Clean Architecture: Patterns, Practices, and Principles

by Matthew Renze

Course Overview

Course Overview

Hi. I'm Matthew Renze with Pluralsight, and welcome to Clean Architecture: Patterns, Practices, and Principles. Clean architecture is a set of practices used to create modern software architecture that is simple, understandable, flexible, testable, and maintainable. In addition, clean architecture is a more modern replacement for the traditional three layered database-centric architecture that we've been using for decades. As an overview of this course, first you'll learn about clean architecture, what it is, and why it's important. Then, you'll learn about a series of modern architectural practices, like domain-centric architecture, application layers, command query responsibility separation, functional organization, and microservice architectures. Finally, you'll see how these architectural practices make your architecture more testable, more maintainable, and allows the architecture to evolve over time. By the end of this course you'll have the skills necessary to understand and implement these clean architecture practices on your own software projects. In addition, throughout this course we'll be walking through an open source application that implements all of these practices, so you can see how they're implemented first hand. As an introductory course, there are no prerequisites for this course. However, having basic experience with at least one C like programming language, and basic knowledge of software architecture will be beneficial, so please join us today at Pluralsight, and learn how to create highly effective and maintainable software with clean architecture.

Introduction

Introduction

Hello, and welcome to Clean Architecture: Patterns, Practices, and Principles. I am Matthew Renze with Pluralsight, and in this course we'll learn about clean architecture, a modern approach to creating highly maintainable software architecture, but first I'd like to start with a story. We'll call it, A Tale of Two Architectures. Imagine for me, if you will, that we have two architects. We'll call the, John and Jane. John and Jane have roughly the same years of experience, the same job title, and will both cost the same amount to hire. We hire them both to build two new buildings for our company in two different cities. The buildings will house roughly the same number of employees, doing roughly the same type of work in both places. John is an expert in classical architecture. He is constantly pushing the boundaries of classical architectural design using cutting edge technology. In addition, he uses all of his favorite tools and techniques on each of his new projects. His design is elegant. It's a classical design, yet it's state of the art, and he designed it just the way he likes it. In fact, his design completely wows the board of directors. Once he's finished he hands the blueprints to the construction crew, wishes them well, and moves on to design his next masterpiece. Then, once the building is completed, he shows up to the grand opening ceremony to reap the rewards of his accomplishment. Jane, however, has a very different approach to architecture. First, she starts by chatting with the future inhabitants of the building, that is the employees, the maintenance crew, and she even talks to the construction foreman. She finds out what their needs are, what they find valuable, and what the experience level of the construction crew is. Then she designs the architecture of the building with the inhabitants in mind, considering their needs with each decision that is made. Once she finishes the blueprints she doesn't just hand them off to the construction crew and disappear. Instead, she works side by side with them each day helping them understand and implement the design, gathering feedback, and adjusting the blueprints as necessary, as they learn better ways to solve problems that arise. These two architects and their two approaches are going to produce two entirely different architectures. As an employee, a building maintenance person or a construction worker, which of these two architects do you want designing the building that you'll be working within? Now this is a bit of a loaded question, and we're clearly setting up John to be a straw man, just so that we can knock him down, but the allegory captures the essence of what we're going to be discussing during this course, an idea called clean architecture.

Overview

As an overview of this course, first we'll discuss clean architecture, what it is, and why it's important. Next, we'll look at domain-centric architectures, and why we might want to place our domain model at the center of our architecture. Then we'll discuss the application layer, and why we might want to embed the use cases of our system into an application layer. Next, we'll discuss Command Query Responsibility Separation or CQRS, and why we might want to

separate the command and query stacks of our architecture. Then, we'll discuss the screaming architecture practice, and how it can help us organize our folder structure and namespaces using functional cohesion. Next, we'll look at microservices, and how we might want to horizontally partition our application by bounded contexts. Then we'll learn about testable architecture practices, and how our architecture makes testing easier, not harder, and finally, we'll learn about how we can evolve our architecture to reduce risks due to uncertainty in changing requirements over the life of the project. In addition, in each module of this course we'll see sample code from an application built using these various patterns, practices, and principles. The purpose of this course is to teach you about clean architecture, a philosophy of architectural essentialism. That is, a way of designing software architecture for what is essential in an architecture versus what is just a detail. We do this by using a set of modern patterns, practices, and principles for creating simple, understandable, adaptable, testable, and maintainable architecture. In addition, this course lays the foundation of a new style of architecture designed to act as an alternative to the traditional three-layered database-centric architecture we've been using for decades. Essentially, this new architectural style can be used as a starting point when creating an architecture that allows you to evolve the architecture as the needs of the system change over time. Because a topic as wide as software architecture can grow out of control very fast, we're going to have to keep this course constrained in a few key ways. First, we're going to focus on architecture for creating Enterprise applications. That is, applications designed to solve business problems, since statistically this is most likely what the majority of you are working on day to day in your careers. These are often referred to as line of business applications, however, if you're not familiar with either of these two terms, essentially we'll be focusing on a more modern interpretation of the traditional three-layer database-centric architecture found common in these types of business applications. Second, the architectural practices we'll be focusing on are designed to support an agile software development process. If you're not building software in an iterative and evolutionary manner these practices might not be directly applicable to you for your particular project. However, if you're not using agile practices you still might find this approach of building software to be highly insightful. Finally, this course could have easily been several hours long, given how much information I wanted to teach you about clean architecture, however, due to limited time and attention spans we're just going to focus on the top seven ideas that have helped me clean up my architecture the most. If you're interested in going much deeper I'll point you in the direction of additional sources of information at the end of the last module of this course. The intended audience for this course are developers who want to learn how to implement clean architecture practices into their applications, and architects who want to update the, knowledge of modern architectural practices, however, this course is widely applicable to anyone in the software industry who wants to understand modern software architecture practices, and how to create maintainable software architecture. Since this is an introductory course there are no required prerequisites; however, having basic programming experience with at least one C like programming language will help you understand the C# code used in these demos. In addition, having basic knowledge of software architecture will be beneficial as well; however, if you don't have sufficient experience with

either of these two optional prerequisites don't worry, I'll take time to explain things throughout this course, so now let's learn about clean architecture and why we should strive to keep our architecture clean.

What Is Clean Code?

Just to make sure we're all on the same page, we need to define what software architecture is. Well, it's not easy to define. There are a few general concepts that we can probably all agree upon. First, it's high level or at least higher level than the code we write. Second, it has to do with the structure of software or how things are organized. Third, it typically involves layers, which are vertical partitions of the system. Fourth, it typically involves components of some kind, which are typically horizontal partitions within each layer. Finally, it involves the relationships between these things, that is how they're all wired together. When we're discussing architecture we can focus on one of many levels of abstraction. Starting at the top, that is the most abstract representation of software, we have the system, which can be represented as a set of one or more subsystems, which are typically divided into one or more layers, which are often subdivided into one or more components, which contain classes that contain data and methods. Our primary focus for the conceptual information in this course will mainly be at the abstraction levels of layers and components. However, during our demos we'll also see examples of the classes, data, and methods used to create our architecture. In addition, we'll also see an alternative representation of components as small subsystems when we get into microservice architectures, so now let's learn about clean architecture, what it is, and why it's important. I think we all have a general intuition about what messy architecture looks like versus what clean architecture looks like. In addition, I think this intuition maps very well to two equally delicious types of Italian food. First, bad or messy architecture looks and feels like spaghetti to us. It's just a messy pile of code with an occasional meatball thrown in here or there. It's almost impossible to navigate from the start of one noodle to the end of that same noodle, and it's very difficult to add or replace a noodle without disrupting all of the neighboring noodles. Next, good, clean architecture looks and feels like lasagna to us. It has nice, consistent, horizontal noodle boundaries that uniformly divide the various layers of filling. When a piece is too big we cut it into smaller components with crisp, clean, orthogonal lines of demarcation, and it has a nice presentation layer riding on top of an edible stack of functionality. So, in more technical terms, what is bad or messy architecture? Well, it's architecture that is complex, but due to accidental complexity rather that necessary complexity. It's incoherent in the sense that the parts don't seem like they fit together. It's rigid, that is the architecture resists change or makes it difficult to evolve the architecture over time. it's brittle, touching a part of the code over here might break another part of the code somewhere else, it's untestable, that is, you'd really like to write unit tests and integration tests, but the architecture fights you each step of the way, and ultimately, all of these things lead to an architecture that's unmaintainable over the life of the project. On the other hand, we know we have good, clean architecture when it is simple or at least it's only as complex as is necessary, and that complexity is not accidental. It's understandable, that is it's easy to reason about the software as a whole, it's flexible, we can easily adapt the system to meet changing requirements, it's emergent, the architecture

evolves over the life of the project, it's testable, the architecture makes testing easier, not harder, and ultimately all of this leads to an architecture that's more maintainable over the life of the project. The way I sum up clean architecture is that it's architecture that is designed for the inhabitants of the architecture, not for the architect or for the machine. Clean architecture is a philosophy of architectural essentialism. It's about focusing on what is truly essential to the software's architecture versus what is just an implementation detail. By designing for the inhabitants we mean the people that will be living within the architecture during the life of the project. This means the users of the system, the developers building the system, and the developers maintaining the system. By not designing for the architect we mean that the architect should put aside his or her own desires, preferences, and wishes, and only consider what is best for the inhabitants of the architecture with each decision that is made. By not designing for the machine we mean that we should optimize the architecture first for the needs of the inhabitants, that is the users and the developers, and only optimize for the machine when the cost of performance issues to the users, who are inhabitants of the architecture, outweighs the benefit of a clean design to the developers who are also inhabitants of the architecture. Essentially, we want to avoid premature optimization, which as visionary computer scientist Donald Knuth says, is the root of all evil in software development. Please note that this is the key point of this entire course. If you only take one thing away from this course let it be this. We need to be designing our architecture first for the inhabitants of the architecture, everything else, our egos, shiny new frameworks, performance optimizations, etc., should all be of secondary concern.

Why Invest in Clean Code?

So why should we invest in clean architecture? The primary justification for investing in clean architecture is mainly a cost benefit argument. Our goal as software architects is to minimize the cost of creating and maintaining the software while we maximize the benefit that is the business value that the software provides. Our overall goal, thus, is to maximize the return on investment, that is the ROI of the software project as a whole. Clean architecture attempts to do this in several ways. First, clean architecture focuses on the essential needs of the primary inhabitants of the system; that is the users. We attempt to build a system that mirrors the use cases and mental models of the users by embedding these use cases and mental models in both our architecture and our code. Second, clean architecture builds only what is necessary when it is necessary. We attempt to create only the features and corresponding architecture that are necessary to solve the immediate needs of the users in order of each features perceived business value. We attempt to do this without creating any accidental complexity, unnecessary features, premature performance optimizations or architectural embellishments. This helps to reduce the cost of creating the system. Third, clean architecture optimizes for maintainability. For an average enterprise application with a sufficiently long lifecycle, say 10 years or so, we spend significantly more time and money maintaining the system than we do creating it. Several sources that I've seen indicate that roughly 60 to 80% of the cost of the life of the software application comes from maintenance, so if we optimize for maintainability, which clean architecture does, we in theory reduce

the cost of maintaining the system. This focus on value adding and cost reducing activities attempts to maximize the return on investment of the software as a whole. In addition, there are several other value adding and cost reducing benefits that clean architecture provides; however, we'll discuss these in more detail throughout this course. One final note before we move on to our next topic. Have you ever noticed how any time you ask an architect a question his or her initial response is invariably, it depends. How long is this going to take? It depends. How much is this going to cost? It depends. Why are you covered in spaghetti sauce? It depends. All joking aside though, there really is a good reason for this, and it's because context is king in the land of architecture. Every answer really does depend on several often conflicting constraints and optimization goals that the architecture's subject to, many of which are outside of the control of the architecture. Every time we make an architectural decision we're trading one thing for something else. Often times we're trading one kind of complexity that we don't want for another kind of complexity that we also don't want either, but we would prefer to have over the first kind of complexity. As we mentioned earlier, the ultimate goal of an architect is to make decisions that trade off things like complexity, in such a way that minimizes cost and maximizes business value, thus maximizing a return on investment of the software project as a whole. In many cases, these patterns, practices, and principles that we're going to learn about may not minimize cost or maximize business value for your specific project because of the specific context surrounding your project and your business as a whole. It is very important for the architect to understand the goals of the business and the context that the business exists within, and then align the architecture with the goals of the business and this context in mind. For example, if we were building embedded software for an industrial control system with very rigid requirements it might make more sense to use a more traditional software architecture and traditional software development practices. However, if we're building the next Uber we're likely dealing with a product in a market with a high degree of uncertainty, and thus eliminating risk due to this uncertainty is our primary business objective. Using an agile software development process, and these architecture patterns, practices, and principles will allow us to quickly evolve the product and the architecture as we learn more. This will likely work very well for us because it helps us to minimize risk for the business. This is why it's important to learn the various patterns, practices, and principles, understand the pros and cons of each, and understand within which context they are effective or not. Ultimately, you need to use your best judgement, make small mistakes, and use feedback from these small mistakes to improve over time. So if you ask me in the discussion boards if you should implement one of these practices on your specific project at work my answer will almost certainly be it depends. So now let's take a look at the demo application that we'll be working with throughout this course.

Demo

Now let's take a look at a demo of the application that we'll be using for all of the code demos in this course. To teach these clean architecture concepts I've created a simple demo application that demonstrates the key ideas of the practices we'll be discussing without all of the extra complexities of a full blown enterprise application. To keep the

demo simple, understandable, and relatable I've created a very basic sales application that will allow an employee to record the sale of a product to a customer for our fictitious company. The sales application has five pages, a Home page, which contains an introduction, and links to the other pages, a Customers page, which contains a read only list of our customers, an Employees page, which contains a read only list of our employees, a Products page, which contains a read only list of our products and their prices, and a Sales page, which contains a list of the sales transactions in the system. A sales transaction is composed of the following fields, an ID, which uniquely identifies the sales transaction, a date on which the sale occurred, a customer that the product was sold to, an employee who sold the product, a product that was sold, the unit price of the product, the quantity of the products sold, and the total price of the sale transaction. We can create a sale by clicking on the Create Sale button. This takes us to the Create Sale page, which we can use to create a new sale. We can select a customer, select an employee, select a product, and assign a quantity of products to be sold. Then when we click on the Create button the sales transaction is recorded, the inventory system is notified that a sale occurred, and we are redirected to the Sales page. We can view the details of a sale by clicking on the ID of a sale. This will take us to the Sale Details page, which contains all of the information relevant to the sale. Throughout this course we're going to focus primarily on the code pertaining to the sales functionality that we just saw. This demo application incorporates several technologies that are of importance. First, the entire demo application has been built with VisualStudio. NET 2015. All the code was created with C# using the. NET framework, 4. 5. The application uses ASP. NET MVC 5 with Razor syntax for the views. The database is Microsoft SQL Server 2014. The application uses Entity Framework 6 as an ORM, that is Object Relational Mapper, to map objects to entities within the database. We use structure map three as a dependency injection framework to wire up all of our dependencies together at runtime, we use NUnit for unit testing, Moq for a test mocking framework, AutoMoq to make mocking and unit testing easier, and SpecFlow for behavior driven design based acceptance testing. However, you're more than welcome to use whatever tools, technologies, and frameworks you'd like to implement your own clean architecture. These practices work well with a wide variety of technologies, many of which were specifically designed to support this style of architecture. In addition, I could have chosen to use all of the latest versions of these technologies in creating these demos; however, I felt that it was more important to use more stable and widely available versions of these technologies to make the demos more accessible to a wider audience. Our solution is divided into seven projects. Starting with the conceptual top of the application we have Presentation. This project contains the user interface into the application. Application; this project contains abstractions corresponding to the use cases of the application. Domain; this project contains abstractions corresponding to the problem domain, that is the business domain. Persistence; this project provides an interface into the persistent storage medium, that is the database. Infrastructure; this project contains an interface into the operating system and third party dependencies. Common; this project contains crosscutting concerns, that is aspects of the application that all projects depend upon, and specification. This project contains acceptance tests verifying the correct functionality of the application. We'll be working with all of these projects throughout the demos. The database that this demo application uses is a Microsoft SQL Server 2014 database; however, the demo application will also work with SQL

Express, Microsoft's free version of SQL Server, if you prefer. The database consists of four tables, Customers, containing our customer data, Employees, containing our employee data, Products, containing our product data, and Sales, containing data for our sales transactions with relationships to the other three tables in our database. In the spirit of the open-source software movement, I've made all of the source code and demos of this course available as free open-source software. This software can be downloaded from Pluralsight's website using the download link on the Exercise Files page. In addition, you can also view, download, and modify this source code from my GitHub repository at the URL found on the course description page. I encourage you to download Explorer and play around with the code using these demos. It will be significantly easier to grasp the concepts that are being taught in this course if you spend some time becoming familiar with the demo code itself. In addition, feel free to use, modify, and redistribute this code in accordance with the included open-source license.

Summary

In this module first we saw an overview of the course, we covered the purpose of the course, its focus, and the intended audience. Next, we learned about clean architecture, what it is, why it's important, and how context is critical with each architectural decision we make. Then we saw a demo where we introduced the application that we'll be using for all of the demos in this course. We saw the user interface, the technologies, the project files, and the database. In the next module we'll learn about domain centric architectures.

Domain-centric Architecture

Introduction

Hello again, and welcome to our next module on clean architecture. I'm Matthew Renze with Pluralsight, and in this module we'll learn about Domain-centric Architecture. As an overview of this module, first we'll learn about domain-centric architecture and how it differs from database-centric architecture. Next, we'll learn about three types of domain-centric architectures. Then, we'll learn about the pros and cons of implementing a domain-centric architecture. Finally, we'll see a demo of our domain-centric demo application, so let's get started.

Domain-centric Architecture

We all know that the earth is at the center of the solar system, and that the sun, the moon, and the planets revolve around the earth. Correct? Well, they don't anymore, but they did at one time. However, a very smart guy by the name of Nicolaus Copernicus changed the way we look at our solar system. Rather than assuming that the earth was

at the center of the solar system, old Nick had this crazy idea that the sun might be at the center of our solar system instead. This shift in perspective turned out to produce a better model of the solar system, in the sense that it's both simpler, and yet it provides more explanatory power. It's essentially a more elegant model of our solar system. A similar shift in thinking is happening in the world of software architecture as we speak. Here we have the classic three-layer database-centric architecture. Its key feature is that the user interface, business logic, and data access layer revolve around the database. The database is essential, and thus, it's at the center of this architecture. However, a new perspective has changed the way many of us look at our architecture. Rather than having the database at the center of our architecture, some of us are putting the domain at the center, and making the database just an implementation detail outside of the architecture. Here the domain is essential, and the database is just a detail. So why has this happened? I think this change in perspective is best summed up by a quote from Robert C. Martin, better known in the software industry as Uncle Bob. He says, "the first concern of the architect is to make sure that the house is usable, it is not to ensure that the house is made of brick. " This change in architectural perspective is being caused by a change in perspective about what is essential in an architecture versus what is just an implementation detail. Using our building architecture metaphor, when we're building a house what is essential versus what is a detail? The space inside of a house is essential. Without empty space to inhabit the house would serve no purpose. The usability of the house is essential. If the house didn't contain rooms and features to support our primary needs, again, the house would not serve its purpose. However, the building material is just an implementation detail. We could build it out of brick, stone, wood or many other materials. In addition, the ornamentation is just a detail. We could still live in this house whether it had Victorian molding, gold trim, French doors or no ornamentation at all. The things that are essential in a house are so because they support the primary needs of the inhabitants of the house. Everything else is just an implementation detail. In clean architecture the same holds true. What is essential are the things that support the primary needs of the inhabitants of the architecture. The domain model is essential. Without it the system would not represent the mental models of the users. The use cases are essential. Without them the system would not solve the user's problems. However, the presentation is just a detail. We can deliver the UI in web forms, ASP. NET MVC or as a single page JavaScript application, and the persistence is just a detail. We can store the data in a relational database, no a SQL database, or as plain old JSON files. Now don't get me wrong, the presentation and persistence technologies are very important. They are just not essential to solving the problem that the user is attempting to solve with the application. Once we've changed our perspective about what is essential versus what is a detail in software architecture, we can start to see why a transition from database-centric architecture to domain-centric architecture is occurring. With database-centric architectures the database is essential, so the database is at the center of the application, and all dependencies point towards the database. With domain-centric architectures the domain and use cases are essential, and the presentation and persistence are just a detail, so the domain is at the center of the application wrapped in an application layer and all dependencies point towards the domain. So now let's take a look at a few types of domain-centric architectures. First, we have Alistair Cockburn's hexagonal architecture. It's a layered architecture with the application layer, and thus transitively, the

domain at the center of the architecture. In addition, it's a plugin architecture which includes ports and adapters. Essentially, the outer layers of the architecture are adapting the inner application layer to the various presentation mediums, persistence mediums, and external systems. You can run, and thus test, an isolation in this entire application architecture without a UI, a database or any external dependencies. Next, with the onion architecture by Jeffrey Palermo. This is also a layered architecture with the domain at the center surrounded by an application layer. The outer layers consist of a thin UI as a presentation layer, and an infrastructure layer, which includes persistence. In addition, all dependencies point towards the center of the architecture, that is no inner layer knows about any outer layer. Once again, you can test this application architecture in isolation without a UI, a database or any external dependencies. Finally, we have the clean architecture by Uncle Bob. Once again, it's a layered architecture with the domain, that is the entities, at the center surrounded by an application layer, that is the use cases. The outer layer consists of ports and adapters adapting the application core to the external dependencies via controllers, gateways, and presenters. In addition, Uncle Bob goes one step further by incorporating Ivar Jacobson's BCE architecture pattern to explain how the presentation layer and the application layer should be wired up. All three of these architectures have roughly the same benefits. However, as Mark Seaman has pointed out, all of these domain-centric architectures are essentially just focusing on different aspects of the same key set of ideas. Essentially, the all put the domain model at the center, wrap it in an application layer, which embeds the use cases, adapts the application to the implementation details, and all dependencies should point inwards towards the domain. So why would we want to use a domain-centric architecture? First, focus is placed on the domain, which is essential to the inhabitants of the architecture, that is the users and the developers which, as we've learned in the previous module, provides several benefits and cost reductions for our software. Second, there is less coupling between the domain logic and the implementation details, for example, the presentation, database, and operating system. This allows the system to be more flexible and adaptable, and we can much more easily evolve the architecture over time. Third, using a domain-centric architecture allows us to incorporate Domain Driven Design, which is a great set of strategies by Eric Evans for handling business domains with a high degree of complexity. However, that's an entire course on its own, but I'll show you where to go to learn more at the end of this course. So why would we not want to use a domain-centric architecture? First, change is difficult. Most developers come out of college having only been taught the traditional three layer database-centric architecture. In addition, it may also be the only architectural model that the architect knows well enough to offer guidance on. Second, it requires more thought to implement a domain-centric design. You need to know what classes belong in the domain layer, and what classes belong in the application layer rather than just throwing everything in a business logic layer. Third, it has a higher initial cost to implement this architecture compared to a traditional three layer database-centric architecture. However, it typically pays itself off if the application is complex enough, and has a long enough life cycle, which is the case with most modern applications these days. Now let's see a demo of a domain-centric architecture in action.

Demo

Now let's take a look at a domain-centric architecture in action. It's important to note that I've created the simplest domain possible to communicate these architectural ideas. In most modern applications the domain model would be significantly more complex than this. Well, I personally prefer to use more powerful domain modeling techniques, like Domain Driven Design and Event Sourcing, for dealing with complex domains. Right now we're just interested in learning the basics, so a more complex domain with more advanced domain-centric practices will just get in the way of learning. However, I'll point you in the direction of additional information, so you can learn these more advanced domain modeling practices at the end of this course. Let's begin by looking at a graph of the project dependencies in our solution. We're just going to focus on the most important projects and compile time project references to keep things simple for now. Starting at the top, we have our Presentation project. It has a dependency on the Application project, and thus transitively, an indirect dependency on the domain via the application project. Next, we have the application project itself. This project has a direct dependency on the domain project. Then we have the persistence project. It has an indirect dependency on the domain via the application project, and a direct dependency on the domain via a project reference as well. This direct dependency is necessary because we're using Microsoft entity framework to map our domain entities to our database tables. This direct dependency isn't necessary in general for clean architecture, however, it is necessary in this case in order to use Entity Framework as our object relational mapper. Finally, we have the infrastructure project. It has an indirect dependency on the domain via the application project. The key idea here is that the domain is at the center of the architecture, and all dependencies point either directly or indirectly towards the domain. Now let's take a look at our domain project. We organize our folders, classes, and namespaces via a concept we refer to as functional cohesion, which we'll discuss in a later module. For the time being, it's easiest to say that we're organizing things by the key entities in our domain. In domain driven design we more precisely refer to these as the aggregate root entities in our domain model. That is, the top most entities with persistent identity in a graph of dependent entities and value objects. Each of these folders contains the aggregate root that the folder is named after. For example, Customers, Employees, Products, and Sales, and any domain entities, value objects or domain services that these entities depend upon. In addition, we have a common folder for classes and interfaces that are shared between the aggregate roots. For example, the IEntity interface, which allows all entities to be worked with generically as objects with persistent identity. That is, they each have a unique identifier. Finally, let's take a look at the Sale class. As I mentioned before, I'm trying to keep things simple, so we aren't using a rich domain model or concepts like parent child entity relationships, value objects or domain services like we would be using with Domain Driven Design or event sourcing. In addition, if we were using Domain Driven Design we'd have classes with methods corresponding to actions named in the language of the business, rather than the relatively anemic domain model composed of mostly property getters and setters, which I'm using for simplicity's sake. Also, if we're using event sourcing we'd be applying events to our domain entities to modify their state rather than simply calling property getters and setters. As we can see, this class implements the IEntity

interface, so that it can be used generically like all of the entities in our domain. At the top of the class we have private fields, which contain data for our class. Next, we have the auto-implemented properties for this class. These store data without requiring backing fields, like the private fields above. Some of these store primitive data types and data structures like Id and date, while others store references to entities, like Customer, Employee, and Product. Below this we have property getters and setters for the properties that use backing fields, and logic within the setters. For example, unitPrice and Quantity. These setters have logic in order to update the total price property, so that it reflects the total price given the unit price and quantity of products sold. Finally, we have a private method called UpdateTotalPrice, which executes the domain logic to update the total price of the sale. As we can see, we've kept the domain logic and code very simple, so that we can focus on architectural practices, rather than deal with a complex domain during our code demos. What's important to note for this demo is that our domain project contains our domain model, which is an abstract representation of the business problem being solved. All of our classes, properties, and methods should correspond to concepts that exist in the business world in the language of the business. In addition, ideally there should be no abstraction leakage. That is, there shouldn't be any classes, attributes or aspects in our domain model that pertain to the implementation details of the application; details like persistence, infrastructure or presentation. So now that we've seen what a simple domain model looks like in domain-centric architecture we can focus on the other important aspects of a clean architecture.

Summary

In this module, first we learned about domain-centric architecture and how it differs from database-centric architecture. Next, we learned about three types of domain-centric architectures. Then we learned about the pros and cons of implementing a domain centric architecture. Finally, we saw a demo of our domain-centric demo application. In the next module we'll learn about the application layer, and why we may or may not want to implement an application layer in our architecture.

Application Layer

Introduction

Welcome back to our course on clean architecture. I'm Matthew Renze with Pluralsight, and in this module we'll learn about using an application layer in our architecture. As an overview of this module, first we'll learn about architectures with an application layer, and how this differs from a traditional three layer architecture. Next, we'll learn about the dependency inversion principle, and how we can use it to make our architecture more flexible and maintainable.

Then, we'll evaluate the pros and cons of implementing an application layer. Finally, we'll see a demo where we'll walk through the code used to implement a command in the application layer of our demo application.

Application Layer

Just to make sure we're all on the same page, what are layers? Layers are boundaries or vertical partitions of an application designed to represent different levels of abstraction, maintain the single responsibility principle, isolate developer roles and skills, help support multiple implementations, and assist with varying rates of change. Essentially, layers are the way that we slice an application into manageable units of complexity. Let's start with what we already know. The classic three-layer database-centric architecture. First, we have our user interface layer, which provides the user with an interface into the application. Next, we have our business layer, which contains the business logic of the application. Finally, we have our data access layer, which contains the logic to read and write to the database. This works just fine for simple CRUD applications, that is applications that perform basic Create, Read, Update, and Delete operations on data within the database; however, it doesn't work all that well with complex or rich domain models. In addition, there's a lot of ambiguity about where application level abstractions versus domain level abstractions should go. This has led to the development of a more modern four-layer domain-centric architecture. First, we have our presentation layer, which provides the user with an interface into the application. Second, we have an application layer, which embeds the use cases of the application as executable code and abstractions. Third, we have a domain layer, which contains only the domain logic of the application. Fourth, we have the infrastructure layer. Oftentimes it makes sense to divide this layer into one or more projects. For example, a common variation is to create a separate project for persistence, and a separate project for the remaining infrastructure. For example, in this diagram we have the persistence portion of the infrastructure layer, which provides the application with an interface to the database or other persistent storage. Then, we have the rest of the infrastructure layer, which provides the application with an interface to the operating system and other third party components. Finally, we have our cross-cutting concerns, which are aspects of the application that cross all the layers of the system. There are obviously a few variations on this architecture. For example, multiple user interfaces, adding a web service layer, separate projects for external dependencies, but the general structure is essentially the same. Here we have the application layer. It implements use cases as executable code, for example, a customer searches for a product, adds it to their cart, and pays with a credit card. We structure this executable use case code as high-level representations of application logic. For example, we might have a query that searches for our product for our customer or a command that adds a product to their shopping cart. The application layer knows about the domain layer, that is, it has a dependency on the domain, but it does not know about the presentation, persistence or infrastructure layers. That is, there are no dependencies on the outer layers of the application. The application layer, however, does contain interfaces for its dependencies that these respective outer layers then implement. Then we use an IoC framework, that is an Inversion of Control framework, and dependency injection to wire up all the interfaces and their implementations at run time.

In addition to the dependency arrows in our diagram in orange I've also added additional arrows in blue to indicate the flow of control through the application. We can follow the flow of control through the application from the users at the top of the diagram down to the database and operating system at the bottom of the diagram. In addition, we can follow the dependency arrows, which flow both upwards and downwards towards the domain. Now some of you may be wondering if I've accidentally drawn two of these orange dependency arrows in this diagram upside down, specifically, the arrow between the persistence project and the application project, and the arrow between the infrastructure project and the application project. Now this is a completely reasonable assumption because it's probably quite different than what you've seen with the classic three layer architecture. However, these inverted dependencies are in fact correct. When building this more modern architecture we utilize the dependency inversion principle. It states that abstraction should not depend on details, rather, details should depend on abstractions. So in the persistence and infrastructure layers we implement the inversion of control patter. That is, our dependencies oppose the flow of control in our application. This provides several benefits, such as providing independent deployability. That is, we can replace an implementation in production without affecting the abstraction that it depends upon. It also makes architecture more flexible and maintainable as well. For example, we can swap out our persistence medium and infrastructure dependencies without having negative side effects ripple throughout both the application and domain layers. This is highly useful for agile applications where we often defer implementation decisions as late as possible when we have a much better understanding of the specific needs of our application and its implementations. This is a strategy referred to in Agile software development as the last responsible moment, an idea which we'll talk more about in the last module of this course. Please also note that sometimes we need to add an additional dependency from the persistence project directly to the domain project when using an Object Relational Mapper, that is an ORM. This is the dashed orange arrow in the diagram on the right. This is necessary for the ORM to map domain entities to tables in the database since the persistence layer needs to know about the entities contained in the domain layer. Using an ORM is optional when creating clean architecture, but it can save a tremendous amount of development, time, and effort if used correctly. Here's a quick visual example to show how all of these classes and interfaces are all wired together in our demo application. We have our presentation project, application project, domain project, persistence project, infrastructure project, and our cross-cutting concerns project. In the presentation project we have a SalesController that has a dependency on the ICreateSalesCommand interface in the application project. The CreateSaleCommand class in the application project implements this interface. This class contains the high level application logic that fulfills the use case for creating a new sale. The class has a dependency on the IDatabaseService interface and the IInventoryService interface, both of which are contained in the application project as well. The DatabaseService class in the persistence project implements the IDatabaseService interface, and the InventoryService class in the infrastructure project implements the IInventoryService interface. As we can see, all of the dependencies point towards the application, and thus transitively towards the domain. All of the details, that is implementations, depend upon abstractions, which are interfaces, and we utilize in version of control for both the persistence projects dependency on the on the

application project, and the infrastructure project's dependency on the application project as well. Our cross-cutting concerns are a bit different, as there are typically multiple projects that all have dependencies upon them. For our cross-cutting concerns we store both the interfaces and the implementations in the cross-cutting concerns project. For example, the IDateService interface and the DateService class, which implements this interface, are both contained in the cross-cutting concerns project. This is because multiple projects need to reference the IDateService interface, so it must be contained in the cross-cutting concerns project referenced by all of the other projects. So why would we want to implement an application layer in our architecture? First, with an application layer we're placing focus on the use cases of the system, which, as we've stated before, are essential to the primary inhabitants of the architecture, that is the users. This provides us with the same benefits that we discussed in the first module of this course. Second, we embed our use cases as high level, executable code, which then delegate low level steps to other classes. This makes it very easy to understand the code, the use cases intention, and to reason about the software as a whole. This is highly beneficial for developers creating the application, and the developers maintaining the application. Third, it follows the Dependency Inversion Principle, which, as we explained earlier, makes our code more flexible and maintainable. In addition, it allows us to defer implementation decisions until later in the project, and thus evolve the architecture over time. There are also several reasons why we might not want to implement an application layer. First, the primary reason is that there's an additional cost to creating and maintaining this layer. Layers in software architecture are expensive to create and maintain, so we generally want to keep the number of layers in our system as small as possible. Second, we need to spend extra time thinking about what belongs in the application layer, versus what belongs in the domain layer, rather than just throwing it all in a business logic layer. Third, the inversion of control between the application layer and the infrastructure layer is often counterintuitive for novice developers. However, after you've wrapped your brain around it, it becomes quite a bit easier to reason about. So now let's take a look at how the application layer is implemented in our clean architecture demo.

Demo

It's really important to note, again, that this demo application is the simplest application that I could create to demonstrate the basic concepts of this set of architectural patterns, practices, and principles. In addition, I've also stripped out other common aspects of modern software applications like validation, security, logging, event handlers, and much more. What's important right now is to capture the essence of these practices. Everything else will just get in the way of learning. However, I'll attempt to point you in the right direction to learn more about these other aspects of software architecture at the end of this course. Let's begin by taking a look at the application project. Again, we see that our filters correspond to the key entities, that is the aggregate roots in our application. For example, customers, employees, products, and sales. These folders contain the commands, queries, and other application services that correspond to the use cases surrounding these key entities. In addition, we have a folder called Interfaces, which

houses all of the interfaces, that is the high level abstractions, that represent the external dependencies contained in the presentation, persistence, and infrastructure projects. For example, the IDatabaseService interface and the IInventoryService interface. From the application projects point of view it doesn't care how the low level details of these dependencies are implemented, it is only concerned with the high level, abstract representations, that is the interfaces, that these dependencies will implement. It's up to each class that implements these interfaces, for example, the DatabaseService class and the InventoryService class to decide how they want to implement the low level details. In order to follow the flow of control through the application let's start with the SalesController class in the presentation layer. The SalesController class is an ASP. NET MVC controller that handles displaying web pages associated with the sales entities. It doesn't matter what type of presentation framework we're using, we could just as easily have used Winforms, Webforms or built a simple console application, and the general concepts that we'll be demonstrating will be the same. What's important to note is that the SalesController class is using dependency injection via constructor injection to inject interfaces from the application project into itself. These interfaces provide an abstract representation of the commands and queries contained in the application project that the SalesController will use to complete its tasks. The SalesController then delegates the work to these commands and queries, so that they can perform the appropriate work, and either modify application state or return the results of the query. The SalesController isn't interested in knowing how any of this work is done. It's only concerned with how to present this information to the users. For example, the SalesController's Index method simply executes the Get-SaleListQuery, and displays the results as a view. The detail method executes the Get-SaleDetailQuery, and returns a detail view. The Create method returns a view that allows the user to enter the information necessary to create a new sale, it does this with the ViewModel to hold the data necessary to create that sale, and the HTTP overload of the create method, takes in that ViewModel filled with the data the user entered, and passes it into the CreateSaleCommand as an argument, then redirects the user back to the index page. Now I'll look at the CreateSaleCommand to see how this application logic is implemented. In the next module we'll take a look at the two queries, that is the Get-SalesListQuery, and the Get-SaleDetailsQuery as well. First, let's start by looking at the ICreateSaleCommand interface in the application project. This interface creates the method signature that the CreateSaleCommand class will implement. We have just a single method called Execute that takes a CreateSaleModel. This model will contain the data necessary to create a new sale. Next, let's take a look at the CreateSaleModel class in the application project. This class contains all of the data necessary to create a sale. As we saw previously, this object, and its data, will be passed from the SalesController into the execute method of the CreateSaleCommand. Please note that this class is using the unique identifier for each related entity necessary to create a new sale. That is, CustomerId, EmployeeId, and ProductId. In addition, this class also contains data to represent the quantity of products sold. Now let's take a look at the CreateSaleCommand class itself in the application project. First, the CreateSaleCommand implements the ICreateSaleCommand interface. This is necessary so that our dependency injection framework can inject this class in place of the interface at runtime. This way the SalesController in the presentation layer, which uses the ICreateSaleCommand interface, will be working directly with

the CreateSaleCommand class instead. The class contains definitions for four private readonly fields for each of our four dependencies. They are the IDateService, which provides us with current date and time information. This interface is implemented by the dateService in the common project, which contains cross-cutting concerns, IDatabaseService, which provides us with access to the database. This interface is implemented by the database service class in the persistence project. ISaleFactory, which creates a new sale entity given the necessary data. This interface is implemented by the SaleFactory class in the application project itself. And finally, the IInventoryService, which provides us with an interface to the inventory service, and allows us to communicate with the inventory system. This interface is implemented by the inventory service class in the infrastructure project. All four of these dependencies are injected into the constructor of this class via dependency injection. The Execute method of the CreateSaleCommand class takes a single argument that is a CreateSaleModel class, and returns void. That is, the command does not return anything to the SalesController. The logic for this command should read like high level application logic, which delegates low level implementation logic to the respective implementation classes. We start by getting a date using the GetDate method of the dateService. Next, we get a single customer from the customers repository in the database service that matches the CustomerId in the CreateSaleModel class. Then we get the matching employee from the Employees repository, and the matching product from the Products repository. For quality we'll also create a variable called Quantity to temporarily hold the data from our CreateSaleModel's quantity property. Now we'll use a sale factory to create a new sale entity. Using a factory keeps the low level details for how to create a sale out of our high level application logic. In addition, it provides us with a seam in the code to make testing much easier, which we'll see in a later module. Now we add the newly created sale to the sales repository contained in the database service class. Save the new record to the database, and notify the inventory system that a sale has occurred, passing it the unique identifier for the product that was sold, and the quantity of products that were sold. The high level application logic in this command represents the use case for creating a new sale. The code is very simple, straightforward, and easy to read and understand. In fact, it practically reads like constructions telling us exactly how to create a sale. Please note that there are more advanced design patterns for handling commands and command logic. For example, we could have used the Gang of Four command object pattern or the command handler pattern common in CQRS architectures instead; however, I'm attempting to keep things as simple as possible for this demo. Now let's take a look at some of the dependencies that our CreateSaleCommand depends upon. First, let's look at the IDatabaseService interface. This interface is contained in the application project, since it is a high level application abstraction. It contains the application's interface to the database or their persistent stores using the repository pattern, and the unit of work pattern. The repository pattern represents entities contained in the database table as a collection of domain objects using a simple collection like interface. That is, it provides methods to add, get, and remove objects from the database, which in turn performs the appropriate create, read, update, and delete operations when the changes are saved to the database. The unit of work pattern maintains a list of domain objects affected by a command being executed in the application, and coordinates the writing of these changes to the database. So with these two patterns in place we can access our customers, employees, products, and sales as if they

were a collection of objects rather than deal with them like database tables. In addition, we have a Save method, which is used to save changes to the database using the unit of work pattern. The IDatabaseService interface is implemented by the DatabaseService class in the Persistence project. This class inherits from DbContext, which is a base class used by Microsoft Entity Framework to handle all of the low level tasks to map domain objects to database records. All of the low level code contained in this class is specific to Microsoft Entity Framework, and thus, our application logic isn't concerned with any of these details. This means that we can swap this implementation out with another implementation without breaking any of the application or domain logic. For example, we could replace entity framework with another ORM like in Hibernate or just write around JSON serializer implementation instead. However, because we're following the contract that the IDatabaseService interface specifies, changing this implementation should ideally not cause any changes in the application or domain projects. I've chosen to use the IDbSet Interface from Entity Framework in the IDatabaseService center base to keep things simple for this course. IDbSet represents a collection of Entities just like the repository pattern Entails. However it's an Interface specific to Entity Framework. Using this Enterface directly isn't a very clean practice as it leaks the IDbSet obstraction from Entity Framework and into our application layer. In addition, exposing all of the repositories and the save method in the single IDatabaseService interface potentially violates the interface segregation principle. So, if you'd like to see a much cleaner, yet slightly more complex implementation please see the following branch in my GetHub repository. This solution completely eliminates all application level dependencies on entity Framework as well as the IDatabase Service Interface. It does this by using a seperate repository interface and coresponding class for each agrigate root entity to impliment the repository pattern. In addition it uses a standalone IUnitOfWork interface and corresponding class to implement the UnitofWork Pattern. The remaining dependencies, that is the DateService, InventoryService and SalesFactory look ruffly the same as what we just saw with the DatabaseService Dependancy. They all use a C# Interface to define the high-level application interface. Encapsulate the low-level implentation details in a corresponding emplimentation class. And deligate the low-level implimentaion details to the correponding class that implimates the interface. The only real difference with the other three dependencies are the following. First, the inventory service class is contained in the infrustucture project since it doesn't specifically pertain to persistence. Second, both the IDateService interface and it's corresponding date service class are both stored in the Common Project since the DateService across And third, the ISaleFactory Interface and the SaleFactory class are both contained in the application project since the saleFactory is an application level service. I encourage you take time to down load the sample code and take a look at these other dependencies. Seeing multiple examples will really help you to solidify these ideas in your mind. We covered a lot of code in this demo in a short amount of time. If you had difficulty following along I recommend that you spend some time becoming familiar with the sample code, and then I encourage you to watch the demo again, to make sure that you fully grasped all of the concepts that we're discussing. If you've understood the concepts we've covered so far, the rest of the demos in this course should be relatively easy to follow along. If, however, you're having difficulty following the demos it might be worth taking a bit of extra time to clear things up now before we move onto the next module.

Summary

In this module first we learned about architectures with an application layer, and how this differs from a traditional three-layer architecture. Next, we learned about the dependency inversion principle, and how we can use it to make our architecture more flexible and maintainable. Then we learned about the pros and cons of implementing an application layer. Finally, we saw a demo where we walk through the code used to implement a command in the application layer of our demo application. In the next module we'll learn about command query responsibility separation in our architecture.

Commands and Queries

Introduction

Welcome to our next module on Clean Architecture. I'm Matthew Renze with Pluralsight, and in this module we'll learn about using commands and queries in our architecture. As an overview of this module, first, we'll learn about commands and queries, and how we should strive to keep them separated in our architecture. Next, we'll learn about three types of CQRS architectures. Then, we'll discuss the pros and cons of CQRS practices. Finally, we'll take a look at the query side of our demo application, so let's get started.

Commands and Queries

Back in 1988 Bertrand Meyer taught us that there were two kinds of methods in object oriented software. First, we have a command. A command does something, which means that it should modify the state of the system, but it should not return a value. Next, we have queries. A query answers a question, which means that it should not modify the state of the system, and it should return a value. Bertrand taught us that we should attempt to maintain command query separation where possible. There are many reasons why this is a good idea. For example, to avoid nasty side effects that hide in methods that violate this principle. However, as Martin Fowler points out, this is not always possible. For example, if you have a stack, and you want to pop the first item on the stack, the pop method removes the top item from the stack, which is a command, and returns that top item, which is a query. In addition, if you want to create a new database record you create the database record, which is a command, but you might also need to return the newly created ID, which is a query, so there are clearly exceptions to this rule, but in general we should strive to maintain command query separation where possible. Command Query Responsibility Separation architectures or CQRS architectures expand this concept of command query separation to the architectural level. In general, we're dividing the architecture into a command stack, and a query stack, starting at the application layer. This is done for various reasons. The primary reason is that queries should be optimized for reading data, whereas

commands should be optimized for writing data. Commands execute behaviors in the domain model, mutate state, raise events, and write to the database. Queries use whatever means is most suitable to retrieve data from the database, project it into a format for presentation, and display it to the user. This change increases both the performance of the commands and queries, but equally important, it increases the clarity of the respective code. CQRS is domain-centric architecture done in a smart way. It knows when to talk to the domain via commands, and when to talk directly to the database via queries. There are three main types of CQRS, which we'll take a look at next. The first type of CQRS, we'll call it single-database CQRS for lack of an existing standardized name. This type of CQRS has a single database that is either a third normal form relational database or some type of NoSQL database. Commands execute behavior in the domain, which modify a state, which is then saved to the database through the persistence layer, which is often an ORM, that is an Object Relational Mapper like in Hibernate or Entity Framework. Queries are executed directly against the database using a thin data access layer, which his either an ORM using projections, LINQ to SQL, SQL scripts or stored procedures. This single database CQRS is the simplest of the three types of CQRS architectures. The second type of CQRS architecture we'll call Two-database CQRS. This type of CQRS has both a read database and a write database. The command stack has its own database optimized for write operations. For example, a third normal form relational database or a NoSQL database. The query stack, however, uses a database optimized for read operations. For example, a first Normal Form relational database or some other denormalized read optimize data store. The modifications to the right database are pushed into the read database either as a single coordinated transaction across both databases or using an eventual consistency pattern. That is, the two databases may be out of sync temporarily, but will always eventually become in sync, typically on the order of milliseconds. This type of CQRS is more complex than the first, but can afford orders of magnitude improvements in performance on the read side of the system. This makes quite a bit of sense because we generally spend orders of magnitude more time reading from a database than we do writing to it. The third type of CQRS system is typically referred to as event sourcing. The main difference here is that we do not store the current state of our entities in a normalized data store. Instead, we store just the state modifications to the entities over time, represented as events that have occurred to the entities. We store this historical record of all events in a persistence medium called an event store. When we need to use an entity in its current state we replay the events that have occurred to that entity, and we end up with the current state of the entity. Then, once we've reconstructed the current state of the entity we execute our domain logic, and modify the state of the entities accordingly. This new event will then be stored in our event store, so that it can be replayed as needed. Finally, we push the current state of our entity out to the read database, so our read queries will still be extremely fast. This is the most complex of the three types of CQRS, but it has some very powerful benefits in exchange for the additional costs. First, since the current state of each entity can only be derived by replaying the sequence of events that have occurred to that entity, the event store acts as a complete and guaranteed to be true audit trail for the entire system. This is highly valuable in heavily regulated industries where this type of auditability is necessary. Second, we can reconstruct the state of an entire entity at any point in time. This is useful for determining what the state of an entity was at any previous point in time in the system,

and this is also very useful for debugging. Third, we can replay events to observe what happened in the system. This is very useful for diagnostics and debugging. In addition, this is also very useful for load testing and regression testing in a test environment using existing production events that have occurred in the system. Fourth, we can project the current state of our entities into more than one type of read optimized datastore. For example, we can simultaneously populate and then query fast text indexing services like Lucene, graph databases, OLAP cubes, In-memory databases, and more. This means that each query can request data from the datastore optimized for querying and presenting the corresponding data. Finally, we can rebuild our production database just by playing the events. All we need to do to get the current state of our system is replay all of the events that have occurred in the past, and we end up with the current state of the system. These are some very powerful features if your architecture needs them; however, it can also be a significant additional expense if you don't actually need any of these features. In addition, despite the fact that most people new to event sourcing assume that the right side of the system will be slow, in reality these systems are actually much faster than you would imagine. There are also several types of optimizations that can be added, like generating periodic snapshots, if the performance of the right side of the system becomes an issue. So why would we want to use a CQRS architecture? First, if you're implementing domain-centric design, implementing CQRS is more efficient from a coding perspective. Commands are coded to use the rich domain models to modify state, and queries are coded directly against the database to read data. Second, by using CQRS we're optimizing the performance of both the query side and the command side for their respective purposes. Depending upon which type of CQRS we implement, this can mean orders of magnitude improvements and performance. Third, by using event sourcing we gain all of the benefits we discussed previously, and a few more that were not discussed. As systems become more complex or require high degrees of auditability these features can become highly valuable to both the business and to the developers. So why would we not want to use CQRS? First, there's an intentional inconsistency in the design of the command stack versus the query stack. Inconsistency in general makes software more complex and difficult to reason about; however, we gain consistency within each stack as a tradeoff. Second, with two-database CQRS, having both a read and write database is more complex, and potentially introduces an eventual consistency model to your databases. Third, event sourcing entails higher cost to create and maintain the event sourcing features. If you'll not be deriving sufficient business value from these additional features event sourcing might not pay for itself in the long run. So now let's take a look at the query side of our demo application.

Demo

Now let's take a look at the query side of our demo application. Once again, I'm trying to keep things as simple as possible to teach you just the basic concepts without overwhelming you with the unnecessary complexities. As a result, I'm just going to show you an implementation of single database CQRS. That is, the first type of CQRS architecture that we looked at, which uses just a single database for both the read and the write side of the

application. However, at the end of this course I'll point you in the right direction to learn how to implement the more advanced CQRS architecture concepts. Let's start again with the SalesController class. As you may recall from the last demo, we injected two queries into the constructor of the controller via dependency injection. Then we used the first of these two queries in the index method to return a list of sales transactions, and we used a second query in the details method to return the details for a single sales transaction. Now let's take a look at the IGetSalesListQuery interface in the application project. This interface contains a single method called Execute that takes no arguments, and returns a list of SalesListItemModels. The SalesListItemModel class is a simple data transfer object that is a DTO in the application project that holds the data necessary to display a list of sales to the user. It only contains the data for a list of sales; however, it is not concerned with how this data will be presented to the user. That's the responsibility of the presentation layer. This data transfer object contains the following fields; sales ID, the sales transaction date, CustomerName, EmployeeName, ProductName, the UnitPrice of the product, the Quantity of products sold, and the TotalPrice of the sales transaction. The GetSalesListQuery class implements the IGetSalesListQuery interface. Once again, we're using dependency injection to substitute this GetSalesListQuery class in place of the IGetSalesListQuery interface at runtime. So the SalesController will be delegating the call to the Execute method on the interface to the Execute method in this class. At the top of this class we have a single private readonly field, which contains a reference to the IDatabaseService interface. We inject this interface into the constructor of the class at runtime using constructor injection in our IoC framework. The Execute method for this class will create the database and return a list of sales list item models. We're going to be using LINQ to entities with projections and fluent syntax to both simplify and optimize the query, so let's break this down, so that we can understand each of the concepts I just mentioned. First, LINQ, spelled L-I-N-Q, stands for Language Integrated Query. It's a language feature in the. NET framework that allows you to write query expressions in the. NET languages, like C#. LINQ to entities is a feature of that allows you to use LINQ queries against entities and Entity Framework. Entity Framework then generates the appropriate SQL query to produce the data necessary to hydrate the entities. Fluent syntax is one of the two styles of LINQ queries, the other being query expressions. It uses a concept known as method chaining to pass data from a child method into its parent method, and so on up the links of the chain. Thus, we can chain together a series of methods to compose a complete LINQ query. A projection is a subset of data with a different shape than the actual data stored in the database. For example, rather than pulling back all of the customer fields, employee fields, and product fields, which Entity Framework would do by default to create this list. Instead, we just want the customer name, employee name, and the product name. Let's read through this we and see exactly what it's doing. First, we're accessing the Sales Repository of the database service interface. Next, we project data into our sales list item model using a LINQ select method. In the class initializer of the SalesListItemModel class we set the following properties. We set the ID property of each SalesListItemModel to the Id of our corresponding Sales Entity, Date to the date of the sales transaction, CustomerName to the Name property of the Customer entity, EmployeeName to the name of the Employee, ProductName to the name of the Product, and we also set UnitPrice, Quantity, and TotalPrice to the corresponding values of the sales entity. Finally, we call ToList on the results of our sales query, so that we're returning

a list instead of an IEnumerable. This is beneficial for a few reasons, for example, to avoid multiple iterations over an IEnumerable collection, and to provide list semantics rather than innumerable collection semantics to the presentation layer. Writing your LINQ query with a projection in this way is significantly different in terms of the SQL query that Entity Framework will generate, compared to if we used Entity Framework or any other ORM with either eager loading or lazy loading of related entities. In fact, to demonstrate how a projection simplifies the SQL query executed against the database, let's compare the three different types of queries. That is, an eagerly loaded query, a lazy loaded query, and finally, our projected query used in the demo application. The first query was generated by Entity Framework using a lazy load of all of the related entities, in order to hydrate the entities necessary to produce the sales list. As we can see, this generates 10 queries, that is 1 query for the sales data, and 1 query for each of the 3 customers, employees, and products related to the sales. In addition, when all 10 of these queries have been executed these queries have essentially loaded the entire database that is all rows in all four tables rather than only the data necessary to generate the sales list. This would produce very slow query response times, and put excessive load on the database if the ORM was used this way. The second query was generated using an eager load of all of the related entities. That is, it loaded the related entities on demand, rather than one at a time like our previous query. As we can see, this reduced the number of queries down from 10 to a single query; however, this query is still loading the entire database, that is all fields from all 4 tables when it is executed. While there are only two database fields in each of the related entities, that is customers, employees, and products, if each of these entities had dozens of fields and dozens of rows they would all need to be loaded in order to produce the sales list that we're trying to create. Finally, here's the query that was produced by the LINQ to entities projection we created. As we can see, there's only a single query that is generated by our ORM. In addition, this query returns only the fields necessary to produce the list of sales transactions to display on the user interface. This is significantly more performant, in terms of query time, and significantly more efficient, in terms of database load, than our two previous examples. While this is a very simple example in this demo application, you can see how this simple change of separating commands and queries in our architecture, and using the data access method most appropriate for the query side of the system, can produce dramatically different results for your query performance in database load. The GetSaleDetailQuery works pretty much the same as the GetSaleListQuery we just saw. Once again, we use constructor injection to inject our database service dependency. In the Execute method we take in the unique identifier for the sale that we want to return details for. Then we use LINQ to entities with fluid and syntax to filter and project the entity into a model to return to the presentation layer. First, we access the sales repository of the database service. We select all rows where the ID of the sales record matches the specified saleId, we project just the required fields into the properties of the SaleDetailModel. Then we select a single element from this list, since it should not be possible to have duplicate sales IDs in our database, and we return that single sale detail model to the presentation layer. While we used LINQ to entities with Microsoft Entity Framework for the queries in this demo, we could have just as easily have used projections within Hibernate, LINQ to SQL, embedded SQL queries, stored procedures, or many other data access methods. The key concept is that you should strive to separate your commands and queries, and use the style of

coding that works best for the respective command and query stacks in your system. Command should read like high level instructions working with entities in the domain model to modify, state, and save changes, while queries should bypass the domain model, and query the database directly, by whatever means is most appropriate to display the necessary data to the users. Implementing the other two kinds of CQRS, that is to database CQRS and event sourcing, incorporates these same principles, but does so in more advanced and powerful ways.

Summary

In this module, first we learned about commands and queries, and how we should strive to keep them separated in our architecture. Next, we learned about three types of CQRS architectures. Then we discussed the pros and cons of CQRS practices. Finally, we took a look at the query side of our demo application. In the next module we'll learn how to use functional cohesion to organize our classes, folders, and namespaces.

Functional Organization

Introduction

Hello again, and welcome back to Clean Architecture. I'm Matthew Renze with Pluralsight, and in this module we'll learn about Functional Organization in our architecture. As an overview of this module, first we'll learn about the screaming architecture practice. Next, we'll learn about functional versus categorical cohesion. Then we'll learn the pros and cons of using functional organization. Finally, we'll see a demo showing the benefits of this style of organization.

Functional Organization

The screaming architecture practice is based on the idea that your software's architecture should scream the intent of the system, hence the name screaming architecture. We do this by organizing our architecture around the use cases of the system. Use cases are representations of a user's interaction with the system. For example, interactions like getting a list of all customers, purchasing a product or paying a vendor. The screaming architecture practice is best explained using a metaphor about the architecture of buildings. Let's take a look at this blueprint. We have some bedrooms, a dining room, a living room, kitchen, and a bathroom. It's pretty easy for us to determine the intent of this architecture by quickly scanning across the blueprint. This is clearly the blueprint for a residential building of some kind, and the intent of this architecture is to facilitate a residential living environment. The rooms of this building embody the use cases of the building. We sleep in a bedroom, we cook in a kitchen, we eat in a dining room, and so

on. Simply by looking at the rooms contained in an architectural blueprint, which represent the use cases of the building, we can quickly determine the function and intent of the architecture of the building. Now rather than looking at the blueprints for a building let's take a look at the bill of materials for a building instead. We can see that we have some appliances, cabinets, doors, fixtures, and more, but it's very difficult to determine the intent of this architecture. By looking at a list of components used to create the building, rather than the rooms that support the building's use cases, it's much more difficult to determine the function or intent of the architecture of a building. The relationship between the organization of software architecture and the ease of discovering the intent of the architecture is governed by similar principles. We can organize our application's folder structure and namespaces according to the components that are used to build the software, components like models, views, and controllers or we can organize our folder structure and namespaces according to the use cases of the system, concepts that pertain to user's interactions with objects in the system like customers, products, and vendors. For example, let's take a look at two representations of the same software architecture. On the left we have the typical MVC folder structure. Things we all recognize as MVC components, like models, views, and controllers. On the right, however, we have the same web application organized by its high-level use cases like customers, products, and vendors. It's very difficult to determine the intent of the software on the left, but it's much easier to determine the intent of the software on the right. This might sound like an arbitrary decision, whether to organize first by components or by use cases, but there are some definite pros and cons to both of these equally reasonable ways to organize our software. The primary advantage is the functional cohesion; that is, organizing by use cases is generally more efficient than categorical cohesion; that is, organizing by component types because it better models the way we maintain, navigate, and reason about software. Essentially, categorical cohesion is like storing your silverware forks next to your pitch forks and tuning fork just because they're all three forks of some kind, whereas functional cohesion is like storing your forks next to your knives and spoons because we use all three utensils when we're eating. In fact, we can extend the idea of using blueprints for our building architecture metaphor to both visualize and conceptualize our application architecture. Here we have a tree map of the application layer of our demo application. A tree map is a data visualization that allows us to visualize hierarchical data, like the hierarchical relationship between the classes, folders, and namespaces in our demo application. The main rectangle on the screen represents the application project in our overall solution. Each of the large, colored rectangles represents the first folder level within the application project; that is, the aggregate root entity folders. The smaller rectangles represent the individual classes and interfaces contained in those aggregate root folders. The size of each rectangle corresponds to the size of the respective class or interface file that it represents. The colors correspond to each aggregate root entity folder that the classes and interfaces are contained within. As we can see by looking at the blueprint of our code, it is very easy to determine what the function and intent of this application is. In addition, it's easy to see that the items that are used together are grouped together via functional cohesion. This is a principle referred to as special locality; that is, it is often more efficient to keep items that are used together near one another in physical space. In terms of our software project, this means keeping items that are used together near one another in the folder structure of our file system. In fact, we

can extend this metaphor one step further and imagine that our code is a multi-story building with each floor representing a layer of the architecture. On the ground floor we have our infrastructure layer, which contains both persistence and the remaining infrastructure classes. On the second floor we have our domain layer, which contains our domain entities. On the third floor we have our application layer, which contains our use cases, and on the top floor we have our presentation layer, which contains our user interface. When we conceptualize and visualize our software this way it becomes pretty apparent to us what the organization of a clean architecture looks like, versus what a messy architecture looks like. Simply by imagining having to live and work inside of this hypothetical building made of code we can get an intuition for whether the organization of our architecture is inhabitable or not. So why would we want to use functional organization in our architecture? First, when we organize by function we utilize the principle of spatial locality; that is, items that are used together live together, just like our forks, knives, and spoons. Second, it's much easier to find things and navigate the folder structure. If we want to work with the employee objects, like the employee models, views, and controllers we just navigate to the employees folder in the presentation layer, and they're all contained in that folder. Third, it helps us to avoid vendor and framework lock in because we're not forced into the folder structure that the vendor insists that we use to implement their framework. So why would we not want to use functional organization in our architecture? First, we lose the ability to use the default conventions of our frameworks, so we typically have to tell the framework where things are located. Second, we often lose the automatic scaffolding features of our frameworks because they are typically designed to create automatically generated template files by categorical cohesion, since it's easier for the frameworks. Third, categorical cohesion is easier during the creation process; that is, figuring out where something should go when you created it, however, it makes things more difficult over the rest of the life of the project, so you'll typically end up paying more over the life of the project, especially if the project is complex and has long lifecycle. So now let's see how we've used functional organization to organize the classes, folders, and namespaces of our demo application.

Demo

Now we'll take a look at how we've used functional cohesion to organize the classes, folders, and namespaces in our demo application. While there are various recommendations for how best to organize classes, folders, and namespaces via functional cohesion, in practice, I typically use a functional organization strategy specific to each layer of the architecture. For example, I organize my presentation layer by the aggregate root entity corresponding with each screen or web page, the application layer by the aggregate root corresponding with each use case, the domain layer by the aggregate root of each domain entity, the persistence layer by the aggregate root corresponding with each database table, the infrastructure layer by operating system components and third party external dependencies, and cross-cutting concerns by the specific aspect of the system that the class pertains to, for example, date and time, logging, security, configuration, etc. In addition, if you've built an application that follows good user experience and information architecture practices your screen should roughly correspond to the use

cases and aggregate root entities in the system. For example, the sales screen allows users to perform sales use cases with the sales entities. The same holds true for your persistence layer as well. For example, the sales tables in the database pertain to the sales use cases and sales entities. So almost all of the layers of your application end up being organized roughly by the aggregate roots of the system. So now let's take a look at the classes, folders, and namespaces in our demo application. Starting with the presentation project we can see that we have top level folders corresponding to the screens in our application, which roughly align with the aggregate root entities of our system. For example, customers, employees, products, and sales. In addition, we have a home folder for the home screen of our application as well, and a few other miscellaneous folders, like the ASP application start folder, ASP. NET Content folder for our CSS files, Dependencies folder for our structure map dependency injection files, and a Shared folder for the files shared between multiple top level root folders. The magic that tells ASP. NET MVC to use this folder structure, rather than the default folder conventions, is this CustomRazorViewEngine class. I'm not going to go into the details of how this works, since this is a general purpose course on architecture, and not a course on ASP. NET MVC 5; however, for those of you interested in learning how to do this my website contains an article with step by step instructions, a sample project, and a video describing exactly how to set this up. You can find the article at the following URL. The application project is organized by the aggregate root entities corresponding with the use cases of the system. For example, we have folders for Customers, Employees, Products, and Sales. In addition, we have a folder that contains the application level interfaces that will be implemented by other projects. The domain project is organized by aggregate root entities that compose the problem domain. For example, again, we have Customers, Employees, Products, and Sales. In addition, we have a folder for the classes and interfaces that are shared between these aggregate roots. Persistence is organized by the aggregate roots that correspond to the database tables in the system. For example, once again, we have Customers, Employees, Products, and Sales. Each of these folders contains the entity framework configuration classes used to generate the database for our demo application. Infrastructure is organized by the operating system components and the third party external systems that our application talks to. For example, in our demo application we have a folder for the classes pertaining to the inventory system. In addition, we have a folder containing the classes and interfaces involved in networking, which is a feature provided by the operating system. Cross-cutting concerns are organized by the aspects of the system that the classes and interfaces pertain to. For example, in our demo application we have just a single cross-cutting concern for the application; that is, a date service, and another cross-cutting concern for testing; that is, a mock DB set extension class. Both of these aspects are shared by multiple projects in the application, so they have been placed in the cross-cutting concerns project. In general, we would likely have many more folders contained in this cross-cutting concerns project, such as folders to contain classes and interfaces for logging, security, auditing, and more. We'll cover the specifications project in depth when we get to our module on testable architecture later in this course. However, for now, just note that it is also organized by aggregate roots as well. The organization of your applications Classes, Folders, and Namespaces may need to be quite different from what you've seen in this demo application based on the specific needs, and the context of the application you're building. What's important to note from the

example you've just seen is that we use functional cohesion in order to organize the Classes, Folders, and Namespaces of the system. In addition, it adheres to the screaming architecture practice; that is, the organization of each layer in the system screams the intent of the application. This makes it very easy to navigate the file system, find what you're looking for, and to work with things that are commonly used together. In addition, it is very easy to reason about the software based on the way it is organized.

Summary

In this module first we learned about the screaming architecture practice. Next, we learned about functional versus categorical cohesion. Then, we learned about the pros and cons of using a functional organization. Finally, we saw a demo showing the benefits of this style of organization. In the next module we'll learn about microservices and microservice architectures.

Microservices

Introduction

Hello again, and welcome to our next module on Clean Architecture. I'm Matthew Renze with Pluralsight, and in this module we'll learn about Microservices. As an overview of this module, first we'll learn about bounded context and how they can be used to subdivide a large domain model. Next, we'll learn about microservices, and how they can be used to subdivide a monolithic architecture. Then, we'll learn about the pros and cons of using microservices. Finally, we'll see how our demo application communicates with an inventory microservice.

Microservices

First, let's start with what we already know, components. Components are how we would typically subdivide the layers of our architecture once it has grown beyond a manageable size. We typically implement components as separate projects within our overall solution, then when we build these projects we create output files like DLLs and C#, assemblies in. NET, jar files in Java, and gem files in Ruby. This allows us to work on the components individually, integrate them as necessary, and deploy them independently. Our users interact with the application typically through a composite user interface, which calls the appropriate component stack, and presents the user interface as a unified view of the system. In addition, all of the data for each component stack are typically stored in a single database. Let's assume, hypothetically, that we're building an application to perform sales and support tasks. Our problem domain contains the following nouns. On the sales side we have a sales opportunity, a contact for the sales

opportunity, a sales person, a product to be sold, and a sales territory. On the support side of the problem domain we have a support ticket, a customer, a support person, a product being supported, and a resolution to the support ticket. In reality, there would likely be several other nouns in both columns, but we'll just keep this list short to make this example more manageable. In the old days we would have modeled this just like we'd been traditionally taught. We would have created a single unified domain model by merging the nouns that pointed to the same types of physical objects and mental concepts as a single entity. For example, a product is a product whether it's on the sales side or the support side, so we would model that as a single product entity. A contact becomes a customer on the support side, so we would have modeled that as a single customer entity with a bunch of nullable fields for when it was being used as a contact. A sales person and a support person are both employees, so we'd represent them as a single employee entity, and this strategy seems reasonable enough; however, as we try to model larger and larger domains it becomes progressively harder to create a single unified domain model. The problem is that models are only applicable within a specific context. When we try to force a model to work within multiple contexts things just don't feel right. We have extra unused properties, methods, and validation rules that are applicable in one context, but not in other contexts. In addition, we have variations on terminology. For example, do we have a sales person, a support person or is this just an employee? A bounded context is the recognition of a specific contextual scope within which a specific model is valid. We constrain our domain model by each bounded context, and subdivide our models appropriately. Then we communicate state transitions of our cross boundary models from one domain to the other. We do this through clearly defined interfaces using either coordinated transactions or an eventual consistency model. This leads us to microservices. Microservice architectures subdivide monolithic applications; that is, the divide a single, large application into smaller subsystems. These microservices communicate with one another using clearly defined interfaces, typically over lightweight web protocols, for example, JSON over HTTP via rest APIs. Microservices can also subdivide larger teams into smaller development teams; that is, one team for each microservice or set of microservices. These services are also very independent of one another. Each one can have its own persistence medium, programming language, architecture, and operating system. In addition, you can independently deploy each microservice and independently scale them as needed, which is very beneficial for cloud scale applications. I should also note that microservices are similar in concept to service oriented architectures; however, they don't explicitly prescribe the use of an Enterprise service bus, along with several other key differences. Two very frequent questions that often come up when discussing microservices are how big should each microservice be, and where should I draw the boundaries of each microservice? This is where bounded contexts come in. Microservices have a natural alignment to the boundaries of bounded contexts. Ideally, in most cases we want each domain, each microservice, each database, and each corresponding development team to line up. Doing so provides several benefits. For example, this maximizes the cohesion of domain entities in each bounded context and microservice, and it minimizes the coupling relative to any other way we could have partitioned the system. In addition, this allows each team to focus on only a single domain of knowledge. They don't need to know about the intimate details of any other team's domain, database or microservices. They just need to know how to

communicate with the well-defined interfaces of those other microservices. This also means that each domain and microservice has a consistent data model; that is, all entities contained within a bounded context are consistent with one another, even if they are temporarily inconsistent with other microservices using an eventual consistency model. In addition, within each microservice each team can use whatever technologies, patterns, practices or principles work best to solve the specific business problems of their domain according to the specific constraints and business objectives of their respective microservices. This could entail using different architectures, persistence mediums, programming languages or even operating systems for each individual microservice based on the specific needs of that domain, and the knowledge and skills of each development team. This is highly valuable for agile software development teams using agile practices to develop large business applications. However, I should note that there's still a lot of debate in the industry as to how small microservices should be. Some experts advocate that they should be smaller than I'm suggesting, while others think that they should be larger. In addition, there's debate over where to draw the boundaries of each microservice. For example, a slightly different approach for organizing microservices is to have a single microservice for each aggregate root within each bounded context, but then have all of those microservices talk to a single database for their respective bounded context. However, information is still exchanged between bounded context via each bounded context set of microservices. There are definite pros and cons to both of these equally reasonable ways to organize your microservices; however, it's up to each of you to decide what makes the most sense for your project and architect your microservices accordingly. If we think back to our metaphor in module one of bad architecture looking and feeling like spaghetti and good architecture looking and feeling like lasagna, then microservices should look and feel like ravioli to us. Each raviolo, which is the singular form of the word ravioli, is a self-contained package of Italian food. They maintain high cohesion and low coupling, they each hide their internal contents from one another, and you can swap one raviolo with another without disrupting the internal contents of the other ravioli. There are clearly advantages to eating pasta packaged this way, relative to just globbing one giant mess of meat, sauce, and dough on someone's plate. Similarly, there are clear advantages to using microservices over monoliths in certain contexts, subject to various constraints and business objectives as well. So why would we want to use microservices? First, the cost curve of microservices is flatter than the cost curve of monolithic applications as a function of the size of the system being built. That is, the cost to build microservices is initially higher than monoliths for small systems, but the cost of building microservices grows much more slowly as the system gets bigger, so for projects with large domains and sufficiently long life cycles, using microservices in this context can, in theory, reduce the overall cost of the project. Second, subdividing a monolithic application into microservices based on bounded contexts creates systems with high cohesion and low coupling. This isn't just in terms of the cohesion and coupling in our code base, but also high cohesion and low coupling for our development teams, domain knowledge, database models, and more. Third, microservices offer independence. Essentially, you can use whatever technologies, design patterns, and development practices are most appropriate for each specific microservice and corresponding team. So why would we not want to use microservices? First, microservices have a higher up front cost than a single domain. It's not until later in a large software project that the cost curves intersect

and microservices eventually become less expensive than monoliths. If we have a small domain, a small total team size or a short project life cycle, then creating a set of microservices could likely be more expensive than creating a single application with a single domain. Second, there's a phenomena observed in the software industry referred to as Conway's' Law. To paraphrase, it states that organizations which design systems are constrained to produce systems that mirror the communication structures of their organizations. So organizations that are top-down, command and control bureaucracies, like traditional waterfall project teams, are more likely to produce top-down command and control monolithic applications; whereas, organizations that are bottom-up, self-organizing adhocracies like agile project teams, are more likely to produce bottom-up, peer-to-peer microservice architectures. If your organization and team are not set up in a manner that can communicate and coordinate effectively in the way that microservices entail, your organizational structure will likely experience significant resistance to creating microservice architectures. Third, there are additional costs in building distributed systems. You spend extra time and effort when building distributed systems dealing with network latency, fault tolerance, load balancing, and more. This is why most experts strongly advise first starting with a single application, and only break the application into microservices when the pain of dealing with a monolithic application becomes greater than the additional cost and complexity of an equivalent set of microservices. For me, this point in time almost always occurs when you try overlapping multiple bounded contexts into a single monolithic application. So now let's take a look at how our demo application communicates with a microservice.

Demo

Now let's look at how our demo application would function as a microservice. Up until now we've assumed that this demo application was a stand-alone application with a user interface for users to interact with the application, which also talked to a stand along inventory system. However, now we're going to imagine that our demo application is a microservice contained in a larger set of microservices that make up the full system. In order fulfillment microservice we'll communicate with our demo application to inform it to create a new sale, and our demo microservice will communicate with an inventory microservice to update the inventory in the system. To pull this off I've added a new project to our solution called service. The new project provides a web based API, which allows the order fulfillment microservice to create a new sale in our demo microservice. This new project contains a SalesController, which is similar to the SalesController in our presentation layer; however, this controller is a web API controller rather than an MVC controller; that is, it provides external systems with an interface into our application using XML or JSON data rather than providing an interface for users to interact with the system via web pages. We'll start with the SalesController, and work our way through the application, next to the CreateSaleCommand, then the InventoryService, down to the WebClientWrapper that sends a message to the inventory microservice. I should note that once again, I'm trying to demonstrate the simplest version of these microservice ideas as possible, so we're just going to be making simple HTTP gets and posts between the various microservices. With real world microservice

architectures we often use more advanced messaging patterns in addition to HTTP gets and posts to communicate commands, queries, and events between microservices. In addition, in the real world we would need to consider fault tolerance, dealing with bad messages, handling exceptions, and more. However, at the end of this course I'll point you in the right direction to learn how to implement these more advanced microservice architecture features. Let's start with the SalesController contained in the service project that I added to our solution. This SalesController is an ASP. NET web API controller, which allows external systems, like our order fulfillment system, to query data and execute commands within our application. The controller inherits from ApiController, which is an ASP. NET class that handles all of the low-level details for our web API controller. In the constructor of this controller we inject two query interfaces and one command interface from our application project via constructor injection. We'll resolve these interfaces to their implementing classes at runtime using our IoC framework. This controller has three methods, a Get method with no arguments, which executes the Get sales list query, and returns a list of sales. A Get method with a single argument, which executes the Get sale details query, and returns the details of the specified sale, and a Create method, which takes a CreateSaleModel, executes the create sale command, and returns an HttpStatusCode indicating that the sale was successfully created. The ICreateSaleCommand interface resolves to the CreateSaleCommand class in the application project. We saw this class during the earlier demo; however, now we're going to look at it again, to see how it communicates with an external microservice. In the constructor of this class we injected the IInventoryService interface into the class via dependency injection. This provided the CreateSaleCommand class with access to the InventoryService class at runtime using our IoC framework. At the bottom of this execute method, after we've performed all of the work to create a new sale, and save the new sale to the database, we call the NotifySaleOccured method on the inventory service interface, and notify the inventory microservice that a sale has occurred. We pass in the unique identifier for the product, and the quantity of that product sold. Please note that in a real world implementation of this type of notification the product. Id would need to be shared key between both systems. For example, a Stock Keeping Unit or SKU, which is a number that uniquely identifies a product across multiple systems. Next, we'll take a look at the IInventoryService interface in the application project. This interface represents the inventory microservice as a high level abstraction. It's not concerned with any of the details of the inventory microservice itself or how to communicate with the inventory microservice, all it cares about is that the inventory microservice can be notified when a product is sold, and it needs to know the productId, and the quantity of products sold when being notified. The InventoryService class in the infrastructure project implements the IInventoryService interface. This class does the actual work of communicating with the inventory microservice. At the top of this class we have some hard-coded configuration values, which, in the real world, would typically not be hard-coded, but rather would most likely be pulled from a configuration service of some kind that would be injected into the constructor of this class via dependency injection. This class has a single dependency that is an IWebClientWrapper interface, which it will use to send an HTTP post message to the inventory microservice. We inject this dependency into the constructor of the class via dependency injection, and resolve it to the WebClientWrapper class that implements this interface at runtime by our IoC framework. The NotifySaleOccured

method takes a productId and a quantity of products sold as arguments. First, it can pose as the URL that is the web address for the endpoint that will receive this message. It does this by replacing the productId placeholder in the address template with the actual productId. Next, it composes a message body using JSON; that is, JavaScript Object Notation, which contains the remaining data for the message; that is, the quantity of products sold. Finally, the inventory service uses the IWebClientWrapper to create an HTTP post to the URL for the corresponding product with the appropriate message body. The IWebClientWrapper interface contained in the infrastructure project defines an interface for making an HTTP post. The interface contains a single method called Post, which takes the address of an HTTP endpoint, a JSON message, and returns void. The WebClientWrapper class implements the IWebClientWrapper interface. This class uses the. NET frameworks WebClient class in order to communicate with web services and external systems via HTTP messages. We refer to this as a wrapper class because it's essentially acting as a wrapper for the underlying web client class. The wrapper is essentially just forwarding the information from it's methods into the method of the WebClient class. It does this with little to no additional logic, so that the class will be low risk, and thus will require very minimal testing. There are several reasons to use a wrapper class like this. The primary reason is to be able to mock out the dependency on the external infrastructure microservice for testing. That is, when we're unit testing our InventoryService class we can substitute the IWebClientWrapper interface with a mock; that is a fake test double, instead of using the actual WebClientWrapper class. This way we can fully test the logic of the InventoryService class without needing an actual inventory microservice to talk to. We'll discuss this idea more in our next module on testable architecture. While this has been a very simple example of how an application can be composed of a set of microservices, it should capture the essence of microservice architectures. We have some external microservices talking to our demo microservice via lightweight protocols like HTTP, queues, and message buses, and our demo microservice talks to other external microservices in the same way. Once again, please keep in mind that in the real world we generally have more complex logic for fault tolerance, handling bad messages, exception handling, and more. In addition, we would likely be communicating with microservices via other communication patterns like message queues, event hubs or message buses.

Summary

In this module first we learned about bounded contexts, and how they can be used to subdivide a large domain model. Next, we learned about microservices and how we can use them to subdivide a monolithic architecture. Then, we learned about the pros and cons of using microservices. Finally, we saw how our demo application communicates as a microservice. In the next module we'll learn about testable architecture, and how clean architecture makes testing easier, not harder.

Testable Architecture

Introduction

Hello again, and welcome to our next module on Clean Architecture. I'm Matthew Renze with Pluralsight, and in this module we'll learn about Testable Architecture. As an overview of this module, first we'll learn about Test-Driven Development, and how it drives the design of an architecture that's testable, reliable, and maintainable. Next, we'll learn about the test automation pyramid, and the various types of tests that we can create for our architecture. Then, we'll discuss the pros and cons of implementing a testable architecture. Finally, we'll see a demo of our testable architecture in action, so let's get started.

Testable Architecture

Let's start with what we already know; the current state of testing in the software industry. Unfortunately, despite great advances in both testing practices and technology to enable software testing, many software developers still do very little automated testing of their code, and when they do it's relatively ineffective or highly inefficient. There are numerous reasons given for why developers don't create high quality automated tests for their code. These reasons include not having enough time to create tests, it's not my job to test software, testing is for testers not developers, and it's too hard to create good tests because our architecture makes testing difficult. While there are compelling counter arguments to each of these claim, and more, we're going to be focusing on the last of these three arguments during this course. We're going to show how clean architecture actually makes testing easier not harder. In addition, how test driven development drives the design of a clean architecture. These two forces work side by side to create a feedback loop that feeds into one another. Test-Driven Development is a software practice where we create a failing test first before we write any production code, and use this test to drive the design of the architecture. We refer to this three-step process of Test-Driven Development as red, green, and refactor. First, we start by creating a failing test for the simplest piece of functionality we need to create. This is the red step of the TDD process. Next, we implement just enough production code to get that failing test to pass. This is the green step of the process. Then, we refactor our existing code; that is, we improve both the test and production code to keep the quality high. This is the refactor step in the process. We repeat this cycle for each piece of functionality in order of increasing complexity, in each method and class until the entire feature is complete. By using the Test-Driven Development practice we are creating a comprehensive suite of tests that covers all code passive importance in our application. In addition, by using TDD the design of each of these classes and methods is being driven by the testing process. This means that all classes and methods will be easily testable by default, since the tests are driving the design. In addition, this coincidentally makes our classes and methods more maintainable because of an interesting parallel in the physics of cohesion and coupling with both testability and maintainability. Essentially, by creating

testable code we are coincidentally creating more maintainable code. More importantly, this comprehensive suite of tests eliminates fear, fear that making changes to our code will cause regression errors in our software. If we can eliminate the fear of change in our architecture we are more likely to embrace change and keep our architecture clean. While I'm personally a very strong advocate of Test-Driven Development, and would really like to cover it here extensively, unfortunately we'll have to defer a more in-depth discussion of TDD to another course. What's important to note for this course is that Test-Driven Development is a key component to creating a clean and testable architecture. There are a variety of types of tests that exist in the world of software testing. Some tests are based on what they are testing. For example, unit tests, integration tests, component tests, service tests, and coded UI tests. Some are based on why they are being tested. For example, functional tests, acceptance tests, smoke tests, and exploratory testing. Still, others are based on how they are being tested. For example, automated tests, semi-automated tests, and manual tests. While we don't have time to cover all the various types of tests that can be applied to our software architecture, we will, however, cover a subset of the most common of these types of tests, and how our clean architecture makes them easier rather than harder to create and maintain. In the book, Succeeding With Agile, Mike Cohn describes a concept he referred to as the test automation pyramid. The test automation pyramid. The test automation pyramid identifies four types of tests. First, we have unit tests, which are automated tests that verify the functionality of an individual unit of code in isolation. Second, we have service tests, which are automated tests that verify the functionality of a set of classes and methods that provide a service to the users. Third, we have coded UI tests, which are automated tests that verify the functionality of the full application from the user interface down to the database. Finally, we have manual tests, which are tests performed by a human that verify the functionality of the full application as well. The test automation pyramid captures the essence that each type of test becomes more costly the further up the pyramid we go. As a result, we want to have a large number of low cost tests and a small number of high cost tests. For example, unit tests are relatively quick and easy to create, run extremely fast, rarely fail due to false positives, and are relatively inexpensive to maintain, so they are much less costly than the other types of tests. However, coded UI tests are much more difficult and time consuming to create. They also run much slower, they're more brittle, relatively unreliable, and relatively more difficult to maintain. So they are relatively more costly than other types of tests lower on the pyramid. Given this, we want to maximize the return on investment from our testing efforts by creating the right balance of each type of test relative to the cost of the test, and the benefit that the test will provide, so we should anticipate creating lots of small, low cost unit tests, some medium cost service tests, a few high cost coded UI tests, and very few repetitive manual tests. Doing so, in theory, gives us the most bang for the buck for our testing efforts. Once final type of test that I'd like to discuss is a type of test called an acceptance test. Acceptance tests verify the functionality of the application like the other tests do; however, the main difference is that they are typically written in the language of the business, and used as the criteria by the product owners and product stakeholders to determine when a certain feature is considered complete and functioning as expected. These tests are often done either as manual tests or as coded UI tests. However, using manual tests and coded UI tests for acceptance testing is problematic for the various reasons we discussed earlier,

so one place where clean architecture really helps us make testing easier, and get the most value out of our acceptance testing, is by allowing us to replace high cost manual encoded UI acceptance tests with automated service level acceptance tests. This is possible with clean architecture because all of our application and domain logic can be tested in isolation. To do so, first, we eliminate the user interface from our acceptance tests. We do this by having our acceptance test work directly with the commands and queries in our application layer. Next, we eliminate the database from our acceptance tests as well. We do this by mocking out the real database with a fake in-memory database instead. Then, we mock out the operating system and any external dependencies in the infrastructure layer. We do this by replacing the dependencies with fake test doubles called mocks that act as surrogates for the real dependencies. We also do the same for our cross-cutting concerns. Acceptance criteria should be written in the language of the business to describe business functionality that the application needs to provide. They should focus on what is essential to the business, and not implementation details. Thus, implementation details like presentation, databases, third party dependencies, etc., should all be irrelevant to these acceptance criteria. In addition, because we've kept our presentation, persistence, and infrastructure projects thin; that is, they contain minimal logic, and no application or domain logic, eliminating them from our acceptance tests should pose little risk from the standpoint of verifying business functionality; however, there can be exceptions, for example, verifying non-functional requirements, security requirements or auditing requirements, but we can handle verifying these types of non-functional requirements in other ways. With this style of acceptance testing our acceptance tests are focused on what is essential in our application; that is, the business domain, and the use cases in the application layer, not the implementation details like databases, operating systems, third party dependencies or cross-cutting concerns. Doing this allows us to minimize the number of coded UI tests in our application, so rather than using coded UI tests for acceptance testing, we can instead reserve them for what are called smoke tests. Smoke tests are a very small number of simple, full-system tests that just verify that the application actually runs, and nothing more when all the pieces are assembled at runtime. This reduces the cost of creating and maintaining a large number of these costly and complex coded UI acceptance tests, and reduces the likelihood of false positives, which degrades our confidence in our comprehensive suite of tests. In addition, this also allows us to minimize the number of manual acceptance tests as well. Thus, we reduce our manual testing costs, and free up our testers to do much more valuable and rewarding work, like exploratory testing, and ensuring a high quality user experience for our end users. We'll see a brief demo of how these service level acceptance tests are implemented in our demo application shortly. So why would we want to use a testable architecture? First, by applying these testable architecture practices we make our code easier to test, and if our code is easy to test we'll be more likely to create and maintain these tests. We want our architecture to encourage developers to write tests and to practice Test-Driven Development. Second, creating testable architecture improves the design of our architecture. This is because the physics of cohesion and coupling of testable code parallels that of maintainable code. So, by virtue of adopting Test-Driven Development, and creating a testable architecture we're actually creating a more maintainable architecture as a result. Third, by creating a comprehensive suite of tests for our architecture we eliminate fear. By eliminating the fear that changes to

our code will break the code we're much more likely to embrace change, continuously refactor our code to improve it, and keep our architecture clean. So why would we not want to create testable architecture? Honestly, there are very few reasons I can think of to not want to create testable architecture. In fact, testability is probably one of the biggest selling points, in my opinion, for adopting clean architecture practices. However, for the sake of counter-argument, I'll do my best to articulate a few cases where you might not want to create a testable architecture. First, there's an initial higher upfront cost to using Test-Driven Development and creating testable architecture, so for a very small project or for disposable software projects, for example, a simple throw away console application with clear and stable requirements, it might be more cost effective to just build the tool and test its narrow functionality by hand. Second, Test-Driven Development requires practice and discipline. It's not something you just learn in a day, and it takes quite a bit of practice to become highly proficient with TDD. In addition, it requires a lot of discipline to keep yourself from falling back into your old coding habits. Third, testable architecture practices usually require buy-in from the whole team. If the whole team isn't creating testable code, running all of the tests, and maintaining these tests, then the test and the testable code will eventually decay and become obsolete, so while I personally think it's very unfortunate, some teams simply just are not interested in creating automated tests or writing testable code. So now let's see our testable architecture in action in our demo application.

Demo

Now we'll see our testable architecture in action in our demo application. This entire demo application was created using Test-Driven Development; that is, I started each feature with a failing unit test, then I created just enough production code to get the test to pass, Then I refactored the code to improve its quality. Finally, I repeated this process until each feature was complete. While I would prefer to show you a demo of me creating the application using the TDD process, unfortunately that would take much more time than we have available, so we'll have to defer that demonstration to another course; however, I would like to show you just a few unit tests in the domain and application layer, and a service level acceptance test for this demo application. All of these tests will involve the functionality to create a new sale that we've already seen in previous demos. During this demo we're going to be using a few tools that you might be unfamiliar with. For example, NUnit, Moq, AutoMoq, and SpecFlow. However, I'll do my best to explain what they do as we proceed through the demo. If you're having difficulty understanding what the tools are doing or how they work, no worries. This is mostly to get you interested in learning more about testing software this way in the hopes that you might invest time learning more about testable architecture in the future. Let's start with something simple. Let's begin by looking at the unit tests in the SaleTests class. This test class contains all of the unit tests for the domain logic of the sale class in the domain project. The class begins with a TestFixture attribute. This attribute tells NUnit, which is our unit test framework, that this class contains tests that should be run as part of our suite of tests. Inside this class we have four private fields; one for our sales class, which is our subject under test, and three more for a customer, employee, and a product, which are entities we will be using while we test the

sales class's functionality. Below this we have some private constant and private static fields to hold values we'll be using for set up and verification. Next, we have the SetUp method of this test class. This method is marked with the SetUp attribute. This attribute tells NUnit that this method should be called before each unit test is run to set up the necessary preconditions for the test. In this method we create our three dependent entities; that is, Customer, Employee, and Product, and then assign them to the private fields we saw above. Finally, we create a new Sale class, which, as I mentioned before, is our subject under test, and assign it to the Sale variable above as well. Now let's take a look at the actual unit tests contained in this Test class. The first test method is called TestSetAndGetId. This unit test will simply test the property getter and setter for the ID property. We mark this test method with the test attribute, which tells NUnit to run this method as a unit test. Inside this method we set the Id of our sale entity to the value contained in the private constant integer field called Id that we saw above. Then, we assert that the value now returned from the sales Id property should match the value property we are expecting. If these two values match the tests passes. If not, the test fails. The remaining unit tests for property getters and setters looks pretty much the same, so we'll just skip over them. These are relatively low value unit tests since there's minimal logic being covered in these tests, however, they are created as a result of using the Test-Driven Development process, and are necessary to maintain a comprehensive suite of tests for our domain logic. The more interesting unit tests in this class are the two unit tests that verify the logic to calculate the total price of the sale. The first of these two tests sets the unit price and verifies the total sales price is recomputed to the correct value. First, we set the sales Quantity project to the quantity of products sold from the Quantity constant we saw at the top of the file. Next, we set the UnitPrice to a simple arbitrary value like 1. 23. Finally, we assert that the value returned from the TotalPrice property matches the expected value of 1. 23. The final test in this test class also tests the total sale price logic, but does so by setting the quantity rather than the unit price, and verifying the correct sale price is returned. Now we'll look at a much more complex example. We'll look at the unit test contained in the CreateSaleCommandTests class. This test class provides unit tests for the production CreateSaleCommand class, the same class we looked at during two of our previous demos. This class creates a new sale, saves the sale to the database, and notifies the inventory service that a sale has occurred. This type of test would generally be extremely difficult to test in isolation if we are using traditional architecture, patterns, practices, and principles; however, as we'll see shortly, it is completely testable in isolation in our clean architecture. Once again, this test class is marked with the TestFixture attribute to indicate to NUnit that it contains tests that need to be run. The class begins with four private fields, a field called command, which holds the CreateSaleCommand that will be used for testing, that is our subject under test, a field called mocker of type AutoMoqer, which is a tool that makes mocking dependencies easier; that is, it will make it easier to swap out interfaces with fake test doubles, a field called CreateSaleModel, which will hold the data for creating a new sale, and a field called sale, which will act as a placeholder for the sale being created. Next, the class contains a series of private constants and static fields to hold values used during our testing process. The SetUp method for this class is marked with the NUnit SetUp attribute, like we saw before, to tell NUnit to run this method before executing each test. Within this set up method we create a new Customer, a new Employee, and a new Product. Then we create a

CreateSaleModel and assign it to our model variable. We create a sale, and assign it to our Sale variable. We construct a new AutoMoqer, and assign it to our mocker variable, then we use our newly created AutoMoqer to get a mock IDateService. This creates a fake test double for our DateService that adheres to the IDateService interface. Once we have our fake DateService test double we set up its GetDate method, and tell it to always return the test Date value that we created at the top of the class. This way any time our SalesController class calls GetDate it will always get the same date no matter what day, month or year we run this test. Next, we set up fake Db sets for our Customers, Employees, Products, and Sales repositories. The details for how this is done are in methods at the bottom of this test class. We'll take a look at these methods later. For now, all you need to know is that these methods are creating a simple, in-memory repository for each type of entity, so that we can work with these test entities without needing an actual database to store them. The first three methods also add a single customer, employee, and product entity to these in-memory repositories. Then, we use AutoMoq to create a new mock SaleFactory. This will create a fake test double for our SaleFactory class. Doing so allows us to decouple the low level logic for how to create a sale domain entity from the high-level application logic for how to create a sale, save it to the database, and notify the inventory system. In addition, by doing so we're adding a seam in our code that will allow us to inject a sale entity, which makes testing easier because we don't have to create a new sale class inside of our CreateSale command. Finally, we use AutoMoqer to create a new CreateSaleCommand, and we assign it to our command variable. Now let's take a look at the actual unit tests in this test class. First, we'll start with the TestExecuteShouldAddSaleToTheDatabase test. Once again, this method is marked with the Test attribute, which tells NUnit that this method is a test method to be run. We've already done all of the necessary set up for this test in the SetUp method we saw earlier, so we don't need to do any additional set up before running this test. Next, we execute the CreateSaleCommand, and pass in our CreateSaleModel as an argument. Then we use AutoMoqer to verify that the Add method of the Mock Sale repository has been called exactly once. If that is the case the test will pass. If not, the test will fail. Next, let's take a look at the test to verify the execute method should save changes to the database. Once again, all of the set up for this test was taken care of in the common SetUp method, so we just execute the command, and then use AutoMoqer to verify that the saved method on the Mock DatabaseService has been called exactly one time. Then we'll look at the test that verifies that the inventory system should be notified that a sale has been created. Once more, the setup of this test has been taken care of in the SetUp method, so we call the Execute method, then we verify that the NotifySaleOccured method on the Mock InventoryService has been called exactly one time with the expected ProductId and the expected Quantity of products sold. Finally, at the bottom of this class we have two private methods to set up the Mock database repositories. These methods are a bit complex, so I won't go into too much detail; however, in general they create a new generic. NET list type to a specific type of entity using an extension method that it created and stored in the common project. Then, they return that list of entities when the specified property corresponding to those entities is called on the Mock DatabaseService, so the CreateSaleCommand class will think that it's talking to a real table of entities in a database when, in fact, it's just talking to an in memory list of those entities instead. I should probably note that typically these two methods would

eventually end up being refactored out of this test class into a shared location once multiple test classes needed this functionality. Finally, let's take a brief look at the service level acceptance test for our demo application. To go in depth would require more time than we have during this demo, but I want to show you the basics of how we're able to isolate our application and domain logic from its external dependencies and create tests in the language of the business to verify the correct business functionality of the application. This part of the demo uses a tool called SpecFlow. SpecFlow allows us to describe a feature and a set of corresponding tests to verify the functionality of that feature, all using the language of the business. Behind the scenes, SpecFlow parses the English text, and matches it to C# code. So we can create, run, and verify a set of business specifications that testers, business analysts, and product stakeholders can read and understand. Once again, we'll be focusing on the functionality to create a sale. We'll start with the Create a Sale feature file. This file describes the Create a Sale feature in the language of the business. It states, as a sales person I want to create a sale to record a sales transaction. The scenario being tested is the most basic test for creating a sale, which we refer to as the happy path in software testing. This happy path test, like the feature it belongs to, is also titled Create a Sale. The test reads, Given the following sale info, followed by a table with one row of sales data, When I create a sale, then the following sales record should be recorded, followed by a table of the expected sales record, and the following sale-occurred notification should be sent to the inventory system, followed by a table of the data to be sent to the inventory system. This test describes the process for creating a sale from a high-level perspective in English and in the language of the business. It discusses customers, employees, products, sales, records, and notifications; however, there's no mention of low-level technical implementation details like user interface components, database tables or external web APIs. The underlying code that actually runs this test is contained in a SpecFlow step file titled, CreateASaleSteps. This is a plain old C# class file that contains executable code. SpecFlow automatically handles mapping the English test steps in the feature file that we just saw to methods in this file. While I won't go into the details of this code or how SpecFlow works let's just scan through the key points. In the constructor of this class we inject a class called AppContext. This class represents the entire application held in isolation. It contains fake test doubles for the database, the inventory system, and all operating system dependencies. We mock out all of these dependencies in the same way we did in the previous unit test demo we saw; however, now we're doing it for all of the key external dependencies in the application. It's essentially like we're putting the entire application and domain layers in a bubble, so that they are running in memory, but completely isolated from the outside world. The first method in this class is tagged with the Given attribute. The text string in the attribute matches the English text for the test step in the feature file we saw earlier. The method itself takes a table of data, which is populated with the table of data we saw in the feature file. This method then performs the logic to set up external dependencies like the date service, and create the input that the user would be inputting into the system. The next method handles the step described as when I create a sale. This method creates the CreateASale command, and passes it the CreateSaleModel, which represents the user input that was created in the previous step. The next method handles the step described as, then the following sales record should be recorded. This uses the table of expected data from the feature file to verify that the correct information was saved

to the database. Finally, we verify the last step; that is, then the following sale-occurred notification should be sent to the inventory system. This method uses the table of expected values to verify that the inventory system was notified with the appropriate data. While I would really like to spend more time on the topics of testable architecture, Test-Driven Development, and acceptance testing using behavior driven design, unfortunately, we need to move on to our final topic. While brief, this module was necessary to demonstrate how clean architecture makes testing easier, and how effectively testing your code drives the design of good, clean architecture. In addition, I really want to spark the idea of why testable architecture, Test-Driven Development, and acceptance testing are so important, so that you'll have the motivation to seek out additional information and training on these topics. Finally, I strongly encourage you to spend time with the downloadable demo project to inspect all of the unit tests and acceptance tests in this project. The demo project has great examples of effective and maintainable unit tests, acceptance tests, and testable classes, all created using Test-Driven Development and behavior driven design practices.

Summary

In this module first we learned about Test-Driven Development, and how it drives the design of an architecture that's testable, reliable, and maintainable. Next, we learned about the test automation pyramid and the various types of tests that we can create for our architecture. Then we discuss the pros and cons of implementing a testable architecture. Finally, we saw a demo of our testable architecture in action. In the next module we'll learn about how to evolve our architecture and wrap things up for the course.

Evolving the Architecture

Introduction

Hello again, and welcome to our final module on Clean Architecture. I'm Matthew Renze with Pluralsight, and in this final module we'll learn how to evolve our architecture over the life of the project. As an overview of this module, first we'll discuss how we can evolve our architecture to reduce risk due to uncertainty and changing requirements. Next, we'll discuss where to go for more information if you'd like to learn more about the topics in this course. Finally, we'll wrap up this module and the course as a whole.

Evolving the Architecture

What we've seen so far in this course isn't the end goal of this set of architectural patterns, practices, and principles; rather, it's just the beginning. What I've attempted to define is a starting point for building modern applications that

will benefit from this set of practices. These are typically applications built using Agile software development practices in an environment with a high degree of risk due to uncertainty and changing requirements caused by changing technologies, changing markets, and changing user preferences. This means that the architecture needs to evolve to minimize this risk due to uncertainty, and to meet these changing requirements. By placing focus on the key abstractions of the domain and application logic at the center of the architecture, and deferring user interface, persistence, third party dependencies, and cross-cutting concerns to implementation details clean architecture allows the application to more easily evolve over time. When we're making these implementation decisions we want to defer these decisions until the moment known as the last responsible moment. This term was coined by Mary and Tom Poppendieck in the book, Lean Software Development: An Agile Toolkit. The last responsible moment is a strategy for avoiding making premature decisions by deferring important and difficult to reverse decisions until a point in time where the cost of not making the decision becomes greater than the cost of making the decision. By delaying these decisions until the last responsible moment we increase the likelihood that we are making well informed decisions because we now have more information. Making implementation decisions too early can be a huge risk on some projects; however, waiting until after the last responsible moment, as the name implies, creates potential risks and technical debt as well. Evolutionary architecture practices are about creating architecture that allows us to more easily defer these decisions until the moment where we've minimized the risk due to making the decision too early, but not waited too long, and accumulated technical debt. For example, by focusing on the domain and application logic, and solving the key problems that our users need to solve, we can validate whether or not our software will actually provide real business value to our users before we invest heavily in the implementation details. In the meantime, we can just use the simplest solution that could possibly work for each of the implementation details, just enough to get us by while we validate the primary value proposition of the application; that is, whether or not our software will provide real business value to our users and our business. In addition, technology will likely change over the life of the project. For example, during the course of a project a new type of database technology may eventually be released, that makes more sense than using the database technology that we'd previously decided upon. Clean architecture makes it much easier to defer implementation decisions or replace existing implementations than an architecture built around a specific implementation. Markets may also change over the life of a project as well. For example, a simple client server application designed to support a few hundred users might balloon in demand to millions of users and need cloud scale, CQRS, and microservices to support the new workload. With clean architecture we build a solid domain and application core first, and then scale that core as large necessary, if and when the need should arise. User preferences may also change over the life of a project as well. For example, we might discover later on in the project that users now want a mobile user interface rather than the desktop user interface that they had previously desired. With this style of architecture we can easily swap out presentation technologies or support multiple user interfaces built upon the key abstractions in the domain and application layer. We want our application architecture to be flexible by default to help protect us against the unpredictability of the future. These architectural patterns, practices, and principles that we've just learned give us

this kind of flexibility and adaptability. So why would we want to create an architecture that evolves over time. First, evolutionary architectures embrace uncertainty. If you're building an application in an environment with a high degree of uncertainty it's better to have an architecture that can evolve as you learn more over time than to try to predict the future and build a rigid architecture based on those, likely to be wrong, predictions. Second, evolutionary architectures embrace change. It's inevitable that on most software projects the requirements will change over the life of the project. Having a flexible architecture allows the architecture to adapt to these changes. Finally, evolutionary architectures help us to reduce certain types of risk. If uncertainty or changing requirements are the biggest risk to your project, which has been the case with most projects I've worked on over the years, then optimizing your architecture for adaptability helps you to reduce this risk. So why would we not want to build an evolutionary architecture? First, if you have a project with very clear requirements, and you've already validated each of those requirements, then there would likely not be much uncertainty, and thus, minimal need to change the application as we learn more. Second, if you have staple requirements; that is, requirements that will not change over the life of the project, then the value of an adaptable architecture would likely not be justified by this cost. However, I feel compelled to point out that I have yet to see a project of any significant complexity in over 17 years of professional experience that has met both of these first two criteria. This may be the case with very small, simple, and disposable applications, but it's unlikely to be the case with large and complex applications with a long lifecycle. Finally, it's important to be aware that there are obviously limitations to the flexibility of clean architecture. It can't work miracles, but it's still significantly more adaptable and maintainable than any of the more traditional styles of architecture I've worked with over the years.

Where to Go Next

Now let's discus where to go to learn more about the topics we discussed in this course. I have a great list of books, courses, and websites that will help you go beyond what we've covered in this course, and take your skills to the next level. First, regarding recommended books, Martin Fowler has a book called Patterns of Enterprise Application Architecture. This book covers many of the patterns that we discussed in this course, and laid the early groundwork for clean architecture practices. It's a great book, in fact, one of the best books I've ever read on software architecture to date. It was originally published in 2003, so while it's a bit out of date, it's still a great source of information. Next, Robert C. Martin, aka Uncle Bob, is working on a new book called, Clean Architecture, which, at the time of the creation of this course, has not been released yet; however, given how influential Uncle Bob has been in the clean architecture movement, and the quality of his previous books on architecture insight, I imagine this book will become the new standard text for this style of architecture. Third, Eric Evans book, Domain-Driven Design, is a must read if you're dealing with a larger complex domain. It provides a set of patterns and practices that will help you to deal with domain complexity. In addition, the whole book is a compelling argument for why you should implement a domain-centric architecture if you have a complex business domain. Finally, if you're interested in learning more

about dependency injection, and using IoC frameworks to resolve dependencies in your code at runtime, Mark Seaman has a great book titled, Dependency Injection in. NET. While the examples are specific to. NET, this book does the best job I've seen to date of explaining the dependency injection pattern and inversion of control frameworks from both a conceptual and a practical standpoint. Next, for additional more in depth courses on Pluralsight the following course may be of interest to you, Domain-Driven Design Fundamentals by Julie Lerman and Steve Smith covers the fundamental concepts of domain-driven design. Domain-Driven Design in Practice by Vladimir Khorikov provides an in depth walk through for applying DDD practices to your application. Modern Software Architecture by Dino Esposito covers domain models, CQRS, and event sourcing in more detail. Microservices Architecture by Rag Dhiman provides in depth coverage of microservices, and Dependency Injection On-Ramp by Jeremy Clark provides a great introduction to dependency injection in IoC frameworks. Third, I have a few recommended websites. While I've tried to put links to the sources of information in each of my slides, there are a few additional links that might be of value to you. First, Martin Fowler, the author of Patterns of Enterprise Application Architecture has his bliki, which is part blog and part wiki, which provides more up to date architectural patterns and practices. In addition, Greg Young and Udi Dahan are the two biggest name in CQRS and event sourcing right now. At the time this course was created I am unaware of any books on either CQRS or event sourcing that I can recommend, so for the time being Greg and Udi's articles and their respective website are the two best sources of information on CQRS and event sourcing that I'm aware of. Finally, my website contains additional information on clean architecture patterns, practices, and principles as well. I have articles, videos, presentations, open source sample projects, and links to additional resources available. In addition, if you're interested in implementing the screaming architecture practice in ASP. NET MVC I have a step by step tutorial on that subject as well, so be sure to check out all of these excellent sources of information to learn more.

Course Summary

Before we wrap things up for this course, and for the series as a whole, feedback is very important to me. I use your feedback to improve each and every one of my courses, so please be sure to take a moment to rate this course, ask questions in the discussion board if there's something in this course that you didn't understand or would like me to clarify, leave comments to let me know what you liked about the course or if you think there's something I could do to improve upon for future courses, and feel free to send me a tweet on Twitter if you liked this course or would like to provide me with feedback in public as well. You can find me on Twitter @matthewrenze. Also, there are plenty of additional clean architecture patterns, practices, and principles that we didn't have time to discuss in this course. If you found this course valuable, and you'd like to see a follow-up course, including advanced patterns, practices, and principles please let me know. As long as I know that the follow-up course would be valuable to enough people I would love to create it. And, if you found this course valuable, please encourage your coworkers and peers to watch it as well. It will be much easier to get everyone onboard with trying these architectural ideas if everyone has an in

depth understanding of them and their value as well. In summary, in this course first we learn that clean architecture is a philosophy of designing architecture for the inhabitants of the architecture, not for the architect or the machine. Next, we learned that domain-centric architectures put the domain at the center of our architecture because the domain model is essential to the inhabitants of the architecture. Then, we learned about the application layer, and how adding it places focus on the user cases of our system, which are essential to the inhabitants of our architecture as well. Next, we learned about command query responsibility separation, and how it allows us to optimize both the command stack and the query stack of our architecture. Then we learned about the screaming architecture practice, and how we can use it to organize our classes, folders, and namespaces via functional cohesion. Next, we learned about bounded contexts, and how we can use them to horizontally partition our architecture into microservices. Then we learned about testable architecture and how clean architecture makes testing easier, not harder. Finally, we learned how to evolve our architecture to reduce risks due to uncertainty and changing requirements over the life of the project. Thank you for joining me for this course on clean architecture. I hope that you have learned some valuable new skills that you'll be able to put to great use, and I hope to see you in another Pluralsight course in the future.

Course author

Matthew Renze

Matthew is a data science consultant, author, and international public speaker. He has over 17 years of professional experience working with tech startups to Fortune 500 companies. He is a...

Course info

Level	Beginner
Rating	★★★★½ (1337)
My rating	★★★★★
Duration	2h 21m
Released	11 Jan 2017

Share course