**Course Overview**

**Course Overview**

Hi everyone. My name is Gill Cleeren, and welcome to my course, Architecting ASP.NET Core Applications: Best Practices. I'm the CTO of Xpirit Belgium and help customers with web and mobile application development. Creating an enterprise ASP.NET Core application can be a challenging task no matter if you choose to create an API combined with Angular, an MVC application or a Razor Pages application. Understanding how to set up an architecture that has proven itself for building real‑world applications and knowing which packages to use is a valuable asset for your next assignment. This course is aimed at giving you the knowledge of how to build a large enterprise‑ready ASP.NET Core API. In this course, you will build this API using clean architecture and by applying best practices and good architectural principles. You'll understand how to create a testable and maintainable code base that lends itself well to testing its different building blocks. You'll see how an API can be consumed from a client‑side application so to understand the full end‑to‑end application. Some of the major topics that we will cover include understand clean architecture and apply it to ASP.NET Core API, learn best practices and foundational architectural principles, including dependency inversion and separation of concerns, create a testable and maintainable code base and learn how to write unit tests for it, see how the API can be exposed using Swagger and consumed from a client‑side application that we will build with Blazor, implement crosscutting concerns in the API, including logging, exception handling, and authentication in a clean way. By the end of the course, you'll understand how a real‑life enterprise application built with ASP.NET Core API can be created. Before beginning the course, you should be familiar with ASP.NET Core, and you should also be familiar with the principles of REST. I hope you'll join me on this journey to learn how to build real‑life APIs powered by good architectural design in the Architecting ASP.NET Core Applications: Best Practices course, here at Pluralsight.

**Course Introduction**

**Module Introduction**

Hello, and welcome to the Architecting ASP.NET Core Applications: Best Practices course, here at Pluralsight. My name is Gill Cleeren, and I will be guiding you through this course. I hope you've come to this course to understand how to set up an architecture for an ASP.NET Core application, be that an API, an MVC application or a Razor Pages application. If that's the case, you're in for a treat. In this very course, I will show you how I typically create the applications for my customers. It's filled with best practices that have proven themselves over and over again. Software architecture is a topic I love spending time on, and applying it to .NET Core, well, that's a match made in heaven for me. I know I sound a bit excited, so let's get started straightaway. This module is an introduction module, so the real learning will only start in the next module. I want to make sure in this module that you understand what we are going to cover in this course and ensure that this course is a good use of your time. So first, I'll set the scene. I'll explain what we're going to solve for our customer, GloboTicket. This course is not a beginner's course. I, therefore, will expect that you are familiar with and preferably have practical knowledge of a certain number of technologies. I will explain what you need to know before continuing this course, remember, making sure that we're making good use of your time. And finally, I'll show you briefly the application we'll be building in this course. It's pretty hard to talk about an architecture without applying things in a practical way, so that's what we'll finish the module with.

**A Solid Architecture for .NET Core Applications at GloboTicket**

But first, let me explain to you a bit about our customer, GloboTicket, and what we are going to build for them. GloboTicket is a market leader in event ticket sales. They have been in the business for over 20 years. Worldwide, event organizers sell their tickets through the GloboTicket system for events taking place all over the world. Let's get to know the people behind the company, shall we? This is Bob T Hickett. Nice to meet you, Bob. Bob is the new IT manager at GloboTicket. He was hired to make GloboTicket the go‑to marketplace for all digital ticket sales. He has a grand vision of where the company needs to be in two years, but Bob has a bit of a problem. The internal development team consists out of a number of skilled developers that have been building the applications for GloboTicket for several years. They have a whole set of applications, both internal and customer facing. Most, if not all, of these applications have been built quite a few years ago, and the team has been using all the .NET technologies to build them. Think Web Forms, not the tech that we are going to conquer the world with. Bob, a clever fellow, noticed this early in the process. And then we have Bob with his grand vision to become the world leader, but stuck with an old technology stack. Things will need to change. Bob has secured budget to start building applications based on a modern, cloud‑ready technology stack. Of course, GloboTicket is a Microsoft‑oriented company when it comes to development, and they want to keep it that way. .NET Core is the way forward, there's no doubt about that. Ready for the cloud, cross‑platform, open source, all the things we need right here, right now. Bob has a good companion in Bethany, the team lead at GloboTicket. She leads the team, thinks about the different systems, sets up the architecture, and much more. But, indeed, that team was a bit behind in terms of knowledge, and, therefore, they have been skilling up quite heavily lately. They have attended a .NET Core training class, they've watched several great courses on .NET Core here at Pluralsight, and went to several conferences to learn about the different aspect of .NET Core and how to build applications with these new technologies. As said, the GloboTicket team is a group eager to learn. They've had a lot of training hours, so how to get started? But then, it hits them. We know all the different things now, but how do we now bring them together to create a real‑world application architecture, which is what we need? We've seen a lot of applications where we started from File, New Project. Is that really the way of doing things? Do we not instead need to think about a real architecture based on best practices? Probably a lot of other companies went through the same thing. We should follow a proven approach and not reinvent the wheel. That's where we come in. You, the learner, are hired as a consultant at GloboTicket to help the team in their quest to create an application architecture for the ASP.NET Core and .NET Core projects. In this course, together, we are going to create a real‑world application architecture that GloboTicket can use for all their ASP.NET Core development. For the application architecture, we will base ourselves on the concepts of clean architecture, often also referred to as onion architecture. Don't worry if you aren't familiar with the concepts of clean architecture, I will be explaining this starting from the next module. So to conclude this introduction, the application architecture will be an implementation of clean architecture principles in ASP.NET Core interwoven with best practices from real‑world applications. We'll use several commonly used packages in this course as well. You'll see me bring in AutoMapper, MediatR, Swagger and Swashbuckle, NSwag, and much more. The end goal, and you'll hear me refer to this many times throughout the course, will be that we have a testable and maintainable application architecture. Oh, and before I forget, the course is recorded with .NET Core 3.1 since that is a long‑term support release. But, if you want to apply all this to a .NET 5 project, feel free to do so. Everything will work just fine. We will be focusing on the architectural principles, not so much on the specific .NET version. The team at GloboTicket is looking forward to seeing and, of course, using the new architecture for their new projects. I hope you are, too.

**What You Need to Know before Starting**

As mentioned, this course is about building an architecture in ASP.NET Core and understanding how to build real‑world, testable, and maintainable applications. There are some prerequisites that I expect you already know that I won't be spending any time on explaining here. Of course, I expect C# knowledge. That's quite normal, I guess. We will be using ASP.NET Core and EF Core. This is not a beginner's course on either of these technologies, so I expect that you have at least some knowledge on the two. In fact, we will be building the application architecture for an API with ASP.NET Core, so I will expect that you are at least familiar with APIs and REST principles as well. Please note, though, that everything you learn in this course is applicable also to ASP.NET Core MVC, Razor Pages, and more. I, however, have chosen to build a UI that uses the API using Blazor, and we will explore that application's code as well. I won't be spending time explaining Blazor, but it's not required to know Blazor to follow along with this course, although. We will spend most of our time creating the API and its architecture. For the mentioned technologies, we have great paths here at Pluralsight that can get you on your way. Here you can see the link for the ASP.NET Core path and the Blazor path. Take a look at these if you want more information. I definitely recommend that you download the samples for this course as you'll see there is quite a lot of code. And before I forget, if you have questions about this course, please let me know in the Discussion tab. One final thing, what do you need? Well, nothing special really. I'll be using Visual Studio 2019 with the .NET Core 3.1 SDK, but, as mentioned, you can follow along just fine with .NET 5 or even older technologies.

**Demo: Looking at the Finished Application**

With all the administration out of the way, let's wrap up this module by taking a look at the finished application. In this course, we will work on the GloboTicket ticket management application. As you can probably get from the name, it's a back‑end application for the GloboTicket staff. In the system, they can create new events, get an overview of ticket sales for the different events, and so on. The application, in the end, is built using an API and a front end using Blazor. Throughout this course, we will focus on exploring the architecture of the .NET Core back end. In the first demo of this course, let me show you the finished application, which, as said, consists out of an API and a Blazor application. Here you can see the finished application, which consists out of a Blazor application and an API, and you can see the Swagger page over here. Let's take a look at the application. I can, here, register or log in. So let's start by logging in. At this point, the application will authenticate to do its API calls. So now we've logged in successfully, now I can browse here to the menu and look at the different pages. I can click on Events to get an overview of all the events coming up here at GloboTicket. I can edit an event and see the details. I can also add a new event here. I can also export the events to an Excel file. From the menu, I can also browse to the Categories, which gives me an overview of all the categories with their planned events. If I want, I can also include past events. I can also here add a category. And finally, we have a page to see all the sales. I have some data here for November 2020. Click on Get Sales, and I get the overview in a page with all the sales that have happened for the selected month. This is the application and its API that we'll be using in this course. And, we are good to go. Before we start looking at the application architecture, I want to spend some time making sure that we're all on the same line about foundational architectural principles, including clean architecture. That will be the focus of the next module

**Understanding Foundational Architectural Principles**

**Module Introduction**

Understanding some of the most foundational architectural principles is, of course, key to building a good architecture for our application. Making sure that we're all on the same page in terms of understanding some of the most key concepts when building a software architecture is really my goal for this module. Welcome to the Understanding Foundational Architectural Principles module, part of the ASP.NET Core Architecture: Best Practice course, here at Pluralsight. I'm Gill Cleeren, and I'll be guiding you through this module. Let's us start by taking a look at what exactly we will be covering here. First, we will look together at some of the base principles that will come into play throughout the entire architecture. If you have been writing software for some time, this will probably sound familiar, but just to be sure we're talking about the same things here, I'm going to go over these without going in too much depth here. Next, I will explain the different commonly used application architecture styles, and then we will explore the concept of clean architecture, which we will be using to build the application architecture for GloboTicket upon. One word of warning, though. This is a more theoretical module. We will start building the architecture from the next module. I'm all set. Let's dive in

**Foundational Architectural Principles**

So, as said, we'll start the module with an overview of some of the foundational architectural principles that are key to understanding the design choices for our application. Which principles do I want to cover here, then? We'll look at dependency inversion, separation of concerns, single responsibility, the DRY, or don't repeat yourself principle, and persistence ignorance. I don't want to go in too much depth here, as said. This should be merely a refresher for these topics, but we will be referring to these and using them throughout the entire course while creating our architecture. So let's start by looking at dependency inversion, one of the SOLID principles. Indeed, dependency inversion is the D in SOLID. Dependency inversion will help us with decoupling different modules within our application. Creating loose coupling is one of the key aspects to creating maintainable and easy‑to‑test software. To apply dependency inversion, we're going to introduce abstractions, typically interfaces. Normally, dependencies between these different classes go from top to bottom, so from high‑level components that contain the business logic to the implementation components. But to create dependencies between these components, it's difficult to swap these and, hence, also difficult to test later on. Once we introduce abstractions to decouple the different components, this problem will be solved and we'll get more loosely coupled components. Dependencies should, thus, be pointing to the abstractions, not to the details. Take a look at an example here. When we write code, typically we will have a number of classes. A depends on B, and B, in turn, depends on C. To use B and C, we'll introduce a reference from the one class to the next; hence, we are creating a tight coupling at compile time. When running this code, A will always use B, and B will always use C. Testing the code, for example in A, becomes more difficult because it's tightly linked to the code in B. Now, what if we introduce dependency inversion here? What I've now changed is that we now have an Interface B, which is implemented by Class B, and, similarly, I've also introduced an Interface C and a Class C. When writing the code in A, though, we're now using the abstraction, so the Interface B, thus creating loose coupling. It's important to understand that both the high level, so the Class A, and the low level, Class B, depends on the same interface, and that's the abstraction. Hence, we are inverting the normal dependency direction. At runtime, the concrete implementations of Class B is used and the normal flow is still used, but different implementations of Interface B can now be used, and, thus, the code is loosely coupled and can be tested much more easily. That is going to be a key aspect throughout his entire course. Separation of concerns is another key principle, and I assume you already have heard of or are already, hopefully, applying today. Separation of concerns is also part of the SOLID principles. This time it's the S we're talking about. At its core, it states that we need to write our applications split up in different blocks of functionality, each covering a separate concern. A concern can really be anything, but each block should just be doing one single job. Don't write your code in large chunks of code that do everything from, for example, going to the database on one line and on the next line updating the UI. Applying separation of concerns, or SoC for short, leads to more componentized, more modular code. Each module will do one single task, and, therefore, it encapsulates all logic for that functionality entirely. For anything else, it will use a different module. A good example is a logging component. When we write our application, we don't want to include code that does the logging in each and every component. Now instead we used a logging component which knows how to do logging. Other components will use the logging component for all their logging needs, really. Then, we have separated out the logging concern. When writing a typical layered architecture, each layer is also basically applying separation of concerns. We're separating the UI from the code that contains the business logic, and that, in turn, is separate from the code that contains the database access logic. Separating out concerns into different components also makes the code more maintainable, another key concept we're after for our architecture. Quite closely related is single responsibility. Single responsibility is a term we know from object orientation, which states that a class should just have one responsibility, which is the single reason to change it. Anything else should go in a new class. This single responsibility is encapsulated entirely by this class. Applying this pattern also leads to more modular systems. We're adding more smaller classes instead of constantly adding functionality to existing ones. Since we don't change existing classes, we'll less often break existing functionality. Now we can bring this OO principle also to the level of application architecture. Think again of a layered architecture, so with a UI layer, a business logic layer, and a data access layer. Each of these layers should be responsible for just one part of the functionality of the system. The UI layer is entirely responsible for the presentation of the application, while working with the data persistence system is entirely up to the data access layer. The DRY principle is another important one. If you've never heard of it, you may be thinking, should I be drying off my code? Of course not. It's an abbreviation for don't repeat yourself. Applying DRY is something I really hope you're doing all the time in your current projects, even if you don't really know that you're doing it. It states that you shouldn't be repeating code to cover a specific functionality throughout your application. If you're doing the same thing in multiple places, you're bound to make errors since you need to make the change in multiple places instead of just one. Less or even better, no duplicated code at all and, thus, encapsulating a certain functionality and specific component makes your code easier to change and will end up giving you less errors. Another principle that will apply in our application is persistence ignorance. We'll see in the next module that we are creating our domain entirely with POCOs, aka plain old CLR objects. Ensuring that these domain classes are not influenced at all by the technology used to persist them is exactly what this principle is stating. This way, the domain we are modeling with our classes shouldn't be influenced by the technology. In our case, that will be EF Core, or Entity Framework Core. So you shouldn't clutter the domain classes with things like base classes that you need to inherit from or attributes that you need to apply on the properties. Applying this principle correctly ensures that the code isn't linked to any technology and gives you the freedom to store the entities wherever you want. You may be thinking I've rarely seen a project where we decided to switch database technologies. While it happens, it might be rare, but applying persistence ignorance can also be useful to store your entities, first in the cache and then in the database. It basically doesn't matter where you store them, they're just plain objects. While, like I said, I'm not going to go deep into these patterns, this short refresher should give you enough understanding to follow along with the rest of the course. My main reason to include them here is that we'll be applying this heavily in our architecture. And don't worry where we use them. I'll show you how and why we are doing so.

**Different Application Architecture Styles**

With these principles out of the way, we're now ready to tackle the next topic, and that is understanding the commonly used application architecture styles. For our application that we'll start building in the next module, we'll be basing ourselves on the principles of clean architecture, and before we look at clean architecture, I would like to start with showing you the different styles that you'll often see and probably have used already in your professional life. The first one is what is often referred to as all‑in‑one architecture. In an all‑in‑one architecture, literally all the code for the application lives in a single project. This could be an ASP.NET Core MVC application, for example, where all the code for the entire application is contained in one single project. I also refer to this often as a File, New Project architecture. This type of application is often the result of starting for a small project using File, New Project, and the team starting to build screens and functionality and to be quick all was placed in a single project. But we all know how things go in software development, right? The application grows. Hey, add this functionality, please, and before you know it, the entire organization is running on what was once a small application. In essence, there's nothing really wrong with this approach. Even in a single project, you can still create layers through folders. But, problems will arise along the way since you will end up with a lot of classes. Now think back to that MVC application. You'll have lots of model classes, lots of controllers, lots of view models, and so on. And indeed, your models will go in a model folder, and the controllers will go in a controllers folder, so you're essentially applying layering using folders. But, nothing is stopping you from putting a controller in a models folder, so it's hard to enforce separation of concerns in the code. And with many files in the same project, the maintainability will also go down dramatically. So I said, while this works, I think it's hardly the way you want to create a maintainable enterprise application with. The second type I want to touch on here is the layered architecture, one I'm sure you're familiar with. Nearly every application is applying layering in some form or the other, which is, of course, good news. A typical layered architecture will essentially enforce splitting up the code over different blocks, typically different projects in your Visual Studio solution. This code is really split up according to its task, its concern, and so layering is really pushing through separation of concerns, which we looked at already before. Because we're splitting the code in different blocks, we're also applying the DRY principle here. We're going to try to write a certain functionality once, and if it's needed from multiple places in our application, we can actually reuse it. We're putting all the code that interacts with the database in the data access layer, and the business logic layer that lives above will use that same code. Code reuse is a great step and a good direction. We'll have the code in one place, and we can reuse it if needed. If we need to change it, we also need to change it in just one location. That, in turn, helps with maintainability. If we need to change something that is used from multiple places, we only need to do so in a single location, resulting in less errors, at least hopefully. Because we applied the separation of concerns too, the code's maintainability will also increase. It's easy to find something, and since a certain component is just doing one single functionality, it'll help not breaking other parts of the application. Finally, layering also brings better pluggability. Assume that we need to switch databases for the project. Through layering, the impact of this change would be limited to just the data access layer. So that's all good news. Here's a schema of a typical layered architecture. At the top we have the presentation layer, which is concerned with interacting with the user and presenting data to that user. That layer will interact with the business logic layer, also known as the service layer. This is the layer that is agnostic to data persistence code, but will contain things such as the business rules, validations, and the like. Finally, the business layer will interact with the data access layer or the persistence layer. This layer contains your database code, so perhaps if you're using EF Core, that will live in here, as well as repositories. There is, to be clear, nothing wrong with this approach. It has been used for many, many applications, and it works fine. But layering also has its disadvantages. For one, although we split up things in separate blocks, these blocks are still, well, dependent on each other. The business logic layer will have a dependency on the data access layer. So while we have applied separation of concerns, I'm still concerned, pun intended, that we still end up with coupled layers. And while that is not a horrible thing, it's possible to do better. And yes, in the end, the code still behaves as a single application. What I mean is that it's hard to split it up, for example, over multiple servers. Now, that shouldn't be a big issue here since we are, in fact, looking to create a monolithic application. But just keep in mind that layering is not going to solve that. If you want to split up your application over multiple environments, giving you the ability to scale parts independently, you need to go the microservices route. That's not part of what I'm covering here, so let's not get distracted.

**Understanding Clean Architecture**

Now that we have revisited other commonly used approaches, the next step is understanding clean architecture. As said, what we're going to use for the application architecture we're going to create for GloboTicket is clean architecture. I have been using clean architecture for quite some time for my application development with customers, and I see great results with it. Let me explain why I think this is a great architecture to use for your next project, too. Clean architecture is an architectural style that is heavily based on the design principles we've already looked at. Basically, applying all these principles correctly will almost immediately bring us to clean architecture, and you'll see, indeed, that these principles will keep popping up all over the place. That's, of course, also the reason why I included then in this module. Using clean architecture is not really difficult. Like I said, it's going to boil down to using these principles correctly. Clean architecture brings a way of structuring the application so that the business object is encapsulated at the heart of the application, making it independent from implementation details. If you followed along, this should ring a bell already. Yes, this has dependency inversion written all over it. Now, clean architecture is really nothing new. It was already introduced in 2012 and is in itself also another reinvention of the wheel. It's actually a variation of other types of architecture you may have heard of before, namely hexagonal or onion architecture. There are differences between clean and onion and hexagonal architecture design. But I don't want to go the full theoretical route, I want to start building the application architecture, which we'll do very soon. Clean architecture is trying to give us the best in terms of testable, maintainable, and scalable code. One of its key features is that it's based heavily on separation of concerns, most of them plain layered architecture. Remember that I said that although we have layers, they were still referencing each other. Well, that problem will go away with clean architecture. This will be possible because in clean architecture, things are also based around dependency inversion. Indeed, we'll use abstractions and the core, so the business logic, will be entirely independent of the UI or of the database code. There won't even be references to these projects as you'll come to understand over the course of the next modules. I'm no hero in the kitchen, far from it, but I know enough to understand that an onion in real life has some concentric circles. Well, closely related to onion architecture, clean architecture also is based around concentric circles. Here you can see a schema that represents clean architecture, which is, as said, composed out of circles. Each circle contains a layer. To explore how the code will be structured with clean architecture, we need to start in the middle and work towards the outside of the circles. In the middle, we'll find the core, which contains abstractions and the domain entities. Around the entities, we'll find the application logic, so the services which work on the entities, and it's also considered part of the core. One very important aspect is that the core is completely agnostic to any other circles that lie around it. That means that the core has completely no knowledge of any implementation details or mechanisms, such as knowing how the data is stored. The latter, so the implementation mechanisms, live in the infrastructure project or projects, but now take a look at the arrows here, which represent dependencies. They point inwards, so towards the abstraction. Remember, this is exactly what dependency inversion is all about. In the core, we're working with abstractions for pretty much everything. Some abstractions are still satisfied within the core, so in the domain services, but most of them will be fulfilled or plugged in from the outside. This is different from traditional layered architecture where the business logic points to the infrastructure layer. Finally, we also have the UI, the user interface, which is also an outer circle. Nothing in the core knows anything about how the data is represented for the user, so this is clean architecture in a nutshell. To summarize, clean architecture is a concentric model where the layers are different circles. At the very center we have the domain entities and interfaces. Still part of what is considered the core, we have the business logic and the application logic. This layer knows absolutely nothing about the implementation details. So when we translate this to a Visual Studio project, there won't be any references to packages that have anything to do with things like data persistence or logging. The latter belongs in the next circle, which is the infrastructure layer. This contains the mechanisms, the implementations, for the abstractions defined in the core, it has a dependency on the core. Finally, we have the UI circle, which is also dependent on the core, so there, too, we'll have a reference to the core project. To be able to create this type of architecture, there will be two principles which are utterly important. I've already mentioned dependency inversion, which will be used heavily. At runtime, the dependencies defined in the core will be plugged in from other layers. But to keep things, of course, testable, we'll need to create these abstractions, and when we want to test, we can plug in a different test implementation. Secondly, and we haven't touched on this yet, is the mediator pattern. At its core, the mediator will also help us quite a lot to achieve a high level of loose coupling through enabling messaging between different objects instead of creating tightly coupled objects. We will learn more about this very soon, don't worry. You may be in your head already trying to map this to a regular layered architecture, and I think that's a great idea. Let me help you a bit by giving you some more info on what code will now, in clean architecture, go where exactly. So what do we place in the core project? Well, entities. So, the domain definitely is part of what we refer to as the core. Secondly, contracts or interfaces or abstractions. They go in there, too. You'll see soon that we'll have contracts in there that will be implemented in the core, as well as in the infrastructure. It is common to have quite a few abstractions in your core project, and you'll see we'll also have exceptions, so custom exceptions in here. More on that later in the course. Now, one thing I want to repeat here is that you should never have any code nor dependencies in the core projects on anything infrastructure related. So you shouldn't have any EF Core code in your application, nor any logging code, and so on. The latter should actually go in the infrastructure layer. In here, we'll add implementations for contracts defined in the core project. So if we write in the core that want to save something inside the core, it will basically stop there. We don't include how it must be saved. The actual implementation goes in the infrastructure layer. Data access code, so for example EF Core code, will live here. So will the logging implementation. So, how do we log. Clients for other APIs, which again are also infrastructure code, should also be placed in the infrastructure project, just like anything that has to do with identity or file access. Finally, what goes in the UI layer then? Well, I guess this one is pretty straightforward. I call it UI, but it can also be an API. We'll be starting in the next project to create an ASP.NET Core API, but all principles will work fine for MVC or a Razor Pages application as well. Specific ASP.NET Core code should also be part of this layer, and that includes, for example, custom middleware or filters. They don't belong in the core since they are dependent on ASP.NET Core itself. Of course, the UI layer will need to interact with the rest of the code. In a layered architecture, the UI goes to the business logic layer. In clean architecture, the UI will communicate with the core project. But I can already say I want to have this communication to be as loosely coupled as possible; therefore, we'll introduce MediatR and the mediator pattern, but more on that later. But this will result also in very lightweight controller actions, which I also think is good design. Now that you already have an understanding of clean architecture, some of its benefits will start becoming visible too. Writing code this way will make it so that your code is independent of the UI or used frameworks. We can focus our attention on writing, well, the business logic. The code is, as said, entirely decoupled from the database that we're using. Remember that we earlier said we'd like to have persistence ignorance? Well, here you go. And the end result will be a much more maintainable code base. We'll have a good structure and a solid architectures that can cope with changes. Since it will be easy to run tests for this code because of separation of concerns, we'll be able to make changes with confidence. Later in the course, I'll prove that the different layers are actually easy to test in isolation, as well as together through unit tests and integration tests. Just as a glimpse in the not‑so‑distant future, this is what we'll be working towards. This is how I will be organizing the code. You'll see that we have a core folder with several projects in there. Next we'll have the Infrastructure folder, which contains identity, as well as EF core so that will be the persistence logic. Then we have the API, so that will be the outer layer. And I also have a Blazor application that will be talking with that API. Finally, we have a test folder, which contains several tests. Over the course of the next modules, we'll explore this in a lot of detail. I hope you already liking what you see here. As said, I think that for an application such as the one that we're building for GloboTicket, an enterprise application that will have changing requirements overtime, clean architecture is a good choice. One word of warning, though. Clean architecture can actually be more work, and so I think it is not a perfect fit for every application. It might be overkill for small projects, but remember what happens with small projects? They become large applications before we know it.

**Summary**

Now with that said, we've reached the end of this module, which was mostly theory, and I want to apologize for that, but I will make up for it with a lot of demos starting in the next module. To summarize this module, we have seen two important topics. We started with some foundational design principles, each of which plays an important role when trying to create a testable and maintainable application architecture. We then looked at what we will be using in this course, and it is clean architecture. You've seen the different building blocks of an architecture based on this foundation. And now we're ready to start using these in practice. In the next module, we're going to start building the architecture for GloboTicket and, of course, we'll start with the core project.

**Setting up the Application Core**

**Module Introduction**

You now have understood the basics of clean architecture. Seeing you have come to this new module means you're interested in learning more on how we can put this into practice. Welcome to this module called Setting up the Application Architecture where we will be using .NET Core to create a core project of the application architecture we're going to set up for GloboTicket. I'm still Gill Cleeren, and if you have any questions, feel free to reach out via the discussion board on the course page for this course. This will be a very important module since we are going to be setting up the application core for the architecture. Just like the foundation for your house is important, the core will be important for our application architecture. So what will be spending time on this module then? Well, as said, we are going to create the architecture by building a real application; otherwise, things would be too abstract if you ask me. So okay, we are going to be working on an application, so it's going to be important that we understand the business requirements for the application, so we'll start with that. Then, we'll set up the solution in Visual Studio as we will see which technology we'll be using where. Then, we'll really start working on the application core of the application. We'll first need to create the domain. We'll hopefully get enough information where we receive the business requirements to do this. And finally, and this will be definitely the meat of this module, we'll design the application project, also part of what I consider the core of the architecture. Indeed, as you'll see, I'm splitting things up in a few projects, but that's more of a personal preference. In this application project, I'll be adding contracts, we'll look at packages such as AutoMapper that we'll use here, we'll bring in logic for validation, and we'll use FluentValidation for that, and finally, we'll see what happens if exceptions are happening, such as validations and how we should respond to that. As you can already guess, this will be a packed module. Let's get started straightaway.

**Understanding the Business Requirements**

But before we can build just, well, anything, we need to understand what we're going to build. Remember, we're the consultant hired by GloboTicket, and so we're going to work with the internal team on the GloboTicket ticket management system. But what exactly does this new system need to be capable of? Of course, earlier in this course, I already showed you the finished application, but in real life it is rare that someone will show you the finished application before you're actually going to build it. At least I have not seen this happening, so I'm going to guess that this also applies for you. I, therefore, think it's a good idea to set up a meeting to try to understand the business requirements because only then will we know what we need to build. And probably things will change over time, but this should at least already give us the knowledge we need to get started. And we are lucky, we get to meet Mary Goodsale of GloboTicket. Mary is the account manager at GloboTicket, and she uses the current system on a daily basis. She can definitely help us understanding the requirements for the new system. After interviews we had with Mary, the list of requirements can be distilled to the following. The new system will need to make it possible to manage events. So according to Mary, users must be able to enter new events in the system, update existing events, and delete existing events as well. Next, a screen must also be created where the user can see all events per category. And, of course, looking at events without there being any ticket sales, it would be silly, and thus a screen that shows all orders must also be included in the first release. Those are the high‑level requirements we got from Mary. If we look at this list of requirements, there are definitely a couple of entities that jump out, if you ask me. First, events. That will definitely be a key concept. From what I understand, events will also belong to a category, and users will also be able to buy tickets for events and, thus, there will be orders for events as well. These entities will be the domain you will be using in this version of the application. I sat down a bit more with Mary, and we also created a couple of wireframes. When creating a system, I often prefer fast and short feedback cycles, and I find that wireframes help me a lot when discussing a new system. Here's what we came up with. When users arrive in the application, they'll see a menu on the left and a welcome message. Nothing really special, I guess, and not something we need to take into account when creating business rules for the domain. When clicking on events, users will enter the event management part of the application. Upon opening this part, the first thing we'll get to see is a list of events with their details. I can add a new event or edit an existing event. When editing, also the delete option is shown to remove the event from the system. There are some rules on the Events entity that we need to take into account. It's required to specify a name, a price, and a date for the event. The combination of date and name for an event must also be unique in the system. As we understood in our small domain discussion, indeed, events belong to a category. By clicking on Categories in the menu, we'll see a list off all categories, each with their events. I can also define a new category. Of course, a rule will need to be implemented to require the name for the category when creating a new one, and its name should not be longer than 50 characters. Finally, we'll also need a screen where the user can take a look at the sales for a given month, so across all events. The user can select a month, and then a list will be shown containing all the orders received for that month. I think that list will become pretty long, but we'll see how this turns out.

**Setting up the Application Architecture**

Now that you have a clear understanding of the requirements, we're ready to set up the application architecture. From a bird's‑eye perspective, this is what we'll be working on. We'll store all data in a SQL Server database since GloboTicket is already using that. We'll focus in this course on building the API that exposes all core functionality. By the end of the course, when the API is finished, I'll show you the Blazor application that will be using the API. Now since it's an API, it's possible to have other clients that connect with the functionality that we're building, including a Xamarin app, a client‑side Angular or React application, but also a desktop app or even a server‑side MVC application will work fine. In terms of used technologies, the API will be built with a ASP.NET Core, and it's this API that we will be building using clean architecture. For actual data access, we'll be using Entity Framework Core. The client app we'll include will use Blazor. As mentioned before, you don't need to know Blazor since I won't be diving deep into the code for this client‑side application, as it's not the focus of this course. We are using .NET Core 3.1, and as for all class libraries, I use .NET Standard. All the code that we'll look at here will also work fine with .NET 5. Please note that I won't be explaining the details of ASP.NET Core, nor Entity Framework Core either. This is expected knowledge at this point.

**Demo: Creating the Visual Studio Solution**

As a first demo in this module, let us set up the solution and add a basic structure already. So I'm going to, in the first modules of this course, take you through the creation of the entire application architecture. And I really want to start from the blank slate, so I'll start from a blank solution. So here in Visual Studio we have a Blank Solution template. I'll take that one, and I'll call my solution GloboTicket.TicketManagement, and I will click Create. I'm not going to be writing any code just yet, but I am already going to prepare a couple of things. I'm going to be adding a couple of solution folders. I'll bring in a solution folder where I'll put all my sources, and I'll do the same for another solution folder that I'll be using to place all my tests in later on. There we go. I'm also going to already under src bring in a couple of more solution folders, one where I'll put the API, another one that I'll call Core, another one for the Infrastructure, and finally, one where I output my Blazer application later on, and that will be the UI solution folder. There we go. So it is already in place. We're ready to start adding projects, which I'll be doing in the next demo.

**Creating the Domain Project**

We have already talked about the domain that we'll use in the application after our interview with Mary. Let's implement that first. The domain will definitely be part of the core of the application. I typically place the domain in a separate project, and I'm going to do that here too. But the main entities that we'll create will be POCOs. Remember from earlier that we need to do so to ensure persistence ignorance. And since will be using EF Core later on, that won't be a problem.

**Demo: Creating the Domain Project**

In this demo, I will add the solution to the project that contains the domain for our application, and we'll bring in the correct domain entities. So time to start adding projects to our solution, and I'll start under Core by adding a new .NET Standard project. .NET Standard allows us to also share code to other technologies, so I'll use that for all my class libraries here. So I'll search for .NET Standard Class Library, of course take the C# version, and I will call this library GloboTicket.TicketManagement.Domain, since this will actually contain my domain entities. We'll delete the class, Class1, and we'll create our own classes. Now remember from the interview we had with Mary, we'll need events, categories, and orders. So, I'm going to add these entities first. I'll bring in a folder called Entities, and in there I'll paste a number of classes that you can also find, by the way, in the assets of this course. So we'll start with an event. An event has an ID, a name, a price, and so on, and also has a link to a category. A category has an ID, a name, and also link to events. And finally, we'll have an order that contains just the number of regular properties, including the ID of the order, the user ID, and so on. I'll also bring in one more class, and I'm going to put that under a different folder called Common. I'm going to bring in, let's say, a base entity, and I'm going to call that the AuditableEntity. Let's bring in that class, and you'll see in a minute what it will do. It will contain a couple of base properties, and I just want every other entity to have full logging for, let's say, tracking purposes in my data store. Here is my AuditableEntity, which has a CreatedBy and a CreatedDate and a LastModifiedBy and LastModifiedDate. And what I'll do is I'll make all my other entities inherit from this base class, and later on we'll see how we can also automatically fill this in as a way of tracking data. So I'll make sure that now all the other entities inhabit from that AuditableEntity. If you build this and that seems to work fine, we already have the domain project ready.

**Designing the Application Project**

With the domain part ready, the first part of the core project is already done, but now comes the bigger part. In the rest of this module, we'll be spending time creating the application project, part of the core of our application architecture. Now fasten your seat belts as quite a lot of concepts will be coming your direction. Just to be clear again, the application project is part of the core of the application architecture. It's what you see highlighted here in our architecture schema. While creating our application architecture, we have to keep our main goals in sight. Application core must remain loosely coupled and have no links to the implementation parts, part of the rest of the application. We'll rely heavily for this on contracts, so interfaces, which we'll start creating in the app core. Next, as mentioned before, the second concept that will help us in achieving the required level of loose coupling is messaging. And to do messaging between components, we avoid direct concrete references from the one component to the next. Instead, we'll have an intermediate, a mediator. We'll bring this one in very soon as well. Contracts first, so abstractions. It, as said, is going to be a fundamental concept while working on the application core. You'll see that we end up with quite a large number of interfaces in the application core. And yes, that is normal. We'll create our code against interfaces. Some of these interfaces will be implemented in the core, but most will actually be implemented in the infrastructure later on. The first place where we'll encounter the use of contracts will be repositories. In our application architecture, we're using the repository pattern. In general, the role of the repository is mediating between the application code and the data mapping layer. It's a common pattern to use when your application performs data access operations. Often, the repository is used in combination with another pattern, the unit of work. It is debatable whether you should use a repository or not when you're already using Entity Framework with its database contexts since that already fulfills a lot of the functionalities of a repository. I still often use them so I have included them here in our architecture. As already said, the architecture we're using here is a proposed architecture. Many different opinions exist. I am, however, not going to bring in a manual unit of work since I believe that is entirely handled by the EF Core DbContext, which we'll be using in the next module. As said, the repository will handle data access operations. Data access in this context is not just accessing data in a database. It can also be about data that lives in a service or in a file. The repository pattern is used to create a layer of indirection between the consumer of the data and the actual data access code. This way, the application code isn't littered with data access details. I think this is really applying the DRY pattern, so the don't repeat yourself pattern. What you will do in the demo in just a minute is creating a generic repository with generic methods like a generic Add, Remove, and GetByIdAsync. Of course, it'll be just the contract that we are creating, not the implementation just yet. Next to a generic repository, we'll bring in specific repositories where needed. The specific repositories will contain well more specific methods, such as a GetAllTicketSalesForMonth, which is not covered in the generic repository.

**Demo: Creating the Application Project**

Time to head back to Visual Studio. In the next demo, we'll bring in the Application project, and we'll create the first contracts, including the one for the repository. I've gone ahead and already created another .NET Standard project here called the GloboTicket.TicketManagement.Application project. It's again a .NET Standard project. So in here, we'll create our Application logic. You'll see quite a lot of code that goes into this project, which is also the focus of this module, that is. So, I'm going to start by bringing in some contracts, and just to organize things, I'm going to create a new folder, and I'm going to call that Contracts. As said, some contracts will be implemented in the Core, but most contracts will actually be implemented in the Infrastructure. I'm actually going to create some more sub folders here just to organize things. Our application isn't going to be that big, but I can imagine in a real enterprise application, you'll have a lot of contracts, so I do recommend to add some extra organization here. I'll bring in a folder for Persistence, since Persistence will be the project under Infrastructure, that is where I have put my database interaction code later on. So I've added here the Persistence folder, and in there we're going to bring in a couple of contracts already. One that we'll need to do all the interaction with my database is going to be a repository, and I'm going to bring in an interface called the IAsyncRepository. The IAsyncRepository will be a base interface that is going to contain, well, some generic methods on the repository. Here you see the interface, and let me start by bringing in the correct using statements, there we go. So what I have here is a declaration of what an AsyncRepository should actually be capable of. It should actually be capable of, in a generic way, getting an entity by ID, everything is going to have a GUID as a key, so that's okay, I'm also going to be able to get a read‑only list to list all entities of a certain type. It should be able to add, update, and delete an entity. Notice that I've also specified here a constraint that T should be a class. Now with this generic repository, we're not going to be able to build, let's say, an entire enterprise application. We are going to need some more specific repository methods, and for that reason, I'm also going to bring in a couple of other repositories specific for the entities in our domain. So for example, I'll bring in another interface, and it's going to be the IEventRepository. That will be an interface as well that implements the IAsyncRepository and Event. And it seems that it doesn't know about Event yet, and that's actually quite normal because I have not added the reference to Domain project yet, so let's do that now. Here we go, and if we now try to resolve this, now it will find my domain project. There we go. And we'll do the same for the OrderRepository and the CategoryRepository. So we now have the OrderRepository and the CategoryRepository added as well. So of course, they're just contracts, no implementation yet, that won't be added here in the Application project anyway, but we'll need those to actually write our business logic. So with that added, you see that we are already adding quite a few contracts in our Application project. We are going to be talking in our Application mainly to these contracts, those abstractions, and we'll implement those later on in the Persistence and in the Infrastructure project, but that will be something for the next module. Let's now go back to the slides and see what else we need to add here in the Application project.

**Using MediatR and AutoMapper**

Writing only contracts won't get us very far. We'll need to write some actual business logic as well. That we will start doing now, and I'll take you step‑by‑step through how I've implemented this here. We'll start with introducing a mediator, again, with the goal of achieving a high level of loose coupling. We've touched on the concept of a mediator already, but let's now take a look at it in detail. First, what's the mediator pattern exactly then? Well, it's actually a pretty easy pattern. Normally, when we have different objects that need to work together, we'll create references from the one object to the other, and the next one, and another one. I think you get the message. While in essence there's nothing wrong with this, it creates tight links between them, decreasing the ability to test them in isolation later on, since we'll have a hard time separating them. Using a mediator, we're introducing an intermediate, an object itself that encapsulates how objects will interact with each other. Our original objects know the mediator, and when one object needs to say something to another object, it just says this to the mediator, which will inform the interested other objects. No tight links anymore, our original objects just know the mediator. Now you may be thinking, where are we going to introduce this? Well, I'm going to wrap our business logic in objects that will be triggered when a certain request comes in. This will result in loose coupling: from the outside of our application project, a message can just be sent in that contains the request of what needs to be done, and that will be the message. A handler inside of the application project will then trigger, handle the message, effectively executing our business logic. By writing our code this way, it will be much simpler to handle changes. The business logic is totally independent, and when it's being changed, no other components are impacted. Also, since the business logic is encapsulated, it's a lot easier to test it also in isolation. This all sounds very nice, but should we then be writing ourselves a mediator? Well, we could do that, but there are open source solutions available, which have a proven track record. So why reinvent the wheel when it's already done? What I would like to use here is a package called MediatR. The GitHub page states that the library is indeed a simple .NET mediator pattern implementation, and that's exactly what we need. To start using MediatR in our application project, we simply add a NuGet package and we'll need to register it in the ConfigureServices of our API later on. I'll show you how we can expose this from our application project in the demo. The library is very straightforward to use. As said, we'll need just two parts, a message that contains what we need to do, and the handler, which will contain a logic to handle the message. In MediatR, we have the IRequest interface for wrapping the message, and the IRequestHandler for handling a specific type of message. Assume that we're writing the business logic that will return the list of all events in GloboTicket. Some object, and this will be the controller, as you'll see later on, will request that we get a list of all events. Hey, this is inter‑object communication, so that will be a message. I'll thus create an object that wraps this message, which is nothing more than a class that implements that IRequest interface. It's a generic interface, the type parameter points to the type of data being returned. So this is the message. How will it be handled then? Well, quite simply, we'll need to write a handler of a given type. That will be here this class which is the request handler for the IRequest type we just created. Creating a request handler is done using a class that implements the IRequestHandler interface, and the type parameter is the IRequest it handles. When the message is received, the interface method Handle will be called. And in there, we'll write the logic to handle the message. In our case, this will return, using the repository probably, the list of all events. A MediatR is a very nice library that contains even more functionality, but we're not using all of it. The reason is that I want to do some things, well, manually, let's say. But if you're interested in more, take a look at its more advanced capabilities, including the pipeline behaviors to add automatic logging, validation, and caching. As said, I won't be using these here in this course. I will be bringing in those functionalities later on manually. Now, before we head back to the demo, I need to introduce one more library that will make our lives easier. Our application layer will get back from the repository, well, entities. But I don't want to pass entities to the consumer, right? We'll create separate objects that contain just the properties we need, referred to as view models. But I'm not interested in writing manually the code to create this new object and transfer all the data from property to property. The good news is that there is another great library called AutoMapper that will do this for us. As the name gives away, it does exactly what it says it will do, namely, avoiding that we need to write mapping code for one object to another. AutoMapper is open source and also free to use and just takes an object and will automatically create a target object and fill the properties with the values. Most of it goes automatically, so, for example, if the properties have the same name. But it is also flexible enough to allow you to write extra logic that contains more complex mapping logic. Using AutoMapper is, just like MediatR, pretty simple. You bring in a NuGet package called AutoMapper. Next, you register it in the Startup, so in the services collection. Finally, you need to specify to it which types it should be mapping to which. This happens in a so‑called profile. I will show you how to do this in the demo. And as said, most of it then happens automatically, or should I say auto‑mappically? Here's a simple example of using AutoMapper. Mapper is here an instance of AutoMapper. And we pass it an object, in our case, allEvents. That's a list of entities, and these now need to be mapped to a list of view models. Using mapper.Map, the view model object will be instantiated, and the data of the properties is copied over.

**Demo: Writing the Application Logic in the Request Handler**

We have now already covered quite a few concepts. Let's head back to Visual Studio and see how we can make this happen. We'll bring in MediatR and AutoMapper. We'll then create a request and a request handler and we'll prepare things so that our libraries can be registered in the service collection. If I would get paid for adding new files, new contracts, then I would have already made quite enough money today. But that is of course, not very useful. We'll need to actually also write some implementation code. And that is what I'm going to be doing next. So we are going to now together write the business logic to get a list of all events. So let's do that next. So to write our business code, I'm going to need a couple of packages. I'm actually going to be using AutoMapper and MediatR, as explained in the slides. AutoMapper will be used to map between entities. MediatR will be used to write those requests and request handlers, so the messages and the message handlers, that is. To make this easy, I'm going to do to my application project, and I'll bring in another item group that contains the references to the packages that I'll need. So I've brought in AutoMapper with the extension for the dependency injection, and also MediatR. If I save this file, you can also see in the NuGet Package Manager that these packages have indeed been added to my application. So what I'll do now is I'll wrap what I'm going to do in an object. That is going to be my request. So what I'll do now is I'll wrap what I want to do, that would be getting all the events in a list in an object, so this is just going to be let's say the message of what I want to do. I want to get a list of all events. So I'm going to wrap that in a class in an object, and that is going to be called the GetEventsListQuery. I'm calling it query because I want to retrieve something, and what I want to retrieve is a list of all events. So I need to make this class a message, and that I'm going to do by letting it implement the IRequest interface. The parameter, the type parameter, that is, for IRequest is going to be the type of data that this query is going to be getting back, and it's going to be a list of EventListVms. That is another class that I'm going to be returning. I'm going to create a specific object, a view model, that I'm going to return for my client application. It's going to be a view model that is going to contain just the properties to visualize in a list, just enough information so that I don't return too much data. I just add that class because I haven't created that. So I'm going to create a new class, and it's going to be the EventListVm, and it's going to be containing, like I said, the data that I want to visualize for an event when shown in a list. It contains just a number of base properties, not all the properties of event, but just the ones I'll need in the list view. And now this query here is also satisfied. So this is the message. Now I also need a message handler that is going to be triggered let's say when this message is being sent. And that is going to contain the actual business logic. So that handler is going to be triggered by MediatR. So I'm going to create another class, and that is going to be the GetEventsListQueryHandler handling the GetEventsListQuery That is going to be another class that now needs to implement another interface from MediatR. So I'm specifying here that this implements the IRequestHandler for GetEventsListQuery, that's my message type, and it is going to be returning the list of EventListVMs. Now it does give me still a red squiggly. That is kind of normal because I still need to implement the method where I'm going to handle the message, and that method is effectively called Handle. So this method will be called automatically when a GetEventsListQuery is fired off, and this handler will pick it up, so that's going to contain my business logic. Let me implement that here. Now if I base it in a bit of code, let me first make sure that all the using statements have been added correctly, so what does this code now do? Well, let me take you through it. In the constructor, I'm getting in a mapper, that is AutoMapper, as you can see here. I'm also going to be using an IAsyncRepository in events. Indeed, this query handler is my business logic and is going to work with the repositories to get the list of events. To construct the injection, I'm going to get an instance of both eventRepository and AutoMapper. Now, the actual magic is happening here in the Handle method as set. I'm going to use my eventRepository. I'm going to actually use the ListAllAsync method, which we defined on the IAsyncRepository, so the base interface. And I'm going to get all the events ordered in by date. That gives me a list of entities, an IOrderedEnumerable of events, so my domain entities. I don't want to return entities to my client. I want to return objects that I'm in control of that only contain the properties I want to return, and those are available on my EventListVm. Now I don't want to write mapping code myself, so I'm going to use AutoMapper for that. And in AutoMapper, I can use the Map method here as defining the type I want to map to, that is, the list of EventListVms, and I'm going to use allEvents as the object that I want to be mapping from. So what is going to be returned is a list of event list view models. Now, AutoMapper does need some more information for this to actually work. It does need what is known as a profile. A profile will contain mapping information so that AutoMapper knows I should actually be able to try to map from this type to this type. AutoMapper does a lot of work automatically, but sometimes you'll need to help it a bit. But if the properties on the EventListVm have the same names as the ones on the actual entity, then it will do the mapping itself. If we then just create a profile, the mapping will be done automatically for us, if we define that in a profile, that is. So I'm going to go here to my application project, and I'm going to create a folder called Profiles. Now, a Profile is really nothing to be scared of, It is just a simple class that contains what AutoMapper should be knowing in terms of mappings between types. I'm going to call this the MappingProfile class. It is, like I said, just a class, and it does need to inherit from the base Profile class. That is a class that comes with AutoMapper. In the constructor, I'm going to write my mappings. So I'm going to create a mapping between Event and EventListVm, and I'm going to specify that it has to be capable of knowing about this mapping in the two ways, so from Event to EventListVm, and vice versa. That I'm doing here using the ReverseMap. Let me bring in the correct using statements for both Event and EventListVm. There we go. We'll add more mapping profiles as we go along. I want to point out one more thing here. As we can see here, I've used the IAsyncRepository in event. Of course, we don't have an implementation yet. That'll be plugged into dependency injection, basically, the implementation of dependency version, later on. We'll still need to write an actual repository that knows how to handle this in the database. But we don't have that yet, but you can see that in the application object, we're just talking with the abstraction. That's what application project is going to be doing, just talking with the abstractions. I think at this point my GetEventList implementation is more or less ready. So I hope you understand this. But I'm going to bring in yet another one getting the event details, and I'll show you the result of that. I've now brought in a few extra classes which are going to be used to get the event details. So I have here another query, GetEventDetail, which is going to be a little bit different compared to our GetEventListQuery in a sense that they GetEventDetailQuery contains an extra parameter. I need to know which event detail do I need to fetch? So that I've wrapped here in the GUID ID. Notice that the IRequest is now going to be a generic in EventDetailVm. So the return type is now going to be a different VM, a different view model, that contains more properties about the event, event detail information, that is. And notice I also have a nested entity called CategoryDto. I also need to know information about the category of that event. That's a different type. It's again not the actual Category entity that I'm going to be returning, I'm going to be returning a nested CategoryDto containing the ID and the name of the category that I need to return for my event detail. Now, the GetEventDetailQueryHandler is also included here, and I'll take you through that one as well. It's again an IRequestHandler now going to be triggered by the GetEventDetailQuery and is going to return not the list of EventListVms, but a single EventDetailVm. We're again going to use the EventRepository, but we're also going to use a CategoryRepository and we're also going to be using AutoMapper again. The constructor is going to be using, again, dependency injection to get later on concrete implementations of this repository plugged in. In the Handle method, which is going to be triggered when that message is received, I'm going to get in the GetEventDetailQuery, that's the one we just saw, containing the ID of the event I want to get the information of. I'm again going to use my eventRepository, so my generic repository, GetByIdAsync, passing in the ID. That is going to return, in this case, an EventDetailVm, at least that's what I want. So I'm going to again use AutoMapper to map to that target type. Now remember that I also had in here that CategoryDto. That is not going to be returned automatically by my GetByIdAsync for an event. So I'll need my categoryRepository to get the event's category, and that I'm doing here. I then also use another mapping to map a Category entity to a CategoryDto, and then I finally return my EventDetail. Now this will work fine, this will actually compile, that's very fine, there we go, but we also need to add a few more mapping because I've added here a mapping from Event to EventDetailVm and another mapping from Category to CategoryDto. So I need to go back to my mapping profile and I'll need to bring in a mapping from Event to EventDetailVm, and from Category to CategoryDto. There we go. With those in place, this will actually run fine. We can't test that yet, but we'll see that later. Now we need to do one more thing. I've used in my application project now MediatR and AutoMapper. These two packages need to be registered with the service collection of my application. Now, we don't have direct access to the service collection insight of my ASP.NET Core application that we'll add later on. So what I'll do is I'll bring in a new class that is going to add an extension on top of this service collection class, so an extension method, that is. So I'm going to call this the ApplicationServiceRegistration, and I'm going to add an Extension method here. So, as you can see, I've now made this into an Extension method on IServiceCollection, and I'm then registering AutoMapper and MediatR on the services collection. There we go. That is, of course, returning services to the caller. This we'll need later on in my ASP.NET Core application to make sure that AutoMapper and MediatR have been correctly registered. With this in place, let's do another build, and that seems to work fine. So we're in good shape now. Let's get back to the slides.

**Introducing CQRS into the Architecture**

Now, our GloboTicket application will become a large application. And in the larger application, we'll have quite a lot of code. Most of the time, we write model code and more code and more code. And we typically use the same model classes for reading and writing data. This is not an issue when your application isn't too large, but again, we all know what happens with small applications, right? They become large applications. Having the same model for reading and writing data can become overwhelming. You'll get a lot of query methods, each returning probably a different type based on what it's being used for, be that a list or a detail. That same model is then also used for all the save operations, so the create or update methods, which can end up containing quite a lot of logic. On top of that, in the same model, we may need to apply different security configurations for different actions. We may end up with large model classes, which are going to do simply too much. For this reason, we rely often on CQRS, or Command and Query Responsibility Segregation. And I'm going to introduce a simple form of CQRS in our application. Now before I do so, let's make sure you understand what it does exactly. It's mostly an organization thing in our architecture. I'm going to split up our logic over different model classes. Commands will be the ones changing data, so inserting an event or updating an event. They will modify data. Next, the query models are the ones to be reading data. So indeed, splitting up thing's really into smaller, more manageable parts. Look at what I've said about commands. I'll create a command to insert an event, or another one to update an event. They're typically task‑based and can be also placed on a queue for asynchronous processing. Splitting up things using CQRS will definitely have some advantages, right? Of course, separation of concerns comes to mind here again, since indeed, read and write will all be split up. Because of this, we could also scale them independently and apply different security constraints on them. And yes, again, smaller methods will be easier to make changes to without having the risk of breaking things. Now, it's not all perfect, though. Introducing CQRS does introduce some extra complexity, but I must say, it's limited. Because of this, I do recommend using it for more complex, larger applications. So, what problem are we going to be solving using CQRS then? Well, normally, well, without CQRS, we would probably write service classes like this. I have an event service and that will do everything around events. Perhaps it will communicate with different message handlers, like the ones we have already created. Probably one service will also invoke another service. To be clear, there is nothing wrong with this approach. This will work just fine. But I think once the project becomes larger, this service, so the model, might become brittle. If I make a change to this service and another service depends on it, I need to be careful. My model is doing too much, and I risk breaking things. That's where CQRS will come in handy. Here you see the structure I would like to introduce for our application. I'm going to create more smaller classes again that each have a certain responsibility, getting all events, getting the details of an event, adding an event, updating an event, and deleting an event. Small model classes, just like I want them. We'll basically create separate requests to represent the query or command and create separate handlers for each. There might be some overlap being introduced here, so we might actually be violating the DRY principle a little bit here.

**Demo: Adding CQRS into the Application Architecture**

Okay, time to head back to Visual Studio. I'll now show you how we have introduced CQRS into the application. Now let's take a look in the Solution Explorer. What we haven't been doing is creating large service classes. We have, in fact, made a separate class for the handling of getting a list of events, as well as for getting the details of an event. So that is actually already quite a good step. We have been doing, without really knowing, a good job in terms of not creating large service classes. But what I am a bit worried about is, take a look at this Events folder, there's so many classes already in there with just two functionalities, GetEventList and GetEventDetails. So it's going to be very confusing later on when we have extra functionality. So, I'm going to start splitting up things basically by just creating a couple of folders under Events that specify what these classes are used for. So I'm going to reorganize things, and I'll show you the results. So here's what I've come up with. I've basically implemented CQRS on a folder level. I've created the Credits folder, in which I'm going to put all my Application logic. You have the logic GetEventDetails and GetEventList, and that contains small application logic containing the logic only for a specific functionality, in this case getting information about events. We'll also, of course, need to create events, update events, those aren't queries. So under Events, I'll also create another folder where I'll put application logic that will be used to create new events and so on. So that's basically Commands that I'm going to put in there. Now without really specifying it yet, we had already been preparing somewhat to go the CQRS route. I hadn't been creating large service classes, I had already created small handler classes, each containing a certain functionality, and the query logic, I already had written in separate classes. I will now do the same for all the Command logic, so adding, updating, and deleting events. And just to prepare us for the next demos, I'm going to already create a folder here called CreateEvent, which is going to contain only the logic, well, to create a new event, but we'll see that very soon. So here you see how I'm using here a simple version of CQRS in my Application logic.

**Understanding Features**

Now we have looked at the code for working with events, but we'll need to also add code for working with categories, and we'll need code for handling orders as well, and I assume a lot more will be coming into the application. If you place all requests and request handlers in one large folder, I think things will become cluttered again very soon, so I would like to introduce a feature‑based organization in our application. A feature is a vertical slice to the functionality, so I'd say a context. It can be seen a bit as a bounded context, which is a concept of domain‑driven design. It's basically a feature, something that is pretty much standalone. I will then create feature folders inside a Features main folder. So I have a feature Events, other feature categories, and a future Orders. Know that I'll probably also let these features contain their own view models they'll use. View models are the types returned to the caller, and even if they could be shared, I will typically not share them. This way, I can make changes to a given feature, knowing that I won't be impacting another feature or functionality.

**Demo: Organizing the Code Using Features**

Let's return once again to Visual Studio and take a look at how our application is now split up using features. Now without really focusing your attention onto it, I had already put in a Features folder, maybe you had noticed, and inside of that I had created already an Events folder. We were already working feature based, but now you see it more clearly. I have added the other features that we'll have in our application, so that we'll be working with Categories and working with Orders. So under Features I have now added a Categories folder and an Orders folder, and we already had the Events folder. For each of these I have also added again the Commands and Queries folder, in which we'll put our application logic. Let's do that now already for Categories. So we already have some basic Queries for Events in place, but I'm going to do the same thing for Categories. I'm going to paste in a bit of code that you can again find in the assets, and I'll take you through that code. So I've now added again two queries for working with Categories, the GetCategoriesList, which is very similar to what we had with the Events. So I will need in the application the functionality to list out all categories. So I have again a CategoriesListVm, a specific ViewModel for displaying information about the category in a list. I have a GetCategoriesListQuery, which contains no parameters, it will just return all categories. And in the Handler for that, so the GetCategoriesListQueryHandler, I'm going to handle the GetCategoriesListQuery message, and in the handle method I'm going to get all the Categories, order them by name, and map them to a list of CategoriesList here. So that is very similar to what we already had. I have also implemented another query here, which is going to not just get the list of Categories, but for each category it's also going to fetch the list of Events. So now I have a CategoryEventListVm, which contains the information about the category, but also all the events for that specific category, which I have now wrapped in a CategoryEventDto. This will contain basic information about an event. In the query, so in the IRequest, I also have the option to include the history, so basically specifying if I want to get all events or just the ones in the future. And this is also going to be returning a list of CategoryEventListVms. That's this one here. Lets take a look at a handler for this. Now, what you see is different here is I'm not using the IAsyncRepository, but for the first time, I'm going to be using a specific repository, the ICategoryRepository, which we already created, but we haven't added any extra method declarations in there yet. And I'm actually getting an error here because we don't have the method yet GetCategoriesWithEvents on that specific repository. This is the reason why I need a specific repository. I will need to write extra methods, not just the ones defined on the AsyncRepository, the base repository, but specific methods that will, in this case, get the categories with their related events, passing in also that Boolean whether or not I want to include the history. So this is extra business logic I also need to contain in my handler. And then I'm going to use the mapper again to return that again as a list of, in this case, CategoryEventListVms. Now I do need to add that on my CategoryRepository, but this is a quick thing to add, so let's go back to the Contracts, Persistence, CategoryRepository. I'll now bring in this new method here, GetCategoriesWithEvents, passing in that Boolean if I want to get the historical events or not. And now this will build fine. One thing to do, one thing not to forget, let's say, is making sure that these new types that I have now brought in also will be mappable via AutoMap. So let's go back to the Profiles, and bring in two more MappingProfiles between Category and CategoryListVm and Category and CategoryEventListVm. So now AutoMapper knows about these, and if the properties map in terms of name, it will automatically map between them. So there we go, you now see that I've wrapped the functionality for Categories, for working with Categories that is, inside of the Categories Feature folder. We have the Events Feature folder, which already contained our queries, and we'll later on add functionality for the Orders as well. So I've now split up things nicely in these vertical slices in these Feature folders that will contain everything that has to do with Categories, with Events, with Orders, and so on. This is a nice way of structuring the business code that we're writing.

**Demo: Using Commands to Create a New Entity**

So far, we have looked at the code for getting data, so a query, but what does the code look like for creating an entity, say, an event? Well, it is pretty similar. Take a look at this schema first. The creation of a new event will come in as a request again. This time this will be the command, containing all the data for the event to be created. A handler will pick up on this, creating the event in our persistent store using the repository. A value will then be returned. There are different options for what we can return. It could be just a primitive value, like a Boolean indicating if the equation was successful, or a full blown response containing more info about the creation. And note that we'll also need to bring in validation, but we'll look at that in a minute here. First, time for another demo. I'm going to show you the code for creating, updating, and deleting an event. Now so far we've only created queries, and most applications, including ours, also need to be able to enter new data, so that's what I'll do next. I'll now include the code to create a new event, and creating a new event, well, that's really an action, that's a command, and so we'll create the code for that in this Commands/CreateEvent folder. And the first thing I'll need is, again, the message. What am I going to be doing? I'll need to let MediatR know what it needs to react to. And it's going to be the Command, and I'm going to wrap that again in a class that implements the IRequest interface. So I'm going to create here a class, and it's going to be the CreateEventCommand. Notice that I've now suffixed this with Command to be clear that this is going to be an action, a command. And let me put in the code for this. So the CreateEventCommand will be the message about the new event that needs to be created, and it's going to be a set implementing the IRequest. The return type is going to be a Guid. I'll talk about return types later, but this basically means that when I create a new event, I'm going to be returning to the consumer the ID, the GUID, of the newly created event. So this is the message about the to‑be‑created event, so it will contain all the information about the event that I want to create, so the Name, the Price, the Artist, and so on. And just like we did with the Queries, where we want to get Events, I'm also going to create a Handler to handle this CreateEventCommand coming in. It's pretty much the same thing. So let us create another class here. It's going to be the CreateEventCommandHandler. And in here, I'm going to be handling the CreateEventCommand. So I need to implement here the IRequestHandler in CreateEventCommand, and it was going to be returning a Guid. Let's add MediatR here and implement also the Handle method. Let me show you the completed code for this. So here you see now the finished code, where I'm again using a repository and AutoMapper. And the rest of the code is actually pretty simple. What I'm getting in is a CreateEventCommand, and I need to map that to an entity. So basically we are going to do, well, the opposite in this case. I need to map now from a CreateEventCommand into an Event. That's going to be the event that I want to store in the database. So I'm going to use the AddAsync method here to store that event later on in the data store. And I'm supposed to return the ID, the GUID, of the newly created event, and that is what I'm doing here. Now since I'm mapping here from CreateEventCommand to Event, I need to go back to my Profiles and include another MappingProfile that will do exactly that. So with this in place, I think we can now handle creating a new event. The actual implementation, that will come in later. I'm going to bring in the code for the updating and deleting of an event as well. To save us some time, I'm going to paste that in, and I'll show you the result. Here you see the code for updating an event. The UpdateEventCommand is another command that implements IRequest. And I actually don't return anything, because if you think about it, an update call over an API, if everything works okay, will not return anything, so I don't need to actually return something here. The UpdateEventCommandHandler is again pretty similar. It will first fetch an event by ID, and then we'll update it later on. And the DeleteEventCommand. Well, that's pretty simple as well. That is going to contain in the EventCommand the ID of the event to be deleted. And in the DeleteEventCommandHandler, we're going to fetch the event that we want to delete, call DeleteAsync on the eventRepository, and I'm basically returned the default value again. And in the Profiles I've also added now one more mapping for everything to work fine.

**Adding Validation Using Fluent Validation**

In our discussion we had with Mary, we already got some of the business rules. When adding an event, for example, it's required that the new event has at least a value for the name, price, and date. We have earlier, in this module, created the entity for event. So if I ask you to implement these business rules, I think your default solution would be adding attributes on the Event class. Data annotations are very often used and are an easy solution. Now while they work, I would advise against using them. First, for the sake of the persistence ignorance principle, let's try not to go away from POCOs for our domain entities. I also believe the data validation code doesn't really belong there. Secondly, and this is perhaps a bigger issue, not every business rule can be solved with just data annotations. Required string length, they work fine, but if we need to write combined rules, say if the name of the event contains 2020, then the date should not be in 2021, well, then data annotations simply won't cut it. So we'll leave our entities alone and go another route for validation. I would like to bring in another commonly‑used framework, and it is called fluent validation. Fluent validation is, again, an open source project and free to use and can thus be added as a NuGet package to our application architecture. Using fluent validation will lead to maintainable validation code since it uses code, lambda expressions to be exact, to write our validation logic. We can thus write our validation logic entirely separate from the domain entities just like we want it. We can use fluent validation from our code project as well and we'll introduce the validation logic in the handle method of the request handler. You'll also see that the validation logic will be specific per Feature folder. Again, this may look a bit weird as I may need to write validation logic for the same entity twice, but that's the price I want to pay for the flexibility I'm getting this way. Like I said, validation rules and fluent validation will be written using lambda expressions. Your validation logic will be contained typically in a class that inherits from a base class that comes with fluent validation called AbstractValidator, this is, again, a generic class. The type parameter is the type you're writing validation rules for here. Then for each rule we want to create, we write an expression like you see here. Once we have our validator written, we can use it from the request handler code. Here, I'm instantiating our validator first and then I call ValidateAsync passing it the object I want to validate. Now, how do we then handle validation errors? That is another interesting concept, and maybe even more broadly, how do we handle in a clean way, any exceptions that are coming back for our application project so that the consumer, that will be our API later on, knows what happened and can, in turn, also return the correct message. Well for that, I will in the application project create custom exception classes. They are part of the core project and are thus known everywhere. The consumer of the core will get back these exceptions and can handle accordingly. I will include a few exceptions, including the NotFoundException, the BadRequestException, and the ValidationException. NotFoundException, I'll be returning when the request, for example, for a non‑existing event is received. BadRequestException will be used to let the caller know that the received input, for example, to create a new event, is incorrect. ValidationException, I think, is quite clear, it used to return, well, validation errors. Note that these are not the built‑in exceptions, they're custom classes. You can see a sample of one of these exceptions, namely, the NotFoundException, it is a custom class and I let it inherit from the ApplicationException base class and that is a built‑in class.

**Demo: Adding Validation and Custom Exceptions**

Let's return to Visual Studio once more for this module and explore how validation was added to the application architecture. I'm going to show the validation process, as well as custom exceptions. I'll also include one more thing that I haven't looked at in the slides, and that is a custom response object that we can also return, useful in combination with validation errors. Now we can currently create a new event, but I don't have any validation logic yet. I have no logic in here that checks that the data coming in via the CreateEventCommand is actually valid. Now I want to stay away from going to my domain entity, so to my event entity, and start adding attributes. I don't want to go that route. So, what I'm going to do instead is I'm going to bring in fluent validation and application logic that will validate my data inside of my application project. So let's go back first to the application project file and bring in FluentValidation. So I'll bring in FluentValidation and this other package here as well, save that, and now we have support to fluent validation in the application project. So now I want to validate the CreateEventCommand from the CreateEventCommandHandler. So I'm going to do that by writing an extra class that will be the CreateEventCommandValidator. That will contain the validation logic for the CreateEventCommand. Now to make this into a validator, I need to inherit from the AbstractValidator. In CreateEventCommand, the AbstractValidator is a class that will be added through FluentValidation. There we go. I'm going to make this public, and in here I'm going to, in the constructor, write the validation logic, and I'm going to do that using, like I said in the slides, lambda expressions. So I'm going to write a validation RuleFor, in this case, the name of the event, and I'm going to say that it, for example, should not be empty. I don't want this to be an empty string. That's a validation rule. I think there are more interesting things to do than watching me type, so I'm going to show you the finished validation rules for CreateEventCommand. There we go. I've now added a few more, I've added here a rule that says the name shouldn't be empty and if it's empty I'm going to show this message. It also shouldn't be null, and I'm also specifying here the maximum length again with an error message. And you can see that I've also added a validation rule for the date as well as for the price. Those were the validation rules that Mary mentioned to us in our meeting. Now, I have these validation rules wrapped inside of this class, I also now need to use them. Now for that, I'm going to go back to my CreateEventCommandHandler. And before I'm actually going to go to my repository and try to map this, I'm going to validate the incoming requests. I'm going to instantiate the CreateEventCommandValidator, and then I'm going to call the ValidateAsync method onto our CreateEventCommand instance, and this will trigger the validation rules defined in the validator. Now, just triggering those won't do anything, we still need to check the result of that validation. And the result will be captured in this validationResult and that will actually contain the list of errors, if any, that are coming back from that validate. But if there are, what do I then do? Well, then I'm going to, well, in my case, throw an exception. Now, I don't want to use the default exceptions. Why that is will become clear later on. But I am going to include some custom exceptions inside of my application project that I'm going to use for my application logic. The consumer of my application logic will then be able to react to these custom exceptions. I'm going to paste those in and then show them to you. So in my application project I've now added an Exceptions folder, which contains a couple of custom exceptions, such as the BadRequestException, that will be used later on when we, for example, encounter wrong input, an event that is going to be created that basically contains invalid input, it might be null or something. NotFoundException, well, that one will be used to return to the user when you've asked me to update an event, for example, but I don't find that event, that's a NotFoundException. It's a custom exception as well. And we are going to be using the ValidationException. We're going to throw this exception if validation rules are not met. So let's go back to our handler, and in here I'm going to now check if the validationResult.Errors.Count is larger than null, that means that validations have been broken and so I'm now going to throw my own validation exception passing in that validationResult. So now we have some basic validations in here. Now, if you look back at this validator, you could actually say, well, this I could have done using attributes, and well, frankly, you would be right, but I can do a lot more with fluent validation, things that I cannot easily do using attributes. And I'll show you that next here. Now fast forward to a new version of the CreateEvent CommandValidator, in which I've now added a custom validation rule. Well I want to validate if the combination of event name and date is still unique in the database, so I'll need to bring in a repository, the event repository, that's what you see here. And I then use a custom validation rule that will be triggered, and if it's broken, then I'm going to show this message here saying that an event with the same name and date already exists. So to check this rule, I'll need to go to my repository, so in here I've added this IsEventNameAndDateUnique method on the repository, on the contract of the repository that is, and this rule will now also be validated when I'm calling the validation on my CreateEventCommand. In the CreateEventCommandHandler I don't need to do anything new, I just still call the ValidateAsync, which will take into account now that new rule. So I've now nicely separated again my validation rules from my business logic, again nicely applied the separation of concerns here as well. Now the last thing I want to show you here is what I am returning from my CreateEventCommandHandler. As you see here, I'm returning just the EventId, and while that is okay, it might be so that we also want to return the newly created instance in full. Well, that I've also included in the sample application, but for categories. So let me add a bit more code and I'll show you how I can also return a full response for the consumer of my application logic. So again, fast forward, I have now added a command to create a category, and I'll quickly go through the code, but it's very similar. We have, again, a CreateCategoryCommand, which is very simple, it just contains the name of the to be created category, but notice now what thjs IRequest is going to be returning, not simply the GUID of the newly created category, but a CreateCategoryCommand response. If we go to the handler, and we take a look at what is being returned here, we indeed see a bit more logic. I'm again doing validations, there is a validator, just take a look at that at your own pace, it is in the sample download, but notice what I'm returning here if everything goes according to plan. I'm going to create here a new CreateCategoryCommandResponse, which is again a class, part of the CreateCategory folder, that inherits from the BaseResponse. The BaseResponse is now also included as a BaseResponse that can be used by commands to return a response to the consumer. In here, I have basic properties, such as Success, Message, and optionally, also the list of ValidationErrors. In the CreateCategoryCommandResponse that inherits from that BaseResponse, I'm also returning the newly created category, so I'm including the data here also in the response. So in the handler, if everything goes according to plan, then I'm actually going to now return a custom response type for my creation of the category. So this depends a bit on what you want to offer to the consumer of your application logic. If you say well, the ID is enough to know that we have successfully created a new entity, well then you can just return the ID. But here you see that I've included a custom type that we then return to the caller that contains not only the data, but also possibly validation errors, if any. And that can allow the client, the consumer of your code, to align on always getting back at BaseResponse and looking for these specific properties, such as the success property.

**Summary**

I'm happy we are here now. This was quite a complex module. We have touched on many topics to really create a clean architecture for our GloboTicket application. As you have seen, the core projects, so the domain, but mostly the application project contained really the core functionality of the application. We have brought in several concepts and libraries to create a testable and maintainable codebase, which is definitely loosely coupled. The code so far contains no implementation logic. We've also introduced features, as well as validation logic using FluentValidation. Now that we have all this code, you may be thinking, well, that's nice, but I'm still stuck with a lot of interfaces, contracts that have no implementations yet. Indeed, that was the goal really. We have kept the core projects clean of anything infrastructure related. No EF Core, no file access, no service access, just abstractions, just what we need it. But of course, we'll need to add these implementations. Adding the infrastructure or the mechanisms, that will be the goal of the next module, and I hope to see you there as well.

**Creating the Infrastructure Project**

**Module Introduction**

Hello again, and great to see you here for another module of ASP.NET Core architectural fun. I'm having a great time explaining to you how I think we should address the architecture for GloboTicket. In the last module, we have covered a lot of the foundations, and I think we've done a great job. But indeed, we don't have any implementations, any infrastructure yet. Well, that's correct, and this is the way it should be when using clean architecture. We are, in this module, going to bring in yet another layer, another concentric circle around the core, and that will be the infrastructure. Let's see what we will be doing in this module. I'm going to go through a small introduction of the goal of the infrastructure project first to make sure you understand correctly what we'll be adding here. Then, we'll basically be adding two new projects. First, we'll add persistence, and I'll do that using Entity Framework Core. Next, I'll bring in yet another project where I'll bring in support for other infrastructure tasks. All set? Let's do this.

**Understanding the Goal of the Infrastructure Projects**

So let me first try to explain exactly what the goal of the infrastructure layer is and what types of code you should be expecting here in this layer. The core projects contained the business logic, but no mechanisms. It was all quiet abstract, right? And that's what you get with all these abstractions. Just kidding. This is perfectly fine. But indeed, we need to implement somewhere the code to write to the database. Well, that will go in the infrastructure layer. If you look back at our schema that we've been using so far, now we're going to focus on this very layer here. Remember that the core project should never contain references to EF Core packages, logging libraries, and the like. Indeed, they should be added and used in the infrastructure project. So basically, as soon as we introduce a mechanism, an implementation, look at the infrastructure project. Database code will go in here. In our case, we will use Entity Framework while all that code will go in the infrastructure project. So that means the the DbContext and the migrations, also code that interacts with files. So reading from or writing to a file, that will go in here too. Say that you want to use Azure Service Bus in your application, you shouldn't add Service Bus messaging code in the core. You add an interface in the core, and the implementation, it goes here. If you want to interact with another API, you'll typically write a service client. That will go in here too. Logging implementations, same thing. They'll also live here. I hope you get the message. Anything external goes in the infrastructure project. Make sure to keep the core clean. Now how does this then work? Well, we added a lot of contracts in the core. We saw, for example, an IAsyncRepository. Well this repository will require an implementation, and that will go in the infrastructure project. Many interfaces, contracts defined in the core, are implemented in the infrastructure layer. Infrastructure projects will reference the core projects, and it's through dependency inversion that actual implementation will get plugged into the core from the infrastructure at runtime. Dependency injection, basically the implementation of dependency inversion, is used for this.

**Adding Data Persistence with Entity Framework Core**

Now that we know the goal of the infrastructure projects, let's get to it and add the first one. I'll bring in, finally, data persistence using Entity Framework Core. The infrastructure project itself will be just a plain .NET Standard project. To make use of Entity Framework, therefore, we'll need to follow a few steps, but they're not any different from what we normally need to do to bring in EF Core. We'll need to bring in some NuGet packages first. Then we'll create a DbContext. And in the DbContext, I'll create the DbSets for all entities I want to have EF Core manage. Then we'll create a migration. And finally, I need to also register in the Startup, so in the services collection, and I'll be using EF Core. Nothing really special. Of course, remember that we created, in the previous module, a contract for our generic repository, the IAsyncRepository. That repository I now need to implement here. I'll create a base repository implementation for methods defined in the contract, making use of the EF Core DbContext. Just having the base repository won't be enough I think. I'll need specific methods in the repository layer as well, and for that, we'll have specific repository implementations, each which will also need to be backed again by a contract in the core. Otherwise, the core simply can't use it. Here's a part of the base repository implementation that we're going to create. The class indeed implements the contract defined in the core project, but now I have concrete method implementations. I'm using a DbContext instance. And to get, for example, an instance by ID, I'll use the generic GetByIdAsync. But as said, we won't be able to build an entire enterprise application with just a generic repository. We'll need to be able to do more specific queries. And for that, I'll use more specific repositories. These are going to be specific for a given type. Here you can see the CategoryRepository, which implements an interface defined in the core, but also still inherits from the base repository. I have a method specific to categories implemented here.

**Demo: Adding Entity Framework Core**

All right, time for our first demo. I'm going to bring in the infrastructure layer. First, we'll create a new project called Persistence. It will contain the code for storing data in the database. I'll then also, in this project, configure EF Core. Our core projects are in good shape. We have application logic that will do validations on the data and so on. But if you remember, we have in the Persistence folder a lot of repositories, but they are just interfaces. So now we'll start implementing that. So let us go to the Infrastructure folder that we already created earlier in this course, and I'll add another new project. Again, I'll select a .NET Standard project, and I'll bring in the Persistence project, so GloboTicket.TicketManagement.Persistence. Now for reasons that have to do with the compatibility of packages, I do need to make this .NET Standard 2.1. So the persistence project will have to do with storing data, fetching data from my database. So we do a couple of things, and I'll bring in a reference to my application project since I'll be implementing the contracts for the repositories defined in my application project. And next, I'll also bring in a couple of NuGet package references, one to Entity Framework Core SQL Server and also to this ConfigurationExtensions package, which we'll need later as well. So now the packages have been added. That's all good. So now let us go and create the actual logic for the persistence project. We'll start with deleting Class1. Now the first thing I'll do is I'll bring in a database context, the DbContext. It's going to be a simple class, GloboTicketDbContext. That will be my database context. And since we already have brought in the packages for Entity Framework Core, we can use the DbContext base class. So let me bring in the code for this one. You can find the code for this class again in the snippets. Let me take you through the code. What does my DbContext contain? Well, of course, it contains this constructor, and it also contains DbSets for the Event, Category, and Order entities. Then I have overridden the OnModelCreating, and I'm doing two things here. I'm going to apply configurations. I'll come to that in just a second. That has to do with how I want to configure my database. But next, I'm also going to use the OnModelCreating to insert some seed data. So as you can see, I've added a couple of categories. And using the HasData, I'm going to check if that entity already exists in the database. And if not, it's going to be inserted as part of a migration later on. And I'm doing the same thing for events. I'm adding a couple of events here. Now let's focus our attention again on this ApplyConfigurationsFromAssembly. Now I mentioned already that I want to leave my domain entities. I don't want to litter my domain classes with attributes that have to do with how the database should be created. This time not validation, but how the database should be created because that too I can do using attributes. But I want to stay away from that. I am going to instead use configuration classes, which will contain code that specify to the ModelBuilder how the database should be constructed. So I'm going to go and add a new folder here. I'm going to call that Configurations. And in here, I'll add configuration classes for each of my entities, and I'll add just one for now. I'll bring in a class, and I'll bring in configuration for the event entity, so I'll call it EventConfiguration. And that'll contain the code for configuring how the ModelBuilder should handle the event type. So there we go. I've now specified here that I want to have an EntityTypeConfiguration for Event that will specify, in this case, that the name of the event should be required and also have a maximum length of 50. Let's make sure that all the correct using statements have been included. Now by calling in the OnModelCreating, the ApplyConfigurationsFromAssembly, it will search for all configurations included in the assembly and apply them on the ModelBuilder. And there's one more thing I want to show you here all the way at the bottom of the DbContext. I've also included an override of the SaveChangesAsync. In the SaveChangesAsync, I'm of course going to call the base, but I'm also going to add some extra code that will update the properties, the tracking properties that is, on the AuditableEntity. Remember that all my entities inherited from AuditableEntity? Well in here, when an entity is being saved, I'm going to check if it's being added, and I'm going to update the CreatedDate. And if it's being modified, I'm going to update the LastModifiedDate. Now with that in place, my DbContext is already in good shape.

**Demo: Implementing the Persistence Layer**

With this project configured, I'm going to bring in a few more things. First, we need the base repository implementation using the DbContext. Next, we'll bring in more specific repositories. And let's not forget that we also will need some more code to make sure everything is correctly registered later in the service collection. So now we already have DbContext. Let us now create implementations for our repositories, and we have to do a few. So I'm going to add them in a folder called, surprise, surprise, Repositories. In here, I need to implement the contracts that we have defined in our application project. So I need to bring in an implementation for the async repository, as well as for the individual and specific repositories. Let's do that now. I'll bring in first the implementation for the IAsyncRepository, so that will be the base repository, and that will require using statement to contracts persistence. That's normal. I'll bring in also the other using statements and also a using statement to EF Core. So what do we have in here? Well, I'm implementing now the IAsyncRepository, specifying again that the generic T type should be a class. I'm then going to use my GloboTicket DbContext, which I'm then going to insert through dependency injection. We'll need to think about that a bit more when we add this to the services collection. Well, I'll come to that later on. Then I have some basic implementations for the GetByIdAsync, which is just using some base EF Core code to find an entity by ID. In the ListAllAsync, well I'm simply going to get all entities from a given type. I also have an add, an update, and a delete, which are again very simple. I'm going to add an entity to the DbContext, and I'm going to call the SaveChangesAsync, returning also that entity. I have the same thing for the update, as well as for the delete. So that is now my BaseRepository. I also need to add the implementations for the entity‑specific repository. Let's do that next. To save us some time again, I've brought them in already, and I'll show you them now. Let's take a look at the EventRespository first. So the EventRepository inherits from the BaseRepository and Event in this case and also implements the IEventRepository. The IEventRepository had that extra method, IsEventNameAndDateUnique, that we used in the validator to check if that name and date combination was unique. Well I'm writing some custom code here, which indeed cannot be part of the BaseRepository. But I'm going to use my DbContext checking in my events if an event already exists with the given name and date, and I'm going to return the result of that. In the specific CategoryRepository, I have an implementation for the GetCategoriesWithEvents that we also already looked at in the previous module. In there, I'm going to not only get the categories, but for all categories, I'm also going to include all events. That's why I'm using here the .Include. If I don't want the events in the past, I'm going to remove the ones that are in the past. I've also implemented the OrderRepository, which contains a couple of extra methods that we'll need to use later on. But they are already here. And again, the repository implementations can all be found in the snippets that come with the downloads of this course. Now I've added EF Core. I've added my DbContext. If you think about a regular a ASP.NET Core application, what do we still need to do? We need to let my service collection of my ASP.NET Core application know that we are going to work with EF Core, and that we typically do in the Startup. But of course, we don't have a Startup. We're just working with the .NET Standard library here. So again, like we did in the application, I'm going to bring in a class that extends my service collection. So I'm going to bring in a class. This time I'm going to call it PersistenceServiceRegistration where I am going to again extend the service collection, allowing me to make my service registrations from my persistence project. It's important that I do that from the persistence project again. Here is the PersistenceServiceRegistration static class, which is going to extend again this service collection. Let's bring in the correct using statements. Look out when you bring in the IConfiguration one because AutoMapper also has a configuration namespace. There we go. Now Visual studio is happy again. Now let me show you what I've added here. So on the service collection, I'm going to register like normal the DbContext, passing in our own GloboTicket DbContext. And I'm already also specifying that I'm going to use SQL Server, passing in the connection string. Now we don't have that connection string yet. That'll come later when we add the API. At this point, we don't have an app settings yet. That lives in the executing assembly, and that will be the API. I'm also registering the BaseRepository, as well as the custom repository. And just like with blank EF Core, we need to do this in a scoped way so that one instance is created per request. And with this in place, I think we're in good shape already with our persistence project. Let's do a quick build to make sure that we haven't made any mistakes, and that seems to work fine.

**Adding the Infrastructure Project**

Uh‑oh, I just got a call from Mary. She forgot about something in the previous meeting. When a new event is being created, we'll need to have the system send an email to a given address. I think now that we are creating the infrastructure project, this is really a good time to see how this can be added to the architecture. Let's take a look. In this part of the module, I'm going to bring in support for other infrastructure tasks. Let me show you how we can add that to the architecture. So Mary asked us if the system will be able to send an email when an event is being created. Sending an email is definitely infrastructure code. It's an external system we need to work with. When I get this question, I typically use SendGrid, an excellent system for sending emails. SendGrid offers an API and a NuGet package. You register at SendGrid, you'll get a key, and then you can, from code, send emails. No internal mail server is required. You just use theirs. Since this new email functionality will need to be called from our core code, we'll bring in an abstraction for sending emails, which will be used from the core code. Then, we'll need to write an implementation in the infrastructure again, which will at runtime be plugged in.

**Demo: Adding Support for Mailing from the Infrastructure**

All right, let's see this end‑to‑end example of bringing in a new requirement, and you'll see how easy this now has become. This new functionality I'll place in a general infrastructure project, so separate from the Persistence project, just to keep things cleaner. We'll then write a logic to interact with SendGrid, and we'll use this code on the Core code. Finally, we'll need to register this new service also in the ServiceCollection. A change request. My, I think we're all familiar with those. But the purpose of this demo is not only showing you how we can now add the infrastructure layer; I'm also going to show you how easy it now is to actually implement that change request that Mary sent us. So, we need to send an email. So how are we going to do that? Well, we are basically going to, in our application code, just say how we need to send an email. So I'm going to bring in a new IEmailService that will be able to send an email. So in the application project, let us go back to Contracts. And in here I'll now bring in a new folder, and I'll call that Infrastructure because this has nothing to do with Persistence. This really is infrastructure. So I'll bring in an Infrastructure folder, and in here I'll bring in a new contract and I'll call that the IEmailService. This will be my interface. That is just going to know that it needs to send an email. That email type, that will be the thing that I'm going to send, I still need to define that as well. Now for that, I'm going to create a custom model. Now, this is part of the application logic, and it will be the type that contains the properties of the email that needs to be sent. So in here, I'll now go ahead and create a Models folder. I'll create a subfolder here as well. Let's call it Mail, and in here I'll create a new class, Email. That class Email will contain the properties that contain the email message information, the to, the subject, and the body. Now you may be thinking, hey, why did he not put that email type under domain? Because this is a class, right? Well, it's a class indeed, it's a type, but it has nothing to do with the domain that GloboTicket works with. This is just a class, a type that I'll need to define under my application project for my email service to work correctly. And so I go to my IEmailService again, and I'll bring in the using statement here. There we go. So now we can, in our application logic, in the application project, send an email. We create an email, and then we call SendEmail on that. And again, we'll need to implement that in an infrastructure project, and I'll do that in just a second. Now I'm going to do one more thing. I'm going to create another type here under Mail and it's going to be EmailSettings. Now EmailSettings is going to be a class that contains properties that I'm going to read out from settings later on. Settings, defined in an appsettings file, to configure my external email service, which will be SendGrid. So these values will be read out from an appsettings file later on in the API. So now I'm going to implement the application logic that is again totally agnostic of how we are going to send the email. So Mary asked us to send an email when an event was created. Well, that's pretty easy to find. We go to Features, Events, and in there we have the Commands, CreateEvent. So in the CreateEvent command handler, when an event was successfully added, we want to send an email. That's what I want to do. So I'm going to bring in the IEmailService, and I'll also let this be injected through dependency injection as a \_\_\_\_\_ constructive parameter. Now when the event was successfully added, I'm going to send an email. I'm going to create that email here. I'm going to use my own email, I specify the body and the subject, and then in a try‑catch, I'm going to ask the emailService to send that email. Again, no implementation, just fire and forget, in this case. Now put this in a try‑catch because, in fact, I don't want my application to fail should that emailService go wrong. I want to log that, but we'll come to that later on. So I'm just firing off that email using the emailService with SendEmail. All right, our application logic now knows about sending emails. Let's build. That seems to work fine. Let us now go to my Infrastructure folder here again, and we'll bring in yet another project, the actual Infrastructure project. So I'll go here, and I'll create another .NET Standard project. I'll call this GloboTicket.TicketManagement.Infrastructure. And again Class1 is there, and let's delete it. There we go. Now in the project file, I'll bring in a couple of references again. This time I'm going to bring in a package to work with SendGrid. I'm going to now implement the real EmailService using SendGrid. SendGrid is an external mail service that offers you the ability to send emails via their service, and have an API that I'm going to work with that, of course, I don't want to litter my application project with. That is really infrastructure code, so that's why I put this in the Infrastructure project. Let's save all this. And least we forget to add a reference to my application project, because I'll be implementing the IEmailService here. So in here I'll implement the IEmailService, and I'll bring in a new folder called Mail, and I'll bring in a class called EmailService. This EmailService class will, of course, implement my IEmailService interface that lives in the Infrastructure contract. And I need to implement the SendEmail method. And of course we're now also are going to be using email, which is part of the Models folder in my application project, so that, I think, is a good place to store that. Let me implement this SendEmail method here. And this EmailService is nothing really complex. It is going to use the EmailSettings. Those were the ones that we defined earlier in our application project under Models. That'll contain things like the APIKey, the FromAddress, the FromName, that we will also need to put in our appsettings later on. Then I implement the SendEmail, which is just going to use the SendGrid API to send the email. I take my APIKey, I create the subject, the from, and then I finally will send the email using that SendGrid client. Now that all seems good. Now I need to do, again, one more thing. We again need to create a class that extends the ServiceCollection to register the things we are using inside of this project. So again, we'll go here and add a new class, and this time I'll call it the InfrastructureServiceRegistration, and here is the code for that class. I'm again extending the services collection, and I'm registering my EmailService, and I'm also registering that from configuration EmailSettings an instance of EmailSettings should actually be loaded, so those values should be specified later on in the appsettings, but that, again, will be in the API project itself.

**Summary**

And there we go, another module done. We now have another layer in our clean architecture implementation added. Infrastructure is, in general, the place to put all configuration for external services, be it the database, a file, or even an external mail service. I have, as you saw in the demos, split things up into two projects. One is specific to persistence, while the infrastructure will really contain interaction with other services. We are nearly at a point where I will be able to test all of our work. In the next module, we'll bring in the API project that we'll create using ASP.NET Core. See you there.

**Adding an API Using ASP.NET Core**

**Module Introduction**

We have been doing quite a lot of work in the past couple of modules, but apart from a code base that actually builds, we haven't seen anything working yet. That will now change in this module where we are going to create an API that uses our application architecture. Welcome to this module, Adding an API Using ASP.NET Core where we are going to add this API, indeed, using ASP.NET Core. Clients, in our case a Blazor application, will then be able to connect with this API for the back‑end functionality. This API will be the gateway to the core functionality. A very important piece of the puzzle if you ask me. Let's get started. Of course, the way we create the API will also require some thought, and that's what we will be doing in this module. First things first, though. I'm going to create the API project, which will be pretty standard stuff, but I'm going to give you some pointers here. What we will be spending more time on is the next part. How are we going to create the code in the API? Which code will we really have in our API controllers. I'm going to show you a few options, and we'll see that MediatR comes into play here. Next, we'll also take a look at which data we will be returning. Finally, for the consumers of the API to be able to know what functionality the API will be returning, we'll bring in swagger. Lots of things to do. I think it's best if we dive straight in.

**Creating the API Project**

So, as promised, we'll start with just bringing in the API project, which will be a very straightforward task. Like I said, in the intro of this module, we've spent a lot of time on creating the business functionality using good architectural principles, and with neatly followed, clean architecture principles to do so. We have looked at the core and the infrastructure. So here's our diagram again, and we're now basically going to be adding the next piece of the puzzle, and that is the UI, the user interface. The UI will be exposing the functionality to the end user. Now, the UI here is really a broad term. We could bring in an ASP.NET Core MVC or a Razor Pages application, and then, indeed, this server‑side UI would be directly exposing our functionality. We are going to follow this part here, though, and we're going to use instead of a UI an API, which will be exposing the functionality. The UI that we will be adding is a client‑side UI, which happens to be Blazor, but could also be an Angular app or a React app or, well, pretty much everything that can hook into an API. So, by using an API, were basically opening up our application's back end for use with many types of clients, and that is a big plus. But from an architectural point of view, the API we're going to discuss here will take the role of the UI. For that reason, the API will be using our core code, as well as fit into our architectural schema. I've shown you this slide already, and it shows the different projects we have in our solution. Just to be clear here also, we've already built the core and the infrastructure project using .NET Standard, and now I'm going to bring in an ASP.NET Core project for the API. The API that we will create will indeed be created using ASP.NET Core, and I'll be using a ASP.NET Core 3.1. I won't be spending time in this course explaining to you the ins and outs of ASP.NET Core APIs. We have a lot of great material available here on Pluralsight that goes a lot deeper into this. So going forward, I'm going to assume you're all familiar with controllers, the concepts of REST, and the configuration of an ASP.NET Core project. All right, then. In our API application, what we'll need to do is make changes to the Startup class of our application. Remember that we have already created along the way in the different projects we've built extensions for the ServiceCollection class. Well, we'll finally be able to use them here now. I'm going to add them here since the API will be the executing assembly, and it is the one that we'll be starting up. Of course, we'll need to use the code that we've written already. If you follow the architectural schema correctly, we would need from our API just a reference to the core project. Now while that is true, because of the way that the DI container that comes with ASP.NET Core works, we'll also need to include a reference to the infrastructure projects. If you would use, for example, Autofac as a DI container, which is module‑based, this would not be needed, but we are using the default, so we'll need to bring in these references as well. No big deal, though. You'll see that in our code we'll just use to core code from the API.

**Demo: Adding the API Project**

Now before I'm going to show you how we write the interaction between the API and the core code, let's start with adding the API project and configuring it correctly in the first demo of this module. So now we have the foundation ready, let's now expose that functionality over an API. So I'm going to go here to my API solution folder, and I'm going to add another new project, and I'm going to select here an ASP.NET Core application. And I'm going to call that GloboTicket.TicketManagement.Api. And I'll select the Empty template here. So our API, as said, is going to expose functionality, so the first thing I'm going to do is add in a couple of references. And, as said in the slides, I will typically need to expose the application since that is the one I'm going to be using directly, but remember that we also have added these service collection extensions in the infrastructure and persistence project, so I'll need to also add a reference to these. There you go. So we have now added a reference to the application infrastructure and persistence project. Very soon, we'll bring in some controllers, so let us already also bring in a Controls folder. This is the empty project, there's nothing in there yet. What else do I need to do here? Well, I'm going to go to the appsettings, and remember in the persistence, we, of course, used a connection string, and so that connection string isn't defined in persistence, but we need to also bring that in in the API project. So in the appsettings, I'm going to define this connection string. And while I'm at it, I'm also going to bring in email settings. Remember, we have created the email service, and that email service relied on email settings. Email settings also were brought in through reading configuration, a section called EmailSettings, and that, too, needs to be defined here under the appsettings. So, here I have the appsettings, the FromAddress, an ApiKey, that is an API key from send with, and the FromName. Those settings will be read out automatically. Now I say automatically, that's not 100% correct. I need to go to my Startup now. I need to configure my application, so my API. In the API I need to do a couple of things. So, of course, I'm going to be reading out configurations. Let me bring that in first. I'm going to use, of course, the Microsoft.Extensions.Configuration. So, what do I need to add now in the ConfigureServices? Remember that we have added extension methods on ServiceCollection, like the one you see here. Those I will need to call now from the ConfigureServices for my API Startup. So that's what I'll do here. So what did I bring in? AddApplicationServices. That's the one that lives in application. This one did the registration of AutoMapper and MediatR. Also, I'm going to call AddInfrastructureServices. That was the one that was going to read out the email settings and also register the email service as the implementation of the IEmailService. I'm going to do the same for the persistence services, and that was the one I just showed you that brought in the DbContext support, as well as the repositories. So, all of that is now brought together in the Startup of the API. I've also brought in the AddControllers, which is going to bring in support for working with controllers very soon, and I've also opened this up for CORS. I'm not really going to specify a strict CORS policy, but I'm going to open up this API so it can be used from client‑side technologies, which we'll need for our Blazor application. I'm going to also replace the code here in the Configure. So what I have done here? I've basically added support for routing, I've brought in calls, and I've also specified here that we are going to map to controllers, so routing to controllers is automatically now enabled. So let's see if this builds. That's all good. So, what do I now need to do still before I actually can start writing code here in the API? Now if I come to think about it, so we have added a DbContext, but we haven't ran any migrations yet. Now, that is because I couldn't really execute anything. So I'm now going to do that now that I have my API. So I'm going to set my API as the Startup project, and then I'm going to create a migration. Now before I can actually execute the migration, I do think I need to add one more package, indeed. I need to bring in support for the EntityFrameworkCore.Tools package because otherwise I won't be able to create a migration from the Package Manager Console. With that added, save this. I can now go to my Package Manager Console and then set the default project here to my persistence project. Now I'm going to add a migration for my database, I'm going to type here add‑migration, and I'm going to call that InitialMigration. That has now created the migration for my entities, so I have a Categories, Orders, and Events table, which are going to be created taking into account also the things like the maxLength 50 that we have defined. So now I can actually create a database. I'm going to go back to my Package Manager Console and run an update‑database statement from here. There we go. If we now go to View and then SQL Server Object Explorer, because I've used a LocalDB, I need to use this one, and now I go to Databases, and I have quite a few test databases here, but you can see here I've created GloboTicketTicketManagementDb5, and in here I have my Tables and I have my Categories, Events, and Orders. This has also used HasData. If I go to the Events table, I also see some default events that have been created. So that's all good. The API project is in here, and we've done everything to set it up. Now, let's get back to the slides.

**Transitioning from View Services to MediatR**

Now that everything is configured correctly, we need to see what options we have to create the actual controller code, so the code part of the API. There are actually quite a few options here, and what I propose is that I take you through a number of options so we can decide later on which will be the best option. Sounds like a plan? All right, let's do this. When we write an API, in most cases, you'll end up with some code in the controls. If we decide to take the easy route, the default route, let's say, then we'll get a controller that is pretty heavy in terms of the amount of code that it'll contain and the functionality it is covering. It's pretty easy to write a controller that does a lot and, frankly, too much. This means, for example, that the controller will in its action methods first check the incoming data to see if it's valid. For this, we can rely on model binding built into ASP.NET Core. If we know the model is valid, then we can, from the controller, also execute logic, so perhaps by talking with the code in our Core project. Once we receive a response from the Core, we then again in the controller will create a response type, so an instance of the type that we are going to return. This can, for example, be a view model instance. And then finally, again, still from the controller, return an API response containing a status code and a response. That is actually a heavy controller, if you ask me. Aren't we violating the separation of concerns principle here? Should all this code really be living in the controller? I'm not a fan, to say the least. Here's an example of such a heavy controller. You can see the steps that I've just explained. We have model binding to validate the incoming data, then we talk with the Core code, then we get back a response from that, we create a view model that we're going to return, and finally we'll create an API response. Way too much for a controller. At least that's my opinion. A better approach might be that we make the controller lighter already by moving part of what we've just seen to a separate class. I typically refer to this as a view service. It's a separate class which is called from the controller, and this is also part of the API project. And it will take over a lot of the functionality that I've just explained. This view service will then talk with the Core code and return type. The controller's part in this is now less heavy. It'll just talk with this view service and, based on its response, return a status code and a response to the caller. Here you see an example of such a view service. As you can see, a large part of the code as we previously had in the controller has now been moved here. In terms of separation of concerns, we're now in better shape, since this class's functionality is, well, a lot smaller. And the same goes for the API controller itself. It now knows about the view service. Perhaps it will get it in through dependency injection. Although we're moving in the right direction, I still feel that we could do better. If you think about it, the controller is now tightly coupled with that view service, and perhaps we can still improve here. And the approach that I recommend here is using, again, MediatR. So again, there will be a very loose coupling between the code in the controller and the code that we need to work with. In my controller, what I'll do is, I'll have a specific type that represents what we want to do. So in other words, we have a specific type for the query or the command we want to execute. It can be passed in as a parameter, as I'll show you in a minute, or it can be created in the controller. Then, and this is where it gets interesting, the controller will send the object basically to our MediatR, and I'm going to use the MediatR library for that again. The request will then be picked up by the RequestHandler in the Core code where we have defined a RequestHandler already for the types we need. This way, the controllers are becoming very lightweight. The most they'll do is just raise a message, a request, that is picked up by MediatR. We create this way a very loose coupling between our controller and our Core code, and that's perfect, if you ask me. Here you can see an example of what I've just explained, and this example will be for a query that we want to trigger in our Core project. In this action method, we are creating an instance of the query request type. Then, using MediatR, we just fire the message, the request, which will then be picked up by MediatR, handled in the Core code, and then a response will be returned, which is then sent back to the consumer.

**Demo: Adding Controller Code Using MediatR**

Time to see this in action. I'm going to prepare the API project further. We'll bring in the correct packages since now we'll need MediatR to be callable from the API project. And then we'll see how we can build our controllers in a loosely coupled way. So now let's focus on really exposing some controllers that we can actually talk with from the outside. So I'm going to use, as said in the slides, the MediatR‑based approach, so a very loosely coupled approach. So I'm going to bring in support for MediatR. So I'm going to bring in another PackageReference here to the version of MediatR that we also already used in the other projects. So now I can add my controller, and I'm going to start by exposing the category controller, which will be a plane controller, an API controller. So I'll go here, select API controller. I'm going to call it the CategoryController. To save us some time, I'm going to paste in the code to this controller and then take you through that code. So fast forwarded a bit, and here is my finished CategoryController. As you can see, it, is relying on MediatR, which is going to be brought in through dependency injection again. Now MediatR is already added to the services collection and is, thus, also available through dependency injection already because I've used it also already in my other projects. And we've called that from the startup. So we don't need to do anything extra to get it in here via dependency injection, via constructor injection I should say. So here I have now my actual controller actions. First, I have this GetAllCategories, and now notice how this is going to work. From the outside, we're requesting the GetAllCategories. I have an HttpGet that is going to be sent to the /all. So this is going to be addressable via api/category/all. Now what am I going to do here? I need to basically let MediatR know that GetAllCategories is going to be requested. Now GetAllCategories, we have created for that in our application under Features, Categories, Queries, a GetCategoriesListQuery, this one. I basically need to raise one of these, send one of these requests, that is, and that is going to be sent to MediatR. So the only functionality that my controller really needs to do is creating one of those messages. And that is going to send that to MediatR. MediatR has registered in the GetCategoriesListQueryHandler that CategoriesListQuery should be handled by this handler, and it's going to automatically invoke this Handle method here. So without creating a tight coupling between my controller and my query handler, I can invoke my business logic in the application in a very loosely coupled way. Then I am returning that data. As another example, which is actually pretty much the same, I'm going to get the CategoriesListWithEventsQuery. That was another one, that's actually this gray here, which now just requires a value for the includeHistory. And I'm just creating that here in the controller. And then I also use MediatR to send that off. Again, I have defined that this is accessible via an HttpGet that needs to be sent to allwithevents as the endpoint. And what about creating new data, so sending data over the API? Now here's an example. This is the Post method which is going to arrive here in this createAction method. In the body, I'm going to now send in a CreateCategoryCommand. That is just an object, a type that is going to be sent from the client. You'll see very soon how we are going to create that on the client. But now I'm basically exposing this command that we created earlier. It's also going to implement the IRequest in CreateCategoryCommandResponse in this case. It just defines the Name property. So, in other words, this command is automatically going to be created, let's say, by the client and sent to my API. In a very similar way, I'm now going to raise that message, that CreateCategoryCommand, and send that off to MediatR. MediatR is then going to call the CreateCategoryCommandHandler, that will be this one, that will trigger validation and that will in the end return a response. Now it is in place if you haven't made any mistakes along the way, and we should now actually be able to call our categories API. Let's try that out. So we need to browse to api/category/all. And there we go. We are getting then our list of categories. And the other calls will work in a similar way. Let me quickly show you the other controllers, which are also very lightweight and relying on MediatR to also work in a very loosely coupled way. Here you see the EventsController. And as you can see, it's very similar. It also relies on MediatR to create that loose coupling, and also it's going to be a very lightweight controller again. You can see the GetAllEvents, which just creates a GetEventsListQuery and sends that with MediatR. GetEventById, as well as the Create, Update, and Delete also are very similar. Here's the OrderController. There are no actions in there yet, but we'll come to those very soon. So to conclude, using MediatR, we have now created very lightweight and loosely coupled controllers, just the way I like them.

**Deciding Which Objects to Return**

One thing I see often in application architectures is that the types that are being returned from the API controllers to the consumers are far from optimal. Too much data is being sent back. In this case, I don't mean that a very long list of objects is being returned, which is of course also not great, and we'll come to that later, what I mean here, though, is that I see often that way too many properties are being returned to the consumer in the response object, resulting in too much data going over the wire, as well as time lost on both sides on serializing and deserializing that data. Let's see what I think we should be returning instead. I'm a fan of returning what the consumer of the API needs. Of course, this is easier said than done. If you're building a general‑purpose API, it is hard to guess what consumers of the API will need, and this doesn't really apply. But if you are, like we are doing here for GloboTicket, building things end to end, you are basically in control. In that case, an easy split can be made, for example, by looking at where the data for our call will be used. If you are returning, say, a list of events, the objects we are returning should be small and contain only the relevant properties. Often, the result will be, indeed, a generic list, and so you'll often create a view model type that contains just the properties needed in the list. If you think about it, when the user of the client‑side application will be using this data, for most returned instances, they will just see the properties used in the list view. Why would you then return all properties? Then, if a caller asks for the detail, we are returning another type containing all relevant properties. The detail will therefore be also getting a custom type, a view model, that we create, and using AutoMapper, we are mapping the original properties to the properties of the view model. If needed, we can still use a custom nested DTO if we also need to use nested lists, for example. Now when returning data, so far we have just returned the data and nothing more. But here too, there might be even a better solution, and that is using a response type, so a class, a type, which is specifically going to be used to be returned to the consumer of the API. It will wrap, of course, the data, but it will also contain a fixed structure to contain errors and other information. If you use this approach, the consumer of the API will know that always a class of this type will be returned and that errors can be found in a specific property. The response data can be found in another property and so on. This can result in a very clean way of working with the API.

**Demo: Returning View Models and Responses**

All of this may sound a bit abstract. Let's return to Visual Studio and take a look at what our API should be returning. I'm going to show you the different options here using a specific view model for the list and the detail. Next, I'm also going to show you how we can use a response base type and return that from the API. Now let's focus a bit on what my actions are returning. So, here we have the EventsController, and it's pretty much the same for all of them. So let's start with the EventsController here. So here I have the GetAllEvents. And, as you can see, the action is returning a list of EventListVms. The EventListVm was the view model I specifically created in my application to return as the entity that was supposed to be shown in a list. So I'm just returning, in this case, the data, the list of view models to be shown in a list. And GetEventById, well it works pretty much the same. It's, of course, not returning a list, but it's returning a single EventDetailVm that contains all the properties, again, created inside of my application project. So I'm just returning my type here. Notice that I'm never returning the actual database entity. I will always use AutoMapper to map to this VM. When we take a look, also, at the Create, Update, and Delete, notice that I'm using, of course, an HttpPost, Put, and Delete, but I'm also returning something here. In the case of the Create, I'm just returning an Ok, passing in the id, that would be the GUID of the newly‑created event. From a pure REST perspective, this is not really correct. And I should, in fact, also return the newly‑created entity, but I'll come to that in just a second. But here I'm doing the simple approach, just returning the newly‑created id of the new event. The Update is returning NoContent because an update, when it succeeds, shouldn't be returning anything, but it can also return a 404NotFound, for example. I'll come to how this 404NotFound can actually be returned. That has to do with malware, but again, I'll come to that very soon. And here you also see the Delete, which follows the same pattern here, if everything goes correctly, and I will also be returning a NoContent. Now let's take a look at the CategoryController because in the CategoryController, we also see something that we've already touched upon earlier. Take a look here at this Create method here, and this Create method is not just returning the ID of the newly‑created category. Instead, I'm returning this CreateCategoryCommandResponse. And if you remember from earlier in the course, this was a type that I've created inside of my application project, of course, which inherits from the BaseResponse. And this type contains the data, so the newly‑created category, as well as properties it inherits from the base response, including validation errors. So if we create a new category and validation errors occur, then we are going to pass those back in the validation errors inside of the response that we're sending back. So, this is really a nice approach. So I'm creating those validation errors inside of the handler, and those are going to be returned via the controller to the client that is using my API. So, in fact, you could always rely on this response‑based approach so that clients will always know what they are getting back. They will always get back an object, and it contains information about whether or not the API call succeeded, as well as possible validation errors or the newly‑created or updated entity.

**Demo: Adding Support for Returning a CSV File**

One second. I'm getting an incoming call from Mary again. Okay, uh huh, okay. So you'd like to export the list of events to an Excel file, right? Yes, at this stage, we can fit that in quite easily. Alright, bye bye. Okay, so you may have already understood from my call here what's going on. Mary called me, and we need to export a list of events to an Excel file. That is something we don't have support for in the API just yet. I think we'll need to bring that in. Let's see how we can using all the principles we have learned so far fit this into our architecture. In the following demo, I'm going to show you how we can implement exporting to Excel from the API, and we'll need to add this in the different layers. And to save us some time, I've gone ahead and already implemented the functionality that was requested by Mary. So, we need to now export the events to a CSV file. So on the EventsController, I've added a new action. That's the one you see here called ExportEvents. And it's going to be returning a file result because in it, we're not going to be returning JSON. I'm going to be returning a file that can be downloaded to the client. Now the way that we do this is pretty much the same. The functionality to create this file lives in the application project. So I'm going to, again, create a message that will trigger a handler. And that message is now called the GetEventsExportQuery. It is not expecting any parameters. It is going to be returning, however, an EventExportFileVm. That is a bit special, as you can see. It contains a FileName, the ContentType that will be CSV, but it also contains the data, and that will be the CSV file Data that is going to be returned to the client. So that is now wrapped inside of this VM that I'm going to be returning. That is, of course, created by the handler. So I'm going to create, again, a handler, which is going to be handling the GetEventsExportQuery. Now what is special about this one? It's, again, using our repository and AutoMapper, but it's now also using an ICsvExporter. Because if you think about it, creating that physical file, that byte array, that is something that needs to be done by infrastructure code. So I'm creating, again, an interface, the ICsvExporter, which contains a method, ExportEventsToCsv. My application code isn't concerned with how we are going to create a CSV file. This code just knows it needs to be exported to a CSV byte array. And so basically what I'm doing here is I'm going to get all the events, and I'm going to call that \_csvExporter interface and call the ExportEventsToCsv onto it. Then I'm creating my VM, which is going to contain the file data in there. I'll give it a filename, and the ContentType is going to be set to "text/csv", which is then going to be returned. I still need to create an implementation for that CsvExporter, and for that, I've gone to the infrastructure project where I have now added under FileExport a CsvExporter, which is going to be implementing the ICsvExporter. And in here, we simply see some code that is going to export that data to a CSV file. I'm using for that a library. If we go to the infrastructure project file, we see that we've now imported CsvHelper, which is another open‑source library to export to CSV files. So this is infrastructure code. In the InfrastructureServiceRegistration, I shouldn't forget that I also need to register the ICsvExporter to be implemented by the CsvExporter and then through dependency inversion and dependency injection, really. The CsvExporter will be used by the query handler. And the code runs when this ExportEventsToCsv is going to be called. What I hope that you see is that because of the way that we structured the code, we haven't impacted anything that already existed. We nicely created in the EventsController, an extra method, an extra action method, I should say. We created an extra query, and then we created an interface, the ICsvExporter, which is then implemented in infrastructure. But existing code hasn't been influenced. We just have created that new functionality next to what we already had without creating any impact in the already existing code.

**Exposing the API Functionality Using Swagger**

In the final part of this demo we are going to add a Swagger interface on top of the created API. If you have already built APIs, I'm sure you're already familiar with Swagger. Swagger allows us to add a description of the API and its methods. This way, other teams within GloboTicket, in particular the team working on the client projects, will be able to see what the API is capable of. Swagger is actually composed of quite a few things. As mentioned, it can be used to describe the API, but thanks to Swagger a standard specification for the API will also be generated, typically in JSON or in YAML. This is the standard, meaning that other tools can hook into it as well. Quite a few tools exist around Swagger, and we'll use two in this course. First, we'll bring in Swashbuckle as a way to generate the Swagger endpoint and the Swagger documentation. In a later module, we'll also use NSwag, a tool that will look at the API spec and, based on that, generate code. Because Swagger is indeed the standard, it's possible to have a lot of tools based upon it. When adding Swagger to the API, as said, documentation about the API is generated. A JSON file is available, exposing all the information about the API. But, next to that, also a human‑readable version of the API documentation is available in a typical page layout you probably have already seen. Here you can see a screenshot of a Swagger documentation file, which lists out the API endpoints for our API. For each available action which is available, the HTTP verb is shown to be used on that very method. And using this page, also, it's possible to test the API. And I will show you this in the demo. In order for the Swagger documentation file to be available in the API, some packages need to be added. For ASP.NET Core APIs, we can use the open source package Swashbuckle. Indeed, a very interesting name. There are a few packages in this area we need to add to get the full Swagger experience for the API. Swashbuckle.AspNetCore and Swashbuckle.AspNetCore.Swagger bring in the tools and the middleware to expose a JSON endpoint from our API. SwaggerGen will be used to generate the Swagger document based on the routes we have in our API. And SwaggerUI will be used to generate the UI, so that human‑readable documentation.

**Demo: Adding Support for Swagger**

Let's return once more for this module to Visual Studio and bring in support for Swagger in the API project. To enable Swagger, I've added these two packages here, as you can see in the project file. Then in the Startup class, I also need to bring in support for Swagger. As you can see, I've added this AddSwagger method, which I'm calling from ConfigureServices, which is adding support for Swagger. I've specified here that a SwaggerDoc file should also be created. A specific filter is, by the way, needed to also bring support for the CSV file that we are returning over our API as well. That's what I've done here. Then, in the Configure method, I need to bring in some more middleware for Swagger and Swagger UI, and that will also enable the endpoint, swagger.json, which is that machine‑readable JSON file, that OpenAPI specification that can be used by other applications, and we'll see how we can do that from our Blazor app very soon. With this in place, let's run the application. If I now just go to /swagger, you'll see here that we get that documentation file where I can also test my API methods. For example. I can click here on the Category/all, try it out, execute, and I get back the response containing, in this case, the name of all categories. If I browse to swagger.json, we'll indeed get the actual JSON file that describes our API. And this we can then use from other tools to generate code, and we'll see that in the next module. So now we've added support for Swagger on our API.

**Summary**

Now with this, our API is in a good shape, and we can now finally use the functionality we have embedded in the core project. I think the way we have now organized things, where we are using a loosely coupled approach with the API sending messages using MediatR, is a very clean approach. We have been added Swagger, and we have also seen that bringing in a new functionality, the export to Excel, was actually easy to implement. The next test will be, well, the actual testing. Is our architecture easy to test? Let's find out in the next module.

**Testing the Application Code**

**Module Introduction**

We now have a large part of the application architecture ready. We can already work with API, and so that part, the back end, is in good shape, at least that's what I hope. We did a lot of work to create a testable architecture for the application, but so far we haven't really tested if it's testable at all. Let us change that. In this module, Testing the Application Code, I'm going to be showing you how we can effectively test the different parts of the application we have created in the previous modules. Let's dive in. Before we are going to be creating tests, I want to give you a brief overview of the different types of tests we can actually write. Then we'll get to writing tests, and I'll show you how we can write unit tests for the code we have created so far, as well as integration tests, testing time,

**Understanding the Different Test Types**

The main question we need to solve in this module, and frankly, it is quite exciting, if you ask me, did we do a good job? Is the architecture that we have created for GloboTicket's new system actually testable? And will this help them in creating a soft resolution that can be changed easily over time? I'm feeling good about this. We have kept loose coupling as a goal at all times, and that inherently will help us in achieving this. Now let us see how far we get. Before we look at things in practice, let us start with understanding the different types of tests we can and probably should be writing as part of our application architecture. There are three main types of tests that we can distinguish. Unit test is the first one. It is probably the one you are most familiar with. We'll start with this one in just a minute. Secondly, we can write integration tests. And finally, we also have functional tests. So, let's start with unit tests. So, unit testing. A lot has been written about it, but what is a unit test really? Well, I have a small definition here that more or less captures the essence of a unit test. A unit test is code that will typically, in an automated way, invoke code to be tested. It will check an assumption about the behavior of that code under test. So a unit test is really code that tests other code. I'm going to make an assumption about that piece of code's behavior, and that will decide whether or not the test has succeeded. When writing unit tests, there's a couple of things that we need to keep in mind. First, we typically should test the public API of a class. Although there are definitely cases where you need to test the private API as well, in general, your unit test is a consumer of the code, and hence, you'd test the public API. Unit tests also need to run in isolation. That is something we have been trying to achieve with our architecture for some time already. We have applied dependency inversion, and that should make it possible that the code under test is free of dependencies we don't want to include in our test. When running tests, they should also be consistent in what they return. When I run a batch of tests multiple times, the results should always be the same. In order not to be a nuisance for developers, running tests should also be fast. If tests run too long, developers won't be spending any time running the tests. Now very often too, these tests will get automated and will perhaps run as part of a continuous integration build. So why are we creating unit tests then? Of course, to find bugs. We may not find bugs as a direct result of a unit test; however, as mentioned, when we make a change to a part of the code, the unit test can be seen as a guarantee that you haven't introduced a bug while making the change. The result of that test should still be the same. Having unit tests for our code will allow us to make changes without fear. When unit tests still runs without problems, we can be pretty sure that all is still well with our code. By creating tests, the overall quality of our code should also be improving. A unit test can be seen as the consumer for the actual code and makes us think more upfront about the design of our code. And finally, a good battery of unit tests can also be seen as documentation for the code. I said unit tests are the consumer of the code. They describe how we can interact with the public API of our codebase. Very soon we'll be writing some unit tests for the different parts of the application. As a second important type of test, we have integration tests. As the name is giving away, an integration test will be more end to end. We have written are code so that we have most of it independent of the infrastructure, but that code too, that infrastructure code, needs to be tested. And not only does the code need to be tested, we need to make sure that the interaction between the different layers works well. When the dependency is resolved, does everything still work as expected? That's the type of information we want to get from an integration test. Now, typically, also more work to create and set up since we now need to work on setting up testing around infrastructure. This means that we'll, for example, need to work with the database. Sometimes we'll use an in‑memory database for an integration test. EF Core comes with an in‑memory option that can be used for this. Alternatively, when we do testing on an actual database, then only to include code that restores the database in its original form after running the test, for example, by removing the newly added records again. Finally, we have functional tests. This type of test is used to, well, basically checked from the end user's perspective if the system behaves as expected. People like Mary, who are involved in the creation of the system, but from a functional perspective, that is, can definitely help. But we can't expect Mary or someone else to do the testing entirely manually, although that too is required. Functional testing therefore often comes down to bringing in a UI testing framework. We won't be covering that in this course though, as it would take us too far.

**Creating Unit Tests**

In the next part of this module, we are going to get busy writing the actual tests, and we'll start with unit tests. To bring in tests in our project, we need to set up a few things. First, we are going to include a test project where we will have our unit tests in. We have different options here, but I'll be using xUnit for this. Then, since we want to test, for example, the code in the core project in isolation, we'll need to bring in mock versions of the dependencies. The core code has dependencies that will be satisfied at runtime through dependency injection, so we'll need to mock this out. Then we'll be able to write this code in isolation. We can use for the mocking part manually crafted mock versions or a mocking framework. And yes, the testing will be made possible thanks to use of dependencies that will be injected. You'll see that I'm again using a number of common packages when creating the unit tests. First, I'll use Moq as the mocking framework. It makes it very straightforward to create mock versions of the dependencies we'll need in our code under test. I'll also be using Shouldly, an easy‑to‑use assertion framework. Using Shouldly, we can more easily write readable assertions in our code, and I'll show you how to do this in the demo. Finally, as already said, we'll use xUnit as the testing framework. We'll use the template for this, Visual Studio comes with one, but that too is basically a package that is being added.

**Demo: Creating Unit Tests for the Application Code**

Time for demos. First, we'll include unit tests for the core code, and we'll need to bring in some mocks as well. So let us now take a look at writing unit tests. And by means of example, I'm going to show you a unit test for the GetCategoriesListQueryHander. That is business logic, part of our application project. So I'm going to be testing this, and what I'm actually going to be testing is the code in the Handle method, because that's what's going to be triggered when I send a GetCategoriesList query. Now if we look at this code, we see that it has two dependencies, an IAsyncRepository and Category and IMapper, so those I will need to provide an implementation for in my unit test. Under the test folder, I've created this unit test project, and as you can see, it's an xUnit unit testing project. It also has a dependency on Moq and Shouldly, and we're going to be testing code in the Application project, so I also have a reference to the Application project. That's the only thing I should have in terms of references. Now, let us take a look at the unit test itself. You can see the unit test. It's attributed with the Fact attribute, meaning that this is a unit test. I'm going to be instantiating my handler. And in order to be able to do that, I need to provide it with implementations for the two dependencies. In the constructor, I'm creating those. I'm using the Moq framework here to provide an implementation for the IAsyncRepository. I've written a static GetCategoryRepository on a RepositoryMocks class. In here, I've created in the GetCategoryRepository a number of hard‑coded categories. Then, using Mock, I'm creating a mock implementation of the IAsyncRepository stating that when I call the ListAllAsync, it should just return me all categories. And when I'm calling AddAsync, it should be adding that category to the list of categories. That mock version is then returned. I'm also creating an instance of AutoMapper, passing it our MappingProfile class so it knows how to do the mappings. So now we're ready with the setup, so now we can create the GetCategoriesListQuery handler, passing it our mapper and our repository. I can then called in Handle method, which expects a GetCategoriesListQuery. That will then return a list of CategoryListVms. Indeed, that's what we were mapping to here. Now we're going to assert the result, and I'm going to see if it's effectively a list of CategoryListVms that I'm getting back. And because in my mock I created four categories, I'm now also expecting that the number of items in the result should be four. And for these assertions, I'm using the Shouldly framework. So let's run this test, and that seems to have worked fine. Now there's also a test for the creation of a category, which is pretty similar. I'm then going to use the CreateCategory handler. It expects the same dependencies. I'm then going to create a category, which, behind the scenes, is going to create that category. If no validation errors occur, that is, I'm then going to ask my mock repository to list again the categories. And now, because I've added one, the result should be five. I can also run this unit test, and that seems to be fine as well. So now you see how our architecture lends itself well to writing unit tests.

**Demo: Writing Integration Tests for Infrastructure**

We now have some unit tests working, and yes, our application should contain probably quite a few more, but now you understand how they should be created. Let's now turn our attention to creating integration tests. When writing these, we're basically going to be writing a test that integrates several layers. In the first demo, I'm going to be writing an integration test that tests part of the infrastructure code. Under the test folder, there's also an IntegrationTests project, and in there I have an integration test for the DbContext. Now when we want to test something on the DbContext, we actually need to write an integration test. EF Core comes with the in‑memory database, which makes integration tests using a DbContext very easy. Now let's take a look at the GloboTicketDbContext. As you can see, I have two constructors here, I have the regular constructor, and I also have one that expects an ILoggedInUserService. Now I haven't explained this any further in the demos, but I invite you to take a look at this code at your own pace. All this is available with the downloads of the code. The ILoggedInUserService will, in its implementation, returned the name of the logged‑in user. Now that I'm also using here in this SaveChangesAsync method, which is the one I want to write an integration test for now. So the SaveChangesAsync method is going to automatically, when an entity is being saved, set the CreatedDate and CreatedBy, or LastModifiedDate and LastModifiedBy values that are defined on the AuditableEntity. So what I want to test is that this actually works. I've written a test for that, as you can see here. I'm actually going to test that the CreatedBy value is actually set. Now in order test this, I'm going to again provide an implementation, a mock implementation for the ILoggedInUserService, which is going to just return a hard‑coded GUID. That's what you see here. Then in my test, I'm going to create a new event and then call just a plain Add method on that DbSet. By calling the Add, an entity is going to be added. I call the SaveChangesAsync, and then using ShouldBe, I'm going to check that the CreatedBy value is actually set. So here we see how we can easily write an integration test that involves the DB Code text.

**Demo: Writing Integration Tests for the API Controllers**

In the final demo of this module, let's take a look at how we can test the API code, so the code part of the controllers. That too will require an integration test. In the final demo of this module, let's take a look at how we can write an integration test for our API controllers. Now when we want to test an API function, an API method that is, I actually need to be able to talk with a running application. Now this we can easily do using the WebApplicationFactory, which comes with ASP.NET Core. Here's my IntegrationTests project, which contains pretty much the same references that we had up until now but also includes now the Mvc.Testing PackageReference. And you'll also see that I'm, of course, referencing the API project since I need for my test to be able to talk with the API. What I want to test is the category/all API call. Now how am I going to be able to run that test? Well, I've created a CustomWebApplicationFactory, which inherits from the WebApplicationFactory, in which we then set up the WebHost. We pass it here at Startup, so that's the startup of the API project, which will, therefore, configure our application in‑memory. I'm also going to specify that for the GloboTicketDbContext, I'm going to be using an in‑memory version of that DbContext. Now further on, I also load that database with, again, some hardcoded data in this case for categories. And I save those categories to that DbContext, so to that in‑memory database. So this configures our web application that now uses as its database an in‑memory database. So this is the full web application. So this will, behind the scenes, have set it up using my application project, using our infrastructure project, but the main difference is that it's running against an in‑memory database, so I actually can test using that. That's what I'm actually going to be doing here. In my actual test, I'm going to ask for a client against that API. I'm then going to invoke the category/all API call. And I'm going to check that it's effectively returning me a 200 success status code. Once that is true, then I'm also going to read out the result, and I'm going to pause that. The result is effectively JSON, and I'm going to deserialize that in a list of CategoryListVms, which is what our action method was going to return. And then I have a couple of assertions checking that this is effectively the type I'm getting back and that also the result isn't empty.

**Summary**

I think we're in good shape now. This module has proven, I believe, that the application architecture lends itself well to being tested. We've written tests for the important parts of the codebase. Of course, we will need to write different types of tests. We won't get there just writing unit tests. That's why we also looked at writing integration tests in this module. We have a working API. Let's now see how we can plug the UI into this. In the next module, we'll bring in a client‑side application, and we're going to be using Blazor for this.

**Adding a UI**

**Module Introduction**

All right, we have a tested API. Time to, well, really use the API now. In this module, we'll be adding a UI to the solution. Adding the UI and looking at the specific UI code is not the goal of this course, but I want to show you how we can use the API for real, and while doing so, we'll learn about some universal ways to integrate with our API in client‑side technologies. I'm still Gill Cleeren. Also, for this module, I've decided not to change my name. Let me know in the discussion board for this course if you have any questions, and I'll do my best to answer them as soon as possible. The UI technology I'm going to be using here is Blazor. If you don't know Blazor yet, I will give you a very high‑level overview of the technology and some pointers on where to learn more. Next, we're going to learn how a large chunk of the client‑side code that communicates with the API can be generated, and we'll use for that NSwag and NSwagStudio. Next, although it's not the focus of this course, I want to take you through the client‑side code without going into specifics of how the Blazor code can be written. If you're interested in seeing how we can use the API from JavaScript front‑end technology, then too, you'll see a lot of things here that are universally applicable. Finally, I'm expecting another change we'll need to work through. I'm meeting Mary for another meeting.

**Introducing Blazor Client-side**

Now before we start, as promised, I'm going to give you a brief, high‑level overview of Blazor. In this course, we'll use client‑side Blazor, also known as Blazor WebAssembly. So what exactly is Blazor than? Well, great question. Blazor is the technology from Microsoft that allows us to write interactive UI for the web using HTML and C#. Is it like MVC or Razor Pages then? Well, no. MVC and Razor Pages are server‑side technologies, but Blazor is a client‑side technology, and thus allows us to create apps that run client side natively in the browser. So the model is the same as what you would create with Angular or Vue or React, but the main difference is that you are now not using JavaScript. Instead, you're writing C# and HTML. If you're not so much into writing JavaScript, but still want to create client‑side in‑browser apps, Blazor is a great choice. When creating Blazor applications, we actually have two options of how we want to run the code. The model I've just explained is what is known as client‑side Blazor, and it's built on top of WebAssembly. Alternatively, the exact same code can also be run on the server. I'll explain the differences some more in just a minute. As mentioned, Blazor is based on WebAssembly, which is a technology natively supported in modern browsers. Therefore, to run a Blazor application in the browser, the user doesn't need to install anything. Just a modern browser will do fine. No need to install extra plugins. Blazor is very extensible and can in fact integrate with already existing JavaScript components. So anything that is not supported in Blazor yet can still be created in JavaScript and can then be used directly from Blazor. Blazor is entirely based on C# and .NET, so you don't need to learn a lot of new things. All the typical packages you know, all the .NET namespaces, all that transitions, well, to Blazor. You write Blazor apps Visual Studio as well. Again, nothing new to learn. It's pretty easy to get started with if you're already familiar with C# and .NET. Blazor has a very cool feature. We can run the same code in different modes. Let me explain. When we talk about client‑side Blazor, we're talking about the model where things are running using WebAssembly. In this case, your compiled Blazor application is just a file which can be put on a server, and when someone browses to the application, the application is downloaded entirely to the client side and runs from there. Even if you lose the connection, things will keep running fine since, yes, it's running entirely on that client as an in‑browser app. Now, if you want, you can also run the exact same code on the server. In this case, there will be an ASP.NET Core server‑side app that hosts the Blazor app. The application that was running with client side on the client will now run on the server. When the user browses to the app, the initial view is downloaded, and end user will see that. Immediately, a connection is opened up using SignalR. When the user now interacts by clicking on something in the UI, information about that event is sent to the server using SignalR. The running application will get in this information, executes code, and the resulting changes that need to happen to the DOM are sent back to the client, and the browser will then reflect these changes. The cool thing is that it is the same code as we have with client‑side Blazor. You basically run the same code in the different models. We will, in this course, be using client‑side Blazor. The reason I've chosen that here is that you can hopefully see that this is then also using the same model as any typical JavaScript‑based application written in Angular, Vue, or another framework. The way we interact with the back end is what we need to learn about. Of course, there's much more to learn about Blazor than what I have just explained. I have a course here on Pluralsight that will help you getting started with Blazor. Take a look at the link you see on the slide to learn more about Blazor.

**Using NSwag and NSwagStudio**

Like I said, the goal of this course is not explaining you how to build a client app, but my goal is showing you how we can interact with the API that we have so neatly set up using clean architecture, and that's what we'll do here. What I use first is NSwag and NSwagStudio studio, two building blocks that make the consumption off the API that we have created a lot easier in just any client. We have our API. It's a REST API that we've built with C# and .NET Core. But for the outside world, what we've built it with is of no importance. Since it's a REST API, it returns JSON when we send in the request. We've added also, using Swashbuckle, the full description of the API, both in a machine‑readable and a human‑readable format. So now that we have moved on to the client side, we now need to write code that uses the API. That means plain HTTP code that will build up the request, sends it, waits for a response to come back, and once received, will process it. In many cases, that's a lot of boilerplate code that needs to be written in each client‑side technology that'll access the API. I will need to write this in C# to use my API in Blazor. But maybe someone else is using the same API from an Angular app, and so they will be writing the same functionality in TypeScript. Can't we have a lot of this in an automated way? We have, after all, already made the Swagger endpoints available, right? Well, there is another open source project that is a very commonly used for this, and it's called NSwag. NSwag is a toolset which can generate a lot of the boilerplate code that we need to interact with our API. It will do this based on the Swagger spec of the API, so we will basically point it to the JSON endpoint that describes the API and its methods. Using NSwag, all the code to send requests will then be generated. Also, the types required for this to work will be generated for the client side. So in our case, the return types that we're getting back from the API, as well as the types required to send our commands, such as add category, will be generated. That's a lot of manual work we don't need to do anymore. And the good news is that this works for C# and TypeScript. So no matter if you're building a Blazor app where you'd use C# or Angular, Vue, or React app where you'd use TypeScript, it'll work for both. So basically, this is what we'll get. On the left here, we have our API, and it exposes a Swagger endpoint. In our case, we've set that up before. The team that works on the client projects will now use NSwag to generate, based on the specification of the API, the client‑side boilerplate code, for C# or for TypeScript. The generated code will include all the code for interacting with the API, as well as the types, so the classes to use on the client side. Now, how do we then generate this code then really? Well, there are a few options that work all pretty much in the same way. What I'll be using in the demo is NSwagStudio, another open source tool, which is basically a GUI for NSwag. It comes with many options to tweak the generated code. You can also have the code be generated as part of the build or using a CLI. Here you can see a screenshot of its interface, and you can clearly see that the focus lies in the options that you, as a user, will get to influence what code will be generated.

**Demo: Generating Client Code Using Nswag**

In the first demo of this module, I'm going to connect with NSwagStudio to our running API and let it generate code that we'll then use in the next part in the Blazor app. So NSwagStudio is a desktop application, and you can just get it from GitHub. Simply search for NSwagStudio, and you'll get this client‑side application. I've, of course, already installed it here. This is NSwagStudio running, and the first thing I'm going to do is, of course, let it know about my JSON file, so my swagger.json file. So with my application running, this is the swagger.json link, and I've copied that here into NSwagStudio. And then I clicked on Create Local Copy, and then it reads out the JSON file. In our case, then we click on CSharp Client. But, of course, you can also have it generate TypeScript code if you're using your API from a JavaScript‑based application. And I've added a couple of more settings. So I've specified the namespace that I want the generated code to be part of. I've specified that I want to have it generate client classes. I've also specified here that I want to inject the HttpClient using the constructor. That will enable me from a Blazor application later on to pass in the HttpClient to this generated code so that I can also use the HttpClientFactory, which is the preferred way of doing this. I've also asked it to create the DTO types. So those are the classes that it will generate for the client side. And, finally, all the way at the bottom, this is the place where my generated class, the ServiceClient class, is going to be generated. So that's directly in my Blazor application. If I click Generate Files, it will regenerate this file. So every time that you make a change to your API so that the change appears in this regular JSON file, click on Create Local Copy to read in the new JSON file, and then click on Generate Files, and it will regenerate your client‑side code, so your C# code in our case. In the downloads of the course, this Nswag file is also included so that you'll already have all the settings that I'm using available to you. Let's take a look at the generated code. So that lives in the Blazor project, and we'll take a look at the Blazor project in just a minute. But, first, let us take a look at what was generated. So in here in the Services, Base, we have this ServiceClient class. So this is the generated code. It's quite a lot of code. I believe it is close to 1500 lines. It's even more than 1500 lines of code. That is all generated for me. So if you get paid by the number of lines of code, this will definitely get you a raise. Just kidding. Now what is in here? Let's take a look. The most important one probably is the IClient. That's the interface that has all the methods that we have defined on the API, and you see already a few here that we haven't looked at, Authenticate and Register. We'll take a look at those in the next module. But you see also our GetAllCategoriesAsync, which has been generated. Next to all the methods, you also see that the types that we have, for example, the EventListVm, that's a type that we've used quite a few times that contains the eventId, name, date, and imageUrl. Those have also been generated inside of our generated code from NSwagStudio. All of this has been generated, of course, also including the code that will make the actual API call that lives in the client class. The client implements IClient, the interface, and that expects an HttpClient. And that's important because I'm going to use that to be able to inject my own HttpClient into this generated code. In here, we'll find all the methods that we already saw. So, for example, the AuthenticateAsync. Let's not look at that one. Here you see, for example, the GetAllCategoriesAsync, which is then going to make the call to api/category/all. All of this code, we don't need to write it ourselves. It's already been generated for us, so we can now use this. How am I now using it in my application? In the Program class, I'm registering the IClient, which is the interface generated by NSwagStudio, with the client. And so that's our generated class. In the service collection, you can see that I'm registering this here using the AddHttpClient, which is using the IHttpClientFactory. And that's the preferred way of using the HttpClient from a Blazor application or, in general, from just any .NET Core application that uses an API from code. And that's what we are doing here. I'm configuring here also the BaseAddress. Let's now see how we are going to use this from Blazor application code. This is not really specific to Blazor. I have created really service classes in this case inside of my client app that will talk, that will communicate, with that service client. Let's take a look at, for example, the CategoryDataService. The CategoryDataService inherits from the BaseDataService. The constructor accepts an IClient, an IMapper. We're, again, using AutoMapper and a local storage server to store data locally. But then most importantly in the GetAllCategories, we are now calling on the client, that will be the client of the ServiceClient class, the generated code. I'm simply calling onto that the GetAllCategoriesAsync, and that lives in the generated code. And that will then return me simply all categories without me having to write code that goes out to the API, creates the Get request, waits for the response. All of that, I don't need to do. I simply call the GetAllCategoriesAsync that was generated for me using NSwagStudio. The return data is of type CategoryListVm. That's the type that was returned from my API and also was generated in the generated code. I then map that back to, in this case, a ViewModel using AutoMapper, and that I'm then returning to the caller, which will be the UI, and we'll take a look at that in just a minute. So as you can see, using NSwagStudio, we don't have to write all that code manually to interact with our API because we exposed a Swagger endpoint on the API, and I let NSwagStudio look at that endpoint, generate code, while I've saved myself a lot of time.

**Exploring the Client App**

Now that we can save ourselves a lot of time, we're generating all the code to interact with the API, let's now take a look at how we can use this from our Blazor application. And while we're at it, we'll look in a bit more detail at the Blazor app itself. I will show you the code in the demo. Now before doing so though, there are a few points I want to highlight here. In the Blazor app too, we'll make use again of some packages. I'll be using AutoMapper again to map between the different types. This is what I love about Blazor. It's the same tech on both sides of the spectrum. Things that we've learned can easily be applied here too. For the communication with the back end to the REST API, we'll use the HttpClientFactory, not the regular HttpClient. I said we'll include the code that was generated with NSwag, so that's the actual code that'll be communicating with the REST API. We'll also bring in some authentication, a basic form that is. And since we don't have that in the API yet, we'll look at this in the next module.

**Demo: Exploring the Blazor Application**

Demo time. In this demo, we'll take a look at the Blazor app and see how it integrates with the API. So by means of example, I'm going to show you how one screen is built, and they're all pretty much the same. I want to show you a bit of the Blazor code and how it actually talks with our back end. So here's the running app. Let's click on Events, for example, and we'll then see a list of all events. I think you can already imagine which column we are going to be using for that. Let's take a look at the code. The page that you just saw is this EventOverview, and it has a code‑behind file. The EventOverview is just Blazor code and is actually going to loop over a list of events that we're going to expose, and it's then going to visualize all these events in the table that you just saw. Now I said I'm not going to explain you everything that has to do with Blazor. This is pure Blazor code. Again, take a look at that other codes that I mentioned in the slides. But let's see how this is now working together with what we already saw, that code that was generated with NSwagStudio. So this is the class code‑behind that contains the functionality. The EventOverview will use an EventDataService. That is one that also lives here under Services, which is going to work again with the generated code available through the client. In the GetAllEvents, I'm simply going to call the GetAllEventsAsync, which is going to be in that generated code and determine the list of all events in an asynchronous way, that is. That returns me, again, the EventListVms, and I'm going to map that to an EventListViewModel, a local type inside of the Blazor application. I'm using AutoMapper for that. So again, we have defined profiles here, which are going to do the mapping between the different types so that AutoMapper knows between which types it needs to map. So I'm returning that list of EventListViewModels from this EventDataService. So back to the real Blazor code, that EventDataService, as you can see here, is being injected. In Blazor, we are using also dependency injection, but I cannot use constructor injection in a component. So this is actually a Blazor component. The EventDataService, as you can see here in the startup, Blazor doesn't use a startup, but uses a program instead. But I am indeed using here the EventDataService, and I'm registering that with the service collection. Then on that EventDataService in the OnInitializedAsync method, which is going to be called when the page loads and the component loads, I'm going to call that GetAllEvents method, and that's the one I just showed you. That will fill up this collection of EventListViewModels, and that is the one that I was actually binding to in the UI. So that's how I'm listing out the events in my UI. Now this should already give you an idea of how the Blazor application is really built up. We are using components, pages in this case, and those are using our data services, which are working with our generated code to talk with our API.

**Demo: Adding the Paging Functionality End-to-end**

I've just given a demo of the application with Mary, and she was really happy about the app we already have, but she made a good point, the page that lists out the orders is now pretty long, can we add paging to that? Of course we can Mary. In the final demo of this module, we're going to bring in another feature end‑to‑end, namely paging in the oldest page. For this we'll also need to rerun NSwag, and we'll then need to update our Blazor code. For the last time in this course, I'm going to show you that adding a new functionality doesn't have to be a big problem. We can actually pretty easily bring in a new functionality. So, Mary asked us to create this. I'm already showing you the result. Initially, we had a very long list of sales, and I think that's pretty easy to understand, and I don't think it's a good user experience to have a long list of data. Instead, what we should be using is paging, so I have now added functionality for paging onto our API and also in the Blazor app. You can see that I can now page to the list of sales. How did I bring that functionality in? Let me take you through the steps I've gone through here. On the API, and that would be on the OrderController, I have now added a method, GetPagedOrdersForMonth, that is expecting a DateTime, so the month, but also the page and the size. The page will contain, well, the page int that is requested in the UI and size will point to the number of orders I want to return. I have created then in my application code a new GetOrdersForMonthQuery. Indeed, I've added under Orders a new query that also expects the date, the page, and the size. In the Handler, I have now added on the orderRepository a new method, GetPageOrdersForMonth, which expects the DateTime, the page, and the size, but now on the repository level, so that in the orderRepository, in the actual implementation in the Persistence project, we can now go to the dbContext and update the query so that only the right orders are returned. That's what you see here. But notice while we're here, I have also added another method, GetTotalCountOfOrdersForMonth, because my Pager will also need to know how many orders that will be, so I have added that here also on the repository. Indeed, in the Handler I'm also asking the Repository for the total count of orders for that month. And then I'm going to return a new VM, PagedOrdersForMonthVm, which is going to contain the orders as an ICollection of OrdersForMonth, but also the count, so that will be the object that I'm going to return to my client. So, indeed, if we look back at the controller, that's going to send that GetOrdersForMonthQuery, and it's going to send that to MediatR, and that is going to return a PagedOrdersForMonthVm. My client then not only has the orders for that month, but also knows how many orders there are in total that it can use in the Pager control, and is included in the Blazor application to show the number of pages, as well as the list of orders. Now I've added this new method on my API, so I also need NSwag to run again. I have already shown you I need to have NSwag regenerate, let's say, that client‑side code. Once that is done, that code is updated in the ServiceClient. In the OrderDataService, I now have the GetPagedOrdersForMonth method that I've added. That will call that method that is generated in the client and returns the PagedOrdersForMonthVm that I'm mapping to a local type, again, using AutoMapper, that is then being returned. If we then look back at the TicketSales page, we indeed see that we are using again a table to show the orders, and we also use a custom control, a Pager that will list out the number of the pages, as well as the active page. The TicketSales component code will use the OrderDataService, and will go in its GetSales method to the OrderDataService, asking it for the GetPagedOrdersForMonth, passing in the month and year I want the sales for, the page number, and the page size. So the key message here is, it's the same story. We just add a functionality on our API, we add a new query in our application code, we implement the new functionality on the repository, and we call that from our Blazer application after we have run again NSwagStudio to generate that code that talks with the API.

**Summary**

We've reached the end of another module, and we now have a good example of connecting a client‑side app with the API. Technically, we could spend time creating the Blazor app to using clean architecture, but for the sake of simplicity, I haven't done that here. Maybe that's a good exercise for you. What we did see here though is that NSwag can be used for any client to easily scaffold the code that interacts with the API. It'll be a great time saver. And our Blazor app is then happily working together with the API and its architecture that we have created so far. One module to go. We still have a few loose ends that we need to solve back in the API and its layers. In the next and final module of this course, we are going to bring in several crosscutting concerns, such as logging and authentication in the API. See you there.

**Improving on the Application’s Behavior**

**Module Introduction**

We have a working API, which is based on good architectural principles, and we have a working Blazor application. Things are moving along nicely for the architecture and reference app we're creating for GloboTicket, but there are still a few things missing in the architecture. Let's call them crosscutting concerns that'll be needed throughout the entire application. In the last module of this course, Improving the Application's Behavior, we will bring in support for quite a few of these. My goal with this is, again, not teaching you all the details. We have plenty of other resources available for that. Instead, I want to show you how this can be fitted into the architecture. I've selected a few topics that we still need to add into our application architecture. First, I'm going to go into error handling and how we can bring that in all the different levels. We'll need to take a look at how this can be added in the core, as well as on the API so that clients know what happened based on the request they have sent in. Next, I'm going to bring in support for logging, another crosscutting concern that we'll need in the application. And finally, I'm going to implement a base authentication mechanism to authenticate requests on the API. Let us take a look.

**Handling Errors in the API**

And as mentioned, let's see how we are now going to react when some error occurs in the API and other layers. Handling errors correctly and letting clients know what happened is crucial. So in this part, I'm going to explain how we can, in the architecture that we have set up, handle errors. There are definitely different approaches possible here, but I've selected just one. The approach I have taken is actually twofold. First, we'll need to look at the core project since that is the place where errors and exceptions can and will occur. In the core application project, I have to find a number of custom exceptions, classes that inherit from the built‑in ApplicationException. The reason that I'm defining custom classes here is that these classes can also be used from any other project that references the core, so pretty much everywhere. This will include exceptions that happen when we are, for example, asked to retrieve an instance that doesn't exist, so basically a NotFoundException. We'll create our own not found exception instead of using the one that comes with ASP.NET Core. Remember, we're not talking about the API project here, but we are talking about the core application project, so these exceptions will be coming out of the application core. Then, the next task is letting the client know what happened, and this I'm going to do by returning ASP.NET status codes, possibly accompanied by information about what happened. The letter could be, for example, validation errors coming back from the core. I'm going to handle these exceptions in a simple way, namely by adding middleware in the API, which will convert the exception from the application core to an exception known by the clients. The exception classes themselves are pretty straightforward, and you can see an example, the aforementioned NotFoundException. Say that the client asks for an ID that cannot be found. Well, then the core application project will throw such an exception. The class is, as you can see, inheriting from ApplicationException, a built‑in type in .NET Core. In the API, we'll use and set middleware. Now middleware is a typical a ASP.NET Core functionality. It's basically a piece of software component really that is plugged into the request pipeline. When a request is sent to our ASP.NET Core application, it travels through the pipeline, and while doing so, middleware components can work on the request. Each of them can inspect the request, and if needed, perform actions with it. It can change the request, it can pass it on to the next component, or even decide that the request shouldn't go further through the pipeline and thus perform a short‑circuit, sending back a response to the consumer. Middleware can also work on the response, which is what we will be doing. After execution, it'll travel back in the opposite order through the pipeline, and middleware can inspect it again. Our custom middleware component that we'll include will do exactly that. It'll see if an exception has happened, and if so, it'll convert the exception. I will show this in the demo very soon. One thing to watch out for is the order that you apply components in the pipeline. Since the middleware components typically pass the request from the one to the next component in the pipeline, the order that you place them in has an influence on their behavior. Setting up the middleware pipeline happens at the startup of the application in the Configure method, part of the Startup class. Here you can see a visual representation of the request middleware pipeline. When a request arrives, it'll pass through this pipeline on which the middleware components are plugged in. Each component has the ability to inspect a request, act upon it, pass it on to the next component, or simply short‑circuit it. This happens on the request, as well as on the response coming back from the application code. ASP.NET Core comes with quite a few components built in, and some are already plugged in by default. But it's also an extension point. You can create your own middleware components. Now writing custom middleware is not very difficult. A middleware component is just a class that must have a constructor with a parameter of type RequestDelegate. We'll plug it into the pipeline in the Configure method, typically using an extension method, and I'll show you how to do this in the demo. The actual functionality will live in a method called Invoke or InvokeAsync, which must accept the HttpContext parameter. This method will be invoked automatically upon execution of the middleware in the pipeline.

**Demo: Handing Errors**

Demo time again. In the first demo of this module, let us add exception handling in the application architecture and see how this cross‑cutting concern fits in. We need to perform a few tasks, creating the custom exceptions in the core application project to throwing exceptions from the application logic so perhaps when validation fails, and then converting the exception using custom middleware. I have already shown you the custom exceptions that I've created in the application project. So we had the NotFoundException, which inherits from the application exception and typically it will be used if someone wants to update, for example, an event that doesn't exist. The BadRequestException, well, that's more used if the input is, for example, null when we are expecting to update an event. ValidationException, I think the name gives away what that one is going to be used for. It will be used to return to the client validation errors that have been thrown by fluent validation. Let's take a look where I'm using that. For example, let's take a look at the NotFoundException. The NotFoundException, as you can see, is used, for example, from the UpdateEventCommandHandler, that is this one. In here, I'm going to try updating event, and of course, if the event doesn't exist, well it's pretty hard to update it, right? So I'm going to, after the search using the GetbyIdAsync, see if that event is not null, and if it's not null, well, then I'm going to throw one of my custom exceptions saying that that event simply can't be found. Then how are we then handling that in the architecture. And the application code is simply throwing this custom exception, but these exceptions, they aren't useful for a consumer of the API really. So what I've done then is I've created in the API a piece of custom middleware, and it's called the ExceptionHandlerMiddleware. That ExceptionHandlerMiddleware will take a look at the exception that came back. So if an exception is being thrown, I'm going to check which type of exception. Those are our exceptions over here, which of these has been thrown, and then exception is then going to be converted into an HTTP status code. For example, if I'm getting back the NotFoundException, I'm simply going to return an HttpStatusCode.NotFound. Depending on the exception, I'm also going to be returning an error message and then that message is going to be written to the response. This middleware allows me to basically catch all exceptions coming out of the application code. That is why, for example, in the events controller in the update, I've specified here that this is typically going to return a 204NoContent, but it can also return a 404NotFound and that is because that middleware is hooked in on the middleware request pipeline, so our middleware is responsible for converting the exceptions coming back from the application. Now simply creating that middleware won't do anything. I still need to plug it into the middleware pipeline that I'm doing here by creating another extension method, this time on the IApplicationBuilder that gives me access to the middleware pipeline. Here you see that using the UseMiddleware passing in my exception handler middleware, I can plug that in. This extension method, I'm then calling from the startup in the configure method. There I now say use the custom exception handler.

**Adding Logging to the Architecture**

Exception handling was the first cross‑cutting concern that we added into the architecture. Another one that is currently missing is logging. I think just about any application that you build should include logging. Otherwise, it'll be pretty hard, if not impossible, to know what went wrong with the application when it's running in production. Let's see how we can add this. Since ASP.NET Core, adding logging has become a lot easier, since it is basically already built into the platform. To enable it, just like with everything in ASP.NET Core, we need to follow a few steps. Logging is typically configured in the Program class, so not from the Startup class. In the AppSettings.json file, we can specify the options we want to apply for the built‑in logging system. In our code, throughout the application code, we can now write log statements. ASP.NET Core gives us the ILogger in T, and using that, we can write log entries. While we can use the built‑in logging system, I won't be using this one, and I'll instead use Serilog, which we'll also need to configure. Let's take a look at these steps in a bit more detail. As said, ASP.NET Core and .NET Core come with logging support built in. We can therefore tap directly into that system. However, it's a very flexible system. It's provider‑based, which means that we can configure using a provider model how the logging is handled. So, for example, where the log messages should be sent. And it's actually pretty cool, since it's really an extensible model. ASP.NET Core comes with a number of logging providers built in, including Debug and Console. We can write our own providers or use a third‑party one. It is what I'll show you in just a minute when we add Serilog. On the logging engine, quite a few things can actually be configured, often done, like I said, through the AppSettings.json, where we can specify these settings. One of the most important ones is the LogLevel, which can be seen as a filter to specify which level of messages should be sent to which provider. Again, it's not my goal to explain you everything around ASP.NET Code logging. We have dedicated courses for that here on Pluralsight. Here you can see how to write a message to the log. The ILogger instance will be provided through dependency injection. The code itself is agnostic to where we want to look. We just write logger.LogError, and then all configured providers will be triggered. Indeed, we can have multiple providers, but this has no effect on the code throughout the application. The LogError method is simply saying log to all providers that are configured to LogError level or above. So this is where the log level comes into play. ASP.NET Core's logging engine comes with six log levels. Trace is the lowest. Then we have debug, then information, warning, error, and critical. From code, we can thus decide what level should be used, and in the configuration of the provider we specify for which level a provider should be triggered. Now we can do things fully with what we've seen here. But instead of using the default logging providers, we are going to use Serilog. Serilog is again an open source package that we can bring into our application architecture. Serilog is a third‑party logging framework which offers a number of advantages on top of the built‑in logging. Through add‑on packages, it enables us to write our logs to different log destinations, destinations not available in ASP.NET Core's default providers. This includes a file, as well as the database. Serilog also offers something called structured logging. Through structured logging, we get control over how log messages are written, and we can use templates to define how a message should be written to the log output. As mentioned, adding Serilog is done, again, through NuGet. We'll typically need to bring in a number of packages. Writing to a different output typically requires that you bring in another package. It's very nice to see that Serilog offers a very fine‑grained model and just adds what you want to use. Here you can see for example that I've brought in support to write to a file.

**Demo: Adding Logging to the Application**

Let's return to our application and see how it can add logging. We'll use Serilog, and we'll need to configure it properly. Then we'll write log information from within our application core code. Logging is another cross‑cutting concern I want to enable in my API. I want to make sure that if exceptions happen and I can see what has happened in a log file, that I've added support for here using Serilog. As you can see here in the project file of my API, I have added support for Serilog, and I've brought in also a couple of extra packages. This one in particular allows me to also write to a log file, which is what I want to do here. How do I configure this cross‑cutting concern logging then? Logging is one of those things that Microsoft recommends you do not do from the startup, but instead you do it from the Program class. So in here in the static void main, I'm configuring the logger. First, I'm going to read out the app settings. The app settings contain settings for Serilog on how to log and that's what you see here. We're not going to take you through all these settings. You can find that on the Serilog website. Let us focus on how we fit this into the architectural. This logger is the Serilog logger. I'm configuring it by pointing it to that configuration that we've just read out from app settings, and I'm also specifying that it should write to a file with a rollingInterval of Day. You can see here in the Solution Explorer that a new log file is created per day in a specific format also containing the date, and then I create that logger. With that done, logging is configured in my application; Serilog has been added now as a logging provider. I don't need to do anything else for this to work. Then I can go to my application code in, for example, the email service. The email service, if you remember, is part of infrastructure. I now can simply use here the ILogger in EmailService. The logger can now be used to log whether or not sending the email was successful. And do you remember the CreateEventCommandHandler? Well, there, too, I'm injecting a logger. And when the email sending actually didn't work, I don't want the application to break, I don't want an exception to be thrown. I just want to log this, and I'm again using the logger here.

**Authenticating Users**

In the final part of this module, we are going to bring in support for authentication, again, a cross‑cutting concern we'll often need in our applications. In the GloboTicket application architecture that we have created thus far, this hasn't been added, so let's see how we can add this now. I do want to stress that authentication is a complex topic and can easily fill one or more courses. Other authors here on Pluralsight have already created these courses, and thus, the only thing I want to do here is adding a simple mechanism in the API, as well as in the client app so we can ensure that the requests coming in from the Blazor app are effectively authenticated. I'll also show you a simplified version of the authentication in the Blazor app in the demo. Let's first take a look at the authentication on the back end. At the front of the back end, so on the receiving end, we have the API, and that one will be used in my setup here to authenticate requests. Now an API is stateless, meaning that we cannot rely on server‑side functionality to secure it. Each API request that will therefore be coming in should be validated, and that's going to be a task for the API itself. Now, how are we going to perform this authentication then? Are we going to be sending a username and password entered by the user in the Blazor app every time? I don't think, even with encryption enabled, that is a great idea. Instead, we'll rely on a token to be included. In this sample, we'll use the industry standard, a JWT token, or a J‑W‑T token. Using a token, we can, on the API, validate if the request is authenticated and if the user has access to the resources they are trying to access. In the client application, we'll first need to get a hold of a token. This will happen after successful authentication. Once authenticated, the token will be sent back to the client and typically stored also by the client. Then, when a request needs to be made to the API, the token will be included with the request, and it's typically included in the header. Now as that authentication is a large topic and can be done in several ways, what I'm going to show you here is an approach to add authentication on the API using ASP.NET Identity. Other options include IdentityServer, a commonly used library in this context, which is also supported out of the box from a ASP.NET Core. You can also rely on an external provider and configure this to be used from your application. But, as said, we'll use a ASP.NET Identity. The approach is what you see here. We have the Blazor application, which will be first redirected to a login page. On that page, the user needs to enter their credentials, typically a username and password. These are sent to an authentication endpoint that will validate these against the database. And here we are using the ASP.NET Core Identity APIs. If the credentials are valid, we'll generate a token, which is then sent back to the client as the result of the authentication call. The token is a JSON Web Token, a J‑W‑T, or a JWT token. This token is now stored and included again with every subsequent request made to the API. The API will then verify that the request has the token validated and allowed to request through if the token was valid. ASP.NET Core comes with the well‑known Authorize attribute to secure access to the controller. As I will show you in the demo, we'll configure the application for authentication in the Startup class. If our API gets in a request for an action on a controller secured by the Authorize attribute, in the Startup class, we'll use the add authentication, specifying that we'll want to use JWT tokens. I'll show you this in a demo.

**Demo: Adding Authentication Using Tokens**

Now because this is quite complicated stuff, I have split it up in two parts. First, I will show you the API where I have added the AccountController, a new controller, that is. We'll take a look at the Startup where I have now configured the API for using JWT tokens, and we'll see how I've secured the access to other controllers. So let us now take a look in this demo to see how I've now added support for securing the API through Identity. You'll see under infrastructure in the finished code for this module that I've now added also an Identity project. An identity is really infrastructure code. It really has to do with an implementation of that security. So it really belongs under infrastructure. Nobody's in there. Let's take a look. I have in here added another DbContext. The GloboTicketIdentityDbContext, which is going to be using the ApplicationUser. ApplicationUser inherits from the regular IdentityUser, but just adds first name and last name. That application user you can find over here. Now, the most important logic is here under the AuthenticationService. And the AuthenticationService will use plain ASP.NET Identity classes to register and sign in a user. I'm injecting the userManager and the signInManager, and I'm also going to use an instance of JWT settings, which is a class I created myself that will contain the settings that I'm reading it from the AppSettings file. And this class just has two methods, AuthenticateAsync and RegisterAsync, and it actually has a third one, but it's a private one. Let's first take a look at RegisterAsync, which will get in a RegistrationRequest. The RegistrationRequest is actually defined under application models. So, over here. In here, I have all the classes related to data, requests and response, basically coming in and going out of the API in relation to Identity. In the RegisterAsync I'm doing pretty regular Identity stuff. I'm going to register a user. So first, I'm going to see if that username already exists. If that is the case, well, then I'm going to say that the username already exists. Otherwise I'm creating a new application user. I'm going to see if the email already exists. And if that is not the case, well, then I'm going to register the user. In the AuthenticateAsync, I'm going to get in an AuthenticationRequest, which will contain the email and the password. This will be called to try and authenticate the user, and as a result, I'm going to send back a token. So I'm going to again try and find the user. If the user is found, I'm going to try signing him or her in, and if that succeeds, then I'm going to generate a token. That's going to be a JWT Security Token, a built‑in type of ASP.NET Core. In the GenerateToken method, I'm going to generate the token containing a few claims, including the username, email, and the GUID. In the response, I then include the username, the email, and the token, and I sent that back to the caller. Also still in the Identity project, I've added this Extensions class, which will contain the configuration for Identity. Notice that I'm reading out the section JwtSettings from the configuration, and I register the DbContext, I specify that I want to use Identity using ApplicationUser, and I'm going to use for that my DbContext. I register my AuthenticationService, and finally, I configure how authentication should be done. And I specify here that a token, a JWT token, is required. And I specify old parameters for that over here. I also configure how the different authentication events should be handled. So with that, my configuration of identity should be fully done. Let's now go to the API, and in there, we'll now find a new controller. And that will be the accountController. The accountController uses the AuthenticationService. For that, I now have an interface defined again in application contracts. And now I have a subfolder Identity, which then contains that interface. That contains the AuthenticateAsync and RegisterAsync. That is injected through dependency injection. We registered that in the Extensions method. We still need to add that to our Startup. Let's not forget about that. And in here, I just have two methods, two POST methods, one for Authenticate and one for the Register endpoint. They are going to be using the AuthenticateAsync and RegisterAsync methods. In the AppSettings, you'll see here that I've added the settings for the token generation. And in the Startup, I've now added the call to also bring in my identity configuration. Here you can also see, and I've now configured Swagger to also know that it will require a bearer token. And of course, here we have now specified that we are going to use Identity. So we have specified here UseAuthentication. I've also brought in UseAuthorization because I've now specified that these calls are now only accessible if the call is authorized. This authorize attribute will make ASP.NET Core search for the token in the request header, validate it against the specifications that you specified, and only if the token is valid will this call be allowed. This is not have added identity in our application architecture. It's again a cross‑cutting concern that we've added.

**Demo: Adding Authentication to the Blazor App**

In the very final demo of this course, I'm going to show you a way to bring this authentication into the client app. In this demo, I'm now going to show you how our Blazor application is now using this token. I've in fact, on the home page, created a login and register functionality. I've already created the user, so if I click here to login using a username and password, this will now be sent to the AuthenticateAsync endpoint, and if the username and password combination exists, and I will get back a token. And then I can still navigate to the Categories, for example, and that still works because of the fact that my token was included with the request. Let's see how I've done that in the Blazor app. I've included another Service.Authentication service, that is going to accept the username and password, and it's going to build up an AuthenticationRequest. Now where's that AuthenticationRequest coming from the? Good question. Well, it's actually coming from the service client. Because of the fact that I executed NSwag again, it's AuthenticationRequest type expected by my Authentication endpoint is also included in the generated code. So I'm passing in email and password, and then I simply call AuthenticateAsync on my client, which is going to go out to my API, passing in that AuthenticationRequest. That is going to get back an authenticationResponse, which should, if successful, contain a token. If that token is included, then I will store that in the local storage. Blazor can actually access the local storage of the browser and can use that to store an authentication token. On subsequent requests, for example, on the CategoryDataService, I will now go out to the API, only including the BearerToken. In the GetAllCategories, what you see here is that I'm calling the AddBearerToken method. In the AddBearerToken, I'm going to see if we have indeed, in local storage, a token, and if that is the case, I'm going to include that in the AuthorizationHeader as the BearerToken. That token is then going to be included on the request sent to our API. Our API can validate the token, and this can distinguish between authenticated and not authenticated requests. And the rest of the Blazor application, I'm not going to show you, you saw how we can do the actual authentication, but take a look at the finish code if you're interested in more.

**Summary and Course Closing**

And with this, we've reached the end of this module, where we have added several cross‑cutting concerns to the application architecture. You've seen how we can add exception handling, how we can add logging, and how we can bring in a basic form of authentication. Our application architecture proposal is ready. I just had a chat with Bethany, team lead at GloboTicket. She's quite enthusiastic of what we have delivered. The feedback from the developers was also very positive. And what about Bob? Well, he's very positive, too, much more than he was some time ago. He sees that what we have proposed here is a solid foundation for future projects. So I think our job here at GloboTicket is done. The team can now continue building testable and maintainable code. The only thing that's left for me is congratulating you on completing this course. I hope you found it useful, and you will be able to apply this knowledge in future projects. Please give this course a rating, ask questions, and give feedback. Thank you for watching!