cases where testing for a specific value or set of values is required. But it will need to have quite a strong justification to do such a thing.

It's easy to "fool" tests into reporting a successful operation by using specific input parameters or output values. Randomization is required by default unless needed otherwise. For instance, if you have a function `Add(x, y)` if the test is passing specifically `1` and `2` to the function and expects `3` as a return value, then anyone could ignore the actual arithmatic operation and just return `3` all the time from target function. Take that at a larger enterprise level scale as the problem gets more complex and you can see how important and crucial it is to randomize inputs and outputs by default.

You will also notice that we use a DeepCloner(force-net) method to deep clone the `expectedStudent` variable to ensure no modifications have happened to the originally passed in student. For instance, assume an input student value has changed for any of it's attributes internally within the `AddStudentAsync` function. That change won't trigger a failing test unless we dereference the `expectedStudent` variable from the input and returned variables.

We mock the response from the storage broker and execute our subject of test `AddStudentAsync` then we verify the returned student value `actualStudent` matches the expected value `expectedStudent` regardless of the reference.

Finally, we verify all calls are done properly and no additional calls have been made to any of the service dependencies.

Let's make that test pass by writing in an implementation that only satisfies the requirements of the aforementioned test:

```
public async ValueTask<Student> AddStudentAsync(Student student) =>
```

```
        await this.storageBroker.InsertStudentAsync(student);
```

This simple implementation should make our test pass sucessfully. It's important to understand that any implementation should be only enough to pass the failing tests. Nothing more and nothing less.

2.1.3.1 Validation

Foundation services are required to ensure incoming and outgoing data from and to the system are in a good state - they play the role of a gatekeeper between the system and the outside world to ensure the data that goes through is structurally, logically and externally valid before performing any further operations by upstream services. The order of validations here is very intentional. Structural validations are the cheapest of all three types. They ensure a particular attribute or piece of data in general doesn't have a default value if it's required. The opposite of that is the logical validations, where attributes are compared to other attributes with the same entity or any other. Additional logical validations can also include a comparison with a constant value like comparing a student enrollment age to be no less than 5 years of age. Both strucural and logical validations come before the external. As we said, it's simply because we don't want to pay the cost of communicating with an external resource including latency tax if our request is not in a good shape first. For instance, we shouldn't try to post some `Student` object to an external API if the object is `null`. Or if the `Student` model is invalid structurally or logically.

For all types of validations, it's important to understand that some validations are circuit-breaking or requiring an immediate exit from the current flow by throwing an exception or returning a value in some cases. And some other validations are continuous. Let's talk about these two sub categories of validations first.

2.1.3.1.0 Circuit-Breaking Validations

Circuit-breaking validations require an immediate exit from the current flow. For instance, if an object being passed into a function is `null` - there will be no further operations required at that level other than exiting the current flow by throwing an exception or returning a value of some type. Here's an example: In some validation scenario, assume that our `AddStudent` function has a student of value `null` passed into it as follows:

```
Student noStudent = null;

await this.studentService.AddStudentAsync(noStudent);
```

Our `AddStudentAsync` function in this scenario is now required to validate whether the passed in parameter is `null` or not before going any further with any other type of validations or the business logic itself. Something like this:

```
public ValueTask<Student> AddStudentAsync(Student student) =>
TryCatch(async () =>
{
        ValidateStudentOnAdd(student);

        return await this.storageBroker.InsertStudentAsync(student);
});
```

The method in focus here is `ValidateStudentIsNotNull` which is called by the `ValidateStudentOnAddAsync` validations engine. Here's an example of how that routine would be implemented:

```
private static void ValidateStudentIsNotNull(Student student)
{
    if (student is null)
    {
        throw new NullStudentException(
            message: "The student is null.");
    }
}
```

In the function above, we decided to throw the exception with a message immediately instead of going in further. That's an example of circuit-breaking validation type.

But with validations, circuit-breaking isn't always the wise thing to do. Sometimes we want to collect all the issues within a particular request before sending the error report back to the request submitter. Let's talk about that in this next section.

2.1.3.1.1 Continuous Validations

Continuous validations are the opposite of circuit-breaking validations. They don't stop the flow of validations but they definitely stop the flow of logic. In other words, continuous validations ensure no business logic will be executed but they also

ensure other validations of the same type can continue to execute before breaking the circuit. Let's materialize this theory with an example: Assume our student model looks like this:

```
public class Student
{
        public Guid Id {get; set;}
        public string Name {get; set;}
}
```

Assuming that the passed-in `Student` model is not null, it has default values across the board for all its properties. We want to collect all these issues for however many attributes/properties this object has and return a full report back to the requestor. Here's how to do it.

2.1.3.1.1.0 Upsertable Exceptions

A problem of that type requires a special type of exceptions that allow collecting all errors in it's `Data` property. Every native exception out there will contain the `Data` property which is basically a dictionary for a key/value pairs for collecting more information about the issues that caused that exception to occur. The issue with these native exceptions is that they don't have native support for upsertion. Being able to append to an existing list of values against a particular key at any point of time. Here's a native implementation of upserting values in some given dictionary:

```
var someException = new Exception();

if(someException.Data.Contains(someKey))
{
        (someException.Data[someKey] as List<string>)?.Add(someValue);
}
else
{
        someException.Data.Add(someKey, new List<string>{ someValue });
}
```

This implementation can be quite daunting for engineers to think about and test in their service-level implementation. It felt more appropriate to introduce a simple library `Xeption` to simplify the above implementation into something as simple as:

```
var someException = new Xeption();
someException.UpsertDataList(someKey, someValue);
```

Now that we have this library to utilize, the concern of implementing upsertable exceptions has been addressed. This means that we have what it takes to collect our validation errors. But that's not good enough if we don't have a mechanism to break the circuit when we believe that the time is right to do so. We can simply use the native offerings to implement the circuit-breaking directly as follows:

```
if(someException.Data.Count > 0)
{
        throw someException;
}
```

And while this can be easily baked into any existing implementation. It still didn't contribute much to overall look-n-feel of the code. Therefore I have decided to make it a part of the `Xeption` library to be simplified to the following:

```
someException.ThrowIfContainsErrors();
```

That would make our custom validations look something like this:

```
public class InvalidStudentException : Xeption
{
    public InvalidStudentException(string message)
        : base(message)
    { }
}
```

Every custom exception whether it is localized or categorized should essentially adhere to our anemic model principle. This will enforce a strong binding of the exception messages and the testing messages, mainly ensuring messages communicated through the exceptions are the proper messages.

But with continuous validations, the process of collecting these errors conveys more than just a special exception implementation. We will discuss more on this in the next section.

2.1.3.1.1.1 Dynamic Rules

A non-circuit-breaking or continuous validation process will require the ability to pass in dynamic rules at any count or capacity to add these validation errors. A validation rule is a dynamic structure that reports whether the rule has been

violated for its condition; and also the error message that should be reported to the end user to help them fix that issue.

In a scenario where we want to ensure any given Id is valid, a dynamic continuous validation rule would look something like this:

```
private dynamic IsInvalid(Guid id) => new
{
        Condition = id == Guid.Empty,
        Message = "Id is invalid"
};
```

Now our Rule doesn't just report whether a particular attribute is invalid or not. It also has a meaningful human-readable message that helps the consumer of the service understand what makes that very attribute invalid.

It's really important to point out the language engineers must use for validation messages. It will all depend on the potential consumers of your system. A non-engineer will not understand a message such as `Text cannot be null, empty or whitespace` - `null` as a term isn't something that is very commonly used. Engineers must work closely with their consumer or advocates for the people using the system to ensure the language makes sense to them.

Dynamic rules by design will allow engineers to modify both their inputs and outputs without breaking any existing functionality as long as `null` values are considered across the board. Here's another manifestation of a Dynamic Validation Rule:

```
private static dynamic IsNotSame(
        Guid firstId,
        Guid secondId,
        string secondIdName) => new
{
        Condition = firstId != secondId,
        Message = $"Id is not the same as {secondIdName}.",
        HelpLink = "/help/code1234"
};
```

Our dynamic rule now can offer more input parameters and more helpful information in terms of more detailed exception message with links to helpful documentation sites or references for error codes.

2.1.3.1.1.2 Rules & Validations Collector

Now that we have the advanced exceptions and the dynamic validation rules. It's time to put it all together in terms of accepting infinite number of validation rules, examining their condition results and finally break the circuit when all the continuous validations are done. Here's how to do that:

```
private void Validate(
        string message,
        params (dynamic Rule, string Parameter)[] validations)
{
    var invalidStudentException = new InvalidStudentException(message);

    foreach((dynamic rule, string parameter) in validations)
    {
        if(rule.Condition)
        {
            invalidStudentException.UpsertDataList(key: parameter, value:
rule.Message);
        }
    }

    invalidStudentException.ThrowIfContainsErrors();
}
```

The above function now will take any number of validation rules, and the parameters the rule is running against then examine the conditions and upsert the report of errors. This is how we can use the method above:

```
private static void ValidateStudentOnAdd(Student student)
{
        ......

        Validate(
                message: "Student is invalid. Please fix the errors and try
again.",
                (Rule: IsInvalid(student.Id), Parameter: nameof(Student.Id)),
                (Rule: IsInvalid(student.Name), Parameter: nameof(Student.Name)),
                (Rule: IsInvalid(student.Grade), Parameter: nameof(Student.Grade))
        );
}
```

This simplification of writing the rules and validations is the ultimate goal of continuing to provide value to the end users while making the process of engineering the solution pleasant to the engineers themselves.

Now, let's dive deeper into the types of validations that our systems can offer and how to handle them.

2.1.3.1.1.3 Hybrid Continuous Validations

The structure above allows supporting scenarios for nested objects validations. For instance, let's assume that our `Student` model has more than just primitive types in it's structure as follows:

```
public class Student
{
        public Guid Id {get; set;}
        public string Name {get; set;}
        public StudentAddress Address {get; set;}
}
```

Let's assume that the `StudentAddress` model is a set of primitive type properties that are all or some of them at least are required. Considering the `StudentAddress` looks like this:

```
public class StudentAddress
{
        public string Street {get; set;}
        public string City {get; set;}
        public string ZipCode {get; set;}
}
```

In this case, we can't validate `Address` as a property on the `Student` level and also the `Street`, `City` and `ZipCode` at the same level as their parent/container property. That's because it would cause a `NullReferenceException` error if we did so and it might break the circuit unintentionally beyond what local exceptions can handle.

In this case we would need a hybrid approach as follows:

```
private async ValueTask ValidateStudentOnAddAsync(Student student)
{
        ......

        Validate(
                message: "Student is invalid. Please fix the errors and try
again.",
                (Rule: IsInvalid(student.Id), Parameter: nameof(Student.Id)),
                (Rule: IsInvalid(student.Name), Parameter: nameof(Student.Name)),
                (Rule: IsInvalid(student.Address), Parameter:
```

```
nameof(Student.Address))
        );

        ......

        Validate(
                message: "Student address is invalid. Please fix the errors and try
again.",
                (Rule: IsInvalid(student.Address.Street), Parameter:
nameof(StudentAddress.Street)),
                (Rule: IsInvalid(student.Address.City), Parameter:
nameof(StudentAddress.City)),
                (Rule: IsInvalid(student.Address.ZipCode), Parameter:
nameof(StudentAddress.ZipCode))
        );
}
```

In the code above, each level is handled separately. We would break the circuit once we find out that the first round of validations have failed. But if the `Address` property is in good share we can then continue to the next round of deeper validations at the sub-property level with the `StreetAddress` properties (`ZipCode` , `City` and `Street`).

This scenario happens usually with Orchestration-Level validations for virtual models and possibly with API integrations at the Foundation Services level.

2.1.3.1.2 Structural Validations

Validations are three different layers. the first of these layers is the structural validations to ensure certain properties on any given model or a primitive type are not in an invalid structural state.

For instance, a property of type `String` should not be empty, `null` or white space. Another example would be for an input parameter of an `int` type, it should not be at it's `default` state which is `0` when trying to enter an age for instance.

The structural validations ensure that the data is in a good shape before moving forward with any further validations - for instance, we can't possibly validate students that have the minimum number of characters (which is a logical validation) in their names if their first name is structurally invalid by being `null` , empty or whitespace.

Structural validations play the role of identifying the *required* properties on any given model, and while a lot of technologies offer the validation annotations, plugins or libraries to globally enforce data validation rules, I choose to perform the validation programmatically and manually to gain more control of what would be required and what wouldn't in a TDD fashion.

The issue with some of the current implementations on structural and logical validations on data models is that it can be very easily changed under the radar without any unit tests firing any alarms. Check this example for instance:

```
public Student
{
        [Required]
        public string Name {get; set;}
}
```

The above example can be very enticing at a glance from an engineering standpoint. All you have to do is decorate your model attribute with a magical annotation and then all of the sudden your data is being validated.

The problem here is that this pattern combines two different responsibilities or more together all in the same model. Models are supposed to be just a representation of objects in reality - nothing more and nothing less. Some engineers call them anemic models which focuses the responsibility of every single model to only represent the attributes of the real world object it's trying to simulate without any additional details.

But the annotated models now try to inject business logic into their very definitions. This business logic may or may not be needed across all services, brokers or exposing components that uses them.

Structural validations on models may seem like extra work that can be avoided with magical decorations. But in the case of trying to diverge slightly from these validations into a more customized validations, now you will see a new anti-pattern emerge like custom annotations that may or may not be detectable through unit tests.

Let's talk about how to test a structural validation routine:

2.1.3.1.2.0 Testing Structural Validations

Because I truly believe in the importance of TDD, I am going to start showing the implementation of structural validations by writing a failing test for it first.

Let's assume we have a student model, with the following details:

```
public class Student
{
        public Guid Id {get; set;}
}
```

We want to validate that the student Id is not a structurally invalid Id - such as an empty `Guid` - therefore we would write a unit test in the following fashion:

```
[Fact]
private async void
ShouldThrowValidationExceptionOnAddWhenIdIsInvalidAndLogItAsync()
{
        // given
        Student randomStudent = CreateRandomStudent();
        Student inputStudent = randomStudent;
        inputStudent.Id = Guid.Empty;

        var invalidStudentException =
                new InvalidStudentException(
                        message: "Student is invalid. Please fix the errors and try
again.");

        invalidStudentException.AddData(
                key: nameof(Student.Id),
                value: "Id is required"
        );

        var expectedStudentValidationException =
                new StudentValidationException(
                        message: "Student validation error occurred, fix errors and
try again.",
                        innerException: invalidStudentException);

        // when
        ValueTask<Student> addStudentTask =
                this.studentService.AddStudentAsync(inputStudent);

        StudentValidationException actualStudentValidationException =
                await Assert.ThrowsAsync<StudentValidationException>(
                        addStudentTask.AsTask);

        // then
        actualStudentValidationException.Should().BeEquivalentTo(
```

```
                expectedStudentValidationException);

        this.loggingBrokerMock.Verify(broker =>
                broker.LogError(It.Is(SameExceptionAs(
                        expectedStudentValidationException))),
                                Times.Once);

        this.storageBrokerMock.Verify(broker =>
                broker.InsertStudentAsync(It.IsAny<Student>()),
                        Times.Never);

        this.loggingBrokerMock.VerifyNoOtherCalls();
        this.storageBrokerMock.VerifyNoOtherCalls();
        this.dateTimeBrokerMock.VerifyNoOtherCalls();
}
```

In the above test, we created a random student object then assigned an invalid Id value of `Guid.Empty` to the student `Id`.

When the structural validation logic in our foundation service examines the `Id` property, it should throw an exception property describing the issue of validation in our student model. In this case we throw `InvalidStudentException`.

The exception is required to briefly describe the whats, wheres and whys of the validation operation. In our case here the what would be the validation issue occurring, the where would be the Student service and the why would be the property value.

Here's how an `InvalidStudentException` would look like:

```
public class InvalidStudentException : Xeption
{
    public InvalidStudentException(string message)
        : base(message)
    { }
}
```

The above unit test however, requires our `InvalidStudentException` to be wrapped up in a more generic system-level exception, which is `StudentValidationException` - these exceptions are what I call outer-exceptions, they encapsulate all the different situations of validations regardless of their category and communicates the error to upstream services or controllers so they can map that to the proper error code to the consumer of these services.

Our `StudentValidationException` would be implemented as follows:

```
public class StudentValidationException : Xeption
{
    public StudentValidationException(string message, Xeption innerException)
        : base(message, innerException)
    { }
}
```

The string messaging for the outer-validation above will be passed when the exception is initialized from the service class as shown below.

```
private async ValueTask<StudentValidationException>
CreateAndLogValidationExceptionAsync(Xeption exception)
{
    var studentValidationException = new StudentValidationException(
        message: "Student validation error occurred, please check your input and
try again.",
        innerException: exception);

    await this.loggingBroker.LogErrorAsync(studentValidationException);

    return studentValidationException;
}
```

The message in this outer-validation indicates that the issue is in the input, and therefore it requires the input submitter to try again as there are no actions required from the system side to be adjusted.

2.1.3.1.2.1 Implementing Structural Validations

Now, let's look at the other side of the validation process, which is the implementation. Structural validations always come before each and every other type of validations. That's simply because structural validations are the cheapest from an execution and asymptotic time perspective. For instance, It's much cheaper to validation an `Id` is invalid structurally, than sending an API call across to get the exact same answer plus the cost of latency. This all adds up when multi-million requests per second start flowing in. Structural and logical validations in general live in their own partial class to run these validations, for instance, if our service is called `StudentService.cs` then a new file should be created with the name `StudentService.Validations.cs` to encapsulate and visually abstract away the validation rules to ensure clean data are coming in and going out. Here's how an Id validation would look like:

StudentService.Validations.cs

```
private static void ValidateStudentOnAdd(Student student)
{
        ......

    Validate(
                message: "Student is invalid. Please fix the errors and try
again.",
                (Rule: IsInvalid(student.Id), Parameter: nameof(Student.Id)));
}

private static dynamic IsInvalid(Guid id) => new
{
    Condition = id == Guid.Empty,
    Message = "Id is invalid"
};

private void Validate(string message, params (dynamic Rule, string Parameter)[]
validations)
{
    var invalidStudentException = new InvalidStudentException(message);

    foreach((dynamic rule, string parameter) in validations)
    {
        if(rule.Condition)
        {
            invalidStudentException.UpsertDataList(key: parameter, value:
rule.Message);
        }
    }

    invalidStudentException.ThrowIfContainsErrors();
}
```

We have implemented a method to validate the entire student object, with a compilation of all the rules we need to setup to validate structurally and logically the student input object. The most important part to notice about the above code snippet is to ensure the encapsulation of any finer details further away from the main goal of a particular method.

That's the reason why we decided to implement a private static method `IsInvalid` to abstract away the details of what determines a property of type `Guid` is invalid or not. And as we move further in the implementation, we are going to have to implement multiple overloads of the same method to validate other value types structurally and logically.

The purpose of the `ValidateStudent` method is to simply set up the rules and take an action by throwing an exception if any of these rules are violated. There's always an opportunity to aggregate the violation errors rather than throwing too early at the first sign of structural or logical validation issue to be detected.

Now, with the implementation above, we need to call that method to structurally and logically validate our input. Let's make that call in our `AddStudentAsync` method as follows:

StudentService.cs

```
public ValueTask<Student> AddStudentAsync(Student student) =>
TryCatch(async () =>
{
        ValidateStudentOnAdd(student);

        return await this.storageBroker.InsertStudentAsync(student);
});
```

At a glance, you will notice that our method here doesn't necessarily handle any type of exceptions at the logic level. That's because all the exception noise is also abstracted away in a method called `TryCatch`.

`TryCatch` is a concept I invented to allow engineers to focus on what matters the most based on which aspect of the service they are looking at without having to take any shortcuts with the exception handling to make the code a bit more readable.

`TryCatch` methods in general live in another partial class, and an entirely new file called `StudentService.Exceptions.cs` - which is where all exception handling and error reporting happens as I will show you in the following example.

Let's take a look at what a `TryCatch` method looks like:

StudentService.Exceptions.cs

```
private delegate ValueTask<Student> ReturningStudentFunction();

private async ValueTask<Student> TryCatch(ReturningStudentFunction
returningStudentFunction)
{
        try
        {
                return await returningStudentFunction();
```

```
        }
        catch (InvalidStudentException invalidStudentInputException)
        {
                throw await
CreateAndLogValidationExceptionAsync(invalidStudentInputException);
        }
}

private async ValueTask<StudentValidationException>
CreateAndLogValidationExceptionAsync(Xeption exception)
{
    var studentValidationException = new StudentValidationException(
        message: "Student validation error occurred, please check your input and
try again.",
        innerException: exception);

    this.loggingBroker.LogErrorAsync(studentValidationException);

    return studentValidationException;
}
```

The `TryCatch` exception noise-cancellation pattern beautifully takes in any function that returns a particular type as a delegate and handles any thrown exceptions off of that function or it's dependencies.

The main responsibility of a `TryCatch` function is to wrap up a service inner exceptions with outer exceptions to ease-up the reaction of external consumers of that service into only one of the three categories, which are Service Exceptions, Validations Exceptions and Dependency Exceptions. There are sub-types to these exceptions such as Dependency Validation Exceptions but these usually fall under the Validation Exception category as we will discuss in upcoming sections of The Standard.

In a `TryCatch` method, we can add as many inner and external exceptions as we want and map them into local exceptions for upstream services not to have a strong dependency on any particular libraries or external resource models, which we will talk about in detail when we move on to the Mapping responsibility of broker-neighboring (foundation) services.

2.1.3.1.3 Logical Validations

Logical validations are the second in order to structural validations. Their main responsibility by definition is to logically validate whether a structurally valid piece of data is logically valid. For instance, a date of birth for a student could be structurally

valid by having a value of `1/1/1800` but logically, a student that is over 200 years of age is an impossibility.

The most common logical validations are validations for audit fields such as `CreatedBy`, `UpdatedBy`, `CreatedDate`, and `UpdatedDate` - it's logically impossible a new record can be inserted with two different values for the authors of that new record - simply because data can only be inserted by one person at a time. The same goes for the `CreatedDate` and `UpdatedDate` fields - it's logically impossible for a record to be created and updated at the same exact times.

Let's talk about how we can test-drive and implement logical validations:

2.1.3.1.3.0 Testing Logical Validations

In the common case of testing logical validations for audit fields, we want to throw a validation exception that the `UpdatedBy` value is invalid simply because it doesn't match the `CreatedBy` field.

Let's assume our Student model looks as follows:

```
public class Student
{
        String CreatedBy {get; set;}
        String UpdatedBy {get; set;}
        DateTimeOffset CreatedDate {get; set;}
        DateTimeOffset UpdatedDate {get; set;}
}
```

Our two tests to validate these values logically would be as follows:

In this first test example, we would ensure that the `UpdatedBy` field is not the same as the `CreatedBy` field.

```
[Fact]
private async Task
ShouldThrowValidationExceptionOnAddIfAuditPropertiesIsNotTheSameAndLogItAsync()
{
        // given
        DateTimeOffset randomDateTime = GetRandomDateTimeOffset();
        DateTimeOffset now = randomDateTime;
        Student randomStudent = CreateRandomStudent(now);
        Student invalidStudent = randomStudent;
        invalidStudent.CreatedBy = GetRandomString();
        invalidStudent.UpdatedBy = GetRandomString();
        invalidStudent.CreatedDate = now;
```

```
        invalidStudent.UpdatedDate = GetRandomDateTimeOffset();

        var invalidStudentException =
                new InvalidStudentException(
                        message: "Student is invalid. Please fix the errors and try
again.");

        invalidStudentException.AddData(
                key: nameof(Student.UpdatedBy),
                value: $"Text is not the same as {nameof(Student.CreatedBy)}.");

        invalidStudentException.AddData(
                key: nameof(Student.UpdatedDate),
                value: $"Date is not the same as {nameof(Student.CreatedDate)}.");

        var expectedStudentValidationException =
                new StudentValidationException(
                        message: "Student validation error occurred, fix errors and
try again.",
                        innerException: invalidStudentException);

        this.dateTimeBrokerMock.Setup(broker =>
        broker.GetCurrentDateTimeOffsetAsync())
            .ReturnsAsync(now);

        // when
        ValueTask<Student> addStudentTask =
                this.studentService.AddStudentAsync(inputStudent);

        StudentValidationException actualStudentValidationException =
                await Assert.ThrowsAsync<StudentValidationException>(
                        addStudentTask.AsTask);

        // then
        actualStudentValidationException.Should().BeEquivalentTo(
                expectedStudentValidationException);

        this.dateTimeBrokerMock.Verify(broker =>
        broker.GetCurrentDateTimeOffsetAsync(),
            Times.Once);

        this.loggingBrokerMock.Verify(broker =>
                broker.LogError(It.Is(SameExceptionAs(
                        expectedStudentValidationException))),
                                Times.Once);

        this.storageBrokerMock.Verify(broker =>
                broker.InsertStudentAsync(It.IsAny<Student>()),
                        Times.Never);

        this.dateTimeBrokerMock.VerifyNoOtherCalls();
        this.loggingBrokerMock.VerifyNoOtherCalls();
```

```
        this.storageBrokerMock.VerifyNoOtherCalls();
}
```

In the above test, we have changed the value of the `UpdatedBy` field to ensure it completely differs from the `CreatedBy` field - now we expect an `InvalidStudentException` with the `CreatedBy` to be the reason for this validation exception to occur.

```
[Theory]
[InlineData(1)]
[InlineData(-61)]
public async Task
ShouldThrowValidationExceptionOnAddIfCreatedDateIsNotRecentAndLogItAsync(
    int invalidSeconds)
{
    // given
    DateTimeOffset randomDateTime =
        GetRandomDateTimeOffset();

    DateTimeOffset now = randomDateTime;
    Student randomStudent = CreateRandomStudent();
    Student invalidStudent = randomStudent;

    DateTimeOffset invalidDate =
        now.AddSeconds(invalidSeconds);

    invalidStudent.CreatedDate = invalidDate;
    invalidStudent.UpdatedDate = invalidDate;

    var invalidStudentException = new InvalidStudentException(
        message: "Student is invalid, fix the errors and try again.");

    invalidStudentException.AddData(
        key: nameof(Student.CreatedDate),
        values: $"Date is not recent");

    var expectedStudentValidationException =
        new StudentValidationException(
            message: "Student validation error occurred, fix errors and try
again.",
            innerException: invalidStudentException);

    this.dateTimeBrokerMock.Setup(broker =>
        broker.GetCurrentDateTimeOffsetAsync())
            .ReturnsAsync(now);

    // when
    ValueTask<Student> addStudentTask =
        this.studentService.AddStudentAsync(invalidSource);
```

```
    StudentValidationException actualStudentValidationException =
        await Assert.ThrowsAsync<StudentValidationException>(
            addStudentTask.AsTask);

    // then
    actualStudentValidationException.Should().BeEquivalentTo(
        expectedStudentValidationException);

    this.dateTimeBrokerMock.Verify(broker =>
        broker.GetCurrentDateTimeOffsetAsync(),
            Times.Once);

    this.loggingBrokerMock.Verify(broker =>
        broker.LogErrorAsync(It.Is(
            SameExceptionAs(expectedSourceValidationException))),
                Times.Once);

    this.storageBrokerMock.Verify(broker =>
        broker.InsertSourceAsync(It.IsAny<Source>()),
            Times.Never);

    this.dateTimeBrokerMock.VerifyNoOtherCalls();
    this.loggingBrokerMock.VerifyNoOtherCalls();
    this.storageBrokerMock.VerifyNoOtherCalls();
}
```

In the test example want to ensure that the `CreatedDate` is a date current to 60 seconds before or after. We would not want to allow dates that are not recent on an `AddStudent` operation. We then expect an `InvalidStudentException` with the `CreatedDate` to be the reason for this validation exception to occur.

Let's go ahead an write the implementations for these failing tests.

2.1.3.1.3.1 Implementing Logical Validations

Just like we did in the structural validations section, we are going to validate our logical rule(s) as follows:

StudentService.Validations.cs

```
 private async ValueTask ValidateStudentOnAddAsync(Student student)
{
    ValidateStudentIsNotNull(student);

    Validate(
        message: "Student is invalid. Please fix the errors and try again.",
        (Rule: IsInvalid(student.Id), Parameter: nameof(Student.Id)),
        (Rule: IsInvalid(student.Name), Parameter: nameof(Student.Name)),
```

```
        (Rule: IsInvalid(student.CreatedBy), Parameter: nameof(Student.CreatedBy)),
        (Rule: IsInvalid(student.UpdatedBy), Parameter: nameof(Student.UpdatedBy)),
        (Rule: IsInvalid(student.CreatedDate), Parameter:
nameof(Student.CreatedDate)),
        (Rule: IsInvalid(student.UpdatedDate), Parameter:
nameof(Student.UpdatedDate)),

        (Rule: IsNotSame(
            createdBy: student.UpdatedBy,
            updatedBy: student.CreatedBy,
            createdByName: nameof(Student.CreatedBy)),

        Parameter: nameof(Student.UpdatedBy)),

        (Rule: IsDatesNotSame(
            createdDate: student.CreatedDate,
            updatedDate: studente.UpdatedDate,
            nameof(Studente.CreatedDate)),

        Parameter: nameof(Student.UpdatedDate)),

        (Rule: await IsNotRecentAsync(student.CreatedDate), Parameter:
nameof(Student.CreatedDate)));
}
```

In the above implementation, we have implemented our rule validation engine method to validate the student object for the OnAdd operation, with a compilation of all the rules we need to setup to validate structurally and logically the student input object. We then call the logical validation methods `IsInvalid`, `IsValuesNotSame`, `IsDatesNotSame` and `IsNotRecent` to ensure our conditional requirements are met. Here are the example implementations for these methods:

For `IsInvalid`

```
private dynamic IsInvalid(Guid id) => new
{
    Condition = id == Guid.Empty,
    Message = "Id is invalid"
};

private dynamic IsInvalid(string name) => new
{
    Condition = String.IsNullOrWhiteSpace(name),
    Message = "Text is required"
};

private dynamic IsInvalid(DateTimeOffset date) => new
{
    Condition = date == default,
```

```
    Message = "Date is invalid"
};
```

For `IsValuesNotSame`

```
private dynamic IsValuesNotSame(
        string createdBy,
        string updatedBy,
        string createdByName) => new
        {
                Condition = createdBy != updatedBy,
                Message = $"Text is not the same as {createdByName}"
        };
```

For `IsDatesNotSame`

```
private dynamic IsDatesNotSame(
        DateTimeOffset createdDate,
        DateTimeOffset updatedDate,
        string createdDateName) => new
        {
                Condition = createdDate != updatedDate,
                Message = $"Date is not the same as {createdDateName}"
        };
```

For `IsNotRecentAsync`

```
private async ValueTask<dynamic> IsNotRecentAsync(DateTimeOffset date)
{
        var (isNotRecent, startDate, endDate) = await IsDateNotRecentAsync(date);

                return new
                {
                        Condition = isNotRecent,
                        Message = $"Date is not recent. Expected a value between
{startDate} and {endDate} but found {date}"
                };
};

private async ValueTask<(bool IsNotRecent, DateTimeOffset StartDate, DateTimeOffset
EndDate)>
        IsDateNotRecentAsync(DateTimeOffset date)
{
        int pastSeconds = 60;
        int futureSeconds = 0;

        DateTimeOffset currentDateTime =
                await this.dateTimeBroker.GetCurrentDateTimeOffsetAsync();

         if (currentDateTime == default)
```

```
        {
                return (false, default, default);
        }

        TimeSpan timeDifference = currentDateTime.Subtract(date);
        DateTimeOffset startDate = currentDateTime.AddSeconds(-pastSeconds);
        DateTimeOffset endDate = currentDateTime.AddSeconds(futureSeconds);
        bool isNotRecent = timeDifference.TotalSeconds is > 60 or < 0;

        return (isNotRecent, startDate, endDate);
}
```

Finally our `Validate` method would look like this:

```
private static void Validate(string message, params (dynamic Rule, string
Parameter)[] validations)
{
        var invalidStudentException = new InvalidStudentException(message);

        foreach((dynamic rule, string parameter) in validations)
        {
                if(rule.Condition)
                {
                        invalidStudentException.UpsertData(
                                key: parameter,
                                value: rule.Message);
                }
        }

        invalidStudentException.ThrowIfContainsErrors();
}
```

Everything else in both `StudentService.cs` and `StudentService.Exceptions.cs` continues to be exactly the same as we've done above in the structural validations.

Logical validations exceptions, just like any other exceptions that may occur are usually non-critical. However, it all depends on your business case to determine whether a particular logical, structural or even a dependency validation are critical or not, this is when you might need to create a special class of exceptions, something like `InvalidStudentCriticalException` then log it accordingly.

2.1.3.1.4 External Validations

The last type of validations that are usually performed by foundation services is external validations. I define external validations as any form of validation that requires calling an external resource to validate whether a foundation service should proceed with processing incoming data or halt with an exception.

A good example of dependency validations is when we call a broker to retrieve a particular entity by it's id. If the entity returned is not found, or the API broker returns a `NotFound` error - the foundation service is then required to wrap that error in a `ValidationException` and halts all following processes.

External validation exceptions can occur if the returned value did not match the expectation, such as an empty list returned from an API call when trying to insert a new coach of a team - if there is no team members, there can be no coach for instance. The foundation service in this case will be required to raise a local exception to explain the issue, something like `NoTeamMembersFoundException` or something of that nature.

Let's write a failing test for an external validation example:

2.1.3.1.4.0 Testing External Validations

Let's assume we are trying to retrieve a student with an `Id` that doesn't match any records in the database. Here's how we would go about testing this scenario. First off, let's define a `NotFoundStudentException` model as follows:

```
using Xeption;

public class NotFoundStudentException : Xeption
{
        public NotFoundStudentException(string message)
                : base (message) {}
}
```

The above model is the localization aspect of handling the issue. Now let's write a failing test as follows:

```
private async Task
ShouldThrowValidationExceptionOnRetrieveByIdIfStudentNotFoundAndLogItAsync()
{
        // given
        Guid someStudentId = Guid.NewGuid();
        Student nullStudent = null;
        var innerException = new Exception()

        var notFoundStudentException =
                new NotFoundStudentException(
                        message: $"Student not found with the id: {someStudentId}",
                        innerException: innerException.innerException.As<Xeption>
());
```

```
        var expectedStudentValidationException =
                new StudentValidationException(
                        message: "Student validation error occurred, fix errors and
try again.",
                        innerException: notFoundStudentException);

        this.storageBrokerMock.Setup(broker =>
                broker.SelectStudentByIdAsync(someStudentId))
                        .ReturnsAsync(nullStudent);

        // when
        ValueTask<Student> retrieveStudentByIdTask =
                this.studentService.RetrieveStudentByIdAsync(someStudentId);

        StudentValidationException actualStudentValidationException =
                await Assert.ThrowsAsync<StudentValidationException>(
                        retrieveStudentByIdTask.AsTask);

        // then
        actualStudentValidationException.Should().BeEquivalentTo(
                expectedStudentValidationException);

        this.storageBrokerMock.Verify(broker =>
                broker.SelectStudentByIdAsync(someStudentId),
                        Times.Once);

        this.loggingBrokerMock.Verify(broker =>
                broker.LogError(It.Is(SameExceptionAs(
                        expectedStudentValidationException))),
                                Times.Once);

        this.storageBrokerMock.VerifyNoOtherCalls();
        this.loggingBrokerMock.VerifyNoOtherCalls();
        this.dateTimeBrokerMock.VerifyNotOtherCalls();
}
```

The test above requires us to throw a localized exception as in `NotFoundStudentException` when the storage broker returns no values for the given `studentId` and then wrap or categorize this up in `StudentValidationException`.

We choose to wrap the localized exception in a validation exception and not in a dependency validation exception because the initiation of the error originated from our service not from the external resource. If the external resource is the source of the error we would have to categorize this as a `DependencyValidationException` which we will discuss shortly.

Now let's get to the implementation part of this section to make our test pass.

2.1.3.1.4.1 Implementing External Validations

In order to implement an external validation we will need to touch on all different aspects of our service. The core logic, the validation and the exception handling aspects are as follows.

First off, let's build a validation function that will throw a `NotFoundStudentException` if the passed-in parameter is null.

StudentService.Validations.cs

```
private static void VerifyStudentExists(Student maybeStudent, Guid studentId)
{
    if (maybeStudent is null)
    {
        throw new NotFoundStudentException(
            message: $"Student not found with id: {studentId}.");
    }
}
```

This implementation will take care of detecting an issue and issuing a local exception `NotFoundStudentException`. Now let's jump over to the exception handling aspect of our service.

StudentService.Exceptions.cs

```
private async ValueTask<Student> TryCatch(ReturningStudentFunction
returningStudentFunction)
{
    try
    {
        return await returningStudentFunction();
    }
    ..
    catch (NotFoundStudentException notFoundStudentException)
    {
        throw await CreateAndLogValidationExceptionAsync(notFoundStudentException);
    }
}

private async ValueTask<StudentValidationException>
CreateAndLogValidationExceptionAsync(Xeption exception)
{
    var studentValidationException = new StudentValidationException(
        message: "Student validation error occurred, fix errors and try again.",
        innerException: exception);
```

```
    await this.loggingBroker.LogErrorAsync(studentValidationException);

    return studentValidationException;
}
```

The above implementation will take care of categorizing a `NotFoundStudentException` to `StudentValidationException`. The last part is to put the pieces together as follows.

StudentService.cs

```
public ValueTask<Student> RetrieveStudentByIdAsync(Guid studentId) =>
TryCatch(async () =>
{
        ValidateStudentId(studentId);

        Student maybeStudent =
                await this.storageBroker.SelectStudentByIdAsync(studentId);

        ValidateStudentExists(maybeStudent, studentId);

        return maybeStudent;
});
```

The above implementation will ensure that the id is valid, but more importantly that whatever the `storageBroker` returns will be checked for whether it's an object or `null`. Then issue the exception.

There are situations where attempting to retrieve an entity then finding out that it doesn't exist is not necessarily erroneous. This is where Processing Services come in to leverage a higher-order business logic to deal with this more complex scenario.

2.1.3.1.5 Dependency Validations

Dependency validation exceptions can occur because you called an external resource and it returned an error, or returned a value that warrants raising an error. For instance, an API call might return a `404` code, and that's interpreted as an exception if the input was supposed to correspond to an existing object.

A more common example is when a particular input entity is using the same id as an existing entity in the system. In a relational database world, a duplicate key exception would be thrown. In a RESTful API scneario, programmatically applying

the same concept also achieves the same goal for API validations assuming the granularity of the system being called weaken the referential integrity of the overall system data.

There are situations where the faulty response can be expressed in a fashion other than exceptions, but we shall touch on that topic in a more advanced chapters of this Standard.

Let's write a failing test to verify whether we are throwing a `DependencyValidationException` if `Student` model already exists in the storage with the storage broker throwing a `DuplicateKeyException` as a native result of the operation.

2.1.3.1.5.0 Testing Dependency Validations

Let's assume our student model uses an `Id` with the type `Guid` as follows:

```
public class Student
{
        public Guid Id {get; set;}
        public string Name {get; set;}
}
```

our unit test to validate a `DependencyValidation` exception would be thrown in a `DuplicateKey` situation would be as follows:

```
[Fact]
private async Task
ShouldThrowDependencyValidationExceptionOnAddIfStudentAlreadyExistsAndLogItAsync()
{
        // given
        Student someStudent = CreateRandomStudent();

        var duplicateKeyException =
                new DuplicateKeyException(
                        message: "Duplicate key error occurred");

        var alreadyExistsStudentException =
                new AlreadyExistsStudentException(
                        message: "Student already exists occurred.",
                        innerException: duplicateKeyException,
                        data: duplicateKeyException);

        var expectedStudentDependencyValidationException =
                new StudentDependencyValidationException(
                        message: "Student dependency validation error occurred, try
```

```
again.",
                        innerException: alreadyExistsStudentException);

        this.dateTimeBroker.Setup(broker =>
                broker.GetDateTimeOffsetAsync())
                        .ThrowsAsync(duplicateKeyException);

        // when
        ValueTask<Student> addStudentTask =
                this.studentService.AddStudentAsync(someStudent);

        StudentDependencyValidationException
actualStudentDependencyValidationException =
                await Assert.ThrowsAsync<StudentDependencyValidationException>(
                        testcode: addStudentTask.AsTask);

        // then
        actualStudentDependencyValidationException.Should().BeEquivalentTo(
                expectedStudentDependencyValidationException);

        this.dateTimeBroker.Verify(broker =>
                broker.GetDateTimeOffsetAsync(),
                        Times.Once);

        this.loggingBrokerMock.Verify(broker =>
                broker.LogErrorAsync(It.Is(SameExceptionAs(
                        expectedStudentDependencyValidationException))),
                                Times.Once);

        this.storageBrokerMock.Verify(broker =>
                broker.InsertStudentAsync(It.IsAny<Student>()),
                        Times.Never);

        this.dateTimeBroker.VerifyNoOtherCalls();
        this.loggingBrokerMock.VerifyNoOtherCalls();
        this.storageBrokerMock.VerifyNoOtherCalls();
}
```

In the above test, we validate that we wrap a native `DuplicateKeyException` in a local model tailored to the specific model case which is the `AlreadyExistsStudentException` in our example here. then we wrap that again with a generic category exception model which is the `StudentDependencyValidationException`.

There are a couple of rules that govern the construction of dependency validations, which are as follows:

- Rule 1: If a dependency validation is handling another dependency validation from a downstream service, then the inner exception of the downstream exception should be the same for the dependency validation at the current level.

In other words, if some `StudentService` is throwing a `StudentDependencyValidationException` to an upstream service such as `StudentProcessingService` - then it's important that the `StudentProcessingDependencyValidationException` contain the same inner exception as the `StudentDependencyValidationException`. That's because once these exception are mapped into exposure components, such as API controller or UI components, the original validation message needs to propagate through the system and be presented to the end user no matter where it originated from.

Additionally, maintaining the original inner exception guarantees the ability to communicate different error messages through API endpoints. For instance, `AlreadyExistsStudentException` can be communicated as `Conflict` or `409` on an API controller level - this differs from another dependency validation exception such as `InvalidStudentReferenceException` which would be communicated as `FailedDependency` error or `424`.

- Rule 2: If a dependency validation exception is handling a non-dependency validation exception it should take that exception as it's inner exception and not anything else.

These rules ensure that only the local validation exception is what's being propagated not it's native exception from a storage system or an API or any other external dependency.

Which is the case that we have here with our `AlreadyExistsStudentException` and it's `StudentDependencyValidationException` - the native exception is completely hidden away from sight, and the mapping of that native exception and it's inner message is what's being communicated to the end user. This gives the engineers the power to control what's being communicated from the other end of their system instead of letting the native message (which is subject to change) propagate to the end-users.

2.1.3.0.5.1 Implementing Dependency Validations

Depending on where the validation error originates from, the implementation of dependency validations may or may not contain any business logic. As we previously mentioned, if the error is originating from the external resource (which is the case here) - the thrown exception carries with it a Data property and we will propagate this data upstream, then all we have to do is just wrap that error in a local exception then categorize it with an external exception under dependency validation.

To ensure the aforementioned test passed, we are going to need some exception models.

For the `AlreadyExistsStudentException` the implementation would be as follows:

We also need to bring the innerException Data property to the local exception model to ensure the original message is propagated through the system.(as previously mentioned)

```
public class AlreadyExistsStudentException : Xeption
{
    public AlreadyExistsStudentException(
                string message,
                Exception innerException,
                IDictionary data)
        : base (message, innerException, data)
        { }
}
```

We also need the `StudentDependencyValidationException` which should be as follows:

```
public class StudentDependencyValidationException : Xeption
{
    public StudentDependencyValidationException(string message, Xeption
innerException)
        : base (message, innerException) { }
}
```

Now, let's go to the implementation side, let's start with the exception handling logic:

StudentService.Exceptions.cs

```
private delegate ValueTask<Student> ReturningStudentFunction();

private async ValueTask<Student> TryCatch(ReturningStudentFunction
returningStudentFunction)
{
    try
    {
        return await returningStudentFunction();
    }
    ...
    catch (DuplicateKeyException duplicateKeyException)
    {
        var alreadyExistsStudentException =
            new AlreadyExistsStudentException(
                message: "Student already exists occurred.",
                innerException: duplicateKeyException,
                                data: duplicateKeyException.Data);

        throw await
CreateAndLogDependencyValidationExceptionAsync(alreadyExistsStudentException);
    }
}

...

private async ValueTask<StudentDependencyValidationException>
CreateAndLogDependencyValidationExceptionAsync(
        Xeption exception)
{
    var studentDependencyValidationException =
        new StudentDependencyValidationException(
            message: "Student dependency validation error occurred, please try
again.",
            innerException: exception);

    await this.loggingBroker.LogErrorAsync(studentDependencyValidationException);

    return studentDependencyValidationException;
}
```

We created the local inner exception in the catch block of our exception handling process to allow the reusability of our dependency validation exception method for other situations that require that same level of external exceptions.

Everything else stays the same for the referencing of the `TryCatch` method in the `StudentService.cs` file.

2.1.3.2 Mapping

The second responsibility for a foundation service is to play the role of a mapper both ways between local models and non-local models. For instance, if you are leveraging an email service that provides it's own SDKs to integrate with, and your brokers are already wrapping and exposing the APIs for that service, your foundation service is required to map the inputs and outputs of the broker methods into local models. The same situation and more commonly between native non-local exceptions such as the ones we mentioned above with the dependency validation situation, the same aspect applies to just dependency errors or service errors as we will discuss shortly.

2.1.3.2.0 Non-Local Models

Its very common for modern applications to require integration at some point with external services. These services can be local to the overall architecture or distributed system where the application lives, or it can be a 3rd party provider such as some of the popular email services for instance. External services providers invest a lot of effort in developing fluent APIs, SDKs and libraries in every common programming language to make it easy for the engineers to integrate their applications with that 3rd party service. For instance, let's assume a third party email service provider is offering the following API through their SDKs:

```
public interface IEmailServiceProvider
{
        ValueTask<EmailMessage> SendEmailAsync(EmailMessage message);
}
```

Let's consider the model `EmailMessage` is a native model, it comes with the email service provider SDK. your brokers might offer a wrapper around this API by building a contract to abstract away the *functionality* but can't do much with the native models that are passed in or returned out of these functionality. therefore our brokers interface would look something like this:

```
public interface IEmailBroker
{
        ValueTask<EmailMessage> SendEmailMessageAsync(EmailMessage message);
}
```

Then the implementation would something like this:

```
public class EmailBroker : IEmailBroker
{
        public async ValueTask<EmailMessage> SendEmailMessageAsync(EmailMessage
message) =>
                await this.emailServiceProvider.SendEmailAsync(message);
}
```

As we said before, the brokers here have done their part of abstraction by pushing away the actual implementation and the dependencies of the native `EmailServiceProvider` away from our foundation serviecs. But that's only 50% of the job, the abstraction isn't quite fully complete yet until there are no tracks of the native `EmailMessage` model. This is where the foundation services come in to do a test-driven operation of mapping between the native non-local models and your application's local models. therefore its very possible to see a mapping function in a foundation service to abstract away the native model from the rest of your business layer services.

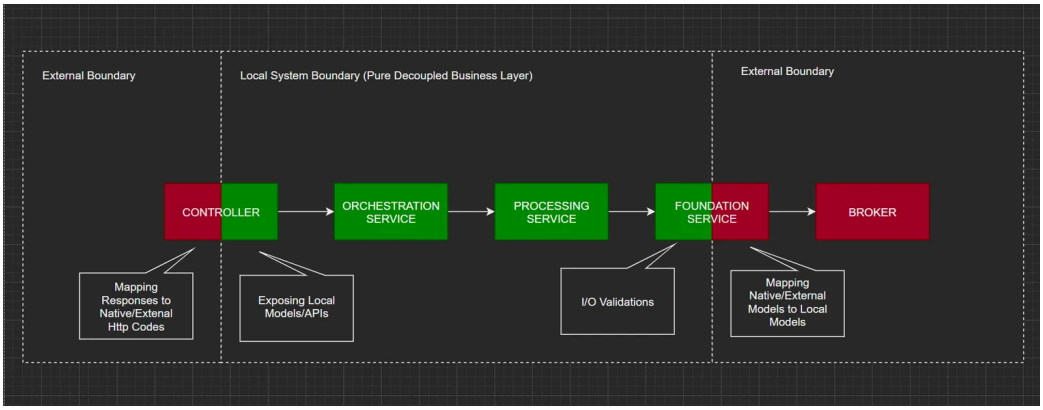
Your foundation service then will be required to support a new local model, let's call it `Email`. Your local model's property may be identical to the external model `EmailMessage` - especially on a primitive data type level. But the new model would be the one and only contract between your pure business logic layer (processing, orchestration, coordination and management services) and your hybrid logic layer like the foundation services. Here's a code snippet for this operation:

```
public async ValueTask<Email> SendEmailMessageAsync(Email email)
{
        EmailMessage inputEmailMessage = MapToEmailMessage(email);
        EmailMessage sentEmailMessage = await
this.emailBroker.SendEmailMessageAsync(inputEmailMessage);

        return MapToEmail(sentEmailMessage);
}
```

Depending on whether the returned message has a status or you would like to return the input message as a sign of a successful operation, both practices are valid in my Standard. It all depends on what makes more sense to the operation you are trying to execute. The code snippet above is an ideal scenario where your code will try to stay true to the value passed in as well as the value returned with all the necessary mapping included.

2.1.3.2.1 Exceptions Mappings

Just like the non-local models, exceptions that are either produced by the external API like the EntityFramework models `DbUpdateException` or any other has to be mapped into local exception models. Handling these non-local exceptions that early before entering the pure-business layer components will prevent any potential tight coupling or dependency on any external model. As it may be very common, that exceptions can be handled differently based on the type of exception and how we want to deal with it internally in the system. For instance, if we are trying to handle a `UserNotFoundException` being thrown from using Microsoft Graph for instance, we might not necessarily want to exit the entire procedure. we might want to continue by adding a user in some other storage for future Graph submittal processing. External APIs should not influence whether your internal operation should halt or not; therefore, handling exceptions on the Foundation layer is the guarantee that this influence is limited within the borders of our external resources handling area of our application and has no impact whatsoever on our core business processes. The following illustration should draw the picture a bit clearer from that perspective:



Here's some common scenarios for mapping native or inner local exceptions to outer exceptions:

Exception	Wrap Inner Exception With	Wrap With	Log Level
NullStudentException	-	StudentValidationException	Error

Exception	Wrap Inner Exception With	Wrap With	Log Level
InvalidStudentException	-	StudentValidationException	Error
SqlException	FailedStudentStorageException	StudentDependencyException	Critical
HttpResponseUrlNotFoundException	FailedStudentApiException	StudentDependencyException	Critical
HttpResponseUnauthorizedException	FailedStudentApiException	StudentDependencyException	Critical
NotFoundStudentException	-	StudentValidationException	Error
HttpResponseNotFoundException	NotFoundStudentException	StudentDependencyValidationException	Error
DuplicateKeyException	AlreadyExistsStudentException	StudentDependencyValidationException	Error
HttpResponseConflictException	AlreadyExistsStudentException	StudentDependencyValidationException	Error
ForeignKeyConstraintConflictException	InvalidStudentReferenceException	StudentDependencyValidationException	Error
DbUpdateConcurrencyException	LockedStudentException	StudentDependencyValidationException	Error
DbUpdateException	FailedStudentStorageException	StudentDependencyException	Error
HttpResponseException	FailedStudentApiException	StudentDependencyException	Error
Exception	FailedStudentServiceException	StudentServiceException	Error

[*] Standardizing Validations & Exceptions

[*] Test-Driving Non-Circuit-Breaking Validations

2.1.4 Self-Sufficient (Dead-End) Services

2.1.4.0 Introduction

A self-sufficient service, also referred to as a dead-end service, is a service whose execution flow does not continue into any external dependency.

These services:

- do not call brokers
- do not call other services
- do not orchestrate external workflows
- contain only internal business or computational logic

The flow effectively ‘ends’ within the service itself, which is why the term dead-end service is sometimes used.

2.1.4.1 Characteristics

A self-sufficient service typically:

- operates entirely in-memory
- produces deterministic outcomes
- relies only on the provided input parameters
- contains pure business rules or calculations
- has no dependency injection requirements

2.1.4.2 Design Guidance

Self-sufficient services should remain:

- focused
- deterministic
- side-effect free where possible
- easy to test without infrastructure setup

Because these services do not depend on external components, they are often simpler to:

- reason about

- unit test
- maintain
- reuse

2.1.4.3 Testing Considerations

While it is generally easier to test self-sufficient services due to their lack of dependencies, there are some unique considerations to keep in mind for exception handling and edge cases.

Logic and validation paths are straightforward to test with various input scenarios, but simulating internal failures or forcing specific execution branches can be more challenging without external dependencies to mock.

In these situations, there may be no injectable seam available for introducing mocked behaviour.
To address this, partial mocking techniques may be used to:

- simulate internal failures
- force deterministic execution branches
- validate internal exception handling behaviour
- isolate difficult-to-reach execution paths

This allows self-sufficient services to remain fully encapsulated while still enabling comprehensive unit test coverage.

Example

The service below is a simple example of a self-sufficient service that calculates the average score for a student based on an array of scores. It includes validation to ensure the scores are valid and demonstrates how to handle exceptions without relying on any external dependencies.

StudentScoreService.cs

```
public partial class StudentScoreService
{
        public ValueTask<double> CalculateAverageScoreAsync(params int[] scores) =>
        TryCatch(async () =>
        {
                ValidateScores(scores);
```

```
                return scores.Average();
        });

        internal virtual void ValidateScores(int[] scores)
        {
                if (scores is null || scores.Length == 0)
                {
                        throw new InvalidStudentScoresException(
                                "Student scores are invalid.");
                }

                if (scores.Any(score => score < 0 || score > 100))
                {
                        throw new InvalidStudentScoresException(
                                "Student scores are invalid.");
                }
        }
}
```

Logic Test

```
[Fact]
public async Task ShouldCalculateAverageScoreAsync()
{
        // given
        int[] scores = [80, 90, 100];

        double expectedAverage = 90;

        var studentScoreService = new StudentScoreService();

        // when
        double actualAverage =
                await studentScoreService.CalculateAverageScoreAsync(scores);

        // then
        actualAverage.Should().Be(expectedAverage);
}
```

(Since there are no dependencies, we do not see any of the usual Setup, Verify or VerifyNoOtherCalls lines)

Validation Test

```
[Fact]
public async Task
ShouldThrowValidationExceptionOnCalculateAverageScoreIfScoresAreInvalidAsync()
{
        // given
        int[] invalidScores = [];
```

```
        var invalidStudentScoresException =
                new InvalidStudentScoresException(
                        "Student scores are invalid.");

        var expectedStudentValidationException =
                new StudentValidationException(
                        message: "Student validation error occurred, fix errors and
try again.",
                        innerException: invalidStudentScoresException);

        var studentScoreService = new StudentScoreService();

        // when
        ValueTask<double> calculateAverageScoreTask =
                studentScoreService.CalculateAverageScoreAsync(invalidScores);

        // then
        StudentValidationException actualException =
                await Assert.ThrowsAsync<StudentValidationException>(
                        calculateAverageScoreTask.AsTask);

        actualException.Should().BeEquivalentTo(
                expectedStudentValidationException);
}
```

(Since there are no dependencies, we do not see any of the usual Setup, Verify or VerifyNoOtherCalls lines)

Exception Test Using Partial Mocking

Since `CalculateAverageScoreAsync` does not have any dependencies, we can use partial mocking to simulate an internal failure and validate that the exception handling logic is working as expected.

Under normal conditions we would use the first dependency executed within the method to mock exceptions, but since there are no dependencies in this service, we will have to use partial mocking of the service itself to throw an exception from the first method called. In this case it is the `ValidateScores` method, which is called for validation inside the `TryCatch` block.

Because `ValidateScores` is already declared as `internal virtual` in the service definition above, Moq can override it on a partial mock of `StudentScoreService`.

Since `ValidateScores` is `internal`, it must be made visible to the test project.

Inside the project file of `University.ClassLibrary`, we can add the following lines to make the internal members visible to the test project:

```
        <ItemGroup>
                <InternalsVisibleTo Include="University.ClassLibrary.Tests.Unit" />
                <InternalsVisibleTo Include="DynamicProxyGenAssembly2" />
        </ItemGroup>
```

(DynamicProxyGenAssembly2 is required for Moq to work with internal members)

Now we can write our test using Moq to create a partial mock of the `StudentScoreService` and set up the `ValidateScores` method to throw an exception.

```
[Fact]
public async Task
ShouldThrowServiceExceptionOnCalculateAverageScoreIfCalculationFailsAsync()
{
        // given
        int[] scores = [80, 90, 100];

        var failedStudentServiceException =
                new Exception("Calculation failed.");

        var expectedStudentServiceException =
                new StudentServiceException(
                        message: "Student service error occurred, contact
support.",
                        innerException: failedStudentServiceException);

        var studentScoreServiceMock =
                new Mock<StudentScoreService> { CallBase = true };

        studentScoreServiceMock
                .Setup(service => service.ValidateScores(scores))
                        .Throws(failedStudentServiceException);

        // when
        ValueTask<double> calculateAverageScoreTask =
                studentScoreServiceMock.Object.CalculateAverageScoreAsync(scores);

        // then
        StudentServiceException actualStudentServiceException =
                await Assert.ThrowsAsync<StudentServiceException>(
                        calculateAverageScoreTask.AsTask);

        actualStudentServiceException.Should().BeEquivalentTo(
                expectedStudentServiceException);
```

```
        studentScoreServiceMock
                .Verify(service => service.ValidateScores(scores),
                        Times.Once);

        studentScoreServiceMock.VerifyNoOtherCalls();
}
```

The mock is created with `CallBase = true`, which instructs Moq to execute the real implementation of any method that has not been explicitly set up. This means `CalculateAverageScoreAsync` runs its actual implementation while `ValidateScores` is intercepted and forced to throw, exercising the `TryCatch` exception handling path without any external dependencies.

Partial Mocking When a Method Has No Arguments

When the method under test has no arguments, such as `RetrievePi` below, there is no validation method that can be mocked directly. The same partial mocking technique still applies, but it requires a small refactor – extract the logic inside the `TryCatch` block into a separate `internal virtual` method so that Moq can intercept it.

Before the refactor:

```
public partial class MathService : IMathService
{
        public double RetrievePi() =>
        TryCatch(() =>
        {
                return Math.PI;
        });
}
```

After extracting the inner logic into a mockable method:

```
public partial class MathService : IMathService
{
        public virtual double RetrievePi() =>
        TryCatch(() =>
        {
                return GetPi();
        });

        internal virtual double GetPi() =>
                Math.PI;
}
```

With this refactor in place, `GetPi` can be set up to throw in the exception test, while `RetrievePi` executes its real implementation via `CallBase = true`:

```
[Fact]
public async Task
ShouldThrowServiceExceptionOnRetrievePiIfServiceErrorOccursAsync()
{
        // given
        var someException =
                new Exception("Some service error.");

        var expectedMathServiceException =
                new MathServiceException(
                        message: "Math service error occurred, contact support.",
                        innerException: someException);

        var mathServiceMock =
                new Mock<MathService> { CallBase = true };

        mathServiceMock
                .Setup(service => service.GetPi())
                        .Throws(someException);

        // when
        ValueTask<double> retrievePiTask =
                mathServiceMock.Object.RetrievePi();

        // then
        MathServiceException actualMathServiceException =
                await Assert.ThrowsAsync<MathServiceException>(
                        retrievePiTask.AsTask);

        actualMathServiceException.Should().BeEquivalentTo(
                expectedMathServiceException);

        mathServiceMock
                .Verify(service => service.GetPi(),
                        Times.Once);

        mathServiceMock.VerifyNoOtherCalls();
}
```

2.2

Processing Services (Higher-Order Business Logic)

2.2.0 Introduction

Processing services are the layer where a higher order of business logic is implemented. They may combine (or orchestrate) two primitive-level functions from their corresponding foundation service to introduce newer functionality. They may also call one primitive function and change the outcome with a little bit of added business logic. Sometimes, processing services are there as a pass-through to introduce balance to the overall architecture.

Processing services are optional, depending on your business need - in a simple CRUD operations API, processing services and all the other categories of services beyond that point will cease to exist as there is no need for a higher order of business logic at that point.

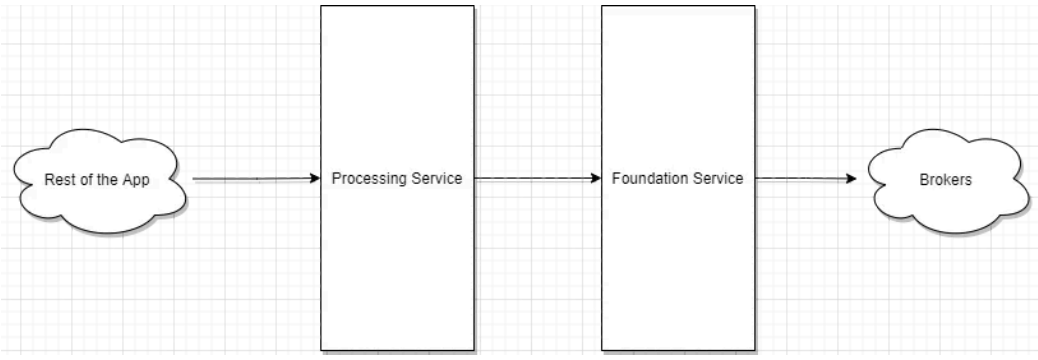
Here's an example of what a Processing service function would look like:

```
public ValueTask<Student> UpsertStudentAsync(Student student) =>
TryCatch(async () =>
{
    ValidateStudentOnUpsert(student);

    ValueTask<IQueryable<Student>> allStudents =
        await this.studentService.RetrieveAllStudentsAsync();

    bool studentExists = allStudents.Any(retrievedStudent =>
        retrievedStudent.Id == student.Id);

    return studentExists switch {
        false => await this.studentService.RegisterStudentAsync(student),
        _ => await this.studentService.ModifyStudentAsync(student)
    };
});
```

Processing services make Foundation services nothing but a layer of validation on top of the existing primitive operations. This means that Processing services functions are beyond primitive and only deal with local models, as we will discuss in the upcoming sections.

2.2.1 On The Map

Processing services live between foundation services and the rest of the application when used. They may not call Entities or Business brokers. Still, they may call Utility brokers such as logging brokers, time brokers, and any other brokers that offer supporting functionality and are not specific to any particular business logic. Here's a visual of where processing services are located on the map of our architecture:



On the right side of a Processing service lies all the non-local models and functionality, whether through the brokers or the models, and the foundation service is trying to map them into local models. On the left side of Processing services are pure local functionality, models, and architecture. Starting from the Processing services, there should be no trace or track of any native or non-local models in the system.

2.2.2 Characteristics

Processing services, in general, are combiners of multiple primitive-level functions to produce a higher-order business logic. But they have many more characteristics than just that; let's talk about those here.

2.2.2.0 Language

The language used in processing services defines the level of complexity and the capabilities it offers. Usually, processing services combine two or more primitive operations from the foundation layer to create a new value.

2.2.2.0.0 Functions Language

The Processing services language changes from primitive operations such as `AddStudentAsync` or `RemoveStudentAsync` to `EnsureStudentExistsAsync` or `UpsertStudentAsync`. They usually offer more advanced business logic operations to support higher-order functionality. Here are some examples of the most common combinations a processing service may offer:

Processing Operation	Primitive Functions
EnsureStudentExistsAsync	RetrieveAllStudentsAsync + AddStudentAsync
UpsertStudentAsync	RetrieveStudentByIdAsync + AddStudentAsync + ModifyStudentAsync
VerifyStudentExistsAsync	RetrieveAllStudentsAsync
TryRemoveStudentAsync	RetrieveStudentByIdAsync + RemoveStudentByIdAsync

As you can see, the combination of primitive functions processing services might also include adding an additional layer of logic on top of the existing primitive operation. For instance, `VerifyStudentExistsAsync` takes advantage of the `RetrieveAllStudentsAsync` primitive function and then adds a boolean logic to verify the returned student by an Id from a query actually exists or not before returning a `boolean`.

2.2.2.0.1 Pass-Through

Processing services may borrow some of the terminology a foundation service uses. For instance, in a pass-through scenario, a processing service could be as simple as `AddStudentAsync`. We will discuss the architecture-balancing scenarios later in this chapter. Unlike Foundation services, Processing services are required to have the identifier `Processing` in their names. For instance, we say `StudentProcessingService`.

2.2.2.0.2 Class-Level Language

More importantly, Processing services must include the name of the entity that is supported by their corresponding Foundation service. For instance, if a Processing service depends on a `TeacherService`, then the Processing service name must be `TeacherProcessingService`.

2.2.2.1 Dependencies

Processing services can only have two types of dependencies: a corresponding Foundation service or a Utility broker. That's simply because Processing services are nothing but an extra higher-order level of business logic orchestrated by combined primitive operations on the Foundation level. Processing services can also use Utility brokers such as `TimeBroker` or `LoggingBroker` to support their reporting aspect, but they shall never interact with an Entity or Business broker.

2.2.2.2 One-Foundation

Processing services can interact with only one Foundation service. In fact, without a foundation service, there can never be a Processing layer. And just like we mentioned above about the language and naming, Processing services take on the exact same entity name as their Foundation dependency. For instance, a processing service that handles higher-order business logic for students will communicate with nothing but its foundation layer, which would be `StudentService`. That means that processing services will have one and only one service as a dependency in its construction or initiation as follows:

```
public class StudentProcessingService
{
    private readonly IStudentService studentService;

    public StudentProcessingService(IStudentService studentService) =>
```

```
        this.studentService = studentService;
}
```

However, processing services may require dependencies on multiple utility brokers such as `DateTimeBroker` or `LoggingBroker` ... etc.

2.2.2.3 Used-Data-Only Validations

Unlike the Foundation layer services, Processing services only validate what they need from their input. For instance, if a Processing service is required to validate that a student entity exists, and its input model just happens to be an entire `Student` entity, it will only validate that the entity is not `null` and that the `Id` of that entity is valid. The rest of the entity is out of the Processing service's concern. Processing services delegate full validations to the layer of services that are concerned with that, which is the Foundation layer. Here's an example:

```
public ValueTask<Student> UpsertStudentAsync(Student student) =>
TryCatch(async () =>
{
    ValidateStudentOnUpsert(student);

    ValueTask<IQueryable<Student>> allStudents =
        await this.studentService.RetrieveAllStudentsAsync();

    bool isStudentExists = allStudents.Any(retrievedStudent =>
        retrievedStudent.Id == student.Id);

    return isStudentExists switch {
        false => await this.studentService.AddStudentAsync(student),
        _ => await this.studentService.ModifyStudentAsync(student.Id)
    };
});
```

Processing services are also not very concerned about outgoing validations except for what they will use within the same routine. Some exceptions may be neccesary as discussed in later sections. For instance, if a Processing service is retrieving a model and will use this model to be passed to another primitive-level function on the Foundation layer, the Processing service will be required to validate that the retrieved model is valid, depending on which attributes of the model it uses. However, processing services will delegate the outgoing validation to the foundation layer for Pass-through scenarios.

2.2.3 Responsibilities

Processing service's main responsibility is to provide higher-order business logic. This happens along with the regular signature mapping and various use-only validations, which we will discuss in detail in this section.

2.2.3.0 Higher-Order Logic

Higher-order business logic are functions that are above primitive. For instance, `AddStudentAsync` function is a primitive function that does one thing and one thing only. But higher-order logic is when we try to provide a function that changes the outcome of a single primitive function like `VerifyStudentExistsAsync`, which returns a boolean value instead of the entire object of the `Student`, or a combination of multiple primitive functions such as `EnsureStudentExistsAsync` which is a function that will only add a given `Student` model if and only if the object mentioned above doesn't already exist in storage. Here are some examples:

2.2.3.0.0 Shifters

The shifter pattern in a higher-order business logic is when the outcome of a particular primitive function is changed from one value to another. Ideally, a primitive type such as a `bool` or `int` is not a completely different type as that would violate the purity principle. For instance, in a shifter pattern, we want to verify whether a student exists. We want only some objects, but we want to know whether they exist in a particular system. Now, this is a case where we only need to interact with one and only one foundation service, and we are shifting the value of the outcome to something else, which should fit perfectly in the realm of the processing services. Here's an example:

```
public ValueTask<bool> VerifyStudentExistsAsync(Guid studentId) =>
TryCatch(async () =>
{
    ValidateStudentId(studentId);

    ValueTask<IQueryable<Student>> allStudents =
        await this.studentService.RetrieveAllStudentsAsync();

    ValidateStudents(allStudents);

    return allStudents.Any(student => student.Id == studentId);
});
```

In the snippet above, we provided higher-order business logic by returning a boolean value indicating whether a particular student with a given `Id` exists in the system. There are cases where your orchestration layer of services isn't really concerned with all the details of a particular entity but just knows whether it exists or not as part of an upper business logic or what we call orchestration.

Here's another popular example of a processing services shifting pattern:

```
public ValueTask<int> RetrieveStudentsCountAsync() =>
TryCatch(async () =>
{
    ValueTask<IQueryable<Student>> allStudents =
        await this.studentService.RetrieveAllStudentsAsync();

    ValidateStudents(allStudents);

    return allStudents.Count();
});
```

In the example above, we provided a function to retrieve the count of all students in a given system. The system's designers determine whether to interpret a `null` value retrieved for all students as an exception case that was not expected to happen or return a `0`, depending on how they manage the outcome. In our case, we validate the outgoing data as much as the incoming data, especially if it will be used within the processing function to ensure further failures do not occur for upstream services.

2.2.3.0.1 Combinations

Combining multiple primitive functions from the foundation layer to achieve a higher-order business logic is one of the main responsibilities of a processing service. As we mentioned before, some of the most popular examples are for ensuring a particular student model exists as follows:

```
public async ValueTask<Student> EnsureStudentExistsAsync(Student student) =>
TryCatch(async () =>
{
    ValidateStudent(student);

    ValueTask<IQueryable<Student>> allStudents =
        await this.studentService.RetrieveAllStudentsAsync();

    Student maybeStudent = allStudents.FirstOrDefault(retrievedStudent =>
        retrievedStudent.Id == student.Id);
```

```
    return maybeStudent switch
    {
        {} => maybeStudent,
        _ => await this.studentService.AddStudentAsync(student)
    };
});
```

In the code snippet above, we combined `RetrieveAllStudentsAsync` with `AddStudentAsync` to achieve a higher-order business logic operation. The `EnsureStudenExistsAsync` operation needs to verify something or an entity exists first before trying to persist it. The terminology around these higher-order business logic routines is very important. Its importance lies mainly in controlling the expectations of the outcome and the inner functionality. However, it also ensures that engineers do not require fewer cognitive resources to understand the underlying capabilities of a particular routine. The conventional language used in all of these services also ensures that redundant capability will not be created mistakenly. For instance, an engineering team without any form of standard might create `TryAddStudentAsync` while already having an existing functionality such as `EnsureStudentExistsAsync`, which does the same thing. With the limitation of the size of capabilities a particular service may have, the convention here ensured redundant work should never occur on any occasion. There are so many different combinations that can produce higher-order business logic. For instance, we may need to implement functionality that ensures a student is removed. We use `EnsureStudentRemovedByIdAsync` to combine a `RetrieveByIdAsync` and a `RemoveByIdAsync` in the same routine. It all depends on what level of capabilities an upstream service may need to implement such a functionality.

2.2.3.1 Signature Mapping

Although processing services operate fully on local models and local contracts, they are still required to map foundation-level services' models to their own local models. For instance, if a foundation service throws `StudentValidationException` then processing services will map that exception to `StudentProcessingDependencyValidationException`. Let's talk about mapping in this section.

2.2.3.1.0 Non-Exception Local Models

In general, processing services are required to map any incoming or outgoing objects with a specific model of its own. But that rule only sometimes applies to non-exception models. For instance, if a `StudentProcessingService` is operating based on a `Student` model, and there's no need for a special model for this service, then the processing service may be permitted to use the exact same model from the foundation layer.

2.2.3.1.1 Exception Models

When processing services handle exceptions from the foundation layer, it is important to understand that exceptions in our Standard are more expressive in their naming conventions and role than any other model. Exceptions here define the what, where, and why every single time they are thrown. For instance, an exception called `StudentProcessingServiceException` indicates the entity of the exception, which is the `Student` entity. Then, it indicates the location of the exception, which is the `StudentProcessingService`. Lastly, it indicates the reason for that exception, which is `ServiceException`, indicating an internal error to the service that is not a validation or a dependency of nature that happened. Just like the foundation layer, processing services will do the following mapping to occurring exceptions from its dependencies:

Exception	Wrap Inner Exception With	Wrap With	Log Level
StudentDependencyValidationException	Any inner exception	StudentProcessingDependencyValidationException	Error
StudentValidationException	Any inner exception	StudentProcessingDependencyValidationException	Error
StudentDependencyException	-	StudentProcessingDependencyException	Error
StudentServiceException	_	StudentProcessingDependencyException	Error
Exception	_	StudentProcessingServiceException	Error

[*] Processing services in Action (Part 1)

[*] Processing services in Action (Part 2)

[*] Processing services in Action (Part 3)

[*] Processing services in Action (Part 4)

2.3

Orchestration Services (Complex Higher Order Logic)

2.3.0 Introduction

Orchestration services combine multiple foundation or processing services to perform a complex logical operation. Their main responsibilities are multi-entity logical operations and delegating the dependencies of those operations to downstream processing or foundation services.

Orchestration services' primary responsibility is encapsulating operations requiring two or three business entities.

```
public async ValueTask<LibraryCard> CreateStudentLibraryCardAsync(LibraryCard
libraryCard) =>
TryCatch(async () =>
{
    ValidateLibraryCard(libraryCard);

    await this.studentProcessingService
        .VerifyEnrolledStudentExistsAsync(libraryCard.StudentId);

    return await
this.libraryCardProcessingService.CreateLibraryCardAsync(libraryCard);
});
```

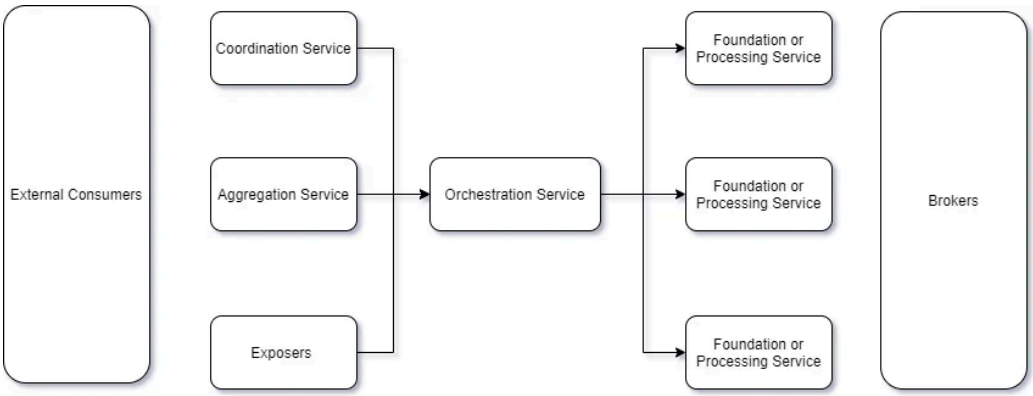
In the above example, the `LibraryCardOrchestrationService` calls both the `StudentProcessingService` and `LibraryCardProcessingService` to perform a complex operation. First, we verify the student's existence and enrollment, then create the library card.

Creating a library card for a given student cannot be performed by simply calling the library card service because the library card service (processing or foundation) needs access to all the details about the student. Therefore, a combination logic is required to ensure that a proper flow is in place.

It's important to understand that orchestration services are only required if we need to combine multi-entity operations, which can be primitive or higher-order. In some architectures, orchestration services might not even exist. That's simply because some microservices might be merely responsible for applying validation logic and persisting and retrieving data from storage, no more or no less.

2.3.1 On The Map

Orchestration services are one of the core business logic components in any system, positioned between single entity services (such as processing or foundation) and advanced logic services such as coordination services, aggregation services, or simply exposers such as controllers, web components, or anything else. Here's a high-level overview of where orchestration services may live:



As shown above, Orchestration services have quite a few dependencies and consumers. They are the core engine of any software. On the right-hand side, you can see an orchestration service's dependencies. Since a processing service is optional based on whether a higher-order business logic is needed, orchestration services can combine multiple foundation services as well.

The existence of an Orchestration service warrants the presence of a Processing service. But that's only sometimes the case. In some situations, all orchestration services need to finalize a business flow to interact with primitive-level functionality.

However, an Orchestration service could have several consumers, such as coordination services (orchestrators of orchestrators), aggregation services, or an exposer. Exposers are like controllers, view services, UI components, or another foundation or processing service in case of putting messages back on a queue - which we will discuss further in our Standard.

2.3.2 Characteristics

In general, orchestration services are concerned with combining single-entity primitive or higher-order business logic operations to execute a successful flow. But you can also think of them as the glue that ties multiple single-entity operations together.

2.3.2.0 Language

Just like Processing services, the language used in Orchestration services defines the level of complexity and the capabilities it offers. Orchestration services usually combine two or more primitive or higher-order operations from multiple single-entity services to execute a successful operation.

2.3.2.0.0 Functions Language

Orchestration services have a common characteristic regarding the language of their functions. Orchestration services are wholistic in most of the language of its function. You will see functions such as `NotifyAllAdmins` where the service pulls all users with an admin type and then calls a notification service.

Orchestration services offer functionality that inches closer to a business language than primitive technical operations. You may see almost an identical expression in a non-technical business requirement matching a function name in an orchestration service. The same pattern continues as one goes to higher and more advanced categories of services within a specific realm of business logic.

2.3.2.0.1 Pass-Through

Orchestration services can also be a pass-through for some operations. For instance, an orchestration service could allow an `AddStudentAsync` to be propagated through the service to unify the source of interactions with the system

at the exposer's level. In this case, orchestration services will use the same terminology a processing or foundation service may use to propagate the operation.

2.3.2.0.2 Class-Level Language

Orchestration services mainly combine multiple operations supporting a particular entity. So, if the primary entity is `Student` and the rest of the entities are just to support an operation mainly targeting a `Student` entity, then the name of the orchestration service would be `StudentOrchestrationService`.

Enforcement of naming conventions ensures that any orchestration service stays focused on a single entity's responsibility concerning multiple other supporting entities.

For instance, creating a library card requires the school enrollment of the student referenced in that library card. In this case, the Orchestration service name will reflect its primary entity, `LibraryCard`. Our orchestration service name would then be `LibraryCardOrchestrationService`.

The opposite is also true. If enrolling a student in a school has associated operations such as creating a library card, then, in this case, a `StudentOrchestrationService` must exist to create a `Student` and all other related entities.

The same idea applies to all exceptions created in an orchestration service, such as `StudentOrchestrationValidationException` and `StudentOrchestrationDependencyException`.

2.3.2.1 Dependencies

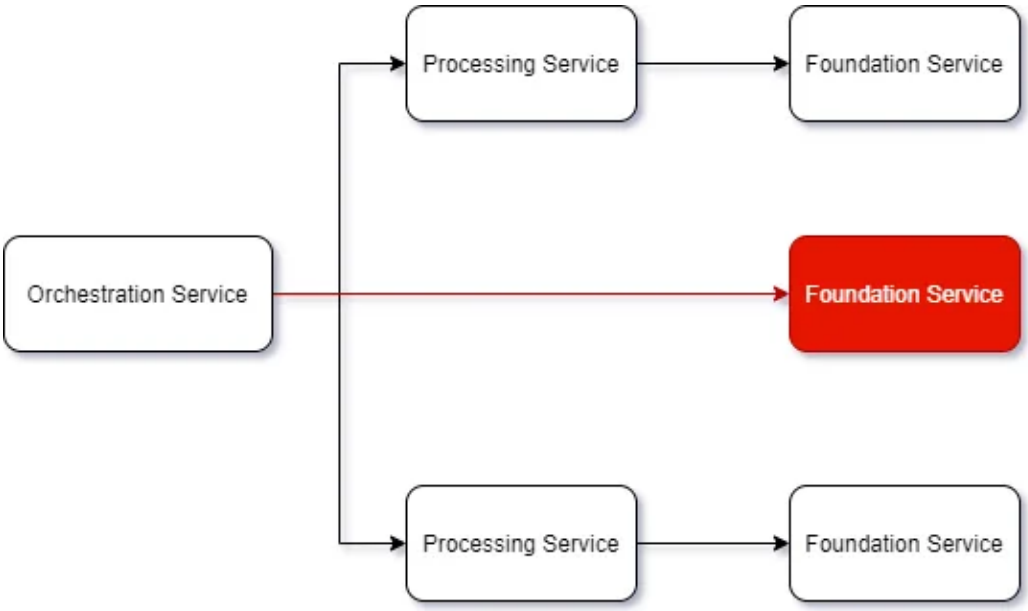
As we mentioned above, Orchestration services might have a more extensive range of dependencies, unlike Processing and Foundation services, because Processing services are optional. Therefore, Orchestration services may have dependencies ranging from foundation services or optional processing services to cross-cutting services such as logging or other utility brokers.

2.3.2.1.0 Dependency Balance (Florance Pattern)

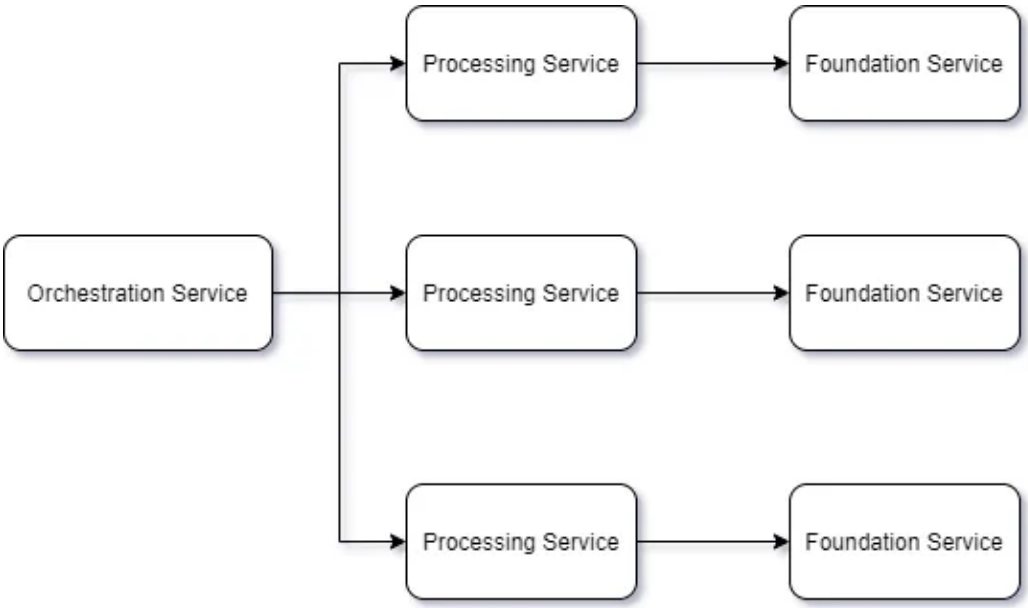
A fundamental rule governing the consistency and balance of orchestration services is the 'Florance Pattern', which dictates that any orchestration service may not combine dependencies from different categories of operation.

That means an Orchestration service cannot combine Foundation and Processing services. The dependencies have to be either all Processings or all Foundation services. That rule doesn't apply to utility broker dependencies, however.

Here's an example of an unbalanced orchestration service dependency:



An additional processing service is required to give a pass-through to a lower-level foundation service to balance the architecture - applying 'Florance Pattern' for symmetry would turn our architecture into the following:



Applying the 'Florance Pattern' might be very costly initially, including creating an entirely new processing service (or multiple) to balance the architecture. However, its benefits outweigh the cost from maintainability, readability, and pluggability perspectives.

2.3.2.1.1 Two-Three

The 'Two-Three' rule is a complexity control rule. This rule dictates that an Orchestration service may have up to three or less than two Processing or Foundation services to run the orchestration. This rule, however, doesn't apply to utility brokers. Orchestration services may have a `DateTimeBroker` or a `LoggingBroker` without restriction. However, an orchestration service may not have an entity broker, such as a `StorageBroker` or a `QueueBroker`, which feeds directly into the core business layer of any service.

This rule, like most of the patterns and concepts in The Standard, is inspired by nature. You can see how the trees branch into twos and threes - the same thing for thunder, blood vessels, and so many other creations around, within, and above us follow the same pattern.

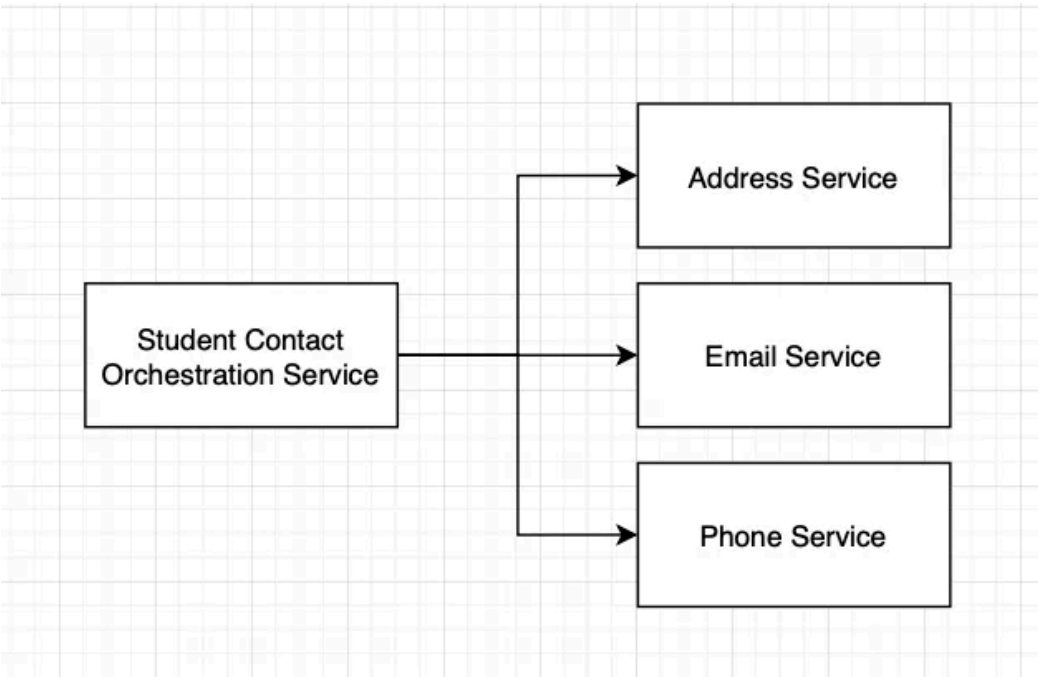
A tree branches as it grows upwards but also in its very roots. And so is the case with Orchestration and Orchestration-Like services. They can branch further upwards, as I will explain here shortly, but also downwards through patterns like the Cul-De-Sac pattern.

The 'Two-Three' rule may require a layer of normalization to the categorical business function. Let's talk about the different mechanisms of normalizing orchestration services.

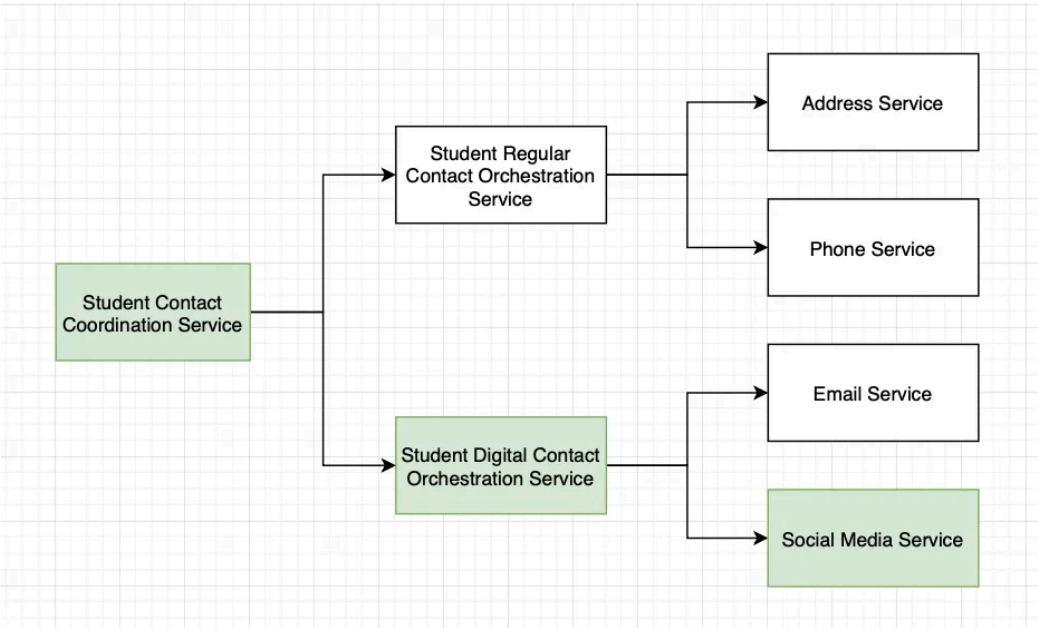
2.3.2.1.1.0 Full-Normalization

There are frequently situations where the current architecture of any given orchestration service ends up with one orchestration service with three dependencies. A new entity processing or foundation service is required to complete an existing process.

For instance, let's say we have a `StudentContactOrchestrationService`, which has dependencies that provide primitive-level functionality for each student `Address`, `Email`, and `Phone`. Here's a visualization of that state:



Now, a new requirement, 'SocialMedia', is added to 'Student', to gather more contact information about how to reach a student. We can go into full-normalization mode by finding common ground that equally splits the contact information entities. For instance, we can break out regular contact information versus digital contact information as in `Address` and `Phone` versus `Email` and `SocialMedia`. This way, we split four dependencies into two, each for their orchestration services as follows:



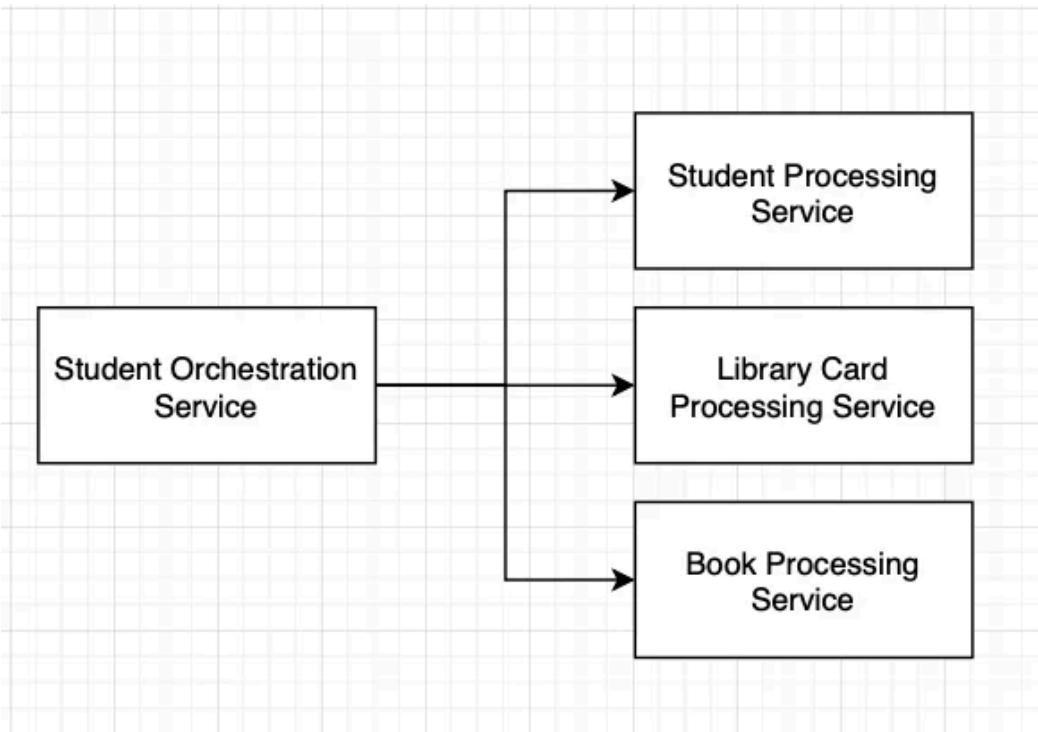
In the figure above, we modified the existing `StudentContactOrchestrationService` into `StudentRegularContactOrchestrationService` and removed one of its dependencies on the `EmailService`.

Additionally, we created a new `StudentDigitalContactOrchestrationService` to have two dependencies on the existing `EmailService` and the latest `SocialMediaService`. Consequently, we need an advanced business logic layer, like a coordination service, to provide student contact information to upstream consumers.

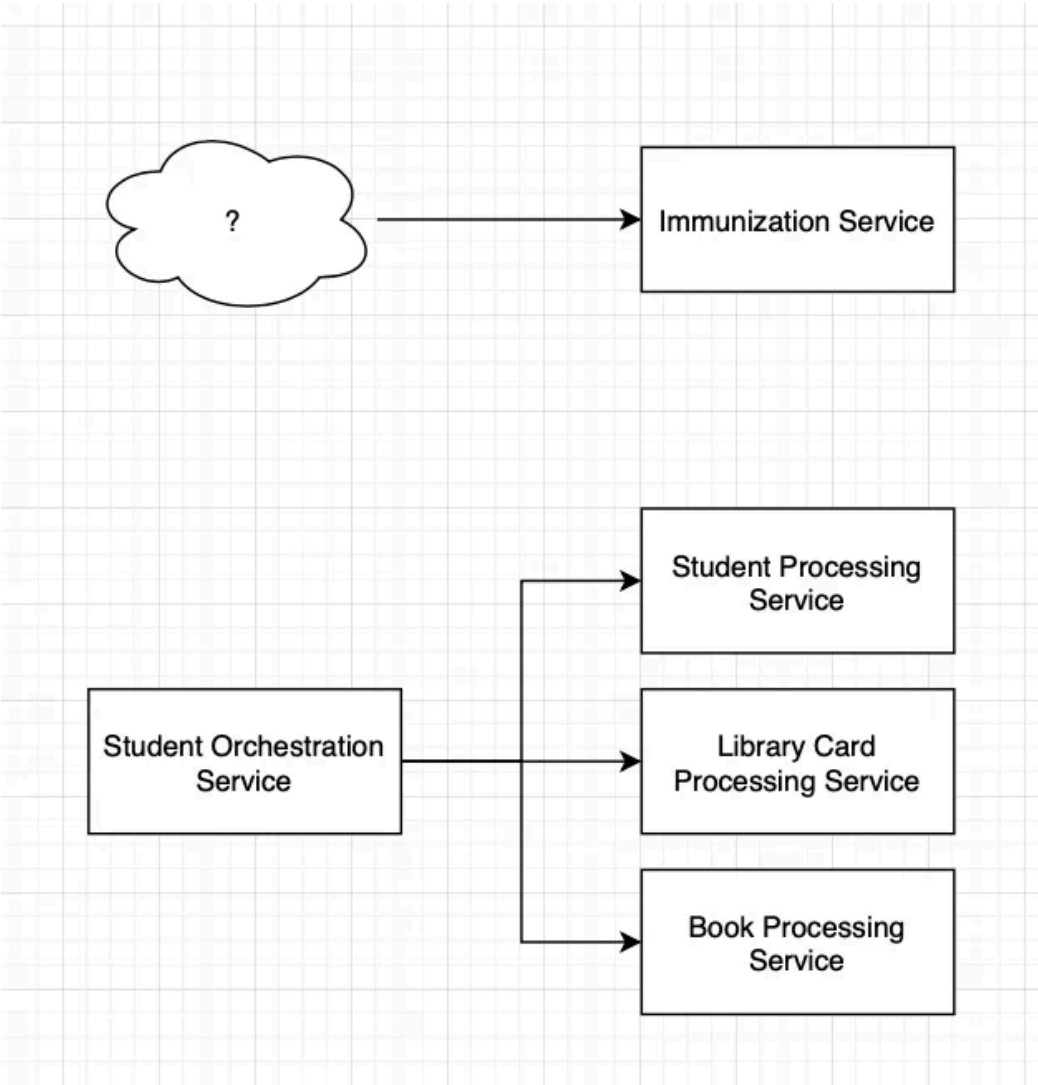
2.3.2.1.1.1 Semi-Normalization

Normalization is more complex than the example above, especially when a core entity has to exist before creating or filling in additional information about related entities.

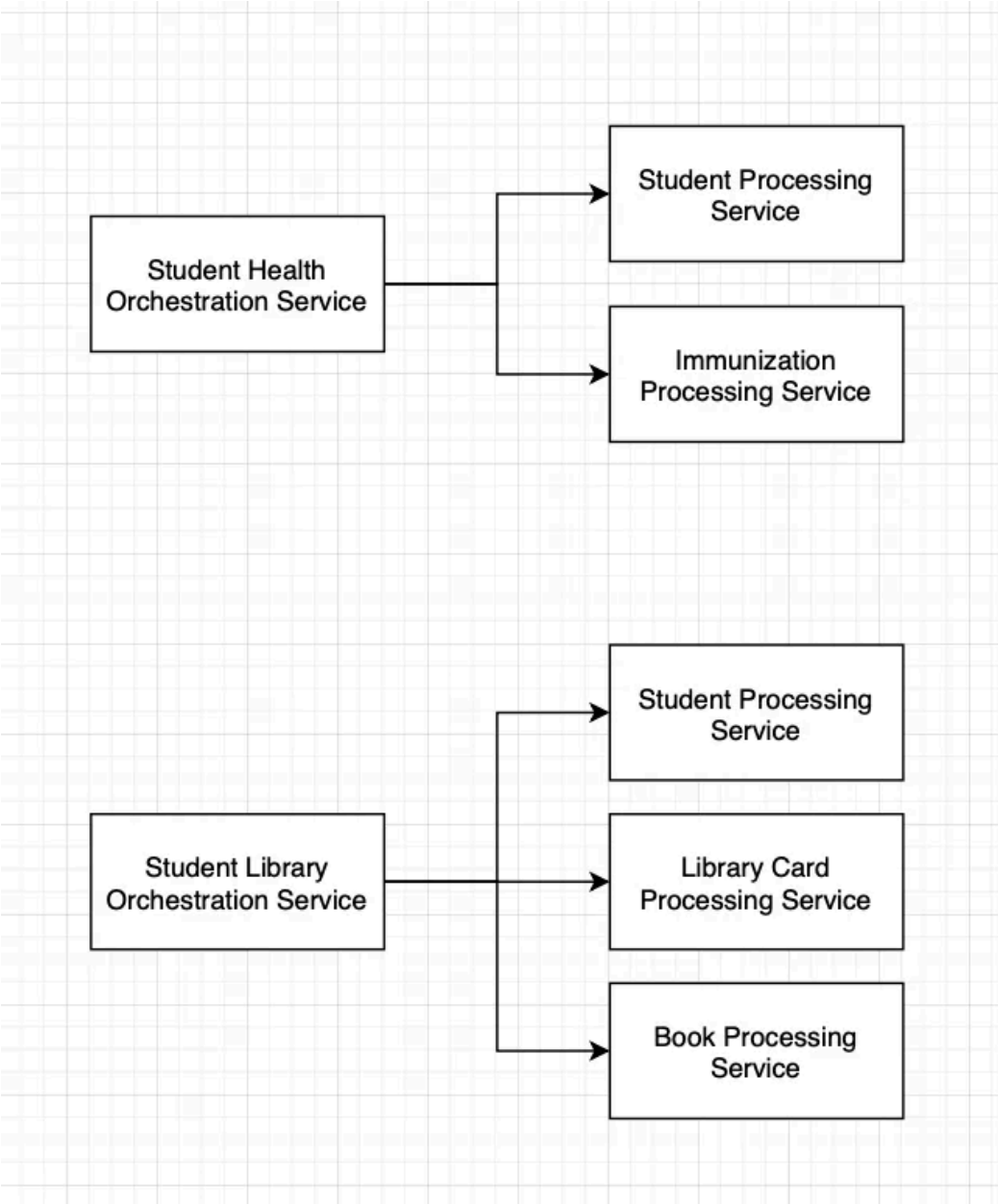
For instance, let's say we have a `StudentRegistrationOrchestrationService` which relies on `StudentProcessingService`, `LibraryCardProcessingService`, and `BookProcessingService` as follows:



But now, we need a new service called' ImmunizationProcessingService' to handle students' immunization records. We need all four services, but we already have a `StudentRegistrationOrchestrationService` that has three dependencies. At this point, a semi-normalization is required for the re-balancing of the architecture to honor the 'Two-Three' rule and eventually control the complexity.



In this case, a further normalization or a split is required to re-balance the architecture. We must think conceptually about the common ground between the primitive entities in a student registration process. Student requirements contain identity, health, and materials. We can, in this scenario, combine `LibraryCard` and `Book` under the same orchestration service as books and libraries are somewhat related. So we have `StudentLibraryOrchestrationService,` and for the other service, we would have `StudentHealthOrchestrationService` as follows:



To complete the registration flow with a new model, a coordination service must pass in advanced business logic to combine these entities. But more importantly, you will notice that each orchestration service has a redundant dependency of `StudentProcessingService` to ensure no virtual dependency on any other orchestration service, creating/providing a student record exists.

Virtual dependencies are very tricky. It's a hidden connection between two services of any category where one service implicitly assumes that a particular entity will be created and present. Virtual dependencies are very dangerous and threaten the proper autonomy of any service. Detecting virtual dependencies early in the design and development process could be a daunting but necessary task to ensure a clean, Standardized architecture is in place.

Just like model changes require database structure migrations and additional logic and validations, a new requirement for a new entity might require restructuring an existing architecture or extending it to a new version, depending on which stage the system receives these new requirements.

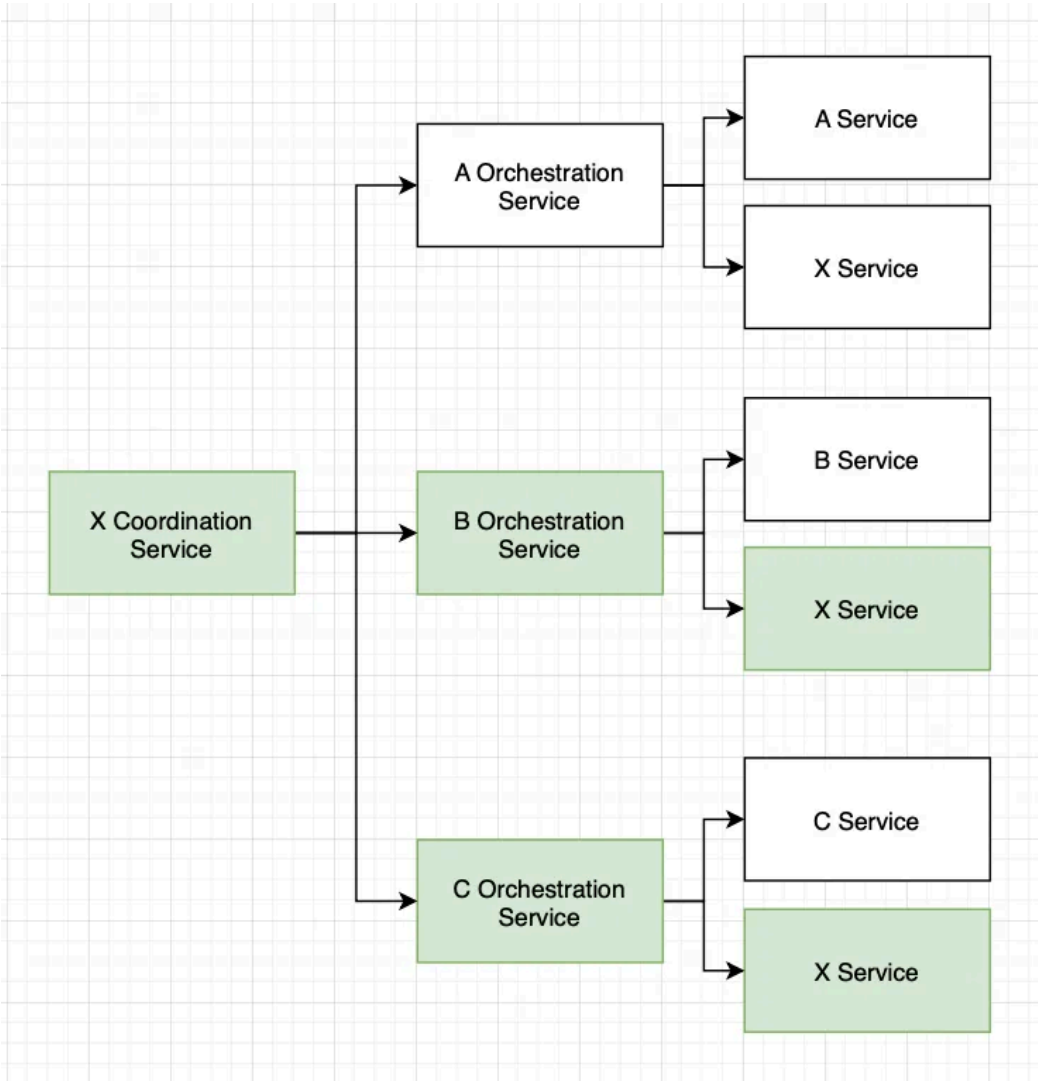
Adding another dependency to an existing orchestration service may be very enticing - but that's where the system starts to diverge from 'The Standard'. And that's when the system begins to become an unmaintainable legacy system. But more importantly, this scenario tests the design principles and standards of craftsmanship of the engineers involved in designing and developing the system.

2.3.2.1.1.2 No-Normalization

Everything, everywhere, is somehow connected. Yet, there are scenarios where higher levels of normalization are challenging to achieve. Sometimes, it might be incomprehensible for the mind to group multiple services under one orchestration service.

Because it's hard for me to come up with an example for multiple entities that have no connection to each other, it couldn't exist. I'm going to rely on some fictional entities to visualize a problem. So, let's assume there are `AService` and `BService` orchestrated together with an `XService`. The existence of `XService` is important to ensure that both `A` and `B` can be created with an assurance that a core entity `X` does exist.

Now, let's say a new service, `CService`, must be added to the mix to complete the existing flow. So, now we have four different dependencies under one orchestration service, and a split is mandatory. Since there's no relationship whatsoever between `A`, `B`, and `C`, a 'No-Normalization' approach becomes the only option to realize a new design as follows:



The above primitive services will be orchestrated with a core service, `X`, and then gathered under a coordination service. This case above is the worst-case scenario, where normalization of any size is impossible. Note that the author of this Standard couldn't come up with a realistic example unlike any others to show you how rare it is to run into that situation, so let a 'No-Normalization' approach be your very last solution if you run out of options.

2.3.2.1.1.3 Meaningful Breakdown

Regardless of the type of normalization you follow, you must ensure that your grouped services represent a common meaning. For instance, putting together a `StudentProcessingService` and `LibraryProcessingService` must require a functional commonality. An excellent example of that would be `StudentRegistrationOrchestrationService`. The registration process requires adding a new student record and creating a library card for that student.

Implementing orchestration services without an intersection between two or three entities per operation defeats the whole purpose of having an orchestration service. This condition is satisfied if at least one intersection between two entities has occurred. An orchestration service may have other 'Pass-Through' operations where we propagate certain routines from their processing or foundation origins if they match the same contract.

Here's an example:

```
public class StudentOrchestrationService
{
    public async ValueTask<Student> RegisterStudentAsync(Student student)
    {
        Student addedStudent =
            await this.studentProcessingService.AddStudentAsync(student);

        LibraryCard libraryCard =
            await this.libraryCardPorcessingService.AddLibraryCardAsync(
                addedStudent.Id);

        return addedStudent;
    }

    public async ValueTask<Student> ModifyStudentAsync(Student student) =>
        await this.studentProcessingService.ModifyStudentAsync(student);
}
```

In the example above, our `StudentOrchestrationService` had an orchestration routine that combined adding a student and creating a library card for that student. It also offers a 'Pass-Through' function for a low-level processing service routine to modify a student.

‘Pass-Through’ routines must have the same contract as the other routines in any orchestration service. Our ‘Pure Contract’ principle dictates that any service should allow the same contract as input and output or primitive types.

2.3.2.2 Contracts

Orchestration services may combine two or three different entities and their operations to achieve a higher business logic. There are two scenarios for contract/models for orchestration services: One that stays true to the primary entity’s purpose and one that is complex - a combinator orchestration service that tries to expose its inner target entities explicitly.

Let’s talk about these two scenarios in detail.

2.3.2.2.0 Physical Contracts

Some orchestration services are still single-purposed, even though they may combine two or three other higher-order routines from multiple entities. For instance, an orchestration service that reacts to messages from some queue and persists in these messages is a single-purposed and single-entity orchestration service.

Let’s take a look at this code snippet:

```
public class StudentOrchestrationService
{
    private readonly IStudentEventProcessingService studentEventProcessingService;
    private readonly IStudentProcessingService studentProcessingService;

    public StudentOrchestrationService(
        IStudentEventProcessingService studentEventProcessingService,
        IStudentProcessingService studentProcessingService)
    {
        this.studentEventProcessingService = studentEventProcessingService;
        this.studentProcessingService = studentProcessingService;
        ListenToEvents();
    }

    public void ListenToEvents() =>
        this.studentEventService.ListenToEvent(UpsertStudentAsync);

    public async ValueTask<Student> UpsertStudentAsync(Student student)
    {
        ...
```

```
        await this.studentProcessingService.UpsertStudentAsync(student);

        ...
    }
}
```

In the above example, the orchestration service still exposes functionality that honors the physical model `Student` and internally communicates with several services that may provide completely different models. These are the scenarios where a single entity has a primary purpose, and all other services are supporting services to ensure a successful flow for that entity.

In our example, the orchestration services listen to a queue for new student messages and use that event to persist any incoming new students in the system. So, the physical contract `Student` is the same language the orchestration service explicitly uses as a model to communicate with upper stream services/exposers or others.

However, there are other scenarios in which a single entity is not the only purpose/target for an orchestration service. Let's discuss that in detail.

2.3.2.2.1 Virtual Contracts

In some scenarios, an orchestration service may be required to create non-physical contracts to complete a particular operation. For instance, consider an orchestration service required to persist a social media post containing a picture. The requirement is to persist the picture in one database and the actual post (comments, authors, and others) in a different database table in a relational model.

The incoming model might be significantly different from the actual physical models. Let's see what that would look like in the real world.

Consider having this model:

```
public class MediaPost
{
    public Guid Id {get; set;}
    public string Content {get; set;}
    public DateTimeOffset Date {get; set;}
    public IEnumerable<string> Base64Images {get; set;}
}
```

The above contract, `MediaPost,` contains two separate physical entities. The first is the actual post, including the `Id`, `Content`, and `Date`, and the second is the list of images attached to that post.

Here's how an orchestration service would react to this incoming virtual model:

```
public async ValueTask<MediaPost> SubmitMediaPostAsync(MediaPost mediaPost)
{
    ...

    Post post = MapToPost(mediaPost);
    List<Media> medias = MapToMedias(mediaPost);

    Post addedPost =
        await this.postProcessingService.AddPostAsync(post);

    List<Medias> addedMedias =
        await this.mediaProcessingService.AddMediasAsync(medias);

    return MapToMediaPost(addedPost, addedMedias);
}

public Post MapToPost(MediaPost mediaPost)
{
    return new Post
    {
        Id = mediaPost.Id,
        Content = mediaPost.Content,
        CreatedDate = mediaPost.Date,
        UpdatedDate = mediaPost.Date
    };
}

public List<Media> MapToMedias(MediaPost mediaPost)
{
    return mediaPost.Base64Images.Select(image =>
        new Media
        {
            Id = Guid.NewGuid(),
            PostId = mediaPost.Id,
            Image = image,
            CreatedDate = mediaPost.Date,
            UpdatedDate = mediaPost.Date
        });
}
```

The above code snippet shows the orchestration service deconstructing a given virtual model/contract, `MediaPost`, into two physical models. Each one has its separate processing service that handles its persistence. There are scenarios

where the virtual model gets deconstructed into one single model with additional details used for validation and verification with downstream processing or foundation services.

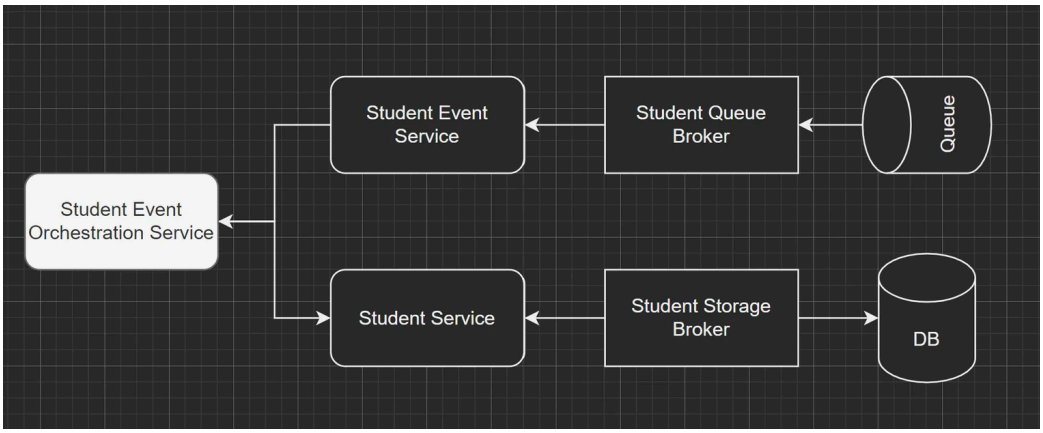
In hybrid situations, the incoming virtual model may have nested physical models, which we can only allow with virtual models. Physical models shall always stay anemic (contains no routines or constructors) and flat (contains no nested models) to control complexity and focus responsibility.

In summary, Orchestration services may create their contracts, which may be physical or virtual. A virtual contract may be a combination of one or many physical (or nested virtual) contracts or simply have its own flat design in terms of properties.

2.3.2.2 Cul-De-Sac

Sometimes, Orchestration services and their equivalent (coordination, management, etc.) may not need an exposer component (controller, for instance). That's because these services may listen to specific events and communicate them back into a Processing or a Foundation service at the same level where the event started or was received.

For example, incoming messages can be received from a subscription to an event service or a queue. In this case, the input for these services isn't necessarily through an exposer component anymore. Imagine building a simple application that gets notified with messages from a queue and then maps these messages into some local model to persist it in storage. In this case, the orchestration service would look something like the following:



The `StudentEventOrchestrationService` listens to messages for new students and immediately converts them into models that can be persisted in the database.

Here's an example:

Let's start with a unit test for this pattern as follows:

```
[Fact]
private void ShouldListenToProfileEvents()
{
   // given . when
   this.profileEventOrchestrationService.ListenToProfileEvents();

   // then
   this.profileEventServiceMock.Verify(service =>
       service.ListenToProfileEvent(
           this.profileEventOrchestrationService.ProcessProfileEventAsync),
               Times.Once);

   this.profileEventService.VerifyNoOtherCalls();
   this.profileServiceMock.VerifyNoOtherCalls();
   this.loggingBrokerMock.VerifyNoOtherCalls();
}

[Fact]
private async Task ShouldAddProfileAsync()
{
   // given
   ProfileEvent randomProfileEvent =
       CreateRandomProfileEvent();

   ProfileEvent inputProfileEvent =
       randomProfileEvent;
```

```
    this.profileServiceMock.Setup(service =>
        service.AddProfileAsync(inputProfileEvent.Profile));

    // when
    await this.profileEventOrchestrationService
        .ProcessProfileEventAsync(inputProfileEvent);

    // then
    this.profileServiceMock.Verify(service =>
        service.AddProfileAsync(inputProfileEvent.Profile),
            Times.Once);

    this.profileServiceMock.VerifyNoOtherCalls();
    this.loggingBrokerMock.VerifyNoOtherCalls();
    this.profileEventServiceMock.VerifyNoOtherCalls();
}
```

The test here indicates that an event listening has to occur first, and then persistence logic in the student service must match the outcome of mapping an incoming message to a given student.

Let's make this test pass.

```
public partial class ProfileEventOrchestrationService :
IProfileEventOrchestrationService
{
    private readonly IProfileEventService profileEventService;
    private readonly IProfileService profileService;
    private readonly ILoggingBroker loggingBroker;

    public ProfileEventOrchestrationService(
        IProfileEventProcessingService profileEventService,
        IProfileProcessingService profileService,
        ILoggingBroker loggingBroker)
    {
        this.profileEventService = profileEventService;
        this.profileService = profileService;
        this.loggingBroker = loggingBroker;
    }

    public void ListenToProfileEvents() =>
    TryCatch(() =>
    {
        this.profileEventService.ListenToProfileEvent(
            ProcessProfileEventAsync);
    });

    public ValueTask ProcessProfileEventAsync(ProfileEvent profileEvent) =>
    TryCatch(async () =>
    {
```

```
        ...

        await this.profileService.AddProfileAsync(profileEvent.Profile);
    });
}
```

In the above example, the Orchestration service constructor subscribes to the events that would come from the `ProfileEventProcessingService`. When an event occurs, the orchestration service calls the `ProcessProfileEventAsync` function to persist the incoming student into the database through a foundation or a processing service at the same level as the event service.

This pattern or characteristic is called the Cul-De-Sac. An incoming message will turn and head in a different direction for a different dependency. This pattern is typical in large enterprise-level applications where eventual consistency is incorporated to ensure the system can scale and become resilient under heavy consumption. This pattern also prevents malicious attacks against your API endpoints since it allows processing queue messages or events whenever the service is ready to process them. We will discuss the details in 'The Standard Architecture.

2.3.3 Responsibilities

Orchestration services provide advanced business logic. It orchestrates multiple flows for multiple entities/models to complete a single flow. Let's discuss in detail what these responsibilities are:

2.3.3.0 Advanced Logic

Orchestration services can only exist by combining multiple routines from multiple entities. These entities may differ in nature but share a standard flow or purpose. For instance, a `LibraryCard` model fundamentally differs from a `Student` model. However, they both share a common purpose regarding the student registration process. Adding a student record is required to register a student, but assigning a library card to that student is required for a successful student registration process.

Orchestration services ensure the correct routines for each entity are integrated and called in the proper order. Additionally, orchestration services are responsible for rolling back a failing operation. These three aspects constitute an orchestration

effort across multiple routines, entities, or contracts.

Let's talk about those in detail.

2.3.3.0.0 Flow Combinations

We spoke earlier about orchestration services combining multiple routines to achieve a common purpose or a single flow. This aspect of orchestration services can serve as both a fundamental characteristic and a responsibility. An orchestration service without at least one routine combining two or three entities is not considered an orchestration. Integrating multiple services without a common purpose is a better-fit definition for aggregation services, which we will discuss later in this services chapter.

However, within the flow combination comes the unification of the contract. I call it mapping and branching. Mapping an incoming model into multiple lower-stream service models and then branching the responsibility across these services.

Just like the previous services, during their flow combination, Orchestration services are responsible for ensuring the purity of the exposed input and output contracts, which becomes a bit more complex when combining multiple models. Orchestration services will continue to be responsible for mapping incoming contracts to their respective downstream services. They will also map back the results from these services into the unified model.

Let's bring back a previous code snippet to illustrate that aspect:

```
public async ValueTask<MediaPost> SubmitMediaPostAsync(MediaPost mediaPost)
{
    ...

    Post post = MapToPost(mediaPost);
    List<Media> medias = MapToMedias(mediaPost);

    Post addedPost =
        await this.postProcessingService.AddPostAsync(post);

    List<Medias> addedMedias =
        await this.mediaProcessingService.AddMediasAsync(medias);

    return MapToMediaPost(addedPost, addedMedias);
}

private Post MapToPost(MediaPost mediaPost)
```

```
{
    return new Post
    {
        Id = mediaPost.Id,
        Content = mediaPost.Content,
        CreatedDate = mediaPost.Date,
        UpdatedDate = mediaPost.Date
    };
}

private List<Media> MapToMedias(MediaPost mediaPost)
{
    return mediaPost.Base64Images.Select(image =>
        new Media
        {
            Id = Guid.NewGuid(),
            PostId = mediaPost.Id,
            Image = image,
            CreatedDate = mediaPost.Date,
            UpdatedDate = mediaPost.Date
        });
}

private MediaPost MapToMediaPost(Post post, List<Media> medias)
{
    return new MediaPost
    {
        Id = post.Id,
        Content = post.Content,
        Date = post.CreatedDate,
        Base64Images = medias.Select(media => media.Image)
    }
}
```

As you can see in the above example, the mapping and branching don't just happen on the way in. But a reverse action has to be taken on the way out. It violates The Standard to return the same input object that was passed in. That takes away any visibility on potential changes to the incoming request during persistence. The duplex mapping should substitute the need to dereference the incoming request to ensure no unexpected internal changes have occurred.

Note that breaking out the mapping logic into its own aspect/partial class file is also recommended—something like `StudentOrchestrationService.Mappings.cs`—to ensure the only thing left is orchestration's business logic.

2.3.3.0.1 Call Order

Calling routines in the correct order can be crucial to any orchestration process. For instance, a library card cannot be created unless a student record is created first. Enforcing the order here can be divided into two different types. Let's discuss those here for a bit.

2.3.3.0.1.0 Natural Order

The natural order here refers to specific flows that cannot be executed unless a prerequisite of input parameters is retrieved or persisted. For instance, imagine a situation where a library card cannot be created unless a student's unique identifier is retrieved first. In this case, we don't have to worry about testing that certain routines were called in the correct order because it comes naturally with the flow.

Here's a code example of this situation:

```
public async ValueTask<LibraryCard> CreateLibraryCardAsync(LibraryCard libraryCard)
{
    Student student = await this.studentProcessingService
        .RetrieveStudentByIdAsync(libraryCard.StudentId));

    return await this.libraryCardProcessingService
        .CreateLibraryCardAsync(libraryCard, student.Name);
}
```

In the example above, having a student `Name` is a requirement to create a library card. Therefore, the orchestration of order here comes naturally as part of the flow without additional effort.

Let's talk about the second type of order - Enforced Order.

2.3.3.0.1.1 Enforced Order

Imagine the same example above, but instead of the library card requiring a student name, it just needs the student `Id` already enclosed in the incoming request model. Something like this:

```
public async ValueTask<LibraryCard> CreateLibraryCardAsync(LibraryCard libraryCard)
{
    await this.studentProcessingService.VerifyEnlistedStudentExistAsync(
        libraryCard.StudentId);

    return await
```

```
this.libraryCardProcessingService.CreateLibraryCardAsync(libraryCard);
}
```

Ensuring a verified enrolled student exists before creating a library card might become a challenge because there is no dependency between the return value of one routine and the input parameters of the next. In other words, the `VerifyEnlistedStudentExistAsync` function returns nothing that the `CreateLibraryCardAsync` function cares about in terms of input parameters.

In this case, an enforced type of order must be implemented through unit tests. A unit test for this routine would require verifying not just that the dependency has been called with the correct parameters but also that they are called in the correct *order* let's take a look at how that would be implemented:

```
[Fact]
private async Task ShouldCreateLibraryCardAsync()
{
    // given
    Student someStudent = CreateRandomStudent();
    LibraryCard randomLibraryCard = CreateRandomLibraryCard();
    LibraryCard inputLibraryCard = randomLibraryCard;
    LibraryCard createdLibraryCard = inputLibraryCard;
    LibraryCard expectedLibraryCard = inputLibraryCard.DeepClone();
    Guid studentId = inputLibraryCard.StudentId;
    var mockSequence = new MockSequence();

    this.studentProcessingServiceMock.InSequence(mockSequence).Setup(service =>
        service.VerifyEnlistedStudentExistAsync(studentId))
            .Returns(someStudent);

    this.libraryCardProcessingServiceMock.InSequence(mockSequence).Setup(service =>
        service.CreateLibraryCardAsync(inputLibraryCard))
            .ReturnsAsync(createdLibraryCard);

    // when
    LibraryCard actualLibraryCard = await this.libraryCardOrchestrationService
        .CreateLibraryCardAsync(inputLibraryCard);

    // then
    actualLibraryCard.Should().BeEquivalentTo(expectedLibraryCard);

    this.studentProcessingServiceMock.Verify(service =>
        service.VerifyEnlistedStudentExistAsync(studentId),
            Times.Once);

    this.libraryCardProcessingServiceMock.Verify(service =>
        service.CreateLibraryCardAsync(inputLibraryCard),
            Times.Once);
```

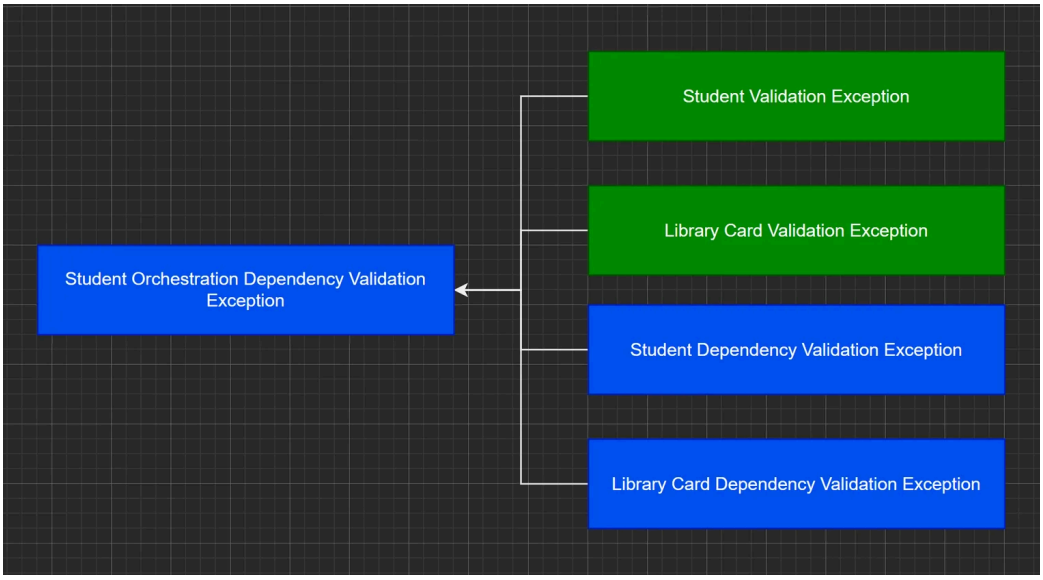
```
    this.studentProcessingServiceMock.VerifyNoOtherCalls();
    this.libraryCardProcessingServiceMock.VerifyNoOtherCalls();
    this.loggingBrokerMock.VerifyNoOtherCalls();
}
```

In the example above, the mock framework is being used to ensure a certain order is enforced when calling these dependencies. This way, we enforce a certain implementation within any given method to ensure that non-naturally connected dependencies are sequentially called in the intended order.

It's more likely that the type of ordering leans more towards enforced than natural when orchestration services reach the maximum number of dependencies.

2.3.3.0.2 Exceptions Mapping (Wrapping & Unwrapping)

This responsibility is very similar to flow combinations. Except in this case, orchestration services unify all the exceptions that may occur from any dependencies into one unified categorical exception model. Let's start with an illustration of what that mapping may look like:



In the illustration above, you will notice that validation and dependency validation exceptions, thrown from downstream dependency services, map into one unified dependency exception at the orchestration level. This practice allows upstream consumers of that same orchestration service to determine the next course of action based on one categorical exception type instead of four, or in the case of three dependencies, it would be six categorical dependencies.

Let's start with a failing test to materialize our idea here:

```
public static TheoryData DependencyValidationExceptions()
{
    string exceptionMessage = GetRandomMessage();
    var innerException = new Xeption(exceptionMessage);

    var studentValidationException =
        new StudentValidationException(
            message: "Student validation error occurred, fix errors and try
again.",
            innerException);

    var studentDependencyValidationException =
        new StudentDependencyValidationException(
            message: "Student dependency validation error occurred, fix errors and
try again.",
            innerException);

    var libraryCardValidationException =
        new LibraryCardValidationException(
            message: "Library card validation error occurred, fix errors and try
again.",
            innerException);

    var libraryCardDependencyValidationException =
        new LibraryCardDependencyValidationException(
            message: "Library card dependency validation error occurred, fix errors
and try again.",
            innerException);

    return new TheoryData<Xeption>
    {
        studentValidationException,
        studentDependencyValidationException,
        libraryCardValidationException,
        libraryCardDependencyValidationException
    };
}


[Theory]
```

```
[MemberData(nameof(DependencyValidationExceptions))]
private async Task
ShouldThrowDependencyValidationExceptionOnCreateIfDependencyValidationErrorOccursAn
dLogItAsync(
    Xeption dependencyValidationException)
{
    // given
    Student someStudent = CreateRandomStudent();

    var expectedStudentOrchestrationDependencyValidationException =
        new StudentOrchestrationDependencyValidationException(
            message: "Student dependency validation error occurred, fix errors and
try again",
            dependencyValidationException.InnerException as Xeption);

    this.studentServiceMock.Setup(service =>
        service.AddStudentAsync(It.IsAny<Student>()))
            .ThrowsAsync(dependencyValidationException);

    // when
    ValueTask<Student> addStudentTask =
        await this.studentOrchestrationService.AddStudentAsync(someStudent);

    StudentOrchestrationDependencyValidationException
        actualStudentOrchestrationDependencyValidationException =
                await
Assert.ThrowsAsync<StudentOrchestrationDependencyValidationException>(
                    addStudentTask.AsTask);

    // then
    actualStudentOrchestrationDependencyValidationException.Should()
        .BeEquivalentTo(expectedStudentOrchestrationDependencyValidationException);

    this.studentServiceMock.Verify(service =>
        service.AddStudentAsync(It.IsAny<Student>()),
            Times.Once);

    this.loggingBrokerMock.Verify(broker =>
        broker.LogError(It.Is(SameExceptionAs(
            expectedStudentOrchestrationDependencyValidationException))),
                Times.Once);

    this.libraryCardServiceMock.Verify(service =>
        service.AddLibraryCardAsync(It.IsAny<Guid>()),
            Times.Once);

    this.studentServiceMock.VerifyNoOtherCalls();
    this.loggingBrokerMock.VerifyNoOtherCalls();
    this.libraryCardServiceMock.VerifyNoOtherCalls();
}
```

Above, we verify that any of our four exception types are mapped into a `StudentOrchestrationDependencyValidationException`. We maintain the original localized exception as an inner exception. But we unwrap the categorical exception at this level to keep the original issue as we go upstream.

These exceptions are mapped under a dependency validation exception because they originate from a dependency or a dependency of a dependency downstream. For instance, if a storage broker throws an exception, it is a dependency validation (something like `DuplicateKeyException`). The broker-neighboring service would map that into a localized `StudentAlreadyExistException` and then wrap that exception in a categorical exception of type `StudentDependencyValidationException`. When that exception propagates upstream to Processing or an Orchestration service, we lose the categorical exception as we have already captured it under the proper scope of mapping. Then, we continue to embed that very localized exception under the current service dependency validation exception.

Let's try to make this test pass:

```
public partial class StudentOrchestrationService
{
    private delegate ValueTask<Student> ReturningStudentFunction();

    private async ValueTask<Student> TryCatch(ReturningStudentFunction
returningStudentFunction)
    {
        try
        {
            return await returningStudentFunction();
        }
        catch (StudentValidationException studentValidationException)
        {
            throw await
CreateAndLogDependencyValidationExceptionAsync(studentValidationException);
        }
        catch (StudentDependencyValidationException
studentDependencyValidationException)
        {
            throw await
CreateAndLogDependencyValidationExceptionAsync(studentDependencyValidationException
);
        }
        catch (LibraryCardValidationException libraryCardValidationException)
        {
            throw await
```

```
CreateAndLogDependencyValidationExceptionAsync(libraryCardValidationException);
        }
        catch (LibraryCardDependencyValidationException
libraryCardDependencyValidationException)
        {
            throw await
CreateAndLogDependencyValidationExceptionAsync(libraryCardDependencyValidationExcep
tion);
        }
    }

    private async ValueTask<StudentOrchestrationDependencyValidationException>
        CreateAndLogDependencyValidationExceptionAsync(Xeption exception)
    {
        var studentOrchestrationDependencyValidationException =
            new StudentOrchestrationDependencyValidationException(
                message: "Student dependency validation error occurred, fix errors
and try again",
                exception.innerException as Xeption);

        await
this.loggingBroker.LogErrorAsync(studentOrchestrationDependencyValidationException)
;

        return studentOrchestrationDependencyValidationException;
    }
}
```

Now we can use the `TryCatch` as follows:

```
public async ValueTask<Student> AddStudentAsync(Student student) =>
TryCatch(async () =>
{
    ...
    Student addedStudent = await this.studentService.AddStudentAsync(student);
    LibraryCard libraryCard = await
this.libraryCard.AddLibraryCard(addedStudent.Id);

    return addedStudent;
});
```

In the implementation, you can see that we mapped all four different types of external downstream services validation exceptions into one categorical exception and then maintained the inner exception for each one.

The same rule applies to dependency exceptions. Dependency exceptions can be both Service and Dependency exceptions from downstream services. For instance, in the above example, calling a student service may produce

`StudentDependencyException` and `StudentServiceException`. These categorical exceptions will be unwrapped from their categorical layer and have their local layer wrapped in one unified new orchestration-level categorical exception under `StudentOrchestrationDependencyException`. The same applies to all other dependency categorical exceptions like `LibraryCardDependencyException` and `LibraryCardServiceException`.

It's crucial to unwrap and wrap localized exceptions from downstream services with categorical exceptions at the current service layer to ensure consistency with the Exposers layer. These exceptions can be easily handled and mapped into whatever the nature of the exposer component dictates. In the case of an Exposer component of type API Controller, the mapping would produce HTTP Status Codes. In the case of UI Exposer components, it would map to text meaningful to end users.

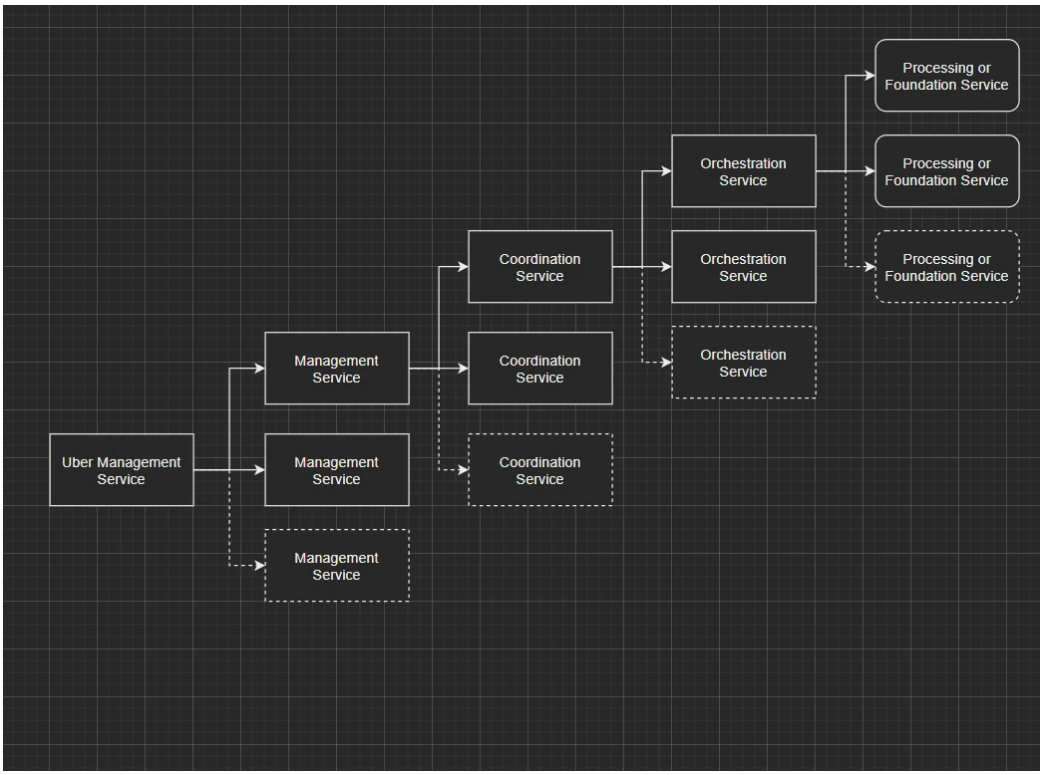
We will discuss further upstream in this Standard when to expose localized inner exceptions details where end-users are not required to take any action exclusive to dependency and service level exceptions.

2.3.4 Variations

Orchestration services vary depending on their position in the overall low-level architecture. For instance, an orchestration service that relies on downstream orchestration services is called a Coordination service. An Orchestration service working with multiple Coordination services as dependencies is called a Management Service. These variants are Orchestration services with uber-level business logic.

2.3.4.0 Variants Levels

Let's take a look at the possible variants for orchestration services and where they would be positioned:



In my personal experience, I've rarely had to resolve to an Uber Management service. The limitation here in terms of dependencies and variations of orchestration-like services is to help engineers rethink the complexity of their logic. But admittedly, there are situations where complexity is an absolute necessity. Therefore, Uber Management services exist as an option.

The following table should guide the process of developing variants of orchestration services based on the level:

Variant	Dependencies	Consumers	Complexity
Orchestration Services	Foundation or Processing Services	Coordination Services	Low
Coordination Services	Orchestration Services	Management Services	Medium

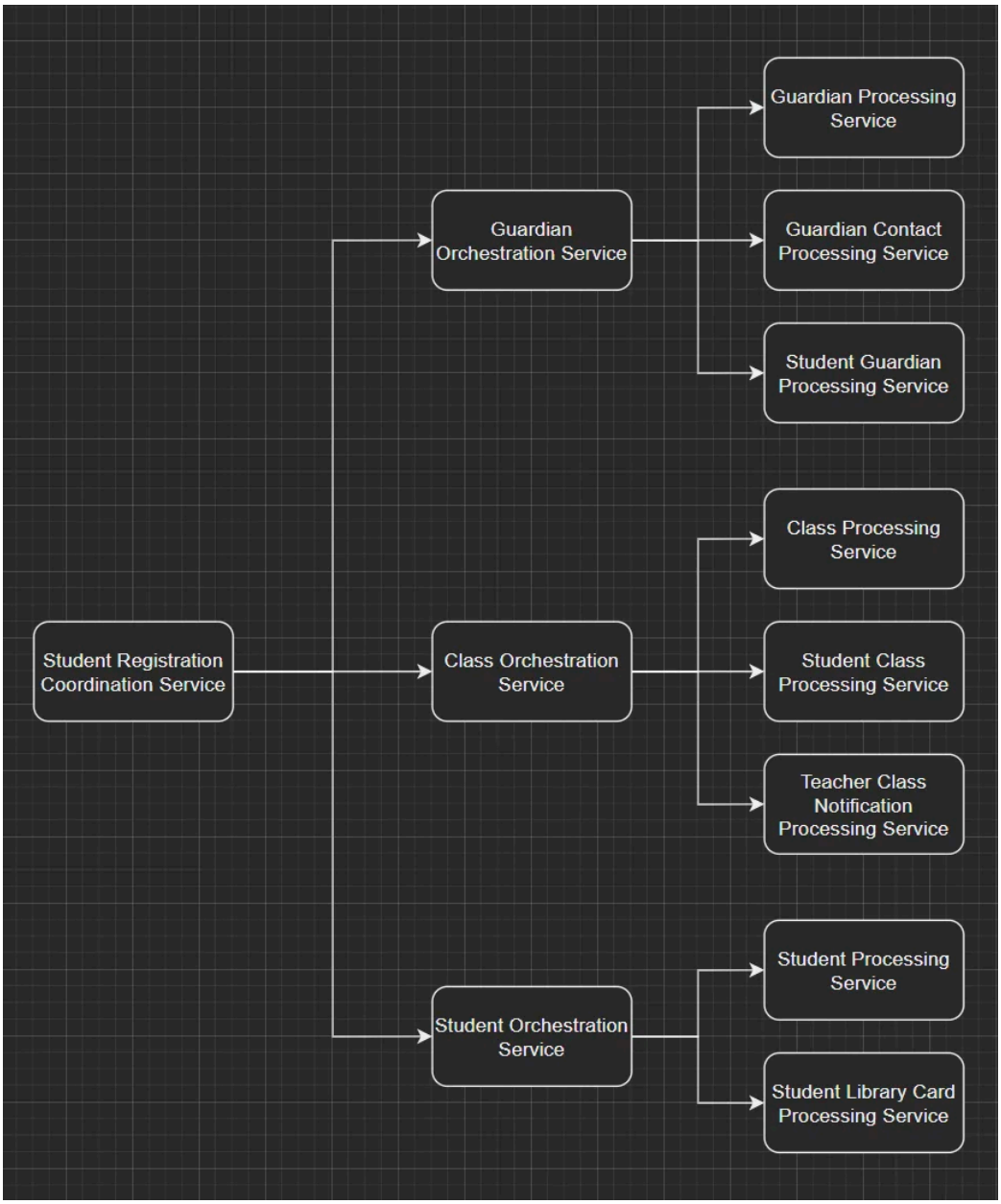
Variant	Dependencies	Consumers	Complexity
Management Services	Coordination Services	Uber Management Services	High
Uber Management Services	Management Services	Aggregation, Views or Exposer Components	Very High

Working beyond Uber Management services in an orchestration manner would require a more profound discussion and a serious consideration of the overall architecture. Future versions of The Standard might be able to address this issue in what I call “The Lake House,” but that is outside of the scope of this version of The Standard.

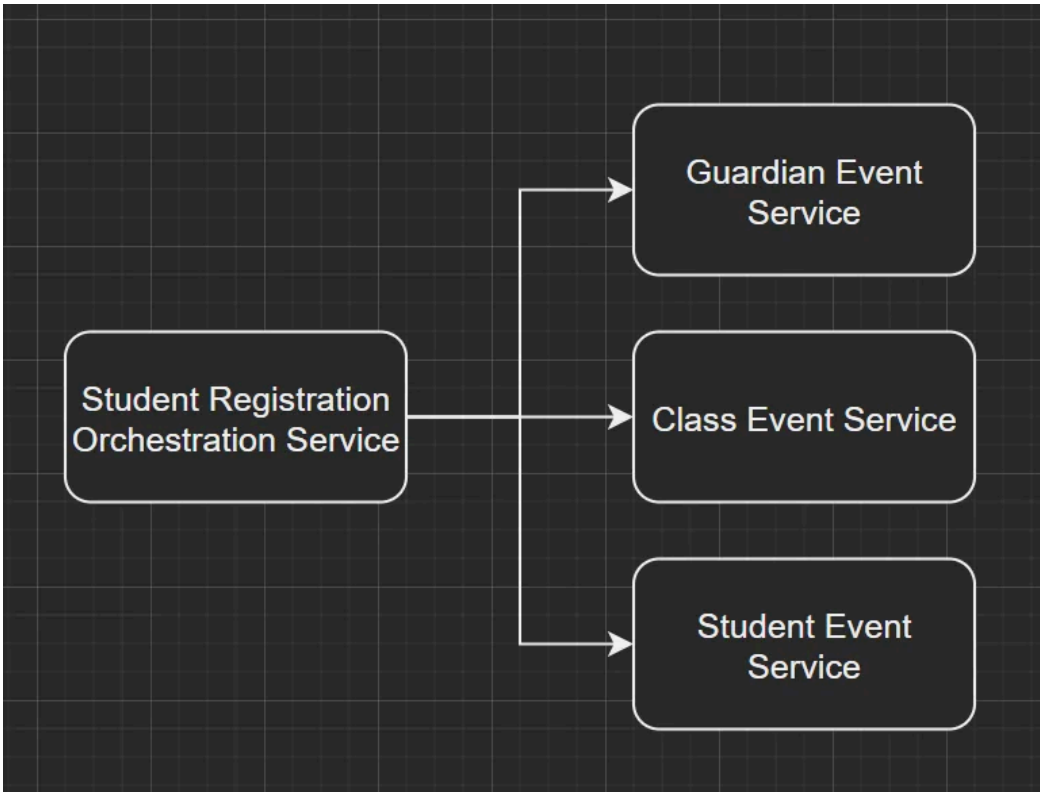
2.3.4.1 Unit of Work

With the variations of orchestration services, I recommend staying true to the concept of unit of work. Every request can do one thing and one thing only, including its prerequisites. For instance, if you need to register a student in a school, you may also need to add a guardian, contact information, and other details. Eventing these actions can significantly decrease the complexity of the flow and lower the risk of failures in downstream services.

Here’s a visualization for a complex single-threaded approach:



The solution above is a working solution for registering a student. We needed to include guardian information, library cards, classes, etc. These dependencies can be broken down into eventing, allowing other services to pick up where the single-threaded services leave off to continue the registration process. Something like this:



Above, the incoming request is turned into events, each of which would notify its orchestration services in a cul-de-sac pattern, as discussed in section 2.3.2.2. That means that a single thread is no longer responsible for the success of each dependency in the system. Instead, every event-listening broker would handle its process in a simplified way.

This approach does not guarantee an immediate response of success or failure to the requestor. It's an eventual consistency pattern where the client would get an `Accepted` message or its equivalent based on the communication protocol to let them know that a process has started. Still, results are only guaranteed once all event logic has been executed.

Note that we can add an extra layer of resiliency to these events by temporarily storing them in Queue-like components or memory-based temporary storages; depending on the criticality of the business.

However, an eventual consistency approach is only sometimes a good solution if the client on the other side is waiting for a response, especially in critical situations where an immediate response is required. One solution to this problem is Fire-n-Observe queues, which we will discuss in the future version of The Standard.

[*] Introduction to Orchestration Services

[*] Cul-De-Sac Pattern for Orchestration Services

[*] Cul-De-Sac Pattern for Coordination Services

2.4

Aggregation Services (The Knot)

2.4.0 Introduction

An Aggregation service's primary responsibility is to expose one single point of contact between the core business logic layer and any exposure layers. It ensures that multiple services of any variation share the same contract to be aggregated and exposed to one exposer component through one logical layer.

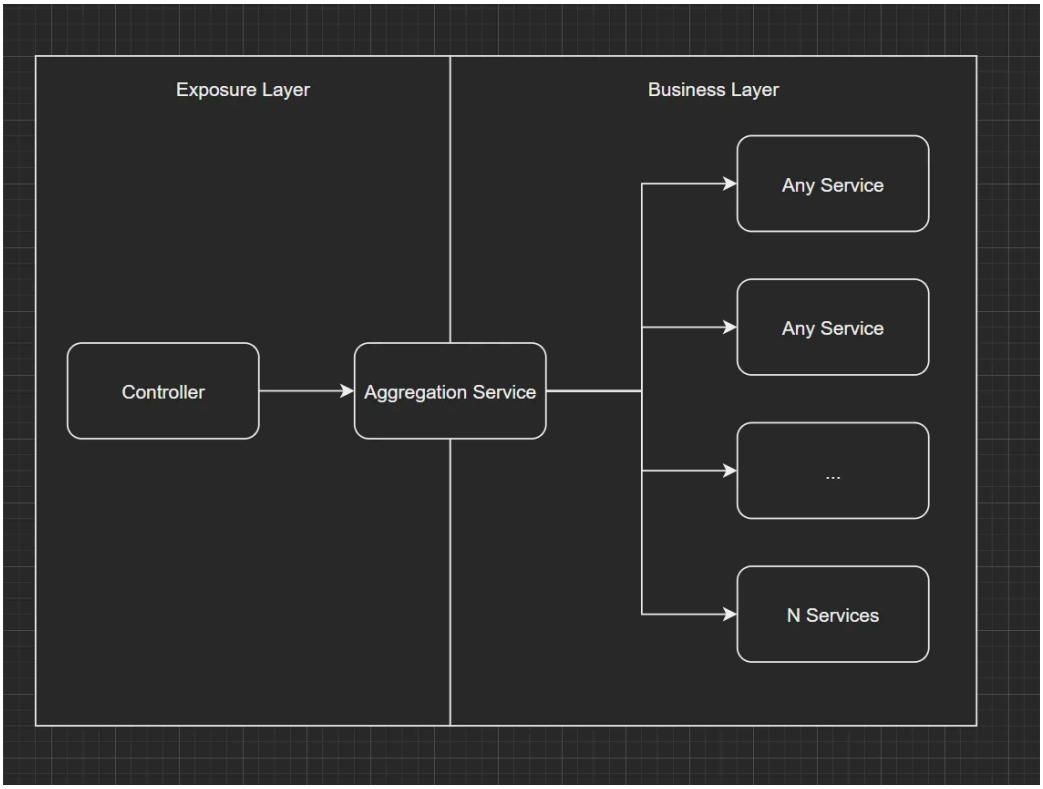
Aggregation services do not hold any business logic in themselves. They are simply a knot that ties together multiple services of any number. They can have any layer of services as dependencies, and it mainly exposes the call to these services accordingly. Here is a code example of an aggregation service:

```
public async ValueTask ProcessStudentAsync(Student student)
{
    await
this.studentRegistrationCoordinationService.RegisterStudentAsync(student);
    await this.studentRecordsCoordinationService.AddStudentRecordAsync(student);
    ...
    ...
    await
this.anyOtherStudentRelatedCoordinationService.DoSomethingWithStudentAsync(student)
;
}
```

As the snippet shows above, an Aggregation service may have any number of calls in any order without limitation. There may be occasions where you may or may not need to return a value to your exposure layers depending on the overall flow and architecture, which we will discuss shortly in this chapter. More importantly, aggregation services should be distinct from orchestration services or variants.

2.4.1 On The Map

Aggregation services always sit on the other end of a core business logic layer. They are the last point of contact between the exposure layers and logic layers. Here is a diagram visualization of where an Aggregation service can be found in the architecture:



Let's discuss the characteristics of Aggregation services.

2.4.2 Characteristics

Aggregation services mainly exist when multiple services share the same contract or primitive types of the same contract, requiring a single exposure point. They mainly exist in hyper-complex applications where multiple services (usually

orchestration or higher but can be lower) require one single point of contact through exposure layers. Let's discuss the main characteristics of Aggregation services in detail.

2.4.2.0 No Dependency Limitation

Unlike any other service, Aggregation services can have any number of dependencies as long as these services have the same variation. For instance, an Aggregation service cannot aggregate between an Orchestration service and a Coordination service. It's a partial Florance-like pattern where services must have the same variation but are not necessarily limited by the number.

The dependencies for Aggregation services are not limited because the service doesn't perform any level of business logic between these services. It doesn't care what these services do or require. It only focuses on exposing these services regardless of what they were called before or after.

Here's what an Aggregation service test would look like:

```
[Fact]
private async Task ShouldProcessStudentAsync()
{
    // given
    Student randomStudent = CreatedRandomStudent();
    Student inputStudent = randomStudent;

    // when
    await this.studentAggregationService.ProcessStudentAsync(inputStudent);

    // then
    this.studentRegistrationCoordinationServiceMock.Verify(service =>
        service.RegisterStudentAsync(student),
            Times.Once);

    this.studentRecordsCoordinationServiceMock.Verify(service =>
        service.AddStudentRecordAsync(student),
            Times.Once);
    ...
    ...

    this.anyOtherStudentRelatedCoordinationServiceMock.Verify(service =>
        service.DoSomethingWithStudentAsync(student),
            Times.Once);

    this.studentRegistrationCoordinationServiceMock.VerifyNoOtherCalls();
```

```
    this.studentRecordsCoordinationServiceMock.VerifyNoOtherCalls();
    ...
    ...
    this.anyOtherStudentRelatedCoordinationServiceMock.VerifyNoOtherCalls();
}
```

As you can see above, we only verify and test for the aggregation aspect of calling these services. No return type is required in this scenario, but there might be one in the pass-through scenarios, which we will discuss shortly.

An implementation of the above test would be as follows:

```
public async ValueTask ProcessStudentAsync(Student student)
{
    await this.studentRegistrationCoordinationService.AddStudentAsync(student);
    await this.studentRecordsCoordinationService.AddStudentRecordAsync(student);
    ...
    ...
    await
this.anyOtherStudentRelatedCoordinationService.DoSomethingWithStudentAsync(student)
;
}
```

2.4.2.1 No Order Validation

By definition, Aggregation services are naturally required to call several dependencies with no limitation. The order of calling these dependencies is also not a concern or a responsibility for Aggregation services because the call-order verification is considered a core business logic, which falls outside the responsibilities of an Aggregation service. That includes both the natural order of verification and the enforced order of verification, as we explained in section 2.3.3.0.1 in the previous chapter.

It violates The Standard to use simple techniques like a mock sequence to test an Aggregation service. It is also a violation to verify reliance on the return value of one service call to initiate a call to the next. These responsibilities are more likely to fall on the next lower layer of an Aggregation service for any orchestration-like service.

2.4.2.2 Basic Validations

Aggregation services are still required to validate whether or not the incoming data is structurally valid at a higher level. For instance, an Aggregation service that takes a `Student` object as an input parameter will only validate if the `student` is `null`.

But that's where it all stops.

There may be an occasion where a dependency requires a property of an input parameter to be passed in, in which case it is also permitted to validate that property value structurally. For instance, if a downstream dependency requires a student name to be passed in. An Aggregation service will still be required to validate if the `Name` is `null`, empty, or just whitespace.

2.4.2.3 Pass-Through

Aggregation services are not required to implement aggregation by performing multiple calls from one method. They can also aggregate by offering pass-through methods for multiple services. For instance, assume we have `studentCoordinationService`, `studentRecordsService`, and `anyOtherStudentRelatedCoordinationService` where each service is independent in terms of business flow. So, an aggregation here is only at the level of exposure, not necessarily the execution level.

Here's a code example:

```
public partial class StudentAggregationService
{
    ...

    public async ValueTask<Student> AddStudentAsync(Student student)
    {
        ...

        return await this.studentCoordinationService.RegisterStudentAsync(student);
    }

    public async ValueTask<Student> AddStudentRecordAsync(Student student)
    {
        ...

        return await
this.studentRecordsCoordinationService.AddStudentRecordAsync(student);
    }

    ...
    ...

    public async ValueTask<Student> DoSomethingWithStudentAsync(Student student)
    {
        ...
```

```
        return await
this.anyOtherStudentRelatedCoordinationService.DoSomethingWithStudentAsync(student)
;
    }
}
```

As you can see above, each service uses the Aggregation service as a pass-through. There's no need for an aggregated routines call in this scenario, but it would still be a very valid scenario for Aggregation services.

2.4.2.4 Optionality

It is essential to mention here that Aggregation services are optional. Unlike foundation services, Aggregation services may or may not exist in architecture. Aggregation services are there to solve a problem with abstraction. This problem may or may not exist based on whether the architecture requires a single exposure point at the border of the core business logic layer. This single responsibility of Aggregation services makes it much simpler to implement its task and perform its function efficiently. Aggregation services being optional is more likely than any other lower-level services, even in the most complex of applications out there.

2.4.2.5 Routine-Level Aggregation

If an aggregation service has to make two different calls from the same dependency amongst other calls, it is recommended that it aggregate for every dependency routine. But that's only from a clean-code perspective, and it doesn't necessarily impact the architecture or the result.

Here's an example:

```
public async ValueTask ProcessStudentAsync(Student student)
{
    await this.studentCoordinationService.AddStudentAsync(student);
    await ProcessStudentRecordAsync(student);
}

private async ValueTask ProcessStudentRecordAsync(Student student)
{
    await this.studentRecordCoordinationService.AddStudentRecordAsync(student);
    await
this.studentRecordCoordinationService.NotifyStudentRecordAdminsAsync(student);
}
```

As previously mentioned, this organizational action doesn't warrant any change in testing or results.

2.4.2.6 Pure Dependency Contracts

An Aggregation service's most important rule/characteristic is that its dependencies (unlike orchestration services) must share the same contract. The input parameter for a public routine in any Aggregation service must be the same for all its dependencies. There may be occasions where a dependency may require a student id instead of the entire student, which is permitted with caution as long as the partial contract isn't a return type of another call within the same routine.

2.4.3 Responsibilities

An Aggregation service's primary responsibility is to offer a single point of contact between exposer components and the rest of the core business logic. But in essence, abstraction is the actual value Aggregation services offer to ensure any business component is pluggable into any system regardless of exposure style.

Let's talk about these responsibilities in detail.

2.4.3.0 Abstraction

An aggregation service successfully assumes responsibility when its clients or consumers have no idea what lies beyond the lines of its implementation. For example, an aggregation service could combine ten different services and expose a single routine in a fire-and-forget scenario.

But even in pass-through scenarios, Aggregation services abstract away any identification of the underlying dependency from exposers at all costs. It only sometimes happens, especially in terms of localized exceptions. Still, it is close enough to make the integration seem as if it is with one single service that's offering all the options natively.

2.4.3.1 Exceptions Aggregation

Aggregation services resemble orchestration-like services when mapping and aggregating exceptions from downstream dependencies. For instance, if `studentCoordinationService` is throwing, `StudentCoordinationValidationException` an Aggregation service would map that into `StudentAggregationDependencyValidationException`. This falls back into the concept of exception unwrapping and then wrapping of localized exceptions, which we discussed in detail in section 2.3.3.0.2 of this Standard.

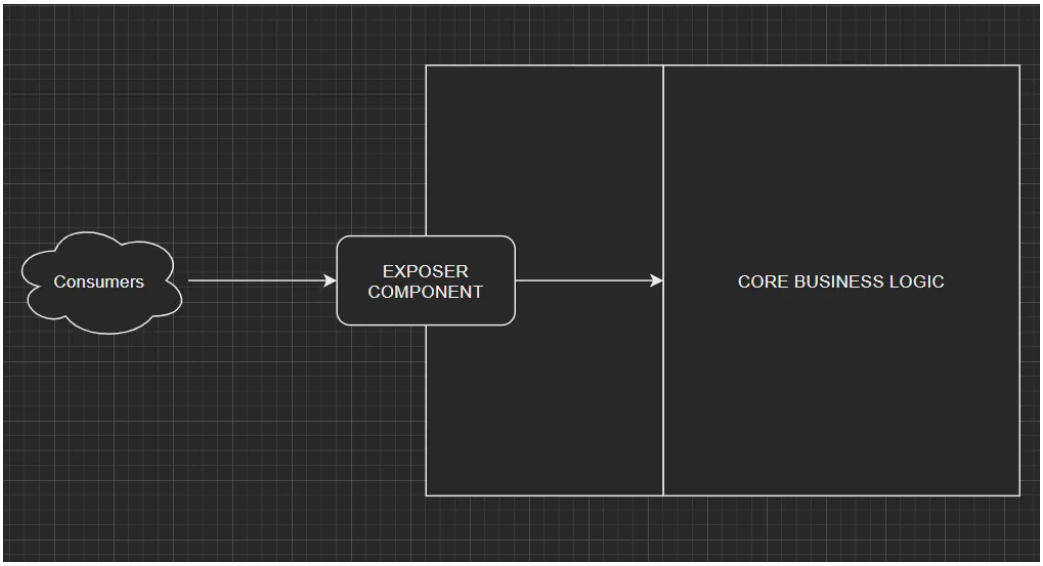
3

Exposers

3.0 Introduction

Exposers are disposable components in any system that expose your core business logic functionality by mapping its responses to a specific protocol. For instance, in RESTful communications, an API controller would be responsible for returning a `200` code for a successful response. The same applies to other protocols, such as gRPC, SOAP, or any other protocol of communication between distributed systems.

Exposer components are similar to Brokers. They are the last point of contact between the core business logic and the outside world. They are built with the intent that they will be detached from the current system at some point in time, allowing the same core logic to integrate with modern systems, protocols, or interfaces.



3.0.0 Purpose

In general, the primary responsibility of exposure components is to allow someone or something to interact with your business logic. For that core purpose, a precise mapping bit by bit to every possible response from your core business logic should be communicated cautiously with the consumer to that logic. I say cautiously because sometimes, specific internal issues in the system are not required to be exposed to the outside world. This mapping of responses can usually be a zero effort as the protocol and the language your code business logic communicates are the same as with libraries produced to run on the system that uses the same technologies or programming languages.

However, there are occasions where the outside world stateless protocol doesn't necessarily match the exact value of a response. In this case, it becomes an exposer component's responsibility to make a successful mapping both ways in and out of the system. API controllers are a great example of that. They will communicate a `4xx` issue when there's a validation exception of some type and return a deserialized JSON value if the communication is successful. However, there are also more details about problem details, error codes, and other levels of mapping and communication that we will discuss in upcoming chapters within this section.

3.0.0.0 Pure Mapping

The most important aspect of exposure components is that they are not allowed to communicate with brokers of any type. They are also not allowed to contain any form of business logic within them. By business logic here, I mean no sequence of routine calls, no iteration, or selection/decision-making. In the same way, it is with brokers only link an existing realm with the outside realm to achieve a specific value.

3.0.1 Types of Exposure Components

Exposure components are of three types: communication protocols, user interfaces, and IO routines. Let's discuss those briefly.

3.0.1.0 Communication Protocols

An exposure component that is a communication protocol can vary from simple RESTful APIs to SOAP communication or gRPC. It can also be a simple client in a library, where consumers would just install the library in their projects and consume your core logic through the client APIs. These examples are all of the same type of exposure components.

The differentiator here is that a communication protocol is usually event-based. It is triggered by an incoming communication and treated with a response of any kind. Communication protocols are usually for system-to-system integrations, but they can be accessible and understandable by humans for testing and debugging purposes.

3.0.1.1 User Interfaces

Another type of exposure component is the user interface. This can vary from Web, mobile, or desktop applications to simple command lines. User interfaces mainly target end-users for communication but can be automated by other systems, especially with command-line interfaces. In this day and age, user interfaces can also include virtual and augmented realities, metaverses, and any other form of software.

There are occasions where Human-Machine-Interfaces (HMI) can also fall into that level of exposure components. For instance, the buttons on a cellphone, keyboards we use daily, and any form of hardware that can interact directly with core business logic interfaces as an exposure component. The same theory applies to the Internet of Things (IoT) components and many others where a human has to utilize a component to leverage a specific capability to their advantage.

3.0.1.2 I/O Components

Some exposure components are not necessarily systems interfacing with another system, and they are not purposely designed to communicate with humans. They are daemons or IO-based components that do something in the background without a trigger. Usually, these components are time-based, and they may leverage existing protocols or interface directly with the core business logic, both of which are viable options.

3.0.2 Single Point of Contact

Exposure components are only allowed to communicate with one and only one service. Integrating with multiple services would turn an exposure component into either orchestration or aggregation services, which are not allowed to exist as core logic in that realm of exposure.

The single point of contact rule also ensures the ease of disposability of the exposure component itself. It ensures the integration is simple and single-purposed enough with controlled dependencies (only one) that it can be rewired to virtually any protocol at any point with the least cost possible.

3.0.3 Examples

Let's take API controllers as an example of a real-world exposure component in any system.

```
[HttpPost]
public async ValueTask<ActionResult<Student>> PostStudentAsync(Student student)
{
    try
    {
        Student registeredStudent =
            await this.studentService.RegisterStudentAsync(student);

        return Created(registeredStudent);
    }
    catch (StudentValidationException studentValidationException)
        when (studentValidationException.InnerException is
AlreadyExistsStudentException)
    {
        return Conflict(studentValidationException.InnerException);
```

```
    }
    catch (StudentValidationException studentValidationException)
    {
        return BadRequest(studentValidationException.InnerException);
    }
    catch (StudentDependencyException studentDependencyException)
    {
        return InternalServerError(studentDependencyException);
    }
    catch (StudentServiceException studentServiceException)
    {
        return InternalServerError(studentServiceException);
    }
}
```

The code snippet above is for an API method that `POST` a student model into the core business logic of a schooling system (OtripleS). In a technology like ASP.NET, controllers handle mapping incoming JSON requests into the `Student` model so that the controller can utilize that model with an integrated system.

However, you will also see that the controller code tries to map every possible categorical exception into its respective REST protocol. This is just a simple snippet to show what an exposure component may look like. But we will talk more about the rules and conditions for controllers in the next chapter of The Standard.

3.0.4 Versioning

3.0.4.0 Release Versioning

Release versioning governs how a system communicates what kind of change has been made at the release level. The Standard uses a strict four-segment format for all release versions:

```
v1.2.3.4
```

Each segment carries a fixed and non-negotiable meaning:

Segment	Meaning
1	Model change
2	Service or routine change

Segment	Meaning
3	Bug fix or configuration change
4	Automated build version

When multiple change types occur in the same release, the highest-order change wins and all lower segments reset to zero. Given a current version of `v1.2.3.4`, the expected next versions for each change type are:

- model change → `v2.0.0.0`
- service or routine change → `v1.3.0.0`
- bug fix or configuration change → `v1.2.4.0`
- automated build of the same code → `v1.2.3.5`
- model + service change together → `v2.0.0.0`

The version number must describe what kind of change happened, not merely that a change happened. That means a model change is never hidden inside the service segment, a service change is never hidden inside the bug or configuration segment, and a build increment is never used to represent a code change.

3.0.4.1 API Versioning

API versioning governs how routes communicate the model and behavior version to the consumers of a system. The initial API is exposed without any version prefix, keeping the route clean and straightforward:

```
api/Students
```

This is the V0 route. When a model change occurs, the route must surface the model version so that consumers understand they are interacting with a structurally different shape:

```
api/V1/Students
```

When the service behavior changes for an already-versioned model, the route surfaces both the model version and the behavior version together as a pairing:

```
api/V1.1/Students
```

In this shape, the number before the dot identifies the model version and the number after the dot identifies the behavior version. This pairing gives consumers a precise signal about both the shape of the data they will receive and the behavior they should expect from the service. If only the behavior changes but the model remains at version zero, the route becomes `api/V0.1/Students` — the model version is still zero, but the behavior version has incremented to one.

3.0.4.2 Deprecation

Deprecation signals that a version of an API or piece of code is no longer the recommended path forward and will eventually be removed. For deprecation to be useful, it must be visible. Consumers must be able to see that an API is deprecated, understand when it is expected to sunset, and know where to go for migration guidance. They must also be given enough time to act before the deprecated version disappears.

For APIs, a valid approach is to surface deprecation metadata through response headers using an attribute applied directly to the controller action:

```
[ApiController]
[Route("api/[controller]")]
public class SampleController : ControllerBase
{
    [HttpGet]
    [DeprecatedApi(
        Sunset = "2024-12-31",
        Warning = "This API is deprecated. Please migrate to v2.",
        Link = "https://example.com/deprecation-info")]
    public IActionResult GetSampleData()
    {
        return Ok(new { message = "Sample data" });
    }
}
```

The `Attrify` NuGet library provides the `DeprecatedApi` attribute and handles the emission of the appropriate deprecation headers automatically, making the lifecycle of an API a compile-time concern rather than a runtime configuration task.

For code that is no longer maintained or supported, the standard `[Obsolete]` attribute makes deprecation visible to maintainers and consumers at compile time:

```
[Obsolete("This version is deprecated and no longer maintained.")]
```

3.0.4.3 Capabilities

Capabilities are a discovery mechanism that allows consumers to understand what a system or provider supports before they ever attempt to invoke it. Without a capabilities surface, a consumer has no choice but to call an operation and react to whatever failure or exception comes back. With one, the consumer can check first and decide safely. This aligns directly with the broader Standard principle of avoiding invalid operations rather than reacting to them after the fact.

In a provider-based architecture, each concrete provider exposes a capabilities property that enumerates the operations it supports for each resource. The consumer queries this property before invoking any operation, making the interaction deterministic and safe:

```
IStudentProvider provider = new StudentProvider();

if (!provider.SupportsOperation("Student", "Add"))
{
    return;
}

await provider.Students.AddAsync(new Student());
```

Capabilities may also be surfaced through Swagger, middleware-based metadata, bespoke capabilities endpoints, or provider-specific capabilities endpoints, depending on the architecture of the system. The goal in every case is the same: turn runtime uncertainty into a deterministic check that the consumer can perform before committing to a call.

3.0.5 Summary

In summary, exposure components are very thin layers that don't contain any intelligence or logic. They are not meant to orchestrate or call multiple core business logic services. They only focus on the duplex mapping aspect of communication between one system and another.

3.1

Communication Protocols

3.1.0 Introduction

In the exposure realm, one of the most common methodologies for building a communication structure between several systems is using a communication protocol. These protocols have evolved over the years from SOAP to REST to many other communication protocols and principles that manifested their technologies to expose APIs to distributed systems.

In the .NET world, technology has evolved with the evolution of architecture from SOA with WCF to Microservices with REST. The evolution continues, but the principles change less often. In these upcoming chapters, we will discuss the most common communication protocols with a standardized way to implement them for enterprise-level applications.

3.1.0.0 Principles & Rules

Communication protocols are required to accomplish two things when integrating with core business logic. Results communication and Error reporting. Let's talk about those briefly:

3.1.0.0.0 Results Communication

Any communication protocol must satisfy the principle of returning a core business logic result. This result can be serialized into a unified language like `JSON` or communicated as is. In the case of API libraries, there is usually no need to serialize and deserialize data. However, that comes with the limitation that only technologies that integrate with these libraries can benefit from it.

Communicating results may also be encapsulated with a status of some kind. In the case of RESTful API communications, a `200` code can accompany the returned serialized `JSON` result. These codes allow the consumers to understand the next

course of action. Some `2xx` results may require a delayed action if the response is just `Accepted` but not necessarily `Created`.

3.1.0.0.1 Error Reports

The core business logic should provide a detailed report of all the validation errors in a particular request. The communication protocols' responsibility is to represent these error reports in their original form or serialize them in a language easily deserialized and convertible back into the Exception original form on the client side.

Serialized error reports shall also have their own codes so the client knows the next course of action. We recommend following a standardized way of communicating errors with documentation, preferably to help guide consumers in developing the best clients for these APIs.

3.1.0.1 Common Types

Let's explore some of the most common types of communication protocols in this section.

3.1.0.1.0 REST

REST is a Representational state transfer protocol with certain constraints that explicitly define the form of communication, error reporting, and its very stateless nature. When this Standard was authored, RESTful APIs were the most common form of communication between distributed systems. RESTful APIs are technology-agnostic. Any technology or a programming language can implement them, and they allow these technologies to communicate with each other statelessly without any hard dependency on the server or the client's choice of technology.

3.1.0.1.1 Libraries

The other most common communication protocol is APIs implemented within libraries. For instance, NuGet packages are published and distributed libraries that allow developers to leverage a localized core business logic or communicate with an external resource to achieve a specific goal.

3.1.0.1.2 Other Types

There are several other types of communication protocols. Some are older, and others are about to emerge in the software industry. These types are like SOAP with manifestations like WCF, gRPC, GraphQL, and several others.

We will mainly focus on RESTful APIs as a more common communication protocol, with a brief touch on Libraries. As we evolve and learn further, so will our Standard, which will include more and more different communication protocols and develop in terms of patterns.

Let's start with RESTful APIs as a communication protocol and then explore the different aspects of the exposer component.

3.1.1

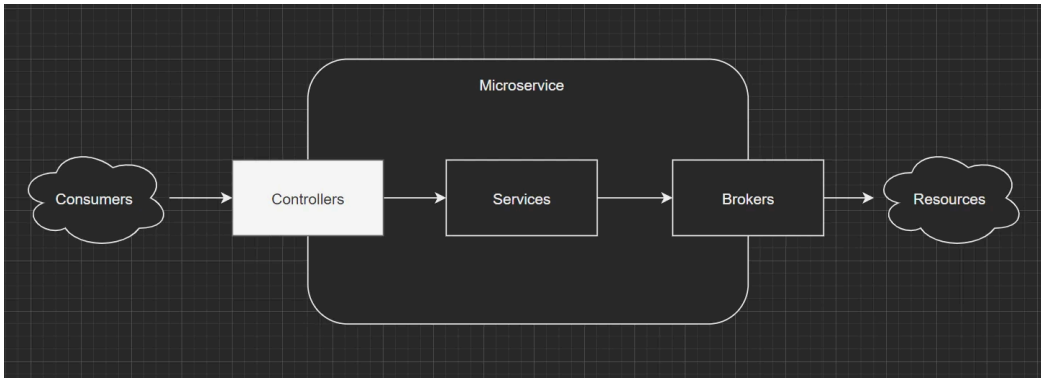
RESTful APIs

3.1.1.0 Introduction

RESTful API controllers are liaisons between the core business logic layer and the outside world. They sit on the other side of any application's core business realm. In a way, API Controllers are just like Brokers. They ensure a successful integration between our core logic and the rest of the world.

3.1.1.1 On the Map

Controllers sit at the edge of any system, regardless of whether this system is a monolithic platform or a simple microservice. API controllers today even apply to smaller lambdas or cloud functions. They act as a trigger to access these resources in any system through REST.



The consumer side of controllers can vary. In production systems, these consumers can be other services that require data from a particular API endpoint. They can be libraries built as wrappers around the controller APIs to provide a local

resource with external data. However, consumers can also be engineers testing endpoints and validating their behaviors through swagger documents.

3.1.1.2 Characteristics

Several rules and principles govern the implementation of RESTful API endpoints. Let's discuss those here.

3.1.1.2.0 Language

Controllers speak a different language when it comes to implementing their methods than services and brokers. For instance, if a broker that interfaces with a storage uses a language such as `InsertStudentAsync` and its corresponding service implementation uses something like `AddStudentAsync`, the controller equivalent will use a RESTful language such as `PostStudentAsync`.

A common controller would use only a handful of terminologies to express a particular operation. Let's draw the map here for clarity:

Controllers	Services	Brokers
Post	Add	Insert
Get	Retrieve	Select
Put	Modify	Update
Delete	Remove	Delete

But it must be important to understand that these operations can be extended and modified to fit the operation we are performing. We will talk about that shortly. The language controllers speak is called HTTP Verbs. Their range is wider than the aforementioned CRUD operations. For instance, there is PATCH, which allows API consumers to update only portions of a particular document. From my experience in productionized applications, PATCH is rarely used today. However, I may expand on them in a special section in future versions of The Standard.

3.1.1.2.0.0 Beyond CRUD Routines

As we mentioned before, a controller can interface with more than just a foundation service. They can interface with higher-order business logic functions. For instance, a processing service may offer an `Upsert` routine. In this case, a typical Http Verb wouldn't be able to satisfy a combinational routine such as an `Upsert`. In this case, we introduce custom http verbs that can support the aforementioned operation.

Custom http verbs can be anything as long as it doesn't conflict with an existing verb or contains non-supported characters. For instance, if we have a system that generate barcodes for some products, the generation process doesn't quite fit a `POST` or a `GET`. Especially when we factor in that these http verbs/methods might already be used for the same route for storing and retrieving barcodes from the system. This is a case where a new http verb can be introduced as `GENERATE` where you can pass the same route with a different verb that a server would understand.

We introduced this capabilty in RESTFulSense library where you can simply define your controller method as follows:

```
[HttpCustom("GENERATE")]
public ActionResult<Barcode> GenerateBarcode() =>
    this.barcodeProcessingService.Generate();
```

A consumer client can call that endpoint also as follows:

```
Barcode generatedBarcode =
    await restfulHttpClient.SendHttpRequestAsync<Student>(
        method: "GENERATE",
        relativeUrl: "api/barcodes",
        cancellationToken: someOrNoneCancellationToken);
```

The same operations can also be done without using a particular library with the native `HttpClient`.

3.1.1.2.0.1 Similar Verbs

Sometimes, especially with basic CRUD operations, you will need the same Http Verb to describe two different routines. For instance, integrating with both `RetrieveById` and `RetrieveAll` resolves to a `Get` operation on the RESTful realm.

In this case, each function will have a different name and route while maintaining the same verb as follows:

```
[HttpGet]
public async ValueTask<ActionResult<IQueryable<Student>>> GetAllStudentsAsync()
{
    ...
}

[HttpGet("{studentId}")]
public async ValueTask<ActionResult<Student>> GetStudentByIdAsync(Guid studentId)
{
    ...
}
```

As you can see above, the differentiator here is simultaneously the function name `GetAllStudents` versus `GetStudentByIdAsync` and the `Route`. We will discuss routes shortly, but the central aspect is the ability to implement multiple routines with different names, even if they resolve to the same Http Verb.

3.1.1.2.0.2 Routes Conventions

RESTful API controllers are accessible through routes. A route is simply a URL combined with an Http verb, so the system knows which routine it needs to call to match that route. For instance, if I need to retrieve a student with id `123`, my API route would be `api/students/123`. And if I want to retrieve all the students in some system, I could just call `api/students` with the `GET` verb.

3.1.1.2.0.2.0 Nouns & Verbs

Routes should never have verbs in them. That's the responsibility of the http verb. For instance, we should never name a route as such: `api/students/get` that is a violation of the naming conventions of The Standard. The rule here is that routes should also have nounes, and http methods should always have verbs. Http methods with customization as mentioned above could have endless number of custom methods against the very same route.

The Standard also enforces the single-noun principle. Routes should not combine mulitple nouns. For instance, instead of saying: `api/studentsubmissions` we should say: `api/student-submissions`. On that same line, retreiving submissions for a particular student can be represented as follows:

`api/students/{studentId}/submissions` the breakdown here is necessary to imply the intersection between resources compared to pulling everything in storage.

3.1.1.2.0.2.1 Controller Routes

The controller class in a simple ASP.NET application can be set at the top of the controller class declaration with a decoration as follows:

```
[ApiController]
[Route("api/[controller]")]
public class StudentsController
{
    ...
}
```

The route there is a template that defines the endpoint to start with `api` and trail by omitting the term "Controller" from the class name. So `StudentsController` would end up being `api/students`. All controllers must have a plural version of the contract they are serving. Unlike services where we say `StudentService`, controllers would be the plural version with `StudentsController`.

3.1.1.2.0.2.2 Routine/Method-Specific Routes

The same idea applies to methods within the controller class. As the code snippet above says, we decorated `GetStudentByIdAsync` with an `HttpGet` decoration with a particular route identified to append to the existing controller overall route. For instance if the controller route is `api/students`, a routine with `HttpGet("{studentId})` would result in a route that looks like this: `api/students/{studentId}`.

The `studentId` then would be mapped in as an input parameter variable that *must* match the variable defined in the route as follows:

```
[HttpGet("{studentId}")]
public async ValueTask<ActionResult<Student>> GetStudentByIdAsync(Guid studentId)
{
    ...
}
```

But sometimes, these routes are not just URL parameters. Sometimes, they contain a more specific parent resources information within them. For instance, we want to post a library card against a particular student record. Our endpoint would look like `api/students/{studentId}/librarycards` with a `POST` verb. In this case, we have to distinguish between these two input parameters with proper naming as follows:

```
[HttpPost("{studentId}/librarycards")]
public async ValueTask<ActionResult<LibraryCard>> PostLibraryCardAsync(Guid
studentId, LibraryCard libraryCard)
{
    ...
}
```

3.1.1.2.0.2.3 Plural-Singular-Plural

When defining routes in a RESTful API, it is important to follow the global naming conventions for these routes. The general rule is to access a collection of resources, then target a particular entity, then again access a collection of resources within that entity, and so on and so forth. For instance, in the library card example above, `api/students/{studentId}/librarycards/{librarycardId}`, we started by accessing all students and then targeting a student with a particular ID. We wanted to access all library cards attached to that student and then target a very particular card by referencing its ID.

That convention works perfectly in one-to-many relationships. But what about one-to-one relationships? Let's assume a student may have one and only one library card at all times. In which case, our route would still look something like this: `api/students/{studentId}/librarycards` with a `POST` verb, and an error would occur as `CONFLICT` if a card is already in place regardless of whether the Ids match or not.

3.1.1.2.0.2.4 Query Parameters & OData

But the route I recommend is the flat-model route. Where every resource lives on its own with its unique routes, in our case here, pulling a library card for a particular student would be as follows: `api/librarycards?studentId={studentId}` or use slightly advanced global technology such as OData where the query would be `api/librarycards?$filter=studentId eq '123'`.

Here's an example of implementing basic query parameters:

```
[HttpPost()]
public async ValueTask<ActionResult<LibraryCard>> PostLibraryCardAsync(Guid
studentId, LibraryCard libraryCard)
{
    ...
}
```

On the OData side, an implementation would be as follows:

```
[HttGet]
[EnableQuery]
public async ValueTask<IQueryable<LibraryCard>> GetAllLibraryCardsAsync()
{
    ...
}
```

The same idea applies to `POST` for a model. Instead of posting towards: `api/students/{studentId}/librarycards` - we can leverage the contract itself to post against `api/librarycards` with a model that contains the student id within. This flat-route idea can simplify the implementation and aligns perfectly with the overall theme of The Standard. We are keeping things simple.

3.1.1.2.0.2.5 X-WWW-Form-UrlEncoded Parameters

The Standard enfornces the concept of single-entity. For instance, we can't have a method as follows in a Standard-compliant system:

```
ValueTask<Teacher> GetTeachersByStudentAsync(Student student);
```

The above is considered a violation because the service that supports this routine explicity handles multiple models or entity types. But The Standard also permits passing primitive parameters such as `string`, `bool` or any other primitive or native type. Controllers/API can also support the same pattern though x-www.form-urlencoded parameters as follows:

On the controller side, you can implement `x-www-form-urlencoded` as follows:

```
    [HttpPost("login")]
    [Consumes("application/x-www-form-urlencoded")]
    public async ValueTask<ActionResult<UserAuthentication>> PostLoginUserAsync(
        [FromForm] string username,
        [FromForm] string password)
```

```
    {
        ....
    }
```

On the consumer side, the implementation would be:

```
    var formUrlEncodedContent =
         new FormUrlEncodedContent(new[]
         {
            new KeyValuePair<string, string>("username", username),
            new KeyValuePair<string, string>("password", password)
         });

    HttpResponseMessage httpResponseMessage =
        await this.apiClient.ExecuteHttpCallAsync(this.httpClient.PostAsync(
            requestUri: $"{userAuthenticationsRelativeUrl}/login",
            content: formUrlEncodedContent));
```

The rule to Services applies to Controllers as well, a routine at this level cannot accept more than 3 parameters - and beyond that point engineers must design the system to accept an actual entity or model and return the same model in the response.

3.1.1.2.1 Codes & Responses

Responses from an API controller must be mapped towards codes and responses. For instance, if we are trying to add a new student to a schooling system. We are going to `POST` a student, and in return, we receive the same body we submitted with a status code `201`, which means the resource has been `Created`.

There are three main categories into which responses can fall. The first is the success category. Both the user and the server have done their part, and the request has been successful. The second category is the User Error Codes, where the user request has an issue of any type. In this case, a `4xx` code will be returned with a detailed error message to help users fix their requests to perform a successful operation. The third case is the System Error Codes, where the system has run into an issue of any type, internal or external, and it needs to communicate a `5xx` code to indicate to the user that something internally has gone wrong with the system and they need to contact support.

Let's talk about those codes and their scenarios in detail here.

3.1.1.2.1.0 Success Codes (2xx)

Success codes indicate a resource has been created, updated, deleted, or retrieved. In some cases, it indicates that a request has been submitted successfully in an eventual consistent manner that may or may not succeed. Here are the details for each:

Code	Method	Details
200	Ok	Used for successful GET, PUT, and DELETE operations.
201	Created	Used for successful POST operations
202	Accepted	Used for request that was delegated but may or may not succeed

Here are some examples for each:

In a retrieve non-post scenario, it's more befitting to return an `Ok` status code as follows:

```
[HttpGet("{studentId}")]
public async ValueTask<ActionResult<Student>> GetStudentByIdAsync(Guid studentId)
{
    Student retrievedStudent =
        await this.studentService.RetrieveStudentByIdAsync(studentId);

    return Ok(retrievedStudent);
}
```

But in a scenario where we have to create a resource, a `Created` is more befitting for this case as follows:

```
[HttpPost)]
public async ValueTask<ActionResult<Student>> PostStudentAsync(Student student)
{
    Student retrievedStudent =
        await this.studentService.AddStudentAsync(student);

    return Created(student);
}
```

In eventual consistency cases, where a resource posted is not persisted yet, we enqueue the request and return an `Accepted` status to indicate a process will start:

```
[HttpPost)]
public async ValueTask<ActionResult<Student>> PostStudentAsync(Student student)
```

```
{
    Student retrievedStudent =
        await this.studentEventService.EnqueueStudentEventAsync(student);

    return Accepted(student);
}
```

The Standard rule for eventual consistency scenarios is to ensure the submitter has a token of some type so requestors can inquire about the status of their request using a different API call. We will discuss these patterns in a different book called The Standard Architecture.

3.1.1.2.1.1 User Error Codes (4xx)

This is the second category of API responses. In this category, a user request has an issue, and the system is required to help the user understand why their request was not successful. For instance, assume a client is submitting a new student to a schooling system. If the student ID is invalid, a `400` or `Bad Request` code should be returned with a problem detail that explains what exactly caused the request to fail.

Controllers are responsible for mapping the core layer categorical exceptions into proper status codes. Here's an example:

```
[HttpGet("{studentId}")]
public async ValueTask<ActionResult<Student>> GetStudentByIdAsync(Guid studentId)
{
    try
    {
        ...
    }
    catch (StudentValidationException studentValidationException)
    {
        return BadRequest(studentValidationException.InnerException)
    }
}
```

So, as shown in this code snippet, we caught a categorical validation exception and mapped it into a `400` error code, which is `BadRequest`. Access to the inner exception here is for the purpose of extracting a problem detail out of the `Data` property on the inner exception, which contains all the dictionary values of the error report.

But sometimes, controllers have to dig deeper. Catching a particular local exception, not just the categorical. For instance, say we want to handle `NotFoundStudentException` with an error code `404` or `NotFound`. Here's how we would accomplish that:

```
[HttpGet("{studentId}")]
public async ValueTask<ActionResult<Student>> GetStudentByIdAsync(Guid studentId)
{
    try
    {
        ...
    }
    catch (StudentValidationException studentValidationException)
        (when studentValidationException.InnerException is
NotFoundStudentException)
    {
        return NotFound(studentValidationException.InnerException)
    }
}
```

In the code snippet above, we had to examine the inner exception type to validate the localized exception from within. This is the advantage of the unwrapping and wrapping process discussed in section 2.3.3.0.2 of The Standard. The controller may examine multiple types within the same block as well as follows:

```
    ...
    catch (StudentCoordinationDependencyValidationException
studentCoordinationDependencyValidationException)
        (when studentValidationException.InnerException
            is NotFoundStudentException
            or NotFoundLibraryCardException
            or NotFoundStudentContactException)
    {
        return NotFound(studentValidationException.InnerException)
    }
    ...
```

With that in mind, let's detail the most common mappings from exceptions to codes:

Code	Method	Exception
400	BadRequest	ValidationException or DependencyValidationException
404	NotFound	NotFoundException

Code	Method	Exception
409	Conflict	AlreadyExistException
423	Locked	LockedException
424	FailedDependency	InvalidReferenceException

There are more `4xx` status codes out there. But they can either be automatically generated by the web framework, like in ASP.NET, or there are no useful scenarios for them yet. For instance, a `401` or `Unauthorized` error can be automatically generated if the controller endpoint is decorated with an authorization requirement.

3.1.1.2.1.2 System Error Codes (5xx)

System error codes are the third and last possible type of code that may occur or be returned from an API endpoint. Their main responsibility is to indicate that the API endpoint consumer is generally at no fault. Something terrible happened in the system, and the engineering team must get involved to resolve the issue. That's why we log our exceptions with a severity level at the core business logic layer so we know how urgent the matter may be.

The most common Http code that can be communicated on a server-side issue is the `500` or `InternalServerError` code. Let's take a look at a code snippet that deals with this situation:

```
[HttpGet("{studentId}")]
public async ValueTask<ActionResult<Student>> GetStudentByIdAsync(Guid studentId)
{
    try
    {
        ...
    }
    ...
    catch (StudentDependencyException studentDependencyException)
    {
        return InternalServerError(studentValidationException)
    }
}
```

In the above snippet, we ignored the inner exception and mainly focused on the categorical exception for security reasons. Primarily to not allow internal server information to be exposed in an API response other than something as simple as `Dependency error occurred, contact support.` Since the API consumer is required to perform no action whatsoever other than creating a ticket for the support team, Ideally, these issues should be caught out of Acceptance Tests, which we will discuss shortly in this chapter. But there are times where there's a server blip that may cause a memory leakage of some sort or any other internal infrastrucrual issues that won't be caught by end-to-end testing in any way.

The types of exceptions that may be handled are smaller regarding server errors. Here are the details:

Code	Method	Exception
500	InternalServerError	DependencyException or ServiceException
507	NotFound	InsufficientStorageException (Internal Only)

There's also an interesting case where two teams agree on a specific swagger document, and the back-end API development team decides to build corresponding API endpoints with methods yet to be implemented to communicate to the other team that the work has yet to start. In this case, the error code `501` is sufficient, just a code for `NotImplemented`.

It is also important to mention that the native `500` error code can be communicated in ASP.NET applications through the `Problem` method. We are relying on a library, `RESTFulSense`, to provide more codes than the native implementation can offer, but more importantly, to provide a problem detail serialization option and deserialization option on the client side.

3.1.1.2.1.3 All Codes

Other than the ones mentioned in previous sections, and for documentation purposes, here are all of the `4xx` and `5xx` codes an API could communicate according to the latest standardized API guidelines:

Status	Code
BadRequest	400

Status	Code
Unauthorized	401
PaymentRequired	402
Forbidden	403
NotFound	404
UrlNotFound	404
MethodNotAllowed	405
NotAcceptable	406
ProxyAuthenticationRequired	407
RequestTimeout	408
Conflict	409
Gone	410
LengthRequired	411
PreconditionFailed	412
RequestEntityTooLarge	413
RequestUriTooLong	414
UnsupportedMediaType	415
RequestedRangeNotSatisfiable	416
ExpectationFailed	417
MisdirectedRequest	421
UnprocessableEntity	422
Locked	423
FailedDependency	424
UpgradeRequired	426
PreconditionRequired	428
TooManyRequests	429

Status	Code
RequestHeaderFieldsTooLarge	431
UnavailableForLegalReasons	451
InternalServerError	500
NotImplemented	501
BadGateway	502
ServiceUnavailable	503
GatewayTimeout	504
HttpVersionNotSupported	505
VariantAlsoNegotiates	506
InsufficientStorage	507
LoopDetected	508
NotExtended	510
NetworkAuthenticationRequired	511

We will explore incorporating some of these codes in future revisions of The Standard as needed.

3.1.1.2.2 Single Dependency

Exposer components can have one and only one dependency. This dependency must be a Service component. It cannot be a Broker or any other native dependency that Brokers may use to pull configurations or any other type of dependencies.

When implementing a controller, the constructor can be implemented as follows:

```
[ApiController]
[Route("api/[controller]")]
public class StudentsController : RESTFulController
{
    private readonly IStudentService studentService;

    public StudentsController(IStudentService studentService) =>
        this.studentService = studentService;
```

```
    ...
    ...
}
```

3.1.1.2.3 Single Contract

This characteristic comes out of the box with the single dependency rule. If Services can only serve and receive one contract, then the same rule will apply to controllers. When passing in IDs or queries, they can return a contract or a list of objects with the same contract or portion of the contract.

3.1.1.3 Organization

Controllers should be located under the `Controllers` folder and belong within a `Controllers` namespace. However, controllers do not need to have their own folders or namespaces as they perform a simple exposure task.

Here's an example of a controller namespace:

```
namespace GitFyle.Core.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ContributionsController : RESTFulController
    {
        ...
    }
}
```

3.1.1.4 Home Controller

Every system should implement an API endpoint that we call `HomeController`. The controller's only responsibility is to return a simple message to indicate that the API is still alive. Here's an example:

```
using Microsoft.AspNetCore.Mvc;

namespace OtripleS.Web.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class HomeController : ControllerBase
```

```
    {
        [HttpGet]
        public ActionResult<string> Get() =>
            Ok("Hello Mario, the princess is in another castle!");
    }
}
```

Home controllers are not required to have any security. They open a gate for heartbeat tests to ensure the system as an entity is running without checking any external dependencies. This practice is very important to help engineers know when the system is down and quickly act on it.

3.1.1.5 Tests

There are three different types of tests that cover API controllers as well as any other exposure layer. These tests are: unit tests, acceptance tests and integration or end-to-end (E2E) tests. Integration tests can vary between smoke testing, availability testing, performance testing and many others. But for the purpose of this chapter, we will focus on unit and acceptance tests.

3.1.1.5.0 Unit Tests

Controllers have a similar type of logic that exists in Services. This logic is the mapping between exceptions coming from a dependency or internally and what these exceptions are being mapped to for the consumer of these APIs. For instance, a `StudentValidationException` can be mapped to a `BadRequest` status code. This logic is tested in unit tests. Let's take a look at an example:

Starting with the test before the implementation, let's assume we have a controller `StudentsController` that retrieves all students. When the call succeeds, the controller should return a `200 OK` status code. Let's write a test for that:

First, let's setup our `StudentsController` class as follows:

```
namespace School.Core.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class StudentsController : RESTFulController
    {
        private readonly IStudentService studentService;
```

```
        public StudentsController(IStudentService studentService) =>
            this.studentService = studentService;

        [HttpGet]
        public async ValueTask<ActionResult<IQueryable<Student>>>
GetAllStudentsAsync()
        {
            return NotImplemented(new NotImplementedException());
        }
    }
```

In the above code snippet, we initialized `StudentsController`, we inherited `RESTFulController` so we can have support for all possible status codes. But additionally, we created a `GetAllStudentsAsync` method that returns `NotImplemented` status code with a `NotImplementedException` exception. Notice the difference here between throwing the exception `NotImplementedException` at the Services layer compared to controllers.

Now, let's move on to writing a controller unit test. Let's setup our `StudentsControllerTest` class as follows:

```
    public partial class StudentsControllerTests : RESTFulController
    {
        private readonly Mock<IStudentService> studentServiceMock;
        private readonly StudentsController studentsController;

        public StudentsControllerTests()
        {
            this.studentServiceMock = new Mock<IStudentService>();

            this.studentsController = new StudentsController(
                studentService: this.studentServiceMock.Object);
        }

        ....
    }
```

In the above example, we did three important things:

1. We made sure the `SourcesControllerTests` class is partial so we can write other files that are still a part of this class but target particular areas and methods.
2. We inherited from `RESTFulController` which is a class that comes from `RESTFulSense` .NET library which we will use later to create the expected response such as `Ok(retrievedStudents)`.

3. We mocked the dependency so we don't actually call the `StudentService` but rather call a controlled mock so we can simulate responses and exceptions depends on the context of the unit test.

Now, let's write a unit test for `GetAllStudentsAsync` controller method as follows:

```
[Fact]
public async Task ShouldReturnOkOnGetAllStudentsAsync()
{
    // given
    List<Student> randomStudents =
        CreateRandomStudents();

    List<Student> returnedStudents =
        randomStudents;

    List<Student> expectedStudents =
        returnedStudents.DeepClone();

    OkObjectResult expectedObjectResult =
        Ok(expectedStudents);

    var expectedActionResult =
        new ActionResult<List<Student>>(
            expectedObjectResult);

    this.studentServiceMock.Setup(service =>
        service.RetrieveAllStudentsAsync())
            .ReturnsAsync(returnedStudents);

    // when
    ActionResult<List<Student>> actualActionResult =
        await this.studentsController
            .GetAllStudentsAsync();

    // then
    actualActionResult.ShouldBeEquivalentTo(
        expectedActionResult);

    this.studentServiceMock.Verify(service =>
        service.RetrieveAllStudentsAsync(),
            Times.Once);

    this.studentServiceMock.VerifyNoOtherCalls();
}
```

In the above test, just like we did with Services unit tests we did the following:

1. We created a list of random students to simulate a response from the service.

2. We cloned the list of students to create an expected response.
3. We created an `OkObjectResult` object to simulate the expected response from the controller.
4. We setup the `studentServiceMock` to return the list of students when `RetrieveAllStudentsAsync` is called.
5. We called the `GetAllStudentsAsync` method on the controller.
6. We verified that response `expectedActionResult` is equivalent to the actual response `actualActionResult`.
7. We verified that the `RetrieveAllStudentsAsync` method was called once.
8. Lastly, we wanted to verify that the controller isn't making any additional unnecessary calls from the dependency.

The above test will fail with expected code being `200 OK` but instead the actual is `501 Not Implemented`. Now, let's make that test pass as following:

```
namespace School.Core.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class StudentsController : RESTFulController
    {
        private readonly IStudentService studentService;

        public StudentsController(IStudentService studentService) =>
            this.studentService = studentService;

        [HttpGet]
        public async ValueTask<ActionResult<IQueryable<Student>>>
GetAllStudentsAsync()
        {
            List<Student> students =
                await this.studentService.RetrieveAllStudentsAsync();

            return Ok(students);
        }
    }
```

In the above code, we implemented `GetAllStudentsAsync` method and now our unit test will successfully pass.

3.1.1.5.1 Acceptance Tests

Here's an example of an acceptance test:

```
[Fact]
public async Task ShouldPostStudentAsync()
{
    // given
    Student randomStudent = CreateRandomStudent();
    Student inputStudent = randomStudent;
    Student expectedStudent = inputStudent;

    // when
    await this.otripleSApiBroker.PostStudentAsync(inputStudent);

    Student actualStudent =
        await this.otripleSApiBroker.GetStudentByIdAsync(inputStudent.Id);

    // then
    actualStudent.Should().BeEquivalentTo(expectedStudent);
    await this.otripleSApiBroker.DeleteStudentByIdAsync(actualStudent.Id);
}
```

Acceptance tests are required to cover every available endpoint on a controller and are responsible for cleaning up any test data after the test is completed. It is also important to mention that resources not owned by the microservice, like the database, must be emulated with applications such as `WireMock` and many others.

Acceptance tests are also implemented after the fact, unlike unit tests. An endpoint has to be fully integrated and functional before a test is written to ensure implementation success is in place.

[*] Controller Unit Tests

[*] Acceptance Tests (Part 1)

[*] Acceptance Tests (Part 2)

3.2

User Interfaces

3.2.0 Introduction

User Interfaces or UI are a type of exposer component mainly targeting humans for interaction with the core business layer, unlike Communication protocols primarily used in distributed systems. UIs are forever evolving in terms of technologies and methodologies with which humans can interact with any given system. This goes from web applications to virtual/augmented realities voice-activated systems and, more recently, brain-waves-activated systems.

Developing user interfaces can be much more challenging in terms of experiences. Today, there needs to be a global standard for intuitive understanding. It heavily relies on culture, commonalities, and many other forever-changing variables. This Standard will outline the principles and rules for building modular, maintainable, and pluggable UI components. However, there will be a different Standard for outlining user experiences, human interactions, and the theory of intuitiveness.

This Standard also briefly highlights specific guidelines regarding rendering choices, server, client, or hybrid, as is the case with the tri-nature of everything. Let's dive deeper into the principles and rules that govern building UI components.

3.2.0.0 Principles & Rules

Like every other exposer component type, UIs must be able to map processes, results, and errors to their consumers. Some of these UI components will require a test-driven approach. Some others are more like Brokers, just wrappers around third-party or native UI components. Let's discuss these principles here.

3.2.0.0.0 Progress (Loading)

The most important principle in building UI components is to develop intelligence to keep the user engaged while a particular process is running, such as a simple spinner or a progress bar to keep users informed at all times of what's happening

behind the scenes.

It violates The Standard to indicate progress if nothing is happening in the background. It falls into the practice of wasting end-users' time and lying to them about the actual status of the system. However, assuming the system is busy working on a particular request, three levels of communication can happen on an exposer component to communicate progress. Let's discuss those in detail:

3.2.0.0.0.0 Basic Progress

The basic progress approach is where you present status with a label like "Waiting ..." or a spinner with no further indication, which is the bare minimum of progress indication. No UI should freeze or stop their hanging while requests are being processed in the background, assuming an eventual consistency pattern is not attainable for the current business need.

Some web applications show a forever progress bar at the top of the page to indicate progress. From an experience perspective, depending on the visibility level of these progress bars, they may be challenging for end users to miss. Some other engineering teams have chosen to play a simple animation to keep users engaged with visual progress without any indication of the details of that progress.

3.2.0.0.0.1 Remaining Progress

Above the bare minimum, there is an indication of remaining time or progress to be completed before the request is processed. An indication such as "40% remaining" or something more specific like "5 minutes remaining ..." to help end-users understand or guestimate how long a time or effort is left; or, there are patterns where engineers would indicate how many tasks remain without showing what these tasks are.

Sometimes, a remaining progress update is as detailed as UI engineers can get. For instance, if you are downloading a file from the internet, you can't be more precise than "saying x percent of the bits remaining to be downloaded" with no further details. Some game developers also choose to visualize the internet speeds and available disk space to keep the end-user engaged in the system. These are all acceptable patterns in this Standard.

3.2.0.0.0.2 Detailed Progress

The highest level of reporting progress is the detailed progress type, where the UI component is fully transparent with its consumers by reporting every step of progress. This type of progress is more common in scientific applications. Engineers in debugging mode may enable a feature where all the underlying activity in the system is visualized through the UI.

This type helps end-users understand what is happening behind the scenes and allows them to communicate better details to support engineers to help them fix an issue if the process fails. However, this process is only sometimes preferred in terms of experience, considering that some details need to be hidden for security reasons.

In summary, selecting the correct type of progress indicator in a UI depends on the business flow, the type of users interacting with the system, and several other variables we will discuss in The Experience Standard.

3.2.0.0.1 Results

UI exposer components will report a result to indicate the completion of a particular request by end-users. Consider registering a new student in a schooling system. There are several ways to show the registration process has been completed successfully. Let's discuss those types of results visualization in detail here.

3.2.0.0.1.0 Simple

The simple indication of success is when the UI reports that the process was completed without further details. You may have seen some of these implementations for this type, such as "Thank you, request submitted" or something as simple as a checkmark with a visualization of green color that indicates success.

Simple results indications, especially with submitted requests rather than retrieved data, may add more details in the next course of action.

3.2.0.0.1.1 Partial Details

An overview of the nature of the request and where it stands in terms of status and timestamps: The other type of results or success indication is to present end-users with partial details. Partial details are usually helpful when it comes to

providing the end-user with a “ticket number” to help end-users follow up on their requests later to inquire about the status. This pattern is typical in e-commerce applications where every purchase request may be returned with a tracking number to help customers and customer support personnel assess the requests.

Detailed results can also be beneficial for visualizing the success process, especially with requests containing multiple parts. Larger requests, such as an application to join a university or the like, may include attachments, multiple pages of details, and confidential information such as payment details or social security numbers.

3.2.0.0.1.2 Full Details

Sometimes, one may prefer to report full details about the submitted request, especially with smaller requests that may help end-users review their requests. Some engineers may display full details as an extra confirmation step before submitting the request. However, full details can also include more than just the requested details. It could include a status update from the server a long with an assigned point of contact or an officer from maintenance and support teams.

It’s a violation of The Standard to redirect end-users at the submittal of their requests with no indication of what happened.

3.2.0.0.2 Error Reports

Error reports primary responsibility is to inform end-users of what happened, why it happened, and the next course of action. Some error reports don’t indicate any course of action, which can be a poor experience depending on the business flow. However, the bare minimum in error reporting is the basic indication of the error with essential details. Let’s talk about those types here.

3.2.0.0.2.0 Informational

The bare minimum of error reports is the informational type, which indicates an error occurred and why it happened, such as “Request failed. Try again” or “Request failed contact support.” There are also informational errors that are time-based, such as “Our servers are currently experiencing a high volume of requests. Please try again later.” These informational error reports are necessary to keep the end-user engaged with the system.

Informational error reports are governed by the context and the type of users receiving them. In a scientific application, the more details, the better. For some other systems, it is important to shift the technical language of the errors to a less technical language. For instance, we can't communicate: "Student id cannot be null, empty, or whitespace". We should select a more readable language, such as "Please provide a valid student Id".

3.2.0.0.2.1 Referential/Implicit Actions

The second type of error report is the referential type. When an error occurs, it automatically informs the support team and returns a reference of a support ticket to end users so they can follow up. You may see this often when video games fail to start or specific applications cannot initialize. Referential error reports are the best for particular business flows since they take care of all the actions, email the end user the reference number, and follow up within a couple of days to report the status.

The fewer actions a system requires users to take after a failure has occurred, the better. Since end-users have already accomplished their tasks in submitting requests, it becomes even more convenient if the original request is queued up, such as with high-volume enterprise systems, so end-users don't have to re-submit the same data.

3.2.0.0.2.2 Actionable

The second type of error report is the actionable report. Errors provide an additional action for the users to go further in their request. For instance, error reports can give a button to try again or submit other details requests back to the engineering and support teams.

Some reports will provide a different route to accomplish the same task in more hybrid legacy and modernized applications. These actionable reports are more convenient than informational reports. However, they would still require their end-users to take more actions and keystrokes, leading to some inconvenience.

3.2.0.0.3 Single Dependency

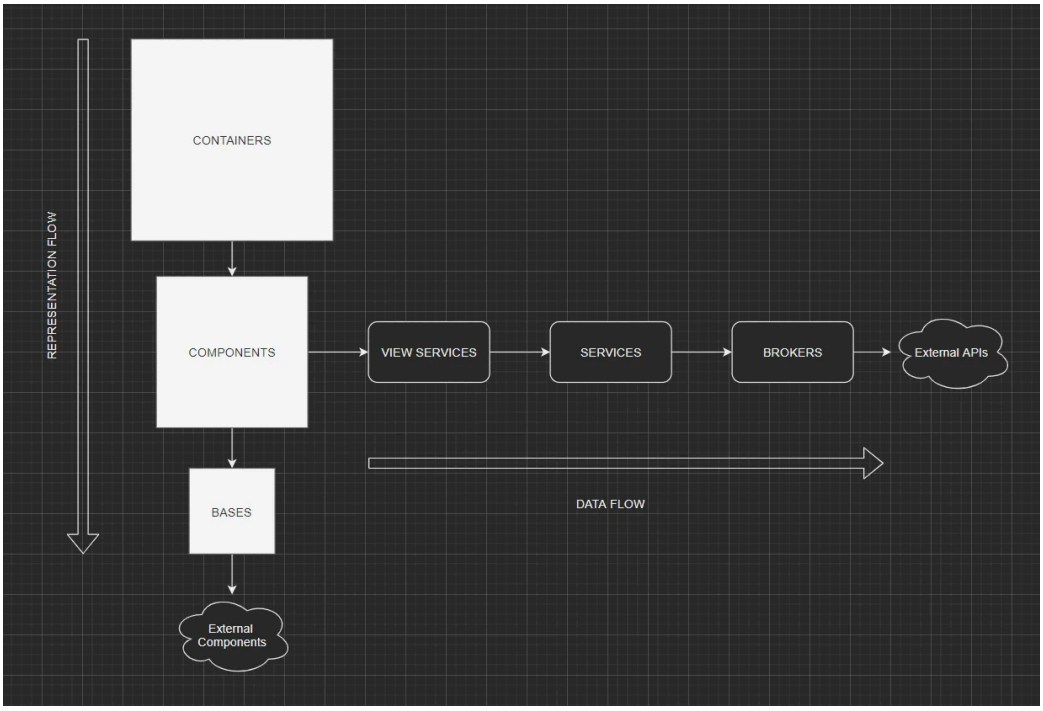
Any exposer component can only integrate with a single dependency at a time. For UI components, contract purity ensures that a UI is not given more data than it needs. To enforce this pattern, a new type of foundational-like service is

implemented, and all other details, such as audit fields, timestamps, and such, are taken care of away from the UI component's sight.

We will talk in detail about view services shortly as we progress talking about UI exposers.

3.2.0.0.4 Anatomy

Just like the data flow in any service. We have brokers → Services → Exposers. UI components also form their own data flow in terms of rendering. Let's take a look at the anatomy of UI exposers in this illustration:



UI exposer components, as shown above, can be Bases, Components, or Containers. Each of these types has a specific responsibility to ensure the maximum maintainability and pluggability of the system, according to The Standard. Let's discuss these three types here:

3.2.0.0.4.0 Bases

Base or Base Components are just like Brokers in the data flow. They are simple thin wrappers around native or 3rd party components. Their primary responsibility is to abstract away the hard dependency on non-local components to allow the system's configuration to switch to any other external or native UI components with the least effort possible.

Base components also make it easier to mock out any external or native component's behavior and focus the effort on ensuring the local component performs as expected. In the following chapter, we will discuss base components for web applications in Blazor and other technologies.

3.2.0.0.4.1 Components

UI Components are a hybrid between a Service and a Controller in the data pipeline. In a way, components contain *some* business logic in handling interactions with certain base components. However, they are also limited by integrating with one and only one view service. Components are test-driven. They require writing tests to ensure they behave as expected. However, they also contain almost no iteration, selection, or sequencing data logic.

The most important aspect of UI components is the intersection between the UI flow and the data flow. They are responsible for leveraging their data dependency (view services) and their base components to become easily pluggable into container components (like pages with routes in web applications).

3.2.0.0.4.2 Containers

Container components are orchestrators/aggregators of components. They are the actual route or page end-users interact with. Containers cannot have any level of UI logic in them, cannot leverage base components, and may have any number of UI components as the business flow requires.

As is the case with every category of components, containers cannot integrate with other containers. The rule applies across the board for every data or UI component.

3.2.0.0.5 UI Component Types

UI components come in all different shapes and sizes. The hosting environment and the type of devices that serve these components play a significant role in determining the technologies and capabilities a particular UI component may have. In this section, we will discuss the different types of UI components.

3.2.0.0.5.0 Web Applications

The most popular type of UI application is web application because of its ease of use. Web applications require no installation of any kind, and they are not dependent on the operating system running the system or the type of devices users may be using. They can run on PCs, tablets, mobile phones, TVs, and watches that support web browsing.

Web frameworks have evolved dramatically in the last few years due to their popularity, as mentioned earlier. Some frameworks allow engineers to write web applications in many programming languages. The evolution of web assembly has also opened the door for engineers to develop even more scalable frameworks with their preferred technologies and languages.

Server-side applications and client-side applications. In terms of rendering, web applications are developed in two different types. In the next few chapters of The Standard, we will discuss the advantages and disadvantages of each type in addition to the hybrid model.

3.2.0.0.5.1 Mobile Applications

The second most popular platform today for developing UIs is the mobile world. Developing mobile applications comes with challenges as they depend heavily on the operating system, the phone's size in terms of resolution, and the available native controls. Mobile applications are also always client-side apps. They are just like Desktop applications. They must be compiled, provisioned, and published to an app store so consumers can download, install, and leverage them daily.

The most significant advantage of mobile applications is that they allow offline interactions, such as mobile games, editing apps, and streaming services with offline capabilities. However, building mobile applications with web frameworks is becoming increasingly popular, as the web is a universal ecosystem that allows

end-users to experience software the same way on their PCs, browsers, and mobile applications. This trend will eventually enable engineers to develop systems for all ecosystems at the least possible cost.

3.2.0.0.5.2 Other Types

We may need to cover other types of UI components in our Standard. These types are console/terminal applications, desktop applications, video games, virtual/augmented reality software, wearable devices, and voice-activated systems. The world of Human-Machine-Interface HMI is evolving so rapidly in the age of the metaverse that we might need to create special chapters for these different types at some point.

3.2.1

Web Applications

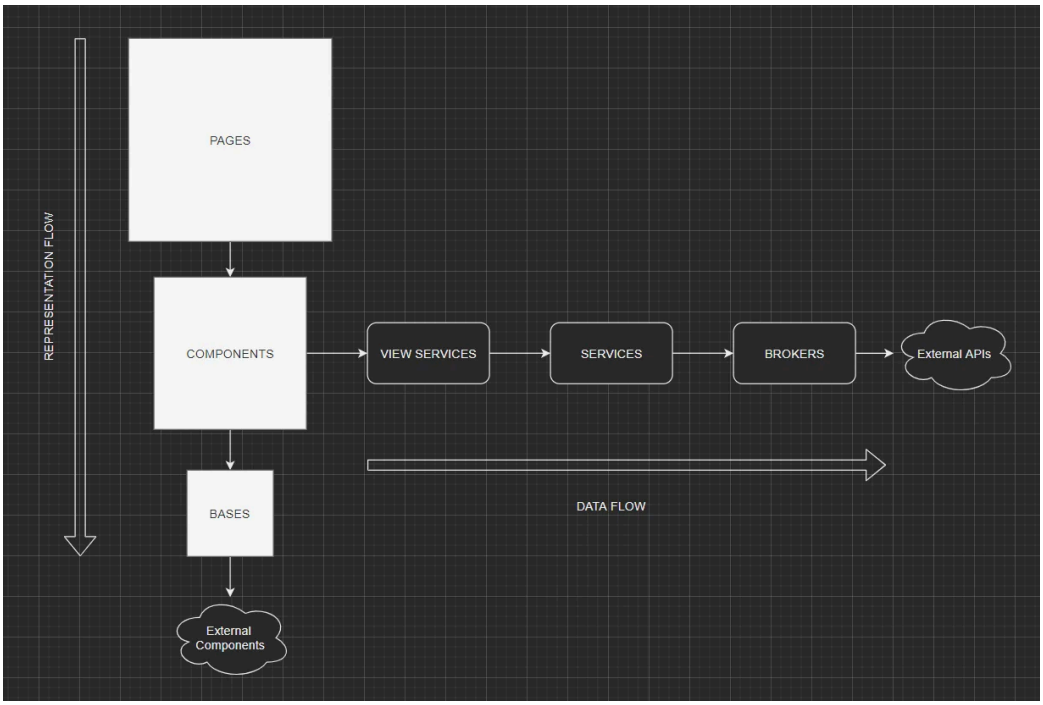
3.2.1.0 Introduction

Web applications are the most common type of exposer components today. They are much easier to use than other known exposer UI components in the software industry. The web software market is also much easier for engineers to publish and update than mobile applications, making it attractive for newer engineers. But more importantly, web applications have a much more diverse set of technologies than mobile applications.

This chapter will use Blazor technology to demonstrate implementing the Standard principles for web applications. However, as mentioned previously, the Standard is technology-agnostic, meaning it can be applied to any web technology without issues.

3.2.1.1 On the Map

Web applications are usually set at the other end of any system. They are the terminals that humans use to interact with the system. Let's take a look at where they are located on the map:



As shown above, web applications are similar to core APIs, except that they have a different group of components in terms of visualization, such as Pages, Components, and Bases. There's an intersection between two main flows in every web application. The presentation flow and the data/business flow. Depending on where a web application lives in terms of high-level architecture, its location determines whether its backend (BFF or Backend of Frontend) is a business flow or just data flow. Let's discuss these details in the characteristics section of this chapter.

3.2.1.2 Characteristics

Brokers, Services, View Services, Bases, Components, and Pages. Web applications usually have six essential components. Since we've already discussed the data flow components in the Services portion of The Standard, this section will discuss the UI aspect (Bases, Components, and Pages) with a slight detail about view services.

Let's discuss these characteristics here.

3.2.1.2.0 Anatomy

UI components consist of base components, core components, and pages. They all separate the responsibility of integration, rendering, and routing users to a particular UI functionality. These components are organized under the Views directory in a clear, domain-driven structure to enhance navigation, consistency, and scalability. The folder structure is as follows:

- **Views**
  - **Bases**: Contains base components that abstract native or third-party components.
  - **Components**: Includes categorized UI components (Foundations, Processings, Orchestrations, Aggregations).
  - **Pages**: Represents the end-user-facing pages.

Let's talk about these types in detail.

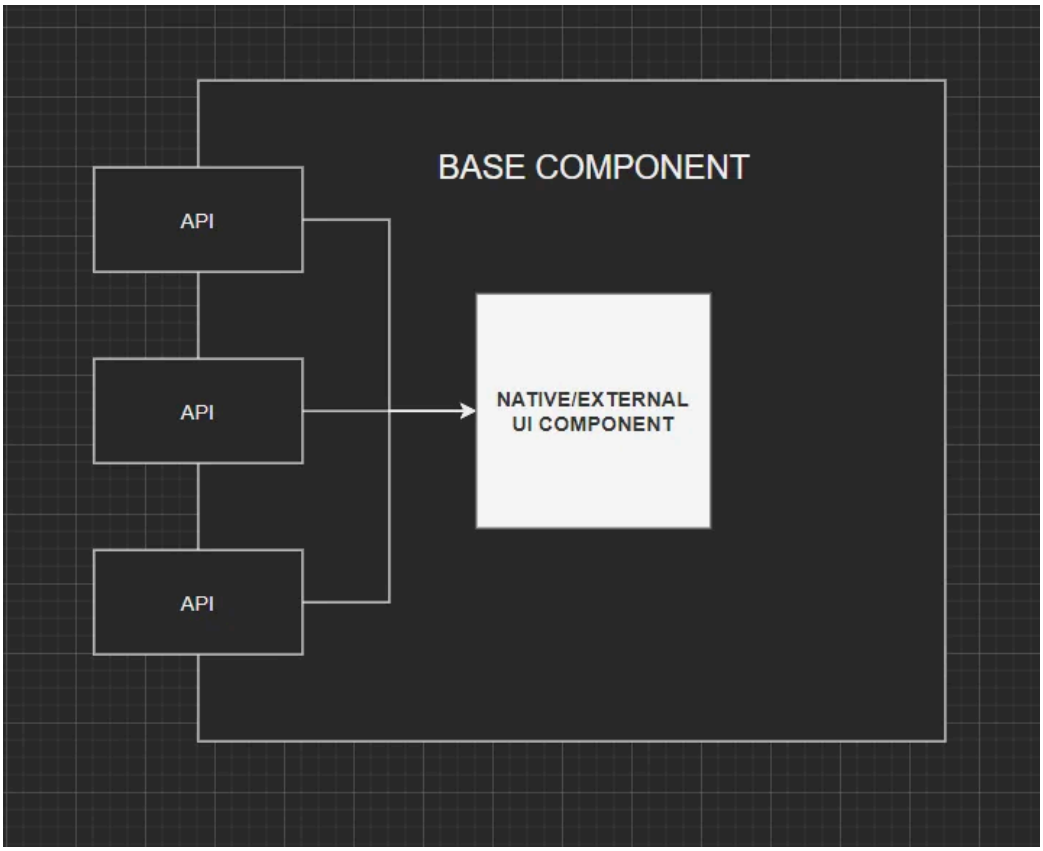
3.2.1.2.0.0 Base Component

Base components are just like brokers. They are wrappers around native or external UI components. Their primary responsibility is to abstract away any hard dependency on non-local UI capability. For example, if we want to offer the ability to create text boxes for data insertion/capture, the native `<input>` tag could provide this capability. However, exposing or leveraging this tag directly in our core UI components is dangerous because it creates a hard dependency on non-abstract UI components. If we decide to use a third-party UI component later, we would need to change these native `<input>` tags across all components that use them, which is not an optimal strategy.

Bases are organized under the `Views/Bases` directory, with each base component residing in a domain-specific folder. For example:

- `Views/Bases/Buttons/ButtonBase.razor`
- `Views/Bases/Toasts/ToastBase.razor`
- `Views/Bases/Divisions/DivisionBase.razor`

Let's take a look at a visualization for base component functionality:



As seen in the above example, base components will wrap an external or native UI component and then expose APIs to seamlessly and programmatically interact with that Component. Occasionally, these APIs will represent parameters, functions, or delegates to interact with the Component based on the business flow.

3.2.1.2.0.0.0 Implementation

Let's take a look at a simple Base component for solving this problem:

```
<input @bind-value=Value />
```

```
public partial class TextBoxBase : ComponentBase
{
    [Parameter]
    public string Value {get; set;}

    public void SetValue(string value) =>
```

```
        this.Value = value;
}
```

In the code above, we wrapped the `<input>` tag with our base component `TextBoxBase` and offered an input parameter `Value` to be passed into that Component so it can pass it down to the native UI element. Additionally, we provided a public function, `SetValue` to allow for programmatically mimicking the user's behavior to test drive the consuming Component of this base element.

3.2.1.2.0.0.1 Utilization

Now, when we try to leverage this base component at the core components level, we can call it as follows:

```
<TextBoxBase @ref=MyTextBox />
```

The `@ref` aspect will allow the backend code to interact with the base component programmatically behind the scenes to call any existing functionality.

3.2.1.2.0.0.2 Restrictions

Components can only use base components. Pages may not use them, and other Base components may not use them. But more importantly, it's preferred those base components would only wrap around one and only one non-local Component.

And just like Brokers, Base Components do not have any business logic. They don't handle exceptions, do any calculations, or any form of sequential, iterative, or selective business logic operations. These operations are either data-based, where they belong to view services and downstream APIs, or UI-based, where they belong to Core Components.

Base components also don't handle exceptions, they don't throw their exceptions, and they don't perform any validations.

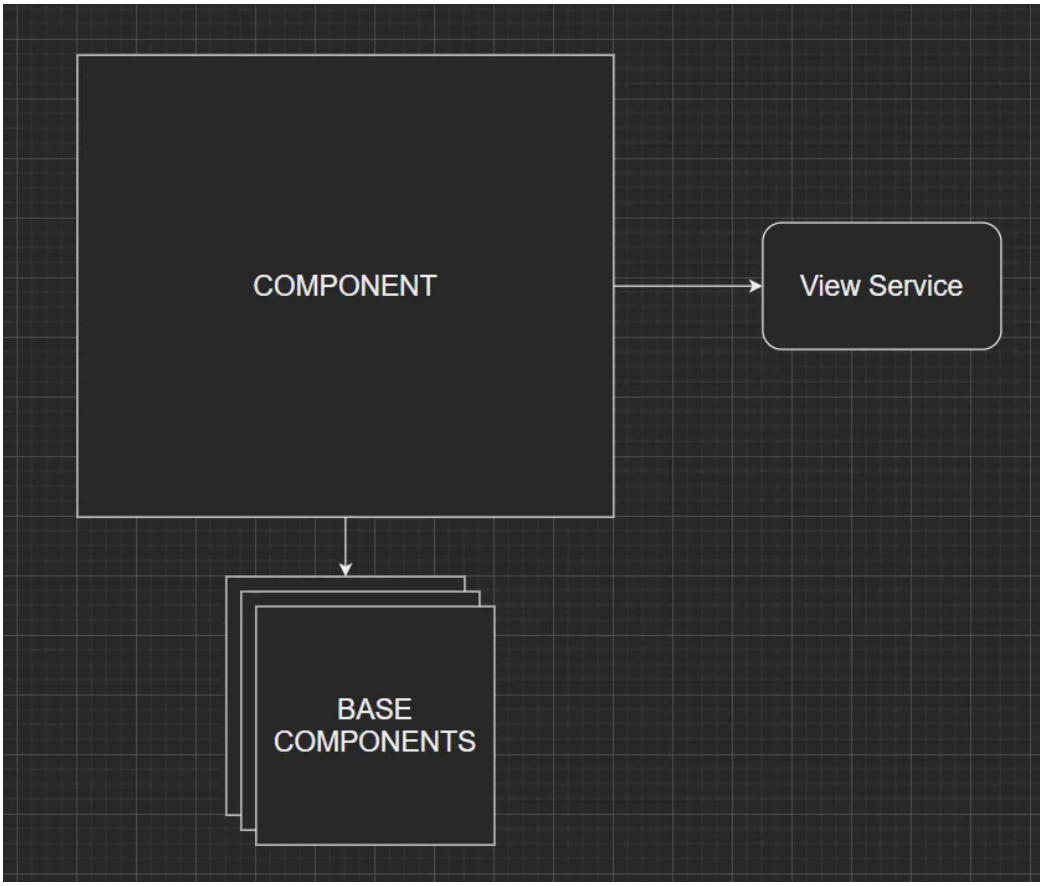
3.2.1.2.0.1 Core Component

Core components are just like services in the data flow. They are test-driven and restricted to one and only one dependency at all times. Core components leverage Base components to perform a business-specific flow. They are less generic than Base components because they orchestrate and communicate with a very particular service for the data flow.

Components are organized under the `Views/Components` directory, further categorized by their type: Foundations, Processings, Orchestrations, and Aggregations. Each component resides in a domain-specific folder with a file name that embeds its category (except for Foundations). For example:

- `Views/Components/Foundations/Subscriptions/SubscriptionsComponent.razor`
- `Views/Components/Processings/UserGrids/UserGridsProcessingComponent.razor`
- `Views/Components/Orchestrations/GroupDialogs/GroupDialogsOrchestrationComponent.razor`

Here's a visualization of core components architecture:



Core components are UI and Data component orchestrators. They will leverage one or many Base components to construct a business-specific flow, such as a student registration form, then send the signal to view services to persist that data and return responses or report errors.

Core Components are three main parts. Elements, Styles, and Actions. Let's discuss these parts here:

3.2.1.2.0.1.0 Elements

Elements are mainly the markup pieces you find in the `.razor` file in any component. These elements should always be Base Components. They are the skeleton of any Core Component. These Elements may or may not expose sub-routines, such as a Button Click, or a reactionary routine, such as a Button color change on hover, and so on.

Elements can be tested in three main ways. Existence, Properties, and Actions.

3.2.1.2.0.1.0.0 Existence

First and foremost, we need to ensure the Element is loaded and is present on the screen. This can be done in three different ways. Either by property assignment, searching by id, or searching for all types. Here are some examples:

3.2.1.2.0.1.0.0.0 Property Assignment

Every Component should have a corresponding property attached to the Element at runtime. For instance, assume we have a `StudentRegistrationComponent` as follows:

```
public class StudentRegistrationComponent: ComponentBase
{
    public TextBoxBase NameTextBox {get; set;}
}
```

In the above code, we defined `NameTextBox` as the same type as the Base Component that will be attached to it. Once that property is defined, we will need to write a failing test that verifies that this Element exists as follows:

```
public void ShouldLoadNameTextBox()
{
    // when
```

```
    this.renderedStudentRegistrationComponent =
        RenderComponent<StudentRegistrationComponent>();

    // then
    this.renderedStudentRegistrationComponent.Instance.NameTextBox
        .Should().NotBeNull();
}
```

The above test will fail. That's simply because no markup corresponds to the `NameTextBox` property on rendering-time. Let's make this test pass by changing the markup in `StudentRegistrationComponent.razor` as follows:

```
<TextBoxBase @ref=NameTextBox>
```

Our test will now pass. That's simply because the property is dynamically instantiated at render time once the page loads.

3.2.1.2.0.1.0.0.1 Searching by Id

Sometimes, Property Assignment is not an option. There are scenarios where components load dynamically a set of nested components that we may not have access to at design time. In this case, searching by ID is our best option to ensure we have the right Component in hand.

Here's an example. Assume we have a list of components that loads dynamically by being given a list of students. We use the student object `Id` as an identifier for every Component. Our code looks as follows:

```
public partial StudentListComponent : ComponentBase
{
    public List<Student> Students {get; set;}

    ....

    public void OnInitialized() =>
        Students = await this.someStudentViewService.RetrieveAllStudentsAsync();
}
```

On load - we call a view service to pull a list of all students asynchronously. We need to take that list and dynamically load a nested view for each student. Let's write a failing test for this first:

```
public void ShouldLoadStudentsAsync()
{
```

```
    // given
    List<Student> randomStudents = CreateRandomStudents();
    ...

    this.someStudentViewService.Setup(service =>
        service.RetrieveAllStudentAsync())
            .ReturnsAsync(randomStudents);

    // when
    this.renderedStudentListComponent =
        RenderComponent<StudentRegistrationComponent>();

    // then
    ....

    foreach(Student student in randomStudents)
    {
        StudentComponent studentComponent =
            this.renderedStudentListComponent.Find($"#{student.Id}")
                as StudentComponent;

        studentComponent.Should().NotBeNull();
    }

    ...
}
```

In the above tests, we looked for components that matched the student `ID`, and verified they existed. Let's make that test pass as follows:

```
<Iterations Items="Students">
  <StudentComponent Value="@context" />
</Iterations>>
```

We use the `PrettyBlazor` library to markup our iteration behavior with the `<Iterations>` tag. Now, our tests should pass by finding and verifying each created Component once they load on the screen.

3.2.1.2.0.1.0.0.2 General Search

There are scenarios where we don't have a key or an `Id` to find the Element. We expect a list of "things" to load on the screen without any data or information on them. In this case, we are going to have to resolve the General search mechanism where we rely on the count of the rendered components against the count that we expect as follows:

```
public void ShouldLoadManyElements()
{
    // given
    int randomCount = GetRandomNumber();

    // when
    this.renderedThingsComponent =
        RenderComponent<StudentRegistrationComponent>();

    // then
    var renderedThings = this.renderedThingsComponent.Find("p");

    renderedThings.Count.Should().Be(randomCount);
}
```

The Standard advises against having unknown-typed components like these loaded on the screen as they give engineers much less control over what's going on. But in gaming scenarios, this could be the only option.

3.2.1.2.0.1.0.1 Properties

The other aspect we consider when developing Core Components is their properties. These could be properties of the Core Component itself or the Base Component. For instance, we want to verify that a `LabelBase` component has property information such as `First Name` or `Last Name`.

Let's start by setting up a test.

```
public class StudentRegistrationComponent: ComponentBase
{
    public LabelBase FirstNameLabel {get; set;}
}
```

In the above code, our `StudentRegistration` component has a label on the screen that is supposed to have a certain value by default for a form. Let's write a failing test for it as follows:

```
public void ShouldHaveFirstNameLabel()
{
    // given
    string expectedFirstNameLabel = "First Name";

    // when
    this.renderedStudentRegistrationComponent =
        RenderComponent<StudentRegistrationComponent>();
```

```
    // then
    ...
    this.renderedStudentRegistrationComponent.Instance.FirstNameLabel.Value
        .Should().Be(expectedFirstNameLabel);

    ...
}
```

The test here will verify the label will always have the property value `First Name`. Let's make it pass.

```
<LabelBase @ref=FirstNameLabel Value="First Name">
```

We verified that the Element exists with the right property or information by simply doing that.

The same thing applies to properties on the Core Component itself, like having view models that load on initialization and then get assigned to certain base components. We will show that example shortly.

3.2.1.2.0.1.0.2 Actions

Testing actions is one of the most important parts of testing any Element. We want to ensure that a certain action is triggered when a button is clicked. These actions can also change a property, create a new element, or trigger another action. There are as many possibilities as there are in the very pattern of Tri-Nature itself.

Let's assume our `StudentRegistrationComponent` is supposed to trigger a call for a `StudentViewService` on the Button click event. Let's start with a simple failing test as follows:

```
[Fact]
public void ShouldSubmitStudent()
{
    // given
    StudentView randomStudentView = CreateRandomStudentView();
    ...

    // when
    this.renderedStudentRegistrationComponent =
        RenderComponent<StudentRegistrationComponent>();

    this.renderedStudentRegistrationComponent.Instance.SubmitButton.Click();

    // then
    this.studentViewServiceMock.Verify(service =>
```

```
        service.AddStudentViewAsync(
            this.renderedStudentRegistrationComponent.Instance.StudentView),
                Times.Once);


    ...
}
```

In the above test, we propose implementing a component that will trigger calling `AddStudentViewAsync` from a `StudentViewService` once the button clicks. This implies a correlation between clicking a button and triggering an action.

Let's write an implementation for this behavior. On the component code side, we should have the following function as follows:

```
public partial class StudentRegistrationComponent : ComponentBase
{
    [Inject]
    public IStudentViewService StudentViewService { get; set; }
    ...
    public StudentView StudentView { get; set; }
    public ButtonBase SubmitButton { get; set; }
    ...

    public async void RegisterStudentAsync() =>
        await this.StudentViewService.AddStudentViewAsync(this.StudentView);
}
```

The above code implements a `RegisterStudentAsync` function that will pass `StudentView` property (data) unto the `StudentViewService` for registration/add. Now, let's attach that function to a UI element on the markup side as follows:

```
<ButtonBase @ref=@SubmitButton
            Label="SUBMIT"
            OnClick=@RegisterStudentAsync />
```

In the above markup, we attached the `SubmitButton` property to the Element and passed the `OnClick` event with the `RegisterStudentAsync` routine. When the button is clicked, the routine will trigger, and we should be able to verify it in our unit tests.

3.2.1.2.0.1.1 Styles

Core Components also carry more than just elements. They have certain styles to ensure the user experience fits the type of business they're trying to accomplish. While Elements or Base Components can also carry their own styles, it's important

to realize that styles are better suited to Core Components to ensure the modularity of Base Components to fit whatever style is enforced by Core Components.

Testing styles are rare in the UI world. Especially when it comes to test-driving styles in C# as code. The Standard enforces the idea of leveraging the same programming language (when possible) across all different aspects of a project. That also includes infrastructure, pipelines, styles, actions, and everything else in between. This principle ensures that the learning curve for engineers working on any project is as minimal as possible, in addition to having standardized patterns.

We will leverage a library called SharpStyles to test styles on Core Components. The library flawlessly translates C# code into CSS styles.

Let's consider a scenario where we want our `SubmitButton` on the registration component above to have a blue color for its background. Let's add a `Style` property on our Component as follows:

First of all, we need to create a C# model with the identifiers we would like to have in our CSS style as follows:

```
public class StudentRegistrationStyle : SharpStyle
{
    [CssClass]
    public SharpStyle SubmitButton { get; set; }
}
```

This model will be translated into a CSS class called `submit-button` when we start rendering the Component. Let's leverage this new model in our Component as follows:

```
public partial class StudentRegistrationComponent : ComponentBase
{
        public StyleBase StyleElement { get; set; }
        public StudentRegistrationStyle StudentRegistrationStyle { get; set; }
        ...
}
```

Now that we have a new property for styles, we need to hook this property to a markup that will transform these styles/models into pure native CSS. We will need to create a `StyleBase` Element/Base Component to take care of the abstraction

side for us - so we don't have any hard dependency on the SharpStyle library as follows:

The markup side of that will look as follows:

```
<style>
    @Style.ToCss()
</style>
```

The code side of the same Element/Base Component will be as follows:

```
public partial class StyleBase : ComponentBase
{
    [Parameter]
    public SharpStyle Style { get; set; }
}
```

Now, let's go ahead and utilize this Base Component in our `StudentRegistrationComponent` as follows:

```
<StyleBase @ref=StyleElement
           Style=StudentRegistrationStyle />
```

Now that we have everything setup, let's write a failing test to require a button to have a blue background color as follows:

```
[Fact]
public void ShouldRenderContainerWithStyles()
{
    // given
    string expectedCssClass = "submit-button";
    ...
    var expectedStyle = new StudentRegistrationStyle
    {
        SubmitButton = new SharpStyle
        {
            BackgroundColor = "blue"
        },
    };

    // when
    this.renderedStudentRegistrationComponent =
        RenderComponent<StudentRegistrationComponent>();

    // then
    this.renderedLabOverviewComponent.Instance.SubmitButton.CssClass
        .Should().BeEquivalentTo(expectedCssClass);
```

```
    this.renderedStudentRegistrationComponent.Instance.StudentRegistrationStyle
        .Should().BeEquivalentTo(expectedStyle);

    this.renderedStudentRegistrationComponent.Instance.StyleElement.Style
        .Should().BeEquivalentTo(expectedStyle);
}
```

With a failing test like this, we can now start writing an implementation to satisfy the following conditions for this test.

On the component code side, let's generate the expected style object:

```
public partial class StudentRegistrationComponent : ComponentBase
{
    public StyleBase StyleElement { get; set; }
    public StudentRegistrationStyle StudentRegistrationStyle { get; set; }
    ...
    protected override void OnInitialized() => SetupStyles();

    public void SetupStyles()
    {
        this.StudentRegistrationStyle = new StudentRegistrationStyle
        {
            SubmitButton = new SharpStyle
            {
                BackgroundColor = "blue"
            },
        };
    }
    ...
}
```

Then, on the markup side, let's attach all the properties to their respective Elements as follows:

```
<StyleBase @ref=StyleElement
           Style=StudentRegistrationStyle />

<ButtonBase CssClass="submit-button" ... />
```

Our tests should pass, and this would be a quick demonstration of a standardized way of testing styles for UI components.

3.2.1.2.0.1.2 Actions

Actions in Core Components are very similar to Actions in Base Components or Elements. It is important, however, to understand that every action can easily be verified by either changing a property or style, creating other components, or simply triggering other actions. It can also be a combination of one or many of those above. For instance, a submit button could change the properties of existing elements by making them disabled while triggering a call/action to another service. It should all be verifiable, as we discussed above.

3.2.1.2.0.1.0 Full Implementation & Tests

Let's take a look at the implementation of a core component.

```
public partial class StudentRegistrationComponent : ComponentBase
{
    [Inject]
    public IStudentViewService StudentViewService {get; set;}

    public StudentRegistrationComponentState State {get; set;}
    public StudentView StudentView {get; set;}
    public TextBoxBase StudentNameTextBox {get; set;}
    public ButtonBase SubmitButton {get; set;}
    public LabelBase StatusLabel {get; set;}

    public void OnInitialized() =>
        this.State == StudentRegistrationComponentState.Content;

    public async Task SubmitStudentAsync()
    {
        try
        {
            this.StudentViewService.AddStudentViewAsync(this.StudentView);
        }
        catch (Exception exception)
        {
            this.State = StudentRegistrationComponentState.Error;
        }
    }
}
```

The above code shows the different types of properties within any given component-the dependency view service maps raw API models/data into consumable UI models. The `State` determines whether a component should be

`Loading`, `Content`, or `Error`. The data view model binds incoming input to one unified model, `StudentView`. The last three are base-level components used to construct the form of registration.

Let's take a look at the markup side of the core component:

```
<Condition Evaluation=IsLoading>
    <Match>
        <LabelBase @ref=StatusLabel Value="Loading ..." />
    </Match>
</Condition>

<Condition Evaluation=IsContent>
    <Match>
        <TextBoxBase @ref=StudentNameTextBox @bind-value=StudentView.Name />
        <ButtonBase @ref=SubmitButton Label="Submit" OnClick=SubmitStudentAsync />
    </Match>
</Condition>

<Condition Evaluation=IsError>
    <Match>
        <LabelBase @ref=StatusLabel Value="Error Occurred" />
    </Match>
</Condition>
```

We linked the references of the student registration component properties to UI components to ensure the rendering of these components and data submission execution.

A component has already loaded state and post-submission states. Let's look at a couple of tests to verify these states.

```
[Fact]
public void ShouldRenderComponent()
{
    // given
    StudentRegistrationComponentState expectedComponentState =
        StudentRegistrationComponentState.Content;

    // when
    this.renderedStudentRegistrationComponent =
        RenderComponent<StudentRegistrationComponent>();

    // then
    this.renderedStudentRegistrationComponent.Instance.StudentView
        .Should().NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.State
```

```
        .Should().Be(expectedComponentState);

    this.renderedStudentRegistrationComponent.Instance.StudentNameTextBox
        .Should().NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.SubmitButton
        .Should().NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.StatusLabel.Value
        .Should().BeNull();

    this.studentViewServiceMock.VerifyNoOtherCalls();
}
```

The test above will verify that all the components are assigned a reference property and that no external dependency calls have been made. It will also validate that the code in the `OnInitialized` function on the component level is validated and performing as expected.

Now, let's take a look at the submittal code validations:

```
[Fact]
public void ShouldSubmitStudentAsync()
{
    // given
    StudentRegistrationComponentState expectedComponentState =
        StudentRegistrationComponentState.Content;

    var inputStudentView = new StudentView
    {
        Name = "Hassan Habib"
    };

    StudentView expectedStudentView = inputStudentView;

    // when
    this.renderedStudentRegistrationComponent =
        RenderComponent<StudentRegistrationComponent>();

    this.renderedStudentRegistrationComponent.Instance.StudentName
        .SetValue(inputStudentView.Name);

    this.renderedStudentRegistrationComponent.Instance.SubmitButton.Click();

    // then
    this.renderedStudentRegistrationComponent.Instance.StudentView
        .Should().NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.StudentView
```

```
        .Should().BeEquivalentTo(expectedStudentView);

    this.renderedStudentRegistrationComponent.Instance.State
        .Should().Be(expectedComponentState);

    this.renderedStudentRegistrationComponent.Instance.StudentNameTextBox
        .Should().NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.StudentNameTextBox.Value
        .Should().BeEquivalentTo(studentView.Name);

    this.renderedStudentRegistrationComponent.Instance.SubmitButton
        .Should().NotBeNull();

    this.renderedStudentRegistrationComponent.Instance.StatusLabel.Value
        .Should().BeNull();

    this.studentViewServiceMock.Verify(service =>
        service.AddStudentAsync(inputStudentView),
            Times.Once);

    this.studentViewServiceMock.VerifyNoOtherCalls();
}
```

The test above validates that on submittal, the student model is populated with the data set programmatically through the base component instance and verifies all these components are rendered on the screen before end-users by validating each base component an assigned instance on runtime or render-time.

3.2.1.2.0.1.1 Restrictions

Core components have similar restrictions to Base components because they cannot call each other at that level. There's a level of Orchestration Core Components that can combine multiple components to exchange messages. Still, they don't render anything independently, the same way Orchestration services delegate all the work to their dependencies.

One view service corresponds to one core component, which renders one and only one view model. However, core components are also not allowed to call more than one view service. And in that, they always stay true to the view model.

View services may do their orchestration-level work in an extremely complex flow, but we recommend keeping things at a flat level. These same view services perform nothing but mapping and adding audit fields and basic structural validations.

3.2.1.2.0.2 Pages

In every web application, pages are a fundamental mandatory container component that needs to exist so end-users can navigate to them. Pages mainly hold a route, communicate a parameter from that route, and combine core-level components to represent a business value.

An excellent example of a page is a dashboard. Dashboard pages contain multiple components, such as tiles, notifications, headers, and sidebars, that reference other pages. Pages don't hold any business logic in and of themselves, but they delegate all route-related operations to their child components.

Let's take a look at a simple page implementation:

```
@page '/registration'

<HeaderComponent />
<StudentRegistrationComponent />
<FooterComponent />
```

Pages are much simpler than core or base components. They don't require unit testing and don't necessarily need backend code. They purely reference their components without reference (unless required) and help serve that content when navigating via a route.

Pages are organized under the `Views/Pages` directory, with each page residing in a domain-specific folder. For example:

- `Views/Pages/Subscriptions/SubscriptionsPage.razor`
- `Views/Pages/Users/UsersPage.razor`
- `Views/Pages/Groups/GroupsPage.razor`

3.2.1.2.0.3 Unobtrusiveness

It's a violation to include code from multiple technologies/languages on the same page for all UI components. For instance, CSS style code, C# code, and HTML markup cannot all exist in the same file. They need to be separated into their own files.

The unobtrusiveness rule helps prevent cognitive pollution for engineers building UI components and makes the system much easier to maintain. That's why every Component can nest its files beneath it if the IDE/Environment used for

development allows for partial implementations as follows:

- StudentRegistrationComponent.razor
  - StudentRegistrationComponent.razor.cs
  - StudentRegistrationComponent.razor.css

The node file here, `.razor`, has all the markup needed to kick off the Component's initialization. At the same time, both nested files are supporting files for simple UI logic code and styling. This level of organization (especially in Blazor) doesn't require any referencing for these nested/support files. Still, this may not be the case for other technologies, so I urge engineers to do their best to fit that model/Standard.

3.2.1.2.0.4 Organization

All UI components are listed under a Views folder in the solution. Let's take a look:

- Views
  - Bases
    - Buttons/ButtonBase.razor
    - Toasts/ToastBase.razor
    - Divisions/DivisionBase.razor
  - Components
    - Foundations
      - Subscriptions/SubscriptionsComponent.razor
    - Processings
      - UserGrids/UserGridsProcessingComponent.razor
    - Orchestrations
      - GroupDialogs/GroupDialogsOrchestrationComponent.razor
  - Pages
    - Subscriptions/SubscriptionsPage.razor
    - Users/UsersPage.razor
    - Groups/GroupsPage.razor

This tri-nature conforming organization should make it easier to shift reusable components and make it also easier to find these components based on their categories. Given the nesting is in place, I will leave it up to the engineers' preference to break down these components further by folders/namespaces or leave them all at the same level.