Building End-to-End Multi-Client Service Oriented Applications

by Miguel Castro

Resume Course

Bookmark Add to Channel Download Course Schedule Reminder

Table of contents Description Transcript Exercise files Discussion Learning Che

Welcome & Course Description

The Course

Hi. This is Miguel Castro, and I am really excited to introduce to you my new course, Building End-to-End Multi-Client Service Oriented Applications. This is not a niche course with a very, very focused technology. This is a course for everybody. This is something that everybody is going to get something out of. This is going to take you through the building of a Service Oriented Application, a complete application from beginning to end covering every layer in the architecture. We're going to cover everything from the database and the data access layer through the business tiers all the way into the service layer, all of the entities on the business side of the wire into the client side of the wire, all the entities there, all of the proxy layers, the different UIs. Yes, there are going to be multiple UIs here. We're going to cover web technologies and we're going to cover desktop technologies. Like I said, this is going to be a course for everybody, and I'm super excited to share it with you. I'm going to be combining a multitude of technologies in the process of developing the application. Essentially I'm going to be answering the proverbial now what question. This is something that comes up in most of my classes and conference sessions where I teach a technology and people learn the technology, but a lot of people still ask okay now what do I do? How do I implement this? How do I wire it all together? Well, this course answers that question by demonstrating not only how to use the technology, but also patterns of implementation and connectivity that let you see how all the layers talk to each other and how the application comes together from beginning to end. Essentially I'm going to show you what travels from where to where, what to what, and how it does it. I'm going to be using a lot of technologies along the way. In fact, most of these

technologies have their very own Pluralsight course already out there. I'll be concentrating on the implementation of the technologies and how they all work together, but don't worry, you're going to see a lot of code, a lot of pattern, a lot of demos, and I think you're going to have a lot of fun watching. I'm going to be incorporating a lot of patterns and techniques, some of them very common, some of them a little more unique to my style of doing things. They solve a lot of common problems that come up during application development. Some of these involve some abstraction patterns like factories, abstract factories, and strategy patterns, and I've also developed some pretty cool lambda expression techniques that save me a lot of code repetition in many areas. All of the layers I'm going to be developing during this application are going to all follow SOLID principles. This is going to allow for really good code manageability and extensibility, and, of course, testability. Testability is going to be pretty important. I'm going to be writing unit tests at several layers of the application and show you exactly how they work and how I do it. I'm also going to be incorporating Dependency Injection throughout from beginning to end. You're going to see dependency injection usage from the data layer all the way up to the service layer, and this is a place where it's commonly not know to be used very often in the WCF service layer. I'm going to show you how you use dependency injection there to inject dependencies into WCF services. I'm going to show you how you can DI aware your proxy classes and inject them in whatever UIs or whatever clients are going to use the proxy classes. So, you're going to see quite a lot of dependency injection usage. And, of course, setting up the app to use dependency injection is going to be the most important thing in order to ensure that the code is all testable.

Course Agenda

This course will be divided into several modules of course, some longer or shorter than others. In the first module, I'm going to jump right in and give you a full demo of the case study application that I'm going to be developing. This is going to be a Car Rental System. The reason I chose a car rental system is because I wanted an application that merited having more than one type of UI, and a car rental system usually has a web UI that the customer is exposed to, and it has a desktop UI that the employee behind the car rental desk is exposed to, and we're going to be developing both of these. The second module is going to describe service oriented architecture and service oriented applications, as well as the technologies that it uses or that are used commonly with it. I'll be talking about how software development has evolved into service orientation, what makes service orientation different and so accepted in the industry. I'll also be talking about the technologies that I'm going to be using in the development of this application. In the third module, I'll be going through the architecture of the car rental system, the use cases and goals of what the application should do, and also the layering and why it's layered the way it is and what the design goals are in the application, or should I say in the architecture of the application. In the next module, I'll be developing the entities used throughout the application. There are two sets of entities here. There's the business side of the wire entities and the client side of the wire entities. I'm going to be describing in detail both of these sets, why they're different, why they exist as two different sets to begin with, and we'll be developing them from beginning to end. The

next module is all about data starting with the database and into data access. This actually signifies the beginning of the business side of the wire, and I'm going to be showing you how I write the data access layer, why I do it the way I do using repository and abstract factory patterns. I'll describe all of it in detail, and, in fact, I'll be developing all of it from scratch. And this is where you're also going to see dependency injection starting to appear. The next module is going to be about WCF services and, when necessary, some business engines or business objects. I'm going to go through the design and development of WCF services along with all the patterns that I use when I develop them including, like I said earlier, dependency injection patterns. I'll also be securing this layer, which means I will be adding security to it in a number of ways. I don't only want to secure against usage of the service operations, but I also want to secure against the data that's attempted to be retrieved to make sure it's the proper person trying to retrieve the right type of data. I'll also be introducing unit testing into this module to show you how to unit test WCF services. The next module will be a pretty short one. It involves service hosting. I'll be showing you how I host these services, but also talking about other options and other alternatives that are available to us regarding service hosting. This module signifies the end of the business side of the wire meaning all of the code that I've developed from the beginning of the course to this point in time can stand on its own no matter what client it's going to service. So, next we're going to move into the client side of the wire, and we're going to jump right in there by developing the service access layer or, as it's more commonly know as, the proxy layer. And, once again, I'll be incorporating dependency injection while I'm doing it. Once the proxy layers are done, then I can jump right in and start developing the user interfaces. The first user interface that I'll be developing is going to be the web site, and I'll be using several technologies for this including ASP. NET MVC, Knockout JS, and Bootstrap for some CSS work. This will also incorporate user authentication. So, you'll see security at this level. You'll also see a combination of S-P-A or SPA, Single-Page Application, style in combination with non-SPA. I'll get into more detail during that module, but you're going to see SPA and non-SPA work together. I'll also be using Web API and jQuery along with several patterns and techniques that'll help me with code reuse and also enable testability. The last module brings us into the desktop application, which I'll be developing using WPF. As with any WPF application, or at least any well written WPF application, I'll be incorporating the MVVM pattern or Model-View-View Model, as well as more dependency injection. In this module, I'll also be adding an abstract factory pattern to the proxy layer developed earlier, and this is going to help me solve a problem that I'll describe during this module.

Ready For Some Fun

Throughout the course, to avoid repetition and unnecessary redundancy, I'll be developing parts of the application from scratch, but obviously not the entire application. For example, when I am explaining entities and developing entities both on the business and the client side where they differ greatly, I'll be coding up one on each side completely from scratch, and, of course, giving you all the detail that's necessary. But since all the other entities are going to follow the exact same pattern, I'm not going to be doing every single one. Likely with services, I'll be

designing and writing one of the services from scratch beginning to end showing you every pattern that I use, every setting that I use, how dependency injection works, how they communicate with entities, how they communicate with business engines, etc., but the other services in the system following the same pattern will be shown in their finished form. When I get to the UI, I'll use a similar process where I'll show you one complicated view and how the controller gets to the view and how Knockout is involved and MVVM and how things communicate with each other and how I inject my proxies in there and how I test everything, but all of the other views are kind of developed in the same way, so I won't be wasting your time with them. I'll show them in their finished form. And the same goes with the WPF application when it comes to its views and its view model. So, either way I think you're going to have some fun, I think you're going to enjoy watching this, and let's get going.

Demo of A Service Oriented Appication

Description and Goals of The Case-Study App

Hi. This is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. In the introduction, I mentioned that I was going to be writing a Car Rental System as the case study for this course. As I stated earlier, the reason I chose a car rental system is because that kind of business typically has more than one point of entry for software, a user side and a customer service side. The customer typically accesses their software through a website in which they can do things like make reservations, view history, manage their account, things like that. CSRs, Customer Service Representatives, usually have some kind of desktop app in front of which they sit all day doing things like booking rentals from reservations, accepting returns, etc. The backend of the system will be driven entirely by WCF, and it's going to be exposing its API to the internal internet, which will drive both the website and the desktop client. Remember, in a web application, the IIS server is the client itself, and that still sits inside an internet. Any further reach extension of any services would be handled by a Web API layer. The customer-facing website will be written entirely in ASP. NET MVC in combination with Knockout JS. This combination is going to give us a real rich user experience and also limit the trips back to the server. The desktop client will be written entirely in WPF, which, in my opinion, is still the premier platform for writing rich desktop applications. The goals of the application I'll be writing during this course are several. The first one is to demonstrate good technique and practice for implementing connectivity among all the different layers that comprise a service oriented application, how things talk to one another, how layers communicate with each other, and using what kind of components. I'll also be showing and demonstrating different patterns, as well as dependency injection, which will be used throughout the entire system on both sides of the wire, the business side and the client side. And a good byproduct of this is that it's going

to effectively make ever layer of my application quite testable. I also want to make sure that I maintain a certain level of cleanliness and elegance in my code in how the code is laid out and stored. Everything from code design patterns demonstrating good separation of concerns all the way to something like folder structure all contribute to ensuring that the system remains manageable now and also as it grows in the future. Also, I feel I should mention this application is also my reference app. This is what I use to provide myself with sort of a kickoff point for the creation of new applications. It's worked continuously for me in the past. Hopefully it will continue to work so in the future. It becomes a test bed for the technology and patterns and techniques that I want to constantly add to my developer's arsenal. Think of it as my version of AdventureWorks, only at a larger scale. So, what I'd like to do now is give you a quick demo and teaser of what it is that we're going to be building together in the upcoming modules. So, I just want you to sit back and relax while my friend Carl Franklin keeps you musically entertained. (Music)

Application Demo

Software Configuration Management

Well, I hope you enjoyed that brief glimpse into the course. You might have noticed that there were quite a few projects in that solution. I assure you they all mean something and that there is absolutely no wasted code here. In the interest of folder simplicity, it's always been my choice to keep the folder structure for the Visual Studio projects flat and then use the project naming to imply any kind of hierarchy or dependency among the projects. Another practice I tend to follow is to separate out my abstractions or my interfaces for many of their implementations. This tends to facilitate unit testing dependencies since much of my classes are actually mocked from their abstractions in testing scenarios. So, let me show you some examples of what I'm talking about in Visual Studio itself. Here's the Car Rental Solution in its finished form. The first thing you'll notice is that I tend to separate everything into solution folders. This has always made it real easy for me to find the projects that I need. This is also directly related to the project naming structure. As you can see, my project naming structure is dot notation, and it follows a path similar to that of a namespace. In fact, in a lot of situations there's a one-to-one correspondence for the namespace and the folder where the class or the code item resides in. Now, you'll notice here that everything starts with CarRental. And then if it belongs in the business layer or in the business side of the wire, I tend to go with. Business on my project names. If it belongs on the client side of the wire, I go with. Client. So, a lot of it is just common sense. And then just to group those things together, I've set up a Business and a Client solution folder. Now, there's a Common folder here, and there's a Core folder. I want to point out what the difference between these two are. You'll notice that the projects in the Common folder start with CarRental. Common, yet the ones in the Core folder start with Core. Common as opposed to CarRental. The idea here is that everything that's in Core is framework level code, which means any future system other than just the car rental system can use all the projects that I put into Core. Now, I'm going to show you in a second where I store this on a folder structure, the actual folder structure itself on the drive.

But in the solution, I keep it as part of the solution here so that it's all easily manageable for me, but I group it in its own solution folder so it's easy to find. Also, notice there's a solution folder called Hosting. This is where the desktop client, the service host, and the website reside. I tend to put these kinds of apps in a hosting folder because they represent executable apps, things that you actually run as opposed to assemblies or DLLs. My Resource Access folder contains the data access layer, and then there's one for the unit tests down here. Now, the other thing I want to bring to your attention is the fact that I've separated out the contracts or interfaces, I tend to call them contacts, into separate projects. So, for example, for the data access layer, things are abstracted out into interfaces, something that you'll learn in one of the modules that I'll cover later, and I have not only a CarRental. Data for the data access layer, but a CarRental. Data. Contracts project, which contains all of the interfaces for the data access layer. The same thing goes for Common here. And then there's also, of course, on the business side of the wire, my services and my contracts are separated out. The reason that I do that is number one to keep better code separation and more of a separation of concerns, but the next reason is because during the course of unit testing my unit tests don't always need the implementation classes, and sometimes they simply require the contract classes or the interfaces. And example of that is if I'm unit testing my WCF services. In such a case, I need access to the services themselves, but I don't need access to the data layer code to actually access the database. I'm going to be mocking that. So, all I would need from that unit test is access to the highlighted project there, the CarRental. Data. Contracts. So, it just allows my unit tests to take one less dependency, or, better put, to not take a dependency on classes that it really doesn't need. Now, let me show you what this looks like in the folder structure because I stated earlier you see a Common and a Core there. Now, the common, this is all common stuff that is going to be usable by every project, but only common to the car rental application as opposed to the Core projects, which are more framework level, which can be common to any other future application. Now, you're seeing this in one solution here pretty flat using the project structure or the project naming to actually imply any kind of dependency or any kind of hierarchy and to also give you the ability to tell the projects apart. Now, if I bring up File Explorer right now, here you notice that in my source folder I have two subfolders, CarRental and Core. Now, in CarRental, that's where all of the CarRental. projects are stored. And as you can see, these are pretty much corresponding to the projects that you see in the Solution folder. Now, if I go to the Core, I keep these in a separate folder called Core. Once again, I show this to you so that you see that there is a slight difference between the storage at the folder structure level and at the solution level. In the solution, I have everything in one solution just for my own manageability. That's just so I don't have to go and manage two different solutions while I'm in the course of developing this application. So, that's how I'm storing things at the solution level. That's how I'm breaking out my projects. You can see that I try to keep things really organized here. When there are a lot of projects like there is here, it's always a good idea to keep things as organized as possible if for nothing more to maintain your own sanity during the development of this, but better yet, to maintain the sanity of every other developer you have on your team that's going to be working on this project.

SOA & Technology

What is SOA

Hi. This is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. In this module, I'll be talking about what SOA is and what makes it different from other architectures and application styles. I'll also be discussing some technologies that help you develop service oriented applications, as well as technologies that I'll be using throughout this course. So, to start with, what exactly is service orientation or service oriented architecture or application? Put simply, it's the decomposition of a system into autonomous or nearly autonomous units of responsibility and exposure. It's really a way to decompose a system, to break a system down into various parts. Now, these parts can take various forms. This decomposition can be functional or it can be volatility based. In either case, what's common in both is that services, the main component in service orientation, essentially are orchestrators or managers that expose this functionality and abstract everything beneath them from the outside world. So, service oriented applications are based on what we call loosely-coupled services. This means that between the services and the way they're exposed to the outside world are industry accepted standards and protocols. Now, behind the services are a multitude of technologies. Just about anything can sit behind a service actually. A service oriented application is essentially an API exposing functionality to the outside world through the standards and protocols that, once again, are industry accepted. A service orientation should not be thought of as a replacement to other programming paradigms meaning it's not doing SOA instead of component orientation or object orientation. In fact, it's more of an evolution of the previous ones, and the previous programming paradigms and ways of doing things can still co-exist in a service oriented application. They're just kind of abstracted away behind the service wall.

Evolution of Software Development

Now, though there are many different programming paradigms that we have read about and we've actually used for several years now, most of it I think kind of originated with procedural programming. This is back when applications were developed by calling upon functions in a certain order or taking certain steps in order to complete a given task. Now, there were several problems with this paradigm, of course. Number one, there was no reuse outside of the application and it was also very linear. There was no real analogy to the real world or to the process that it was trying to solve or the tasks that it trying to solve. This is where the next evolution came in, and this was what we call object oriented programming. This was procedural programming to the next level. Now we separated our application into parts or objects that resembled real world entities that represented the tasks that we were trying to solve. You can actually break this down into something like nouns and verbs where you've got objects which are your nouns and properties, and, of course, the verbs, which are the methods. Objects house both data and behavior of course, and they had a nice correlation to the real world counter parts or the problem, the business problem, that we were trying

to develop for or solve. Now, once again, this had a share of shortcomings, and, of course, once again, we couldn't reuse outside of an application. There was no real plumbing that allowed us to use objects from one application to the other without doing traditional copy and paste. This is where component oriented programming came in. The next step basically took object oriented to the next level and provided a common abstraction layer. We see this as COM, for example, or COM's competitors. And we could now use objects across technologies, across different platforms because of these abstraction layers that provide the coupleness between the objects. Another drawback that component oriented programming had, as well as the previous paradigms, is that you were essentially exposing everything to client. So, once you solved the problem of getting components to talk to one another or the client to be able to access components in different libraries, the client could see everything, and that might not be ideal in many, in my opinion in most, situations. So, this is what evolved into service oriented programming. Service oriented programming took the previous programming paradigms and abstracted them behind what we call a service wall. So, we still use objects, we still use components. Even inside the objects we may use some procedural style programming. We can use other paradigms like functional programming, etc., but what's exposed to the outside world is a lot more restricted and a lot more secure. Things are better managed and things are more encapsulated into what we refer to as services. Now, some drawbacks still exist, but these are more technology based. We still needed a good glue technology to handle a lot of the cumbersome plumbing that we needed to take care of inside services to make sure that they're transactional, to make sure that they're secure, all sorts of different characteristics that I'll discuss in just a few minutes. This is where the technology came into play. This is where WCF and now the Web API come into play.

Characteristics of A Service

Services can be either simple or complex, but either way they share certain characteristics. Simple services, for example, are regular data access services or services that are there for the purpose of just regular data retrieval. For example, just retrieving a product record for a given idea. In this case, the service is nothing more than a managed wrapper, an orchestration wrapper, for the data access layer. There's really no business functionality in between there that's necessary. It's just a matter of calling the service and having the service go to the database and retrieve information. Now, even in a service as simple as this, we should still have proper architectural layering meaning the actual task of retrieving information from the database itself should be delegated to a data access layer, but in this particular example the service would simply become a wrapper for that data access call and provide to that call characteristics that I'll soon talk about. Now, a more complex service example would be something like an ecommerce type of service. This is a service whose job it is to be called from the outside world for the purpose of processing an order in an ecommerce scenario. Now, the business task of processing an ecommerce order involves several steps, and each one of those steps is radically different from the other and involves different types of technology to be used. For example, you may need to take that packaged information that the client sends, which

includes shopping cart information, customer information, product pricing information, credit card billing information, and you may need to do certain things like update the data records of the product inventory to denote that product has been purchased by a certain customer and, of course, update records for the customer history and the customer purchase history. You may need to process the customer credit card information through some kind of billing gateway or payment processing gateway. You would need to build an invoice with line items and email it to the customer. You may need to perform some logging. All of these are separate steps, and they're all initiated by one call to an ecommerce service that says process this order. This is a more complex service. This is a service that is not only calling upon the data access layer to update database information, but also probably a couple of business engines to do things like building an invoice or maybe a notification engine in order to perform a notification of an email or the sending out of an email to the customer, maybe even calling to another service that does all of the payment gateway functionality, maybe a third party service like CyberSource, Authorize. net, Intuit or PayPal or any of those. So, both of these examples, a simple data retrieval or a complex ecommerce processing, are still examples of a service, and they all share certain common functionality or common characteristics. These characteristics are that either one should be simple to call from a client, either one should be the single point of entry for the client. What this means is that the client should not call into the one piece of the order processing service that simply sends out an email notification or processes a credit card. The client needs to just package up a bunch of data and send it to a service and tell the service to do its job, and then call me back when that job is done or call me back if that job fails for some reason. Now, this leads me to another one of the characteristics that all services should have, and that is it should be properly fault handled, which means that the service needs to perform what it needs to perform and account for any potential exceptions or faults as they're called in the service oriented world that may come up. It should also have a way to get those faults back to the client, so the client knows how to properly act upon them. A service, whether simple or complex, should also leave the system in a consistent state. This alludes to the topic of transactions, which means in a simple service you may be talking about just data retrieval or maybe a one-time data update or one table update, but in a complex service like my ecommerce example, we may be talking about several steps. And as far as database work is concerned, we may be talking about updating several parts of the database. Now, all of this needs to be done in the context of a single transaction, which means if anyone of these things fail, if the credit card processing fails or if the creation of a single invoice line item fails, you do not want partial information to be committed. So, you want to back out the entire thing, and then send information back to a customer that's going to tell them I can't process your order at this time. Let's try again a little later. Another characteristic that all services, simple or complex, should share is that they should be secure. Now, this doesn't mean that every single service has to have security applied to it because you may very well have a reason for writing public services that just give information to the outside world and no security is necessary. However, for the most part, I think it should be said that most services are going to have some amount of security on them. A service should also handle threading properly. So, it should allow more than one concurrent user in to perform its services or its functions, or it should allow only one at a time. That's up to the task at hand, but it should all be encapsulated and managed by the service itself. So, as you see, all of these characteristics

are that of a full-blown application, but they're narrowed down to one service, so this one service knows how to do and handle all of these characteristics, all of these things into kind of a black-box, a unit of work access-point. This is the access point that the client will have access to in order to perform that particular task, and the service knows how to handle everything regarding that unit of work in a rock-solid fashion.

A Typical Service Oriented Architecture (visual)

Now, this is an example of a typical service oriented architecture. Now, as you can see, this still has most of the building blocks of your traditional layered architecture. We still have client tiers, we have business tiers, we have data access tiers, database of course. What you're looking at though is the addition of some extra tiers, what we call the service wall or the Service Layer. The Service Layer, as you can see on the business side of the wire, completely blocks out everything underneath it meaning the business engines, the data access layer. Any other layers we put in there are blocked our and are restricted from the outside world, and everything has to pass through that Service Layer. This gives us that layer of encapsulation and that layer of security that we're looking for so that not everything within our business is exposed to the outside world. You'll also notice that there's an additional layer on the client side called the Business Client Layer. Translated to technology terms, this would be called the Service Proxies or the Proxy Layer, and this is what the user interfaces or any other client, whether it's manned or unmanned are going to be using to access the services. These are the low-level layers. These are the layers that actually do low-level communication and use those industry standards and protocols that I spoke of earlier to do the communication through the wire. Now, you'll also notice that the Service Layer, which houses of course the services, in our case that will be WCF Services, that's going down and accessing business engines in some case, but it's also accessing directly to the data access layer in other cases. I'm going to talk extensively about this later on in the next module, and you're going to see this throughout a lot of the code. This is a choice that you make when you design the system. This is the way I design my services. I don't force everything to go through a business layer unless I need to incorporate the services or the functionality that a business layer or a business engine has to offer. If all my service is going to do is access data, I have no problem going straight to a data access layer as long as I do so in an abstracted way so things are properly testable and I do so in an elegant fashion where the code can be managed and extended in the future. But the idea here is the elimination of pass through code, and I'll be talking about that extensively later.

SOA & Course Technologies

The two primary technologies that Microsoft offers for service oriented applications are WCF and now the Web API. Now, I want to stress something very important here. These two are not competing technologies. They offer two very different things. WCF is actually still the preferred technology for writing rich services. It provides a very robust model for exposing services. It provides a very rich option set. A lot of characteristics can be added to your services using

WCF to handle a lot of things for you, things like what I talked about a little earlier, transaction handling, concurrency handling, how the service is instantiated. You can add service behaviors and operation behaviors. There's a very rich set of technologies that go behind WCF that allow you to apply a lot of fine-grained characteristics of services that Web API does not. Now, that being said, WCF, though claiming interoperability, still has certain interoperability limitations primarily due to poorly tooled clients. WCF is SOAP-based, and SOAP requires a certain kind of tooling. Because of the way the SOAP message is structured, the clients need to be able to unparse a SOAP message and read into it properly, and that requires certain tooling. You don't want to do that kind of stuff manually. Trust me. So, this is where Web API has really gained a lot of traction in the fact that because of the fact that it offers REST data or data in a RESTful fashion and in a REST format, it is very interoperable. Pretty much, any client out there has the capability of reading HTTP. And if you can read or access HTTP, you can access the Web API. This is not the case with WCF. So, the two offer very different things. The Web API requires HTTP. In fact, it doesn't abstract it; it completely embraces it whereas WCF can write on other protocols like TCP, MSMQ or other protocols, and some of those are significantly faster than HTTP. However, Web API, because of the fact that it rides strictly on the HTTP protocol, is extremely interoperable, and rest assured just about every client out there, whether it's on the Microsoft platform or not, can read it. So, the two of these together really give us the arsenal that we need to write service oriented applications and offer anything that we want to the outside world whether it's inside or outside the firewall, and you're going to see examples of both of these as we write our application for the rest of this course. Now, there are other technologies that I'm going to be using throughout the course besides WCF and Web API. Other technologies I'll be using include SQL Server, of course, for our backend database. Entity Framework in a code-first capacity will be used for the data access layer and for the entities as well, of course. I'm going to be using the Managed Extensibility Framework or MEF to do my dependency injection. I'm going to be using ASP. NET MVC in combination with Knockout JS to do all the web stuff. Also, some Bootstrap in there. I'm going to be using WPF, Windows Presentation Foundation, for the desktop, and also some assorted libraries for both of these technologies. For example, the Moment library is going to do my data handling in JavaScript, the MahApps library for WPF is going to give me some nice styling in WPF so I don't have to do a lot of work to make it look pretty. I'm going to be using John Papa's ToastR library to do notification displaying on the web. My rules are going to be validated using the Fluent Validation engine. So, there'll be other libraries that I'll be using, and I'll discuss them as I get to them. But before we start writing all the code, we need to design and architect the system. So, the next module we'll go over a little bit of this, and then we're going to jump right into the code.

Architecture & Database

The Database

Hi. This is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. In this module, I'll be covering the database layout of the case study app, as well as the architecture and the service breakdown. The car rental application is going to use a total of just four tables created by me, and the rest of them are created by ASP. NET. The first table is the Account table, and this is simply going to store all user account information, or should I say customer account information. The webpages tables, the ones that start with webpages_ are used by the simple membership provider, which I'm going to explain in detail as we start using it during the creation of the website. This is going to store user password information for me. The actual user account information, like I said, is going to be stored in the Account table. Now, notice the Account table has an ID called AccountId. This obviously is the key for this table, but it's also going to become important in the next couple of tables that I'm going to show you. The Car table is pretty self explanatory. It simply stores vehicle information, and I kept it as simple as possible. So, all we've got here is the basic description, color, year, and the price per day for the rental of this car. The Reservation table stores the reservations that are made by the customer. And the Rental table, of course, stores current car rental information. Now, I want to bring your attention to the AccountId key that is in both the Reservation and the Rental table. Now, in this database, I think it's pretty obvious that this is going to join over to the Account table. Where my design differs a little from traditional design is that right now I'm only one step away from the Account table, but traditionally in a very normalized database, if you were to be two or three steps away from one table up to the Account table, you'd use key joins in order to get from one table to the next one. I do it a little differently. I actually use the AccountId column in every table that's going to contain data that pertains to a specific account no matter how many steps removed it is from the Account table. Now, this may seem like I am not normalizing like I'm supposed to, and in a way you may be right, but it has a couple of benefits. On the database side, visually when you look at a table, the minute that you see that AccountId column you immediately know without guessing that there's data in there that pertains to a specific account. You don't have to do joins to other tables. You don't have to start following the path yourself in order to see if it gets to the Account table. The other benefit is going to show itself later when we start writing code. You're going to see how I do things in the data access layer, and also how I write my services in order to handle the problem of user level authorization meaning authorizing a user only to the data that he or she should have access to. I'll talk more about this when we get to security in the next few slides.

Core use Cases

I typically start designing a new system by attempting to break it down or decompose it into areas that will require management or orchestration. Now, these would eventually translate into services, so I start by identifying who the actors of the system are going to be and what tasks these actors will need to accomplish. This system is going to have two primary actors, customers and employees. Now, as a customer I'll be able to reserve an available car for a given set of dates; I'll be able to cancel an existing reservation, of course; and I'll be able to see my rental history, what cars I've rented in the past. Now, as a car rental employee using the desktop client, I'll be able to manage the inventory of

cars; I'll be able to rent a reserved car to a customer, one that a customer has already reserve or actually perform a rental from scratch; I'll be able to, of course, accept a car when a customer returns it; and I'll be able to see some history information like view all my customers and view their rental history list. Now, obviously this is a simplified version of a car rental system. In the real world such a system would have a much larger amount of functionality and an employee's admin terminal would be able to perform a great many more functions. I tried to limit this application to the handling of just the primary functions that such a system would require, and my use cases reflect that choice; hence, me referring to them as core use cases. Now, it's the definition of these core use cases or actor tasks if you will that will allow me to decide how to break out the services that will provide entry into the middle tier of my system. Remember, I'm writing a service oriented application, which means I'm going to concentrate heavily on the middle tier here because it's going to be designed to service all sorts of different clients and UIs. Now, this is my way of designing and architecting a system and coming up with a breakdown of its use cases and a breakdown of its services. You may have your own way that works for you, and that's absolutely fine. As a matter of fact, all of the stuff that I'm going to show you in this course on how to build services and how to access them and all the pattern in place and how to tie everything together can really apply to any technique that you use to do the initial architecture of your system. So, don't feel you're locked into my way of doing it. This is just a way that has always worked for me in the past, and it's the way that I'm going to demonstrate in this course.

Service Decomposition

Based on the use cases I previously defined, I broke the system into three areas that I felt would serve as top level management for everything downstream. Now, the Inventory service would handle everything that has to do with managing vehicle inventory. Now, this can be from regular CRUD functionality used in maintaining the cars in the database to providing information about car availability. In the future, this service could also handle things like taking cars offline for mechanical upkeep or putting older cars up for sale, etc. The Rental service will handle anything dealing with rentals or reservations. This includes not only the ability to reserve and rent cars, but also to cancel reservations and even remove dead reservations from the system for cars that were reserved, but were never picked up for rental. This service could also provide a user with information on currently rented or currently reserved cars, as well as history of their past rentals. The Account service will deal with everything having to do with customer user accounts. This includes providing employees with customer information or even allowing the customer themselves to manage the information. The only thing this service will not handle regarding accounts is going to be registration and log in since this is going to be handled by the website's membership subsystem. All three services will serve as the gatekeepers of the middle tier, the front line of the API. This means that everything behind them is going to be abstracted from all the clients. The services can either access data directly through one of the data repositories that you'll soon learn about, or if behavior reusability becomes necessary in some cases, a service can call upon a business engine, which would house that behavior. All three of these services are going to be fully testable and are all going to

take advantage of dependency injection that's going to provide them with any additional classes on which they may depend.

Security

The WCF Services will be secured using conventional internet security. This means that standard Windows authentication will be used. Remember, the services sit behind a firewall, as does the IIS server that's going to be hosting the website. The desktop application being internal to the car rental company also sits behind the firewall. So, I can actually use a dedicated user account for the website and then a dedicated admin role for the users of the desktop to keep things simple yet still secure. Only users of the car rental company authorized to use the desktop client would be made members of this role. Later when I develop the services, I'll show you how this security design manifests itself into the code. Customers using the website will be required to create a user account, so good old fashion ASP. NET forms authentication will be used here. A forms authentication has been wrapped by different patterns since the introduction of ASP. NET, and, in fact, new ones are being introduced while I develop this course. I've decided to use the simple membership provider simply because it works, and it provides what I need without unneeded complexity. Forms authentication will also provide the security that I need to secure any Web API services later. Now, so far the security that I've described protects my services from outsiders, but there's still one additional level of security that I'm going to be needing here. I'll need to also secure the data from user to user. This means that just because customer A is an authenticated user into the website and can thus perform functions that end up accessing my WCF Services, they shouldn't be able to access data that pertains to customer B. Now, I'll need to be able to take care of this problem in a manner that's easy and reusable. I started this process earlier when I showed you the database layout. I pointed out tables that had data in it that pertained to specific user accounts and how they contained a column called AccountId. I also mentioned that no matter how many times removed those tables would be from the actual account table, I would still have provided them with an account column. You'll see the code manifestation of this design in the next three modules. When I develop my entities, my data access layer, and then my services, you'll see the pattern that I came up with for handling this problem in an easy and what I think is a very elegant fashion. The pattern and the techniques that I'll show you later will involve any WCF or Web API service that need to know about the current user trying to access information from it. Now, this may sound simple, but it's not as easy as you might think because if you've done any kind of WCF before using internet security, you know that the identity that gets propagated down to the service during a service call, what's know as the primary identity, is that of the caller. And in the case of a website, the caller is not the actual browser user. Remember, it's the IIS server itself. So, what I need to do is that I need to get the actual user of the browser down to the service, but I need to do it in a way that does not impede the developer's productivity. I'll be talking significantly more about this over the next three modules.

The Architecture: Walkthrough

This is the architecture for the car rental system that I'm going to be building throughout this course. Now, as you can see, it's got quite a few components here and quite a number of layers, and it differs slightly from the typical SOA architecture that I showed you in the previous module if you recall. Now, let's go through this from bottom to top. So, at the bottom we have the SQL Server Database of course. I'm going to be using SQL Server 2012 for this, but you can use any version that you want. The next layer up is going to be the Data Access Layer, which you can see will consist of several components, the main one being Data Repositories. I am going to be using a repository pattern here, but using the help of what I also call Data Repository Factories I add my little twist to this, which is going to help me greatly in testability. Now, the Data Access Layer communicates up to either Business Engines or directly to WCF Services depending on the call, depending on the need. And, of course, this WCF Services will have service contracts. Now, conventionally you're taught that when you write WCF Services you always have service contracts, which we will of course, but you also have data contracts. Well, I have Data Contracts here, but I don't use exclusive data contracts only to the WCF Service layer, nor do I use data access entities down at the Data Repositories in the Data Access Layer that are exclusive to the Data Access Layer. This is where my architecture tends to differ from others that you might have seen along the way, and that is the fact that I have this concept of entities on the Server and on the Client-side. And as you can see by the diagram, the Server-side Entities are higher in their box size. This means that these entities have the capability of traveling up and down the side of the wire to which they belong. So, in the case of the Server-side Entities that you see in the lower right, they're be used from the Data layer up to the WCF layer and with everything in between. This means that my Server-side Entities will be used by this Data Access Layer to read in data, and they will be passed to either a Business Engine if the need is there or directly to a WCF Service, and the WCF Service will then use it as a data contract to transport the information across the wire up to the Client-side. Now, this doesn't lock me in to the fact that I'm only limited to what the Server-side Entities can carry because you've got to do what you've got to do to get the job done, and there may be cases where I need to send up different information through a service call or perhaps information that is the combination of more than one entity. When that need comes, I'll simply write my own data contract for it. There's nothing that says that I can't do that. The same thing applies when I have the need to receive data from the Data Access Layer to either a Business Engine or a service. If the call to the Data Access Layer requires for more than one entity to be received or perhaps data in a slightly different shape than that provided by an entity, I'll simply write a custom data transfer object that'll be built by the link query in the data access call. Now, you'll see that right at the service wall line there's my WCF Service Host of course. This can take many, many shapes, and we'll get there during the Service Host module. And then we jump to the Client-side of the wire. And the Client-side of the wire, that same concept of entities that travel up and down the layers exists. So, on the Server-side of the wire, we had the Server-side Entities that go up and down the Server-side. On the Client-side, the call, the WCF call, is received, and remember that call was made by the proxy, and the data is received into Client-side Entities. Or if it was a custom data contract that packaged up the data and sent it through

the service call, then the proxy would receive its client version of that Data Contract. But if it was just a Server-side Entity, there will be a Client-side Entity. Now, these entities differ from one another, and what they each can or cannot do is going to be slightly different, and we're going to get to that in the next module when we cover entities. But the Client-side of the wire also has its own set of Service Contracts. It's also got some WCF Proxies. It's also got this concept of Service Factories, which I'll get to much later in the course, and this is going to help me solve a couple of other problems that are going to come up. You'll see that we have multiple UIs as I promised. We've got the MVC Client, we've got some Knockout there, we've got a WPF Desktop Client. There is a Web API set of services in between there. They're going to be used for a couple of different things. There are a lot of Core Framework classes. And then, of course, there's the manage extensibility framework, which is going to help me with my dependency injection. Now, I could've used any dependency injection container that I wanted. There may be some of you out there that are already cringing and saying oh God why is he using MEF? MEF works for me. It does what I need to do in the area of dependency injection. It comes with the. NET Framework. It's easy to use. I just didn't see a need to bring in a third party, but that doesn't mean that I'm against third party dependency injection containers. In fact, there are all sorts of great ones out there. So, you can use any DI container that you want, of course, in your own application. I just made the decision to use MEF and keep it simple.

The Architecture: Reasons & Goals

I want to explain to you why I do things so differently than what you might have seen in the past. In fact, I do things differently from the way I used to do it in the past, but I think that can be said for just about any developer out there. Traditionally when you look at service oriented architectures or any other architecture, there's a real strict separation of concerns and separation of layers. Now, I still have that separation of layers here. I still have a Business Layer. I have my services in a Service Layer. I have a Data Access Layer. So, I still separate things out they way they should be. Where I differ is in the fact that I'm not as strict of having one layer call only to the layer immediately under it. You saw in the examples I showed you in the previous clip that my services can call to the Data Access Layer, as well as the Business Engine layer depending on what the task at hand is. The reason for this kind of design and the reason that I have these entities that span all the layers on their side of the wire is to avoid pass-through code. For a long time I wrote layers in a very, very strict way, which means I had data contracts only for the services, I had business objects that were only used by the business engines, and I had data access entities that were used only by the Data Access Layer. Then when a service receives a call, what it does is that it makes a call to the Business Engine. The Business Engine in turn makes a call to the Data Access Layer. The Data Access Layer sends a data entity back up to the Business Engine. The Business Engine would then map to a business object. If the mapping was with identical properties, I would use a tool like AutoMapper or some other mapper of course, and then I would pass the object up to the Service Layer. The job of the Service Layer would then be to take that object and once again perform a mapping to a data contract, which then goes up the wire to answer the service call. This would be the case whether

I'm making a service call that is quite complex and possibly requires the job of a business engine, especially for behavior that may need to be reusable, or something as simple as just making a service call to go out and get some data, which a lot of times becomes the majority of your application. What happens there is that I'm writing a lot of what's known as pass-through code. If I have a job for the business engines to do, I will most certainly have one. But if the job of the business engine is nothing more than to serve as an intermediary between the Service Layer and the Data Access Layer and do nothing but pass the object along, then I really don't need it, and I don't want to write that code anymore. There are many out there that are going to disagree with this design. That's perfectly fine. This is just the way that I do things, and it's worked out very well for me. So, by having the Server-side Entities, I can have a reusable class that does not need to get mapped over to another set of classes that look virtually identical. I start with a set of classes down at the Data Layer, and I use those all the way up to the Service Layer. Now, I'm going to have exceptions to that, and I mentioned that in the previous clip, and I'll deal with those exceptions as they come up. But for the most part, I'm not going to need to. For the most part, my Server-side Entities are going to serve me well when they get passed into the business engine, if they get passed into a business engine at all, and just as well when they get passed up to the service. And then the service will treat them as a data contract and send them up the wire to the client. The same thing can be said for the client-side. When I receive something from a call, from a proxy, I don't need to map that over to some kind of model class that's going to be used by the Web API or by ASP. NET. I can go ahead and use that Client-side Entity. In fact, you're going to see in the next module when we design the Client-side Entities, they're actually going to be significantly intelligent objects, and they're going to have quite a lot of functionality. So, that's the reason that I do things the way that I do. If you're not completely convinced, I hope to sell you on it as you see me write the application over the next few modules. So, to recap, we have a flatter model, which means we don't have a lot of object hierarchies here. There's just no need for it. And we have a strict line of separation between the client side and the business side. This means that clients will only communicate with the services. I've stated this a couple of different times already. The Service Layer is my API that is exposed to the clients. Clients have no knowledge of anything below. There is lack of pass-through. This is one of the key things I want to point out because this is one of the key things that differ my architecture from others that you might have seen or that may be more mainstream. There are no unnecessary objects that are there just for data passing purposes. Everything has a purpose, yet I still maintain code cleanliness and good architectural practices. We have multi-use entities on both side of the wire. We're not taking it to an extreme. I don't have one side of entities that crosses the wire from top to bottom. I have two sets, one on the server, one on the client. All the code that I'm going to write is going to be very, very testable, so I have my patterns of abstraction already planned out, and I'm going to show you where I'm going to put them in place. And we're going to use DI throughout the whole system. Both the business and the client side are going to benefit from dependency injection using MEF. Now, every developer is constantly adjusting the way they do things both architecturally or from a coding standpoint. This can be due to a combination of either learning new things and new technologies, new techniques, and new patterns combined with finding faults or shortcomings of previous ways of doing things. And my way of architecting, and later you'll see developing an application, takes no

exception to this. So, I really need to give a shout out to my good friend and fellow Pluralsight author Brian Noyes. Two years of working together and bouncing ideas off each other on a daily basis and performing what we used to call sanity checks have contributed quite a bit to a lot of the elements that I've shown you in the architecture. Concepts like multi-use entities and lack of pass-through code and pass-through layers, these are all things that Brian and I discussed heavily over a period of two years. Brian does have several Pluralsight courses out there, and I strongly urge you to check them out. So, in the next module, we're going to start coding. We're going to stay in Visual Studio as often as we can, and I'm going to start by showing you what these entities are, the Client and the Business-side Entities, how they differ, and we'll code them from scratch.

Entities & Core

Entity Design & Structure

Hi. This is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. In this module, I'll cover the design and development of the now famous entities that you've been hearing so much about. You'll see how the business and client entities differ and will become intimate with the capabilities and functions of each. You'll also learn about some of the core functionality and classes that I'll be using throughout the system. You recall from the architecture description in the last module that there are two sets of entities in this design, one set on the business side and one on the client side of the wire. These two sets do not cross over, but each is a counter part for the other. Each set of entities can travel up and down the layers on its side of the wire avoiding the need to have multiple sets of classes and then use a mapper to copy data between them. In most cases, a business entity will be serialized by a service and sent across the wire where it will be received and deserialized into a counterpart client entity by a service proxy. The same process will apply when going from the proxy to the service. Now, I say most cases because remember, if I have a situation where the data I need to pass across does not fit a particular entity or consists of more than one combined sets of entities, I'll simply supply a custom data contract to both sides of the wire. Each set of entities will contain different code and have its own set of functionality. In fact, the only thing that these entities have in common is their equivalency. This means that the naming of the counterpart entities and their properties must match for the purposes of serialization and deserialization. Now, since the data contract serializer will be the one that WCF uses by default, the namespace for the entities must also match, but I'm going to show you a cool trick for this one. So, I'll start by writing two empty entities, one on each side of the wire, and we'll continue building from there throughout the rest of the module.

Initial Setup for Business and Client Entities

So, here we are in Visual Studio 2012, and I'm going to start by showing you how the entities are laid out, how the projects work, how the namespaces work, and then we're going to get into the inner workings of them as we progress. As promised here, I have two projects, one to store by business entities, and one to store my client entities. Now, the first thing I want to show you what the references or dependencies for these projects are, and they're actually quite simple. The business entities is just a regular project, regular class library. It doesn't have anything that's additional except some of my common framework stuff, which I'll go over later on. And the client entities are the same with one exception. The exception is one NuGet package that I added to the client entities that I will get into shortly, and that's the FluentValidation NuGet package in order to perform validation rules on my client entities. Now, I'm going to start with a simple entity, the car entity. This entity is going to later correspond to the car table in the database. Remember, I have a very simple database, and I'm going to have entities that pretty much match my database. Now, that doesn't always have to be the case. In fact, Entity Framework has some strong ORM functionality that let's you do object relational mapping in case your entity model does not match the database. So, there's no rule that says it has to. But since I'm designing this system from scratch and I've kept things very simple, it happens to be that my entities will match the database. So, I'm going to start with a car entity, which corresponds to the car table in the database, and I'm going to write one on the business side and one on the client side. So, the first thing I'll do is that I'll create the class or the Car, and I'm going to do the same thing for the client project. Now, these classes look the same right now. The first thing I'm going to do is obviously make them public, and I want to point your attention to the namespace first of all, CarRental. Business. Entities, and this one is in CarRental. Client. Entities. Now, this is going to be important because remember these entities are used for serialization and deserialization by WCF Services and Proxies. And for that to work properly, the namespaces have to match. Now, ordinarily you would do this in the data contract attribute with a namespace property, but I'm going to do it a little differently. Now, both of these entities are going to be data contracts. Now, the business entities are going to be a lot lighter weight than the client ones. You'll understand what that means as I progress through the code.

Entities as Data Contracts

Now, they're both going to be data contracts, but the business entity I'm going to be decorating with the DataContract attribute. Now, the DataContract attribute comes from the System. Runtime. Serialization assembly, which I've already added, and all I need to do is decorate my entity with DataContract. Now, the properties inside of them are going to be decorated with DataMember. The rules of the data contract serializer state that if a class is decorated with DataContract, then any member that you want to opt in for serialization must also be decorated with the corresponding DataMember attribute. Now, as of a later version of WCF, or should I say a later version of the data contract serializer, they changed those rules where you can now leave off the DataContract attribute and automatically all public members of the class will be considered DataMembers. This rule has good or bad points. Bad because I tend to like explicitness meaning I like to see what a class is up to, which means by looking at this class and

seeing that it's decorated properly, I know what it's going to be used for. But at the same time, if I have a class where the code starts to get a lot heavier and the class does a lot more things, I tend to find that these attributes pollute the class a little bit visually. So, I'm only going to use the attributes on the business entity. I'm not going to use them on the client-side entity. Now, I'll write the properties of the client-side entity in a few minutes. I want to stay on the business one to talk a little further about it. First of all, let me add the other properties to our business Car entity, and now let me add one property to the Car client entity. Now, the first thing you'll notice is that the entity properties are different. I'm using shortcut notation in the business side, but I'm using long notation on the client side. There's a reason for this, and I'll explain that next.

Data Contract Equivalency

The reason why my client and business entities use a different syntax for their properties is because the business-side entity is going to be a lot simpler. This is all it's going to do. It's going to carry data along the layers of the business side of the wire all the way from starting in the data access layer up to the service layer and anything in between. It needs no additional functionality except to serve as a carrier of data or a data transfer object if you will, whereas the client-side entity will have more functionality that I'll be adding throughout the rest of this module. It may be possibly used in binding scenarios on the web or binding scenarios on the XAML side in which case it would require some kind of property change notification. It will also require validation rules in some cases because this is the entity that the client user interface is going to be interacting with. Now, remember, these two entities must be equivalent meaning their naming must match. So, in both cases I have an entity called Car, and in both cases I have properties that match. Actually the client side of entity only has one property right now, so let me go ahead and add the rest of the properties. I bet you've never seen anybody type that fast, huh? I'll be adding more functionality to the getters and setters of the client-side entity shortly. First I need to talk about namespaces. I've stated a couple of times already that for the data contract serializer to do what it does, the namespaces that these entities serialize into and then serialize out of must match. Now, by default, what's going to happen is that it's going to take the namespaces that I use in. NET, in this case CarRental. Client. Entities and CarRental. Business. Entities, and obviously this will not work. These entities do not have matching namespaces, and we're going to get a mismatch error. So, what's normally done, especially when you're writing WCF Services, one of the more common practices that you've seen would be this. Here I've given the Car DataContract a namespace called Pluralsight. com/carrental. Traditionally these namespaces take a URI format, and all you've got to ensure is that it's a unique URL. Traditionally what you would see somebody do is then go to the other entity and make it equivalent by also adding a Namespace property to the DataContract attribute. Herein lies my problem. My client entities are not decorated, which means it's going to use the automatic DataContract feature of the data contract serializer meaning that if you leave out the DataContract attribute, all public members are automatically considered DataMembers. The reason I chose to go with inferred data contracts on the client entities is to not visually pollute the class. I'm going to be using this class for a lot more things, I'm going

to be adding more functionality, the class is going to get larger and larger, and I thought that having the DataContract attribute on them would just be a little more visual pollution for that class. Now, there's really nothing wrong with having the DataContract attribute on it. It's just a choice that I made. However, in making that choice, I need to find another solution for mapping the namespaces so that they match because as of right now we have a business entity that has a Namespace of Pluralsight. com/carrental and a client entity that has a Namespace of CarRental. Client. Entities. And if I give this class a DataContract attribute for the sole purpose of mapping its namespace so that it matches the business entity, then I have to give all of the members DataMember attributes, and that's what I want to avoid. So, there's another trick that you can use for data namespace mapping, and this involves adding an entry in the AssemblyInfo file of each project. So, the first thing I'm going to do is remove that DataContract Namespace that I added and make this class just a little cleaner. And now I'm going to open up the AssemblyInfo file of my business project here and add an entry at the bottom. As you can see, this is an attribute called Contract Namespace that will allow me to map a CLR namespace to a Datacontract namespace that'll be used in the SOAP message. Now, the trick is to now add the same entry to the AssemblyInfo file of the client entities project, and that will make it so that any time that the data contract serializer runs across a certain CLR namespace during serialization or deserialization, it knows to use the URI namespace that I defined here. And now our namespaces match. Now we have DataContract equivalency on the business side and on the client side.

Business Entity Base Class & Identification Interface

For my next step, I want to focus my attention on the business entity for right now. I've always liked the idea of having a common base class that classes that may share some characteristics either now or in the future can inherit from. This is the case with business entities. What I have here is I have a project called Core. Common and one called Core. Common. Contracts. This is where I store common classes, framework level classes that I'm going to use throughout the entire application. Now, these projects will grow as we progress through the modules, but right now they're just at their minimum form just to accommodate my entities. So, the class that I want to bring your attention to is a class called EntityBase. This is what I use to serve as the base class for all of my business site entities, and this is where I would put common characteristics and common functionality that these entities may share in common. Right now they really don't share anything in common except for the fact that I want them all to inherit from the IExtensibleDataObject interface. This is a data contract serializer interface that allows entities to become version tolerant meaning if they receive data that they cannot accommodate, they would not just blow up and break contract. They would just gracefully accept that data silently, store it temporarily in this property called ExtensionData in case it needs to be passed over somewhere else later. Perhaps in the future I'll come up with other characteristics or functionality that my entities would store in this entity base class, but for right now, I'm just going to make my business entity inherit from EntityBase. The next thing I want to do is let my Car entity implement another interface that comes from my core project. This interface is called IIdentifiableEntity, and it's going to be necessary

when we get into the data access layer in the next module. The job of this interface is to let the data access layer know which one of the properties in this entity serves as the identifier or the ID in the database. Of course this can be done using Entity Framework's key attribute or setting up something in the DB context, which you'll see in the next module, but I don't want the data access layer repositories and the patterns that I've set in place that I'll show you in the next module to have to use reflexion to figure out which is the ID. Now, as you can see, this interface forces you to implement just one property, and all I've got to do here is return which one of the properties is going to serve as the identifier. As I said, when we get into the data access layer in the next module, this interface will come up again, and you'll see where I actually use it.

Business Entity Account Ownership

There's another interface that I'm concerned with on the business side of the wire for my business entity, but this interface is not going to apply to the Car entity because the Car entity does not contain data that pertains to a particular user account. Now, this may sound a little familiar to you. If you recall, when I talked about the database, I mentioned that every table that contains data that pertains to a particular user account would contain an account ID property. Well, this kind of design is now going to manifest itself into code because what I want to do is that I want to create my other entities, and those entities that contain data that applies to a specific user I want to decorate with another interface. So, the first thing I need to do is build my other entities, the Rental entity, the Reservation entity, and the Account entity. I'm only going to do the Rental entity for now because the Reservation and the Account entity will follow the same pattern. So, watch this. I'm going to do this through the magic of Camtasia. Ready. Presto. Here's the Rental entity on the business side. As you can see, it inherits from EntityBase; it has IIdentifiableEntity, which exposes the RentalId property; and now I'm going to give it one more interface. Now, this interface is called IAccountOwnedEntity. And not to implement the interface. Now, here's the implementation for the IAccountOwnedEntity interface. It has one property called OwnerAccountId. And what it's looking for, it's looking for the property within this entity that specifies the ID of the user account. Now, if you look at the properties of the Rental entity, we have RentalId of course, we have CarId, DateRented, DateDue, DateReturned. We also have the AccountId. Just like its matching table, this tells you to which account this rental would apply to. So, this is the property that we want to return here. Now, this is all you're going to see on this for right now, and you're probably not going to see anything about it in the data access layer, but when we get into writing the services and you see how I handle security and how I handle user-based authorization for specific data, that's where this interface and this property will show itself again. I can quickly add the other two entities that I'm going to need, the Reservation entity and the Account entity, and that should finish up our business entity project. And there we go. There's the Reservation entity. As you can see, it follows the exact same pattern. And here's the Account entity, which follows the same pattern. Now, this is the Account entity that corresponds to the account information in the account table, so it doesn't need an I account own entity implementation, just the identifiable entity, which you'll see in the next module

in the data access layer. So, in summary for the business entities, we know that we can use them across all the layers on the business side of the wire, the data layer, the business engine layer, and all the way up to the service layer. The entities map to a database table. They can be used as data contracts on the business side, though as I mentioned earlier, they're not the exclusive data contracts. I may very well have custom data contracts wherever I think I need them. The data access layer will provide ORM mapping information if it's necessary, and this will be done in the data access layer in the DB-Context class as you'll see in the next module. We have a common base class for all our business entities. It's called EntityBase, and it provides any common characteristics that we need now or in the future. And we have a couple of interfaces that the business entities might use. One of them they all use, and that's the IdentifiableEntity interface, and one is used by those entities that carry data that pertains to a specific user account, and that's the IAccountOwnedEntity. Now, let's shift our attention over to the client entities and continue building them.

Property Change Notification & Dirty Tracking: Intro

I mentioned earlier that the client-side entities will be heavier than the business entities and carry more functionality. This is because the client is going to be using these for possibly binding. It may need to do things like validation or dirty tracking or more fancy things that are not necessary on the unattended side meaning the business side. So, the first piece that I want to show you is regarding property change notification, which is something that XAML binding uses heavily, and our client entities are going to be intelligent enough to provide. The other piece of functionality I also want to show you is dirty tracking, so I'm going to show you both of these together, and then we'll get to coding them. So, let me give you a quick overview of what property change notification and dirty tracking will look like once we get into the code. First of all, we're going to be doing property change notification with compile-time safety. This, as opposed to most of the property change handling that you've seen in the past where you raise the property changed event using the property name as a string, I'm going to show you a way that you can do it using a lambda expression so that you get complile-time safety. I'm also going to show you a cool little trick that may or may not come in handy for this application, but it has for me many times in the past, so I think that it's worth showing it to you. It's a trick on how to do events where you disallow duplicate event wiring. You might have had situations in the past where you're manually wiring an event in code and you're doing it in a loop or in a piece of code that may get called more than once. To avoid duplicate event wiring, the norm is to usually do a -= to unwire an event and then a +=. And if the event was not wired to begin with, then the -= will just fall through and not hurt anything. However, if you don't do that, then if you already have an event wired to a specific subscriber and you run that same subscriber through another +=, it's going to do yet another wiring. So, when that event fires, it's going to fire to that subscribed however many times you wired it. So, I'm going to show you a trick to actually avoid this. Then I'm going to build dirty-setting right into property change because we already have a facility that's going to get called any time a property gets set on a client entity. What better facility to use to track dirtiness for that particular object. So, like I said, dirty tracking is

going to be built into the property change notification, and it's going to have its own flag. And we're also going to have a way to obtain the dirty objects from any level in an object graph all the way down meaning if a client object has child objects, we should be able to ask for all dirty objects in a return collection from any level in the object graph and down. We should also not only provide a way of returning a collection of dirty objects, but also checking to see if an object at any level of its graph is dirty. And this will involve short-circuiting because you don't want to continue collecting dirty objects if all you're interested in knowing is any object from this point down dirty. I'm going to accomplish this using a really cool routine that shows you how to walk an object graph. But what's cool about this routine is not the fact that it walks an object graph because many of you listening to this course have probably written code like this in the past, I'm going to show you a very cool lambda technique that allows reusability in a way that I'm willing to bet a lot of you have not seen. So, property change notification, as well as dirty tracking will all be handled by a base class like the EntityBase only this is a base class that the client entities are going to be using, and that's what I'm going to show you next in Visual Studio.

Setting up The Client Entity Base Class

You recall earlier I showed you this class, the EntityBase class. It's used as the base class for all of my business entities. Now I'm going to show you a class that I'm going to use as the base class for all of the client entities. Now I have this class written already, but I want to write a little bit of it from scratch in front of you because I want to describe a couple of things, and then later I'm going to show you the finished version of the class. So, let me create a new code file right now that I'll use for a temporary object base class that I'm going to write from scratch. So, I called this class TempObjectBase because it's just that, it's going to be temporary, and I'm going to remove it later. Now, let me jump to one of the client entities that we created earlier, the Car entity for example. If you recall, we use properties in their long form, which means we have backing storage as a member variable. The reason for that is because there's going to be code that I need to insert in the getter or the setter, or more specifically in the setter of this property. Now, what I want to do here is I that want to execute what's called property change notification, which I described in the previous clip. This means that when a UI, particularly a XAML UI, binds to this class, it wants to wire itself to an event called property changed. That way when the property does truly change, the view receives notification of this and rebinds or refreshes its bindings. This is textbook MVVM. Now, remember I'm not only creating a WPF application later in the course, I'm creating an MVC application too. However, I want to make my client-side entities intelligent enough to accommodate any UI. And any UI means I have to account for the fact that somebody may write a XAML-based application whether it's WPF, Silverlight or a modern app for Windows 8 Store, and they may write it in XAML, which means I have to accommodate the ability for these entities to do a change notification. Now, I can very easily just create the event here. Now, this event actually comes from an interface called INotifyPropertyChanged, which is in System. ComponentModel namespace, but I want to do this in all of my entities, so I'm obviously going to do it in a base class, and that's what this base class is for here. Now, ordinarily we would do something like this. (Typing) And

now that I've implemented the interface, I can very simply just put a kind of helper method around it. (Typing) This is easy enough, and it'll work. So, now if I go back to my Car entity, I can simply make it inherit from this base class. Now, in the setter of all my properties, I can do something like this. Now because property changed, the call to property changed, actually raises an event, this carries a certain amount of weight, of course, as any kind of event raising does, so I want to do it only if the property has truly changed, which means I'll put a quick check around my value set. (Typing) And there we go, a more complete property statement. In fact, I can see some white space here. And, of course, I need to repeat this on every property, which I'll do later on.

Adding Dirty-Tracking Functionality

Now, I told you earlier that I was going to merge dirty tracking and property changed in one feature, or at least make them kind of rely on each other because I already have a facility here for calling something any time a property changes. So, what better place to put the ability to track the fact that this object is dirty. So, one thing that I can do is go to my base class, and then I can do something as simple as this. (Typing) Here I just wrote a simple property that has a Boolean called IsDirty. Now what I can do is something like this. I can provide another overload now to the OnPropertyChanged method that will accept a second argument allowing me to tell the method whether or not to change the dirty state of the object. Something like this. (Typing) Now, take a look at what I did here. I moved the rasing of the event over to the new overload, and then I made the original one call this overload and default the makeDirty argument to true, which means now back to the Car entity when I call OnPropertyChanged or CarId, it will automatically make this object dirty. If I wanted a property to take exception to that, I can then send a false as that second argument. So, now we have the simple facility to track dirty state in an object. Now, later on I'll show you how I write methods that allow me to get all the objects within an object graph, or should I say from this object and down the object graph that happen to be dirty, or to check the dirty state of an object graph.

Event Declaration to Avoid Duplicate Subscribers

Well this event works fine as is, and in fact I can leave it like this for the purposes of my application. However, I want to take it to another level just to give you some extra stuff, which I'll be doing throughout the entire course. I promised you earlier that I would show you an event trick that I have that prevents subscribers from doubly subscribing to the same event. Let me show you what I mean by that. If I scroll down here, I'm going to create just a sample test class real quick. And in this test class, I am going to do the following: I'm going to make this test class inherit from my TestObjectBase, or should I say TempObjectBase so it has notification capability. Now I'm going to write a TestClient class. And in this class I'm going to instantiate my test class. Let's assume this is a unit test or something, and I'm going to wire into its PropertyChanged notification. There it is right there with a sample method. So, now I can do whatever I want in this method. So, I performed this code in the constructor, which doesn't make it very realistic, but let's

assume for argument sake that you have this kind of event wiring in a method somewhere, and that method is part of a routine that may get called more than once maybe out of your control. Maybe it's called automatically or who knows depending on your application of course. But what happens is that if this object doesn't get refreshed, if it's the same object that's still alive when this wiring occurs again, you are going to get double wiring and triple and quadruple. When the object raises the PropertyChanged event, however many times this single subscriber wired to the event, that's how many times the event is going to get fired, and that usually is undesired functionality. So, a very common thing that a lot of developers do, I'm sure you've seen this in the past, is they do something like this in whatever method they feel may get called more than once. And what this does is it ensures that the event gets unwired first and then wired again. And if it wasn't wired to begin with, the -= doesn't hurt anything. But I'm going to show a cleaner way of doing it so you don't have to remember to do this every time. So, what I want to do is that I want to change my event. Right here we have a public event, and I'm going to change to a private event. That's going to be my first step. So, notice I changed it to a private event, and I also changed the name of the event so that it follows more member variable notation, which means I started it with an underscore. And now what I need to do is expose this member variable as an actual public event, and I'm going to do that with an event long form syntax. And we use the delegate of this event, which is PropertyChangedEventHandler, and here's the name of the event. Now, just like a regular property has a getter and a setter, a long form event declaration has an add and a remove. So, let me add these. Ordinarily all I would have to do here is subscribe the value. There is a value argument just like there is in a property statement. I need to subscribe or unsubscribe the value to my private event. So, now what you're looking at here is just about the exact same thing as what I had before with one simple public event statement, so I really haven't added any value. The only value that I've started to add is the same thing as going from short form property notation to long form where once you have a getter and a setter you now have the capacity to put more functionality in your property. Now I can do the same thing for this event. Before I do that, let me fix the compile errors here because now what we actually need to raise and check for null is the private event, not the public exposure of it. There we go. And now for the little trick. And what the little trick does is very simply keep track of the subscribers in a collection and check to see if that collection contains a subscriber that comes into the value argument. So, the first thing I need to do is add a list to keep track of these subscribers. So this list is going to be called PropertyChangedSubscribers. Pretty original name, huh? And now I'm going to alter the add and remove as follows. (Typing) And that's it. So, it's definitely a lot more code, but I've gotten the hang of writing my events like this, and I don't mind the extra code because of the functionality it gives me. Now, when somebody does a += like down here, that += ends up hitting this argument right here, this accessor right here, the add accessor of the PropertyChanged event. And, of course, I check to see if that value and value comes from this guy right here. So, it's for that specific subscriber. And I check to see if that value is in the list. And if it's not in the list, I go ahead and wire the event, and I add the subscriber to the collection. Also, if I go to remove it with a -=, I remove it from the collection. However, if I go to re-add it without removing it, I check to see if it's in the collection. And when I find that specific subscriber is in there already, I don't perform this again. So, even though it's a little more code, it's a nice little clean

trick to ensure that if somebody does a statement like this more than once on an existing object without reinitializing that object, that event will not get wired more than once. So, next I'm going to show you how I eliminate the need to use the propertyName as a string and give you compile-time safety calling OnPropertyChanged.

Compile-Time Safe Property Change Notification

This next trick is an easy one. As a matter of fact, I can claim it to be an original design by me. I've seen this in several places. Particularly, I've seen it in the PRISM framework. Incidentally, Brian Noyes just put out a PRISM Pluralsight course out there for the new version of PRISM, so I strongly urge you to check it out if you happen to be writing Windows 8 apps or even WPF and Silverlight apps as they can both benefit from PRISM. So, what I want to do is that I want the object that inherits from my base class, from my ObjectBase class here or my TempObjectBase in this particular example, to be able to call OnPropertyChanged with something that offers compile-time safety, and a string is not going to. A string is just subject to too much errors from the user. I can misspell it and screw everything up. I can case it improperly. So, what I want to do is that I want to create yet another overload for OnPropertyChanged. Now, this one is not going to receive a string as an argument. It's going to receive an expression. And this type comes from System. Linq. Expressions, so I'll resolve that. And now what I want to do is I now want to take that expression, which is going to be in a lambda form when we see it in use in a few minutes, and I want to extract the propertyName as the string. To extract the propertyName, I need to use some reflexion, and I already have a helper method set up in my Utils folder called ExtractPropertyName. And now that I have the propertyName as a string, I can just leverage the other overload of my method. So, what this enables me to do is the following: I can go back into my entity here, and instead of calling the OnPropertyChanged this way, I can call it this way. So, now if we mess it up, we'll get compile-time safety. Watch this. I can do something as simple as misspell or misscase it, and you'll see that we can't compile now. So, next I'm going to show you how to write some methods that will help us collect some dirty objects.

Object-Graph Walking and Dirty States

So, now I'm going to show you another cool technique that I think you're really going to enjoy. This one has actually helped me quite a bit in the past in many, many different scenarios. So, let's talk about the task at hand. The task at hand is going to any object in my object graph, any object in my system that inherits from ObjectBase, or once again for our example here, TempObjectBase and saying give me all the dirty objects from this point and down. Remember, the IsDirty property is only for one particular object, but you may have a hierarchy going down, you know, 2 levels, 3, 10 levels deep of objects that may or may not be dirty and you want to obtain a collection of all of them. Now, obviously you've got to start at one point and then walk the object graph down. So, the first thing I'm going to do is I'm going to write a method skeleton should I say to get my dirtyObjects. So, now my task here gets a little more

complicated because now I've got to actually write code that's going to perform a recursive walk down the object graph. Now, I'm going to show you a couple of different things here. The first one is a cool way of using lambda expressions to avoid having to use more than one method. I think we've all written code that does recursive routines or recursive walks, and usually you have an entry method that then leverages a secondary method, and that secondary method leverages itself until the recursion is complete and things come back up to the original method that you called and you return some kind of end result. Well, thanks to lambda expressions, we can do this all in one method like this. I'm taking an object, passing it into this lambda expression, and starting a process whereas using reflexion I obtain all of the properties of that object. I have a helper method set up here called GetBrowsableProperties, and I'll explain it in a second. And I obtain a list of all the properties of the object, start iterating through them. If the object is a derivative of TempObjectBase, then I can call upon my lambda variable, which is walk as you can see here, and this is where the recursion takes place. In the walk statement itself, I'm checking to see if that object is dirty, and if it is, I'm simply adding it to a collection. So, after I determine whether the object should be added to the collection or not, then I'm going through its properties and recursively calling walk again. If the property is not a TempObjectBase derivative, but is in fact a list or a collection of some kind, then I can do something with that collection itself if I wanted to. In this case, I do not want to. And then I can iterate through all of the items in that collection. And if they are TempObjectBase derivatives, once again, recursively leverage my walk routine. Then at the end, simply a call to walk(this) will start the object graph walk and collect everything that has an IsDirty property set to true. The end result is this one collection called dirtyObjects. Now, that in itself, I think it's kind of cool. It's a nice way of leveraging lambda expressions, and I think you can probably take this code apart and reuse it for several other purposes. In fact, that's exactly what happened to me. I needed to come up with more than one method that needed to do an object walk to accomplish several things. Some examples of these several things would be a method to get all of the dirtyObjects like what I've just written, but I also needed to write a method to clean all the objects. So, if the object is dirty, set it to clean or set the dirty property to false. So, that's the same kind of object walk, only now I'm performing something else upon the object that is a subject of the walk. Another task that I had was to write a method to determine if there is anything dirty in the object graph, which means instead of collection a collection of dirtyObjects like I'm doing here, I needed to just walk the object graph. And as soon as I encounter one dirty object, I can set a flag to true, and then I needed to stop the object walk. Now, I already have that facility in place here. So, as you can see, I'm only continuing the object walk if this is set to true, but here I've set it to false. I've hard-coded it to false. So, if I wanted to write a method called IsObjectDirty that does an object graph walk, but stops upon the first dirty object it encounters, I would leave everything exactly the same. The difference is that I would not be adding it to a collection, and the minute that I find a dirty object, I would be setting this to true so that the recursion stops there. So, I had three or four different tasks that all required a virtually identical object walk. In fact, this is where it gets cool. It's an entire function of code, and this little part right here is the part that changed from method to method. And in some cases I wanted to act upon the collection, so I had some code here. Everything else around it

was exactly the same. So, this is where the routine got really cool because now I incorporated, once again, the use of lambda expressions to help me out here.

Refactoring Object Walk for Reusability

Let me once again through the magic of Camtasia, replace my GetDirtyObjects with the routine that I'm talking about. And say it with me abracadabra, alakazam, and presto. Okay, so here I have a more general purpose method called WalkObjectGraph. Now, WalkObjectGraph receives three arguments. The first one is a code snippet that is going to act upon the object. The next one is a code snippet that may act upon the collection. And then just because I really live under the Myth Buster's creed of what's worth doing is worth overdoing, I added another argument called exemptProperties whereas I may have a need one day, and I have had in the past where I want to leverage my WalkObjectGraph method or an object, but I want to exempt a couple of properties from that object walk. So, I can just send them in an array here. And as you can see down here, I'm checking to see if they're in the exceptions. And if they are, I don't go through the recursive walk. Now, what's different here? Well, the general object walk functionality is exactly the same, but notice I'm not checking for dirty anywhere. In fact, that's been replaced with this right here. So, now I can send in a code snippet that's a type Func of ObjectBase. Well, in this case it should be TempObjectBase because that's the temporary class that we wrote. In fact, let me change all the instances real quick to TempObjectBase. There we go. And now when I get to the point here where I would normally set or should I say check the object for IsDirty and then add it to a collection, notice that dirty collection is not even here. I'm going to do that in a couple of minutes. So, now what I want to do is execute that snippet, and then return the value for exitWalk. So, the caller of the WalkObjectGraph can then determine whether it's going to short-circuit or not. Then down here where before I said do no functionality for the coll variable, here I may have some functionality that the caller may require, so that's what the second snippet is for, and I invoke the snippet for collection. The rest of the code around this is exactly the same. So, what I did is take the little bit of code in the middle of this function that may vary from use to use, and I replaced it with the invocation of a lambda expression or an action or a func, which would get called in as a lambda expression from the caller. And this is what makes this routine cool because now I can have a method call GetDirtyObjects like I did before, only this time it's going to look a little different. This time the method is going to call upon the WalkObjectGraph like this. So, now we have a GetDirtyObjects method. There's my dirtyObjects collection. Here's where I'm returning it just like in the first example I showed you. But now instead of walking the object graph from here, I'm simply calling the WalkObjectGraph, and I'm sending in the code snippet that is going to do the dirty check and add to the collection and return the false for short-circuiting. And then for the collection action itself, I had nothing to do, so I'm just sending in an empty curly brace construct, and then I'm returning the dirtyObjects at the end. Now, where does this come in really handy? Well, what if I wanted to write that second method that I told you about, the one that cleans all the objects, in other words, sets everything that's dirty back to clean? I can now do it like this. So, now anything I need to write that needs to do an object walk and just act

upon the object that's subject to the walk or act upon the collection property that it may encounter can leverage the WalkObjectGraph in a very, very clean fashion. Now, what happened here? Oh, well the IsDirty just doesn't have a setter, so we'll put one in real quick. So, now let's do one more because it's so easy to do. It's really fast now. What if I wanted to do the method that checks to see if anything is dirty, and as soon as it finds something that's dirty it short-circuits, it stops the object walk? I can do it as simple as this now. As you can see, it's just another call to WalkObjectGraph. Only this one, it's checking to see if the object is dirty. And if it is and I'm setting the variable that I'm returning here to true, and then I'm returning a true here, and that's going to do the short-circuiting because remember down here that's going to be returned in exitWalk. So, if it's not true, then I continue. But if it's true, I'm going to exit the walk, which means I'm going to stop the recursion, come back up to the caller, and simply return my final result. And that's how I'm doing all the functionality having to do with dirty checking or dirty setting from an object down its graph in an easy and what I think is a very clean and cool fashion. You can use this technique for a hundred different things. In fact, I have in the past. This whole notion of having a big function and then separating out chunks of code that may differ if you have to reuse this function is a great subject for this kind of technique. You know, we're used to reusing code in different ways. The latest iteration was with generics. You know, as a matter of fact, my friend Carl Franklin had a great definition for generic usage scenarios, and that is if you have two identical pieces of code and the only thing they differ by is a type, that's a perfect use for generics. Well, here I've taken it to another step. I've given you two or three different identical functions that are quite long and can be even longer where different sections within the functions differ, but the meat around them is exactly the same, and I've refactored it to use this technique, and it's worked out very well for me.

Helpers for Collecting Properties

The last thing I want to point out here is this one method that I told you about earlier, this helper method called GetBrowsableProperties. This returns the properties that I can iterate through in an object, and it doesn't only return them by obviously using reflexion to go get them, but it also checks them to make sure that they're not decorated with a particular attribute. There may be a situation, for example, look at the IsDirty property. This is an actual property, and I don't want this property to be subject to the walk. Now, it wouldn't anyway because the walk is designed to only look at properties that are of type that is derivative of ObjectBase or TempObjectBase. However, I can do this for any property. I have an attribute here called NotNavigable, and by decorating any property in my entity with this attribute, I exempt it from an object walk. This avoids the walk from hitting every single property when that may not be my intention. Now, I already have a facility in the WalkObjectGraph here called visited, and this facility is so that you won't run into stack overflow situations where an object may have a back reference to its parent. You don't want to start that walk over again and then eventually blow the stack out of the water. So, that's what the visited is for. But in cases where the visited is not enough, tagging a property with the NotNavigable attribute may help you out significantly. And, in fact, putting it on non-object based properties is just a good way of making the list that is

collected for the object walk that much shorter. So, next I'll finish putting together the rest of the client entities, and we'll go from there.

The Rest of The Client Entities

So, next I'm going to add the PropertyChanged notification to all the properties of my Car object, and I'm going to start by changing this to the real base class that it's going to use. And I'll show you some more of that base class in a few minutes. So, let's finish up the properties and there we go. So as you can see, every single one is calling OnPropertyChanged and it's using that compile-time safe technique. And I'm going to add another entity for the Rental and also put the other ones in there while I'm at it, the Reservation and the Account entity. So, I'll do the Rental first. Got to love the art of editing, huh? And now I will add the other two. And there we go. So, there's the Account and here's the Reservation. And as you can see, they all inherit from ObjectBase, whereas the business entity is inherited from EntityBase. Now, I can add as much more functionality as I want to these client entities as I need to. So, one of the things that we're going to be doing in this module is also adding validation to the Car entity. I'm not going to do this in this clip. I'll do that in another one, but other functionality may include possibly data annotations for the purposes of binding this to MVC views. It can include all sorts of things. Remember these are going to be the smart set of objects. These are going to be the objects that the client is going to interact with. So, these are the ones that we want the heavy functionality on. And for that reason, and for the reason of the class getting larger and larger, I didn't want to use the DataContract attribute explicitly. I wanted to use the inferred data contract, which means it's a public class so automatically the data contract serializer will treat every public member as if it were a DataMember. And in order to get the namespaces to match, remember the earlier trick, I did that in the AssemblyInfo file right there. Okay, so the next thing that I know that I'm going to need is some validation. And I can probably use validation in all of these, but I'm only going to put it in the Car entity for this example. Validation is one of those really big topics, and unfortunately it's still to this day one of those topics where there is no one solution that meets every single need. Because of the fact that we have object sets that can serve both the web desktop clients today, we most of the time need more than one type of validation. In fact, any XAML application like WPF, Silverlight or Windows 8 modern app or whatever they're calling it today uses validation in a very specific way. It uses an interface called IDataErrorInfo, and that's how XAML produces its validation results to the client. However, MVC does it in a different way. MVC uses data annotations. And if you're using Knockout like we're going to in this course, Knockout is mostly JavaScript client base, so it's got own way of doing validation. And a lot of times a complete system will have double sets of validation rules meaning the same validation rule can be expressed using more than one technology for more than one type of client that's going to address it. So, the validation that I'm going to be adding in this case to the Car entity is going to be validation that's going to be used by any XAML application that's going to bind to our Car entity. Later when we start doing the web stuff, I'm going to be doing validation as well, but I'm going to be doing it in the JavaScript world, so that's not something I'm going to cover right now. So, get ready validation is next.

Validation: Intro

So, I've written full-blown validation engines in the past on more than one occasion in fact, but I wanted to keep things simple for this application and leverage something that is reliable and easy to use. And the product that I found was called FluentValidation. It's available from NuGet, and it's a simple, yet is very extensible validation engine. So, I'm going to show you how to use FluentValidation, but I took it to another level. I wanted to wire validation into my framework, into the core base classes into ObjectBase. I wanted to do this where when you write an entity you inherit from ObjectBase, and everything you do to add validation will then be handled seamlessly by the framework. So, what I did was integrate FluentValidation into my ObjectBase base class. Validation needs to happen at a certain point in an application, and the common denominator of when validation should take place is when a property changes. So, if the value of something changes, it should be revalidated. So, I do my validation when a property changes, and you already know from the previous clips that we have a facility in place that gets called every time a property changes. Now, for validation to be reported back to XAML view, we need to incorporate the help of an interface called IDataErrorInfo that I'll be going over in detail. And here I'm going to be reporting broken rules back up for an object. And these broken rules are stored on a per-object level, which means just like dirty tracking each object is in charge of storing its own broken rules. And, of course, using the WalkObjectGraph technique that I showed you previously, it would be very, very easy to write a method to collect all the broken rules for an entire object graph or from a certain point on down the object graph. So, I'm going to get all of this stated next in the code, and then you'll see a lot more of its usage when we get into the UI, into the desktop application later in the course.

Implementing Validation

So, setting up validation for an object is actually quite easy using FluentValidation, and what I want to do is that I want to add some validation from my Car entity that you're looking at here. Now, in FluentValidation, this means writing a class known as a validator class. So, I'm going to have a class called CarValidator, and I don't want to reuse this class anywhere else, though you can certainly set up validator classes that are reusable. But because I've designed it where I'm not going to be reusing this class, what I'm going to do is write the class inside my Car class, and I just maintain better encapsulation. So, let me do that first. (Typing) There. So I have a class inside my Car entity called CarValidator, and it has to inherit from the FluentValidation class called AbstractValidator, which accepts the class that is being validated or the type that is being validated in the generic argument. Now, the next thing is to create a constructor for this class and put my rules in there. The rules that I want to give my Car entity are very simple. I want the description and color to not be empty meaning they're going to be required properties, I want the rental price to be greater than 0, and I want the year of the car to be greater than 2000 or less than or equal to current year. So, I do this in the constructor of the CarValidator. There. So, now the idea is that FluentValidation can call upon the CarValidator class and execute validation, and then there's a property of the CarValidator for the results of that validation so you can examine them to see if there are any broken rules or not. Now, I want this kind of validation

execution to be what's done behind the scenes and done at a time of my choosing, or should I say a time of the choosing of my framework meaning whenever a property changes. For that, my base class, the ObjectBase class is going to need to know about this class, the CarValidator, so it knows to call it. So, what I did is that I gave my base class, and I'll show you the code for this is a second, a virtual method called GetValidator where I can now say override GetValidator, and here I'm going to return an instance of the CarValidator class. And this is all I have to do to add validation to an object. It's as simple as this. The rest of the magic is all done in the ObjectBase class. So, let's switch there now. This is my full-blown ObjectBase class, which is a lot more complicated than the TempObjectBase that I created in front of you. It has a lot more capability in it, but I want to draw your attention to the validation stuff. Now, this base class is designed to expose a property for ValidationErrors, and it's designed to call upon that Validator automatically. Those are the pieces that I want to show you. First of all, notice that I am calling the Validate method right from the constructor right after I obtain my Validator, which you just saw me override in the Car entity. Now, the reason I'm calling this in the constructor is because the idea here is that when you create a new object out-of-the-box, you immediately want to validate it to report on any broken rules that object may have as a new object. That's why I'm calling validate right in the constructor. I'm also calling it down here in OnPropertyChanged. Now, what exactly is validate? So, validate is essentially calling the Validator. Remember, this is a result of the override that you saw, which our example is an instance of the CarValidator class, and it's running the Validate method and obtaining the results in a property of my ObjectBase class. It's as simple as that. So, it's a very, very simple set of wrapping functionality around FluentValidation that integrates itself with the ObjectBase base class. Now, in my particular implementation for FluentValidation, I've actually decided to just validate the entire object upon the changing of any of its properties. It can certainly be argued that that's not entirely efficient; however, it could also be argued that it is very safe because when you set up the rules, checking a rule for one property may actually check it against the value of another. So, it's a very safe thing to run through validation of the entire object any time any of its properties change. However, I could have made this a little more complex and check the rules only for a given property when that property changes and then maybe figure out a way to set up dependencies if one rule is going to depend on another. But in all honesty, I didn't want to drag out this module into too much validation stuff. Now, an additional nice feature here is that I've given ObjectBase an IsValid property, which checks to see if there are any validation errors out there. So, the nice thing about having one property that does this as a simple Boolean is that you can use it for binding. So, you can use this perhaps to show some kind of error box or anything that you want really on the UI that is just based on the value of one simple property. Now, let's talk about the IDataErrorInfo interface. This interface is wired and called by the XAML application, and it's calling IsDependent on a binding setting in the XAML binding that I'll show you when we get to that module. Now, the way it works is very simple. The columnName or property name that is being validated at that time gets sent into here. Now, this is not where I want to execute my validation. I want that to occur at property change time. But this is where XAML is going to query for any validation results for a specific property. So, what I want to do is that I want to loop through my current ValidationErrors if there are any, and when I see one where the PropertyName matches what came into this indexer property, then I want to add that to a

collection, in this case it's the string builder, and then return it here because I'm returning a string. The idea here is that XAML will automatically visually report any element that is bound to this property in an error template style if the result of this query is not empty. If it is empty, if there are no errors, then it'll just show the element in its regular form. What I mean by validation style or element in regular form, think of a textbox in its regular form or a textbox with a red border around it. That's what I'm talking about. And you're going to see a lot more of this later on. As this course progresses through the modules, you're going to see the Core. Common and Core. Common. Contracts project come up again, and you'll see more and more functionality that I add to this and that I provide to this.

Unit Testing: Intro

The unit tests that I want to write here are going to be primarily for testing core functionality. So, what we need to test are basically the features that I added to my framework, to my ObjectBase class, and that includes testing property change notification. Now we have to test both basic notification and also test that routine that I showed you on how to avoid duplicate subscribing. We'll be testing that as well. And I'm going to test dirty tracking, as well as object cleaning, so everything that has to do with dirtiness and the obtainment of dirty objects. And also, we'll test validation. Now, I want to talk to you quickly about the conventions that I put in place for my unit tests, how I name my unit tests, and this is where I'm going to make some of you out there probably scream bloody murder. I try to have one set of standards for each category of development. Unit tests were a little more difficult for me because I have two sets of standards for naming my unit tests, but I define them and I use them religiously. I don't deviate from these two sets. So, whether you like them or hate them, this is what you're going to see in my unit tests. You chose the standard that fits for you and your team of course. So, first of all, the name of my class to be tested is always followed by the word Tests, and the methods is where I have these two sets of standards. I have tests that may be named using Pascal casing. And when you see that, that means that the name of that test is the exact name of the method to be tested. However, testing a particular method is not always the case in a unit test. An example is what I'm going to show you in the unit tests coming up in the next clip, which are unit tests that test property change notification. There's no one method for property change notification, although it can be argued that yes I have that virtual method called OnPropertyChanged. But in a test like that or a test like for validation, I use a different convention, and that is where I use lower_case_underscore_delimited, something like what you're seeing in the example in orange there where I actually have a sentence describing what this test is going to be. It's a custom name descriptor if you will. So, you'll see both of these scattered throughout the application, but I don't deviate from these two. Once again, you may not like this. You may argue that the lower_case_underscore_delimited can fit both situations, and you're probably right. I chose it this way, and that's what you're going to see going forward.

Unit Testing Core Functionality

The first test I'll write is one to test regular property change notification. Now, I don't want to involve dirty tracking or anything like that in this test, just plain old property change notification. In fact, I don't want it to set the objects dirty state at all. So, what I did is that I have a TestClass here that I didn't write in front of you so not to bore you. It just has all the different properties that I feel I'm going to have. As you can see, it's a TestClass that inherits from ObjectBase. It's got a property called CleanProperty, which does not mark the object as dirty. It's got one called DirtyProp that does mark it as dirty. It's got other properties. It's got a Child property, which is a child object that has its own set of properties. It's got everything that I'm going to need to perform my unit tests. In fact, there's my TestClassValidator so I can do my validation testing as well. The reason I have this class is because I don't want to rely on the entity classes. I like my unit tests to stand alone meaning my unit tests are testing the core framework classes in this particular case. They're going to test the ObjectBase class. They're not going to test the Car entity or the Rental entity, so I don't want to have a dependency on them. So, I'm going to write a very simple test here, and I'm going to call it test_clean_property_change. And in this test, I'm just going to simply create an instance of my TestClass, and then I need to wire into the OnPropertyChanged event so I can test to see if changing a specific property triggered property change notification for that property. So, I want to create a test Boolean flag that I'm going to use to store my result. Now, I'm going to perform the actual wiring of the object's propertyChanged event. As you can see, I could've done this the traditional way and wired the event into another method, but now I've got a separate method that is not really going to be a test method. It's more going to be a support method for this particular test method. With a lambda expression, I can very easily just do this all in the test method itself. So, now in here, I would put whatever code constrict I want that will get called when the propertyChanged event or that TestClass instance gets fired. And in this case, I want to test to see if a specific property called CleanProp got changed. And if it did, I want to set my flag to true. Now, I am going to go ahead and execute a change on that property. And finally, I just need to do my assertion. (Typing) And my test is done. Now, all I've got to do is mark it with a test method argument, and I should be ready to execute it. Now, I'm using CodeRush here, which makes my tests very easy to execute. I can just click on this test tube right here and say Run. And in another window that you can't see, the test is running, but you'll see the result on the little test tube in a second. And there we go. We got a passed test. Now, I won't bore you by writing the additional tests step by step, line by line, but I will describe them and make them appear so you can see them on the screen. And of course, all the sample and downloadable code has all of these tests and more.

The Rest of The Tests and Conclusion

So, the next test that I want is going to check to see if the setting of the object's dirty state works properly. As you can see, I am instantiating the class, I am asserting that the object is clean because it's brand new, and then I'm setting a property that I know happens to set the dirty flag to true. I'm setting that property to something effectively calling property change notification, but also marking the object as dirty. Then I'm asserting that the object is now dirty. The next test is going to check that duplicate subscription avoidance technique that I showed you much earlier. So, here

you can see that I am wiring handlers into the PropertyChanged event, and I'm wiring the same handler. But to help my test out here, I wrote some code where I'm effectively declaring the handlers in code. There's handler1 and handler2 that I'm wiring into the PropertyChanged, and then when they get wired, the changeCounter variable is going to increment. So, as you can see, I'm wiring handler1 three times and handler2 twice, but if my duplication avoidance worked successfully, I should only get two increments of the counter, and that's what I'm asserting for down here. The complete unit test code also includes testing more dirty tracking stuff including the short-circuiting and the tracking to see if something down the object graph has been set as dirty, collecting the total number of potential dirty objects, a couple of other things. The last test that I want to show you for this example is the test on validation. And what I did is that in the TestClass I added a validator, and I made a validation rule for the StringProp property to not be empty meaning it's a required property. And now for this test I'm instantiating the TestClass, I'm asserting that the object is not valid because remember in the constructor of ObjectBase I'm initiating a validation, which means the initial object when you create it initially is going to be invalid because of that validation rule. Then I'm setting that property to some value, and I'm asserting the IsValid again, and this time it should be valid. So, let's see if all of these tests run successfully, and I can do that very simply by clicking on the TestClass here saying Run tests thanks to good old CodeRush, and every single test has passed. In fact, if you see a checkmark on the class itself, it means that all of the tests have passed. So, there you have it. That's it for the entity module. We've created client and business entities. We've created some core framework functionality. We've put some unit tests together. We've got this set of entities now that are ready to be used among all the layers that we're going to create throughout the rest of the course. In the next module, I'll be jumping into the data access layer where I'll be combining Entity Framework Code First with a repository pattern, an abstract factory pattern, and some dependency injection to come up with a nice, elegant, and reusable solution to access any data that I want. So stick around. I've got a lot more cool stuff to show you.

Data Access (starting the business tier)

Intro and EF Code-First Description

Hi. This is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. In this module, we're going to get started writing the data access layer, which is going to be our entry point into the entire business tier. So, let's go over some of the highlights that you're going to see in this module. First of all, you're going to see Entity Framework in a Code-First usage scenario, which is actually my favorite way of using it, and I'll go over all those details. We're going to talk about data repositories, and we're going to write the data repositories

including all of the abstractions, interfaces, the entire core behind it and how it works. We're going to be incorporating dependency injection with the data repositories. Then I'm going to implement an abstract factory pattern into something called a repository factory that I'll explain in detail. And then we're going to go over usage scenarios. How do I actually use the data access layer, which is going to be used from several places, and we're going to do it using unit tests. And then I'm going to cover what I call additional DTOs. I'll get to that in detail at the end of the module. And I'm going to leave you with some references that you can use to learn more about Entity Framework and some of these other technologies. So, let's discuss Entity Framework Code-First. Entity Framework Code-First allows you to do everything manually, which is the way I like to do it. You have full control over it starting with the setting up of entities manually, which in our case it's the business entities. We set those up, and the data access layer will use those as data entities. This bypasses the auto-gem from an already existing database. This would be Entity Framework in a data model first scenario where you allow Entity Framework to actually generate all of the code for you, all of the entities and the DB Context and all that stuff for you, and it becomes really, really bloated, and you don't have absolute control over it. Plus, you want to be able to modify your entities, add whatever you want to them, and though you can do that with the generated stuff, it's not really advisable because remember it's tied directly to the database in a way that it may be subject to regeneration later. So, this way of using Entity Framework is a great way of using as little or as much as you want from it, and there are a lot of parts Entity Framework that I like, but there are also a lot of parts that I really can do without, and I'm not forced to use those or even see them for that matter when I'm using Entity Framework in a Code-First capacity. You control the code 100%. That's the main thing that I like about it. There's no polluting the classes. You have full control over everything. So, like I said, the Business Entities is what is going to be used by Entity Framework for my entire Data Access Layer. Now, there is a class that I've got to set up myself, and normally in the auto-generated mode this would be set up for you, but it would be set up with a lot of excess stuff and stuff that I don't need. So, by me setting up this class called the DB Context class, I can add to it only the pieces that I need. And what I'm going to need to do is define my collection of entities and any kind of ORM configuration that I'm going to be needing. And I do this with the Entity Framework Fluent API, which works really nice, and it keeps everything regarding my object relational mapping nicely encapsulated into one class. So, let me show you a brief pseudocode sample of what this DB Context would look like, and then in the next clip we're going to jump in and write it. So, it's just a class that inherits from the DbContext class. This comes from the Entity Framework. And I need to give it three things. First of all a constructor. That's what's going to point to our connection string. That's how it points to the proper database. Then I need to give it a property for each table that I'm going to be mapping to, and these properties are going to use the DbSet type where the T in the generic is going to be the entity. And then I'm simply going to override a method called OnModelCreating, which is going to give me an argument called DbModelBuilder, and this is where the fluent-interface is, the Fluent API. And this is what I'm going to use to create any kind of object relational mapping rules that I need, and I have just a few of them, and they're all very simple.

Creating The DB-Context

So, here we are back in Visual Studio, and as you can see from the Solution tree, our projects look about the same as the last time we left them with the exception that there are two additional ones. I now have a CarRental. Data project, and I have a CarRental. Data. Contracts project, and I'll get into the differences between these in a few minutes. I'm going to start with the CarRental. Data because this is the primary one in our data access layer, and this where the DbContext class that I spoke of in the last clip will come into play. So, the first thing I'm going to do is that I am going to add a class here called CarRentalContext, which is my DbContext. The idea here is that there is one context class for each database that you're accessing, and, of course, a really large project can have more than one database in which case you would have more than one context class. I only have one, so I am going to give it a constructor. And as I said in the slide in the previous clip, the constructor is what's going to point us to the actual database. So, let's give it the constructor first. In order to point my CarRentalContext class to the proper database, I need to give it the connection string, and I do that by calling into the base constructor. Now, I have no base because I forgot to put the base class here, so let's add that real quick. Now, I have a base class. Let me put in a call to the base constructor. This is my connection string name, CarRental. Now, I'm going to be testing this from a TestClient later, which is going to be just the unit test, so let me show you real quick the App. config for that unit test because remember the only config that matters is the one on your client executable, which in this case is going to be a unit test. Let me open that up real quick. And as you can see, I have a connection string called CarRental that's pointing to the CarRental database that I've set up. The only other thing I need to do in my constructor here is set up my initialization strategy. The initialization strategy can do things like optionally check to see if a database already exists or created, seeded with sample data or any other data, a number of things. I don't need to do much here because I've already created the database. I don't want Entity Framework to do this for me, though it can do it optionally. I simply need to add a line telling the database to use this context class. (Typing) And that's it. My constructor is done. Next, I'm going to be adding the properties that I need to the CarRentalContext, and these are the properties that later the data access layer is going to query using Linq-to-Entities.

DB-Context Properties

The properties I need to set up here are very simple. So simple in fact that I can use the automatic property syntax with just a get seimicolon, set semicolon. These properties are going to represent the tables in my database. So, I only created four tables. I'm only going to need four properties. Let me show you what one of them looks like, and then I'll explain it. (Typing) So, let me go through this one in detail because the other three are going to look just like it. Basically what I need is one property per table that I have, and I have four tables starting with the Account table. So, I set up a property called AccountSet. Now, let me explain to you why I used the word set. I could have used list, election, or I could've said accounts in plural. I don't like to use plurality when I'm pointing to a table. I just find that it gets confusing. Also, if I want to apply any kind of code generation technique to my system later for whatever reason,

it's a lot easier when I'm dealing with the actual name of the table and then a suffix attached to it as opposed to adding the letter S. In the case of account, adding the letter S would pluralize the word, but you know how the English language is. We've got IES, we've got all sorts of other exceptions. What if I had a table called octopus? You can already see where I'm going with this. Now, the AccountSet property is of type DbSet of Account. Account is my data access entity, which in this case, as you can see from the hover IntelliSense, it's my business entity, my account entity that I created in the last module. So, all I need is a property of type DbSet of Account that's gettable and settable. And now I need three other properties that represent my other tables. Now I've given my CarRentalContext class some properties that I can query against later. Now, I've still got to tell Entity Framework when you query something like AccountSet what table am I actually going to point to. Now, you would think that it's enough by it knowing that AccountSet is DbSet of Account. So, you would think that it knows to point to a table called Account. The problem is that Entity Framework by default will not assume that the name of the tables is exactly the same as the name of the entity. It thinks that the tables are of plural syntax to the entity names. So, by default when I go to do a Linq-to-Entities query, Entity Framework is going to query the AccountSet object by creating a sequel query that goes to an Account's table, and I need to tell it not to do that. I need to tell it that my tables are identically named as my entities. And I'll do that next, as well as some other ORM specifics.

Setting up The ORM

So, now I need to give Entity Framework the ORM rules that apply to my scenario here for the CarRentalContext class, things like which of the columns of each table represents the key of the table or any exceptions of properties that are in my entities that I don't have specifically mapped to the database, things like that. And you can do a lot of this with annotations, with attribute decorations in the entities, but remember I don't want to pollute my entities. It's already enough that my business entities have the DataContract attribute applied to them explicitly, which I don't mind because they're very simple classes, but I don't want to over pollute them, and I also don't want things scattered all over the place. I don't want other attributes there that one class may have, another one may not. Entity Framework gives us a very nice facility to do things like that that would otherwise have to be done with annotating properties or classes using attribute decorations with a method override instead where I have one place, a one stop shop to come to, in this case the CarRentalContext class, and I can see every ORM rule in this one location, and more importantly, I have one place to go manage it when I need to. So, this is done from an override called OnModelCreating, which I'll do now. The argument that comes into this method modelBuilder of type DbModelBuilder is what contains the entire Fluent API or Entity Framework. Let me scroll up a bit here. And this is where I can start putting in my conventions and my rules and all of the other mappings or any kind of exceptions that I have. For example, I may have, and I don't in this case, but I may have a situation where a certain entity maps to a table, but they're not named the same. What if I had a table called User, but I wanted to represent that table in an entity called Account? I can certainly do that by doing an ORM mapping. That is what ORM is all about. However, I

want to do it in one place, which is the case here. The first thing I need to do is to let Entity Framework know that whenever I'm going to use a Linq-to-Entities query to query something like the AccountSet or the RentalSet property, it's going to correspond to a table that is named differently than what it thinks it is by default. Now, even though up here I told it AccountSet is of type DbSet of Account, Entity Framework still thinks that even though there's going to be an entity called Account, it's going to be mapped to a table that is plurally named. It's going to look for a table called Accounts and likely a table called Cars, Rentals, and Reservations. And I don't want it to do that. I want it to ignore the fact that there is a difference in naming, and I do that with the modelBuilder argument by tapping into its Fluent API. So, here's the first thing that I'm going to do. (Typing) I'm about to remove a default convention. And like I said, the default is that Entity Framework thinks that it's going up against a plurally named table. (Typing) Now, Entity Framework will know that my tables are going to be named exactly as my entity. It's not going to try to assume any difference in the naming. Now, the next thing I need to do is take into account any maybe events, methods or additional properties or interface implementations that my entities use, and I need to tell Entity Framework to just ignore those things. Now, some of them may be automatically ignored, some of them may not. I like to explicitly ignore things just so that there's no guessing involved. So, if you recall, all of my business entities inherit from the entity base class, and that entity base class implemented the IExtensibleObject interface. We did this in the last module. And that interface gave me one member called ExtensionDataObject, and that was an actual property. And that property had a getter and a setter. And that's important because once it has both, Entity Framework thinks that there's going to be an automatic mapping over to a likely named column in the database. And, of course, none of my database tables had any column called ExtensionDataObject, so I need to tell it to ignore this. And, of course, ExtensionDataObject coming from the IExtensibleData interface comes from the System. Runtime. Serialization, which comes from an assembly of the same name, so my data access layer had to reference that assembly as well. The other two interfaces that I have in my entities were the IIdentifiableEntity interface and the IAccountOwnedEntity interface. Now, the only one I'm really concerned with is the IIdentifiableEntity interface because that's the one that's implemented implicitly by my entities, which means the member that it exposes is a public member, which means it becomes part of the actual interface of the entity itself, and that's why I need Entity Framework to ignore it. (Typing) And that's it for my ignore rules. The next rules are going to be the mapping rules.

Adding ORM Rules

The first mapping rule I need to give Entity Framework here is which column in each one of these tables is considered the key, is considered the ID. Now, in Entity Framework terms, we're talking about entities here, so I need to tell it which one of the properties is considered the key. Now, this has nothing to do with the IIdentifiableEntity interface that I set up in the last module. That's going to be used later on when we set up our repositories. This is just Entity Framework stuff. So, let me tell it which property of the account entity corresponds to the key of the account table. (Typing) This is what's nice about having a Fluent API. It has a really nice flow to it, and things just make sense,

and the IntelliSense helps you a lot here. So, here what I told it is that for the Account entity I have an integer key that corresponds to the AccountId property. And it lets us use a lambda expression much like I showed you in that little trick with the propertyChanged where instead of sending in a string, I can send in a lambda, and this give us compile-time safety. Now I can do this for the other tables as well. And there we go. Easy enough. Now, this would be equivalent to going into each one of these entities and decorating the corresponding property, in this case AccountId, CarId, RentalId, and ReservationId with the key annotation or with the key attribute. But rather than have that spread out in my classes, I have it all nicely managed in one location here. Another rule that I need to give my entities here is a rule telling it which properties to ignore, in other words, which properties are not directly mapped to anything in the database. Now, in this case, I have a property called EntityId, which is the implementation of the IIdentifiableEntity. Now, we've already told it to ignore IIdentifiableEntity here. For some reason it barked at me even though I told it to ignore that because it found the EntityId property, and the fix was simply to tell it to explicitly ignore the EntityId. Now, I can do this with another line that just says modelBuilder. Entity of Account. Ignore, and then point to EntityId property. But because we're dealing with a fluent interface here, a Fluent API, we can do a lot of things in just one line. So, here I can simply just continue here, and the IntelliSense would actually help me. Watch. There's the Ignore, and I've just told it to ignore the EntityId property without having to repeat which entity I'm talking about. Now, I'm going to do the same thing for the other three. (Typing) And there's one more property now that I need to ignore. Now, I'm not sure if you picked up on this earlier, but there is one difference between the database Car table and the Car entity. Let me go to the Car entity to show you want I mean. All of these properties here correspond to columns in the database with the exception of CurrentlyRented. This will be used, but later on. And this is generated on the fly. This property is given its value based on business information or business knowledge that it will calculate during a service call later on. Now, I need to mention obviously you don't always have the ability to set up everything on each layer 100% before you go onto the next layer as we're doing in this course. So, in real life when I developed this application, I didn't immediately add a property called CurrentlyRented to the Car entity until later on when I was writing a service call I figured out that I need one, and then I came back and added it. In real life, I wrote a little bit of each layer, and then as I added more to my UI or as I came up with more needs, I came back and added more service calls. And as of course I added more service calls, I ended up needing to add more data access queries and so on. And that's just the way things work in the real world. So, let's go back to the CarRentalContext, and I need to tell it to now ignore the CurrentlyRented property once again because the database does not have a column that represents this property. I don't need to tell it to ignore anything regarding the IAccountOwnedEntity interface such as the OwnerAccountId that you saw in the last module because that was an explicitly implemented interface, and its members did not become an actual part of the interface of the entity, so Entity Framework will never even see it. The only way to see the member of that interface is to cast to that interface, which Entity Framework is not going to do. So, this actually concludes any of the ORM rules that I need for my application. As you can see they're quite short here. They can get longer. I can do things like this for example. Let's go back to the previous example, even though I'm going to erase this. If the Account entity actually would map to a table called User, I can do it here like this. Before

this Fluent API was made available to us, I would have had to gone into the Account entity and decorate it, the class, with an attribute that told it map over to the user table. And now it gets a little weird because now I'm actually starting to put database specific things in my entities, which I really don't like. I like to keep those worlds as separate as I possibly can, and this Fluent API gives me the ability to do just that. And at the end of this module I'm going to show a slide that has a link to an MSDN page that shows you the documentation for the Fluent API that Entity Framework Code-First uses. So, that's it for setting up the DbContext. Now, we can start writing repositories and actually use this DbContext, and I'll do that next.

Overview of A Data Repository

The data repository patterns I've put in place for this application are designed to make it simple to write data repositories, and also simple to extend current repositories as needs grow. Let's talk about some of the characteristics of the pattern, and then we'll jump to the code and write it. So, the idea is that there will be one data repository per table, and we're going to be Linq-to-Entities inside the methods of these repository classes. I wanted to establish a pattern for doing standard CRUD and then also make the pattern extensible for additional functionality. Standard CRUD of course meaning Create, Replace, Update, and Delete. These are all functions that every table is going to have at its disposal, and there's going to be a pattern of reusability that every repository is going to start from, and it's going to provide these CRUD operations. But then I'm also going to implement a pattern around it to allow me to extend it because obviously any repository class should also have the capability of performing more data operations on that table than just the standard CRUD operations. And the IIdentifiableEntity interface that you've seen a couple of times already is going to rear its ugly head again in this pattern because every repository is meant to act upon an entity that implements this interface. This is how I am achieving the reusability I am with the pattern behind the scenes. Now, the CRUD operations that are going to be common to every repository are going to have a defined interface where they come from, but any custom methods that I add to a repository that will vary from repository to repository will also come from its own interface. Making every repository interface driven means that I can export it through MEF and make it subject to dependency injection so that the proper repositories can be injected into the classes that will use them later on. Let's take a top level view at what this pattern is going to look like, and you're going to see it in code soon. At the bottom here, we have the actual class that is going to contain all of the methods that we use to access data from the account table. This class is going to be called the AccountRepository. Now, this class is going to implement an interface called IAccountRepository that's going to give me the definition for every method in here. Now, the methods in the AcountRepository are going to be broken up into two sets. There's going to be the common CRUD set that's going to be common to every repository class, and then there's going to be the custom methods. All of them are going to come from the IAccountRepository interface who is only going to define the custom operations because the common ones come from a framework interface called the I<T>. In this case, T being account. The generic constraint behind it is an entity that implements IIdentifiableEntity. That's why

that interface has become so important. Now, the Repository class also inherits from a class that's going to give it the overridable methods to perform the CRUD operations and to fill in the blanks where there are differences from repository to repository. This base class is going to be called DataRepositoryBase<T>, or in this case Base<Account>. And this class actually inherits from a framework base class, which we could've inherited directly from. So, I could've made the AccountRepository inherit directly from DataRepositoryBase<Account, CarRentalContext>, but because I'm going to be using all my data repositories in this particular project against the CarRentalContext, I put a shortcut base class in that project called DataRepositoryBase<T>, which is the first data repository base class that you see at the bottom here. And all that is just a shortcut so that we don't have to constantly repeat the CarRentalContext argument, and you're going to see all of this in code. So, this is what the pattern is going to look like, and it'll make a lot more sense as we write the code next.

Core Base Classes & Interfaces

Let's start with this interface, IDataRepository<T>. It's actually quite simple. As you can see, it's an IDataRepository<T> where the T generic argument has a constraint on it and can only be used with entities that implement the IIdentifiableEntity. That's the whole reason for having IIdentifiableEntity. If you remember, IIdentifiableEntity lets you tell the entity which one of its properties is its ID. The whole reason for having this interface is so that I can implement this pattern around it. And now any class that derives from IDataRepository<T> and acts upon T will know which one of the properties in the T entity will be the identifier. More about that a little later. Now, as you can see, the IDataRepository<T> is where I declare my CRUD operations. These are the operations that are going to be common to every single repository class whether it's going to be the AccountRepository, the RentalRepository, ReservationRepository or any future one. And very simply I have an Add, I have two Removes, an Update, and a couple of Gets. I just try to give it as much flexibility as possible. The Add obviously is going to be for an insertion, and the Update for a replacement. And you can see that T is used throughout here. And then I gave it two Removes. One of them so you can remove if you have the identifier or the other one you can remove if you happen to have the entity. I could've done that for the others as well. I decided to do it only from the Remove. And I have two Gets. I have a Get that receives a single entity and one that receives a collection of them. This additional interface here, this is a pattern that I got used to over the years, and that is when I have an interface that has a generic argument, I always inherit from an interface of the same name without the generic argument even if it doesn't have any members. This has come in very, very handy in the past. So, what exactly inherits from the IDataRepository<T>? Well, staying here in the Core framework, and notice that I'm in the Core project. In this case I'm in the Core. Common. Contracts because I keep all of my interfaces in their own project. I move up to the Core. Common, and I have this DataRepositoryBaseClass. And as you can see, DataRepositoryBaseClass takes two generic arguments. The first one is, once again, T being a constraint of IIdentifiableEntity. The other one is what DbContext am I going to work against. Remember the CarRentalContext that I created earlier? Now, also notice that this base class inherits from

IDataRepository<T> that you just saw. And knowing that T is an entity, and now knowing that U is a DbContext, I can now write overridable or abstract methods that my DataRepository will then override and implement on its own. Now, the idea here was for me to write a DataRepository and basically fill in the blanks. That's the terminology I like to use, fill in the blanks. Give the developer the least amount of work to do to write the code that is going to differ from one repository to another. So, in this case, the public methods that are going to be exposed by the DataReposity, the actual Add, the Remove, the Update are going to take care of all of the common code as you can see instantiating the entityContext, doing a SaveChanges, returning an Entity, doing another SaveChanges down here in Remove, doing an update. Here I'm using a simple property mapper that I wrote to perform mapping from one object to another. Performing the retrieval. All of the common code is taken care of by these public methods. Now, there's a little piece inside each of these methods that is what's going to vary from repository to repository because that piece is the actual Linq query, and that's why, as you can see right here, I'm making a call to AddEntity; AddEntity being an abstract method. This is what the repository class that I write later is going to override and fill in the blanks with the Linq query. Here's another one called UpdateEntity. That one is being used down here. This one is calling GetEntities and GetEntity, and there's my definition for those two. So, to start with, the repository class that I'm going to write is very simply going to have to just override these four methods, fill in the blanks with its own Linq query, and all of this code is going to take care of everything else behind the scenes. Let's start writing that account repository now.

CRUD Implementation in The Account Repository

I'm going to start by writing a class called AccountRepository. The naming convention that I've chosen is just the name of the table to which this repository will apply followed by the word repository. Now, I need to say just because this repository implies that it is assigned to a particular table does not mean that I can't break the rules and use a query or write a Linq query that is going to retrieve data from another table. I can certainly do that. The repository is my own logical grouping of methods that apply to that table or at least using one particular table as the primary table. So, I may very well end up with a query later that does a join with two or three tables, but I think it's safe to assume that in every query there is always one primary table that kind of runs the show at least logically, and that's what these repositories are meant to encapsulate. So, let's add a class called AccountRepository. (Typing) Now, this class is going to start by inheriting from the DataRepositoryBaseClass that I just showed you in the last clip. Now, I don't need to implement the IDataRepository interface myself from this class because the DataRepositoryBaseClass already does that for me. (Typing) Now, if you recall the diagram, the diagram showed that my AccountRepository class was inheriting from a DataRepositoryBase<T> only, not of T, U like you see the screen here. The only thing that other class gives me, and I'm going to write it now, is just a little less coding. So, instead of all my repositories having to constantly tell the base class that it's going to use the CarRentalContext since they all do, I just gave myself another class to take care of that for me. Let me write that class real quick. (Typing) Now, notice this DataRepositoryBaseClass is inside my CarRentalData project. It belongs to this data access layer project, whereas the other DataRepositoryBase class is

part of my common framework. So, this one is going to have a T argument, and then it's going to inherit from the core framework base class. (Typing) I'm going to send T to the core framework base class, and then I'm also going to send the type of the DbContext, which is going to be CarRentalContext. (Typing) And then I'll give it a constraint to make sure that it's not used improperly. (Typing) And this constraint is going to be important, especially this new that you see here because this is what's going to allow the base class, which is here to do things like this, which is actually instantiate the DbContext class. So, let's go back to the AccountRepository, and now I can actually remove this. Notice it didn't break compilation because it found a DataRepositoryBase<T>, which is that small one that I just wrote. Now, let's implement the DataRepositoryBaseClass, which was an abstract class and see if there's anything that needs to be implemented. And those are my overridable methods as defined all the way back here in the core framework class. All I've got to do now is fill in the blanks. Notice that I don't have to instantiate my CarRentalContext. It comes into the argument here because AddEntity defined here is being used by the public Add method, which takes care of instantiating my entityContext. All I've got to do is fill in the query that's going to take care of the addition of an entity in the context of an account entity, and that's where it's going to vary from query to query. So, in the case of the AccountRepository, the AddEntity will simply look like this. Because the AddEntity method gives me the entityContext and it gives me the entity itself, all I have to do is add it to the AccountSet, this is standard Entity Framework syntax, and return it. And now I've got to do something similar for Update and GetEntities and GetEntity. So, let's do the update one first, and there's UpdateEntity. And as you can see, it's a Linq-to-Entity statement, a Linq query, but it's working specifically on the AccountSet and with AccountId. Now I can do the query for GetEntities. Once again, just the Linq-to-Entities query. And one last one to GetEntity. Now, these are just the basic CRUD operation, which means when I write the CarRepository or the RentalRepository or the ReservationRepository, it's going to follow the exact same pattern. It's going to inherit from DataRepositoryBase of its appropriate entity and then implement the five overloads or overrides with its own Linq-to-Entities query that applies to that specific set property as defined in my DbContext earlier. Now, I told you earlier that the repository classes are going to be subject to dependency injection, and they can't do that if they don't have an appropriate interface from which they inherit, so that's what I'm going to do next.

Custom Methods & Repository Interface with DI Setup

So, in order for the AccountRepository class to be used in dependency injection or to be used by MEF, it needs to be exported and associated with an interface type. So, what I need to do is give it an interface from which to inherit, and that interface needs to define the entire public interface of the AccountRepository. Now, the AccountRepository does not have a public interface per say when you look at it this way, but if you look at the base class from where it inherits DataRepositoryBase, which takes us to here and in turn takes us to here, here we certainly do have a public interface. We have an Add, a Remove, Remove, Update. Now, remember these methods came from the IDataRepository interface. There they are right there. So, ultimately those CRUD operations in their public form are

available to us from the IDataRepository<T>. Now, I can't inherit this class from IDataRepository<T> and then use that to export it because MEF doesn't let us export with a generic argument, but that's not the only reason I want my own interface. The other reason is because I don't want to limit this repository to just these operations. What if I wanted to add another operation, which I'll do right now? What if I wanted to add an operation called GetByLogin, which means I want to retrieve an account record from the table, not given ID, an integer ID, but actually given an email address, which is used as the login? I can do something like this. I can very simply add my own public method. Now, this method is going to return an Account entity. It's called GetByLogin, and it's simply instantiating the Entity Framework context on its own, performing a query, and returning some results. It's as easy as that. And this query can be as simple or as complex as you allow it to get. Now, let's come up with an interface that serves as the base interface for this class, and that interface is going to reside in my CarRental. Data. Contracts project in the Repository Interface's folder. And I'm going to call that IAccountRepository. So, I give these interfaces the same name as the repository class they represent, but of course using interface syntax. (Typing) Now, remember this interface needs to expose, or should I say define, the public members of the AccountRepository, and we've given the AccountRepository one public method and the other six methods come from the base class. So, this interface can inherit from the IDataRepository<Account> effectively defining the pubic members from that base interface. (Typing) So, now the IAccountRepository interface defines the Add, Get, Remove, all of those public members that the AccountRepository interface is going to use. The one member that it does not define yet is that extra one that we gave it. This one right here, GetByLogin. So, now I need to add this definition to the IAccountRepository interface. (Typing) And now I can let my AccountRepository class implement this interface. (Typing) And that's it. Notice there's no compilation breakage because the IAccountRepository implements GetByLogin, but it also implements Add, Remove, Get, and all those others through its inheritance chain because it takes us all the way back up to here, and here's where those methods are implemented or defined. Now I have an interface that I can use for dependency injection here. (Typing) And if you've used MEF before, you know that by default MEF will use a singleton pattern to resolve types, so you've got to tell it specifically not to use a singleton pattern. (Typing) Now, later when I create my dependency injection container, MEF is going to find this AccountRepository class, associate it with the IAccountRepository. Later I can simply inject IAccountRepository instances into any of the client classes, and MEF is going to know what class to instantiate. That's standard dependency injection usage.

The Rest of The Data Repositories

Now that I have a nice, clean, and elegant pattern to follow, I can add the Rental, Reservation, and Car repositories easily. Now, you don't think I'm actually going to code every single one in front of you, do you? I'm going to use the magic of Camtasia once again. Stand by. (Typing) And there we go. So, now take a look at the RentalRepository for example, and you see that it follows the exact same pattern, DataRepositoryBase<Rental>, it's got its IRentalRepository, there's the overrides from the DataRepositoryBaseClass with a fill in the blank entity queries that

apply to the RentalSet property of the DbContext, and here are some custom methods that I added that I know for a fact that I needed. As I wrote the application, I added these methods as the need came up. And there's GetRentalHistoryByCar, GetCurrentRentalByCar, GetCurrentlyRentedCars. These are all just custom Linq queries. Now, this one is the custom DTO that we'll get to later on. These are all just public methods that I added, and, of course, they now belong to this class, which means if I look in the IRentalRepository, it inherits from IDataRepository<Rental>, and it defines all of those custom ones. Remember the CRUD ones came from the base interface, the IDataRepository<T>, and then all of the additional custom ones are defined here. And, once again, I'll get to this one in a few minutes. That's why it's not compiling right now. And then I'm exporting the RentalRepository using that interface, which means that one interface contains all the public members for the entire DataRepository class. Same thing for ReservationRepository. Once again, a custom DTO that we'll get into, and the CarRepository. Now, I'm going to comment out the code that's not compiling right now because I'm not going to get into DTOs until one of the last clips in the module, and I want to be able to use this in my next example. (Typing) And now we should compile. (Typing) And build succeeded. So, in the next clip, we're going to start using this stuff so you can see how the repositories are actually used.

Setting up A Test Client Class to use DI

Now, I just don't want to write a unit test against the AccountRepository class for example. I want to actually write a test that's more of an integration test. I want to actually write a test that is going to use it in the way that I'm going to use it in my application, and that is actually incorporating dependency injection. After all, I did go through all of the trouble of setting the pattern up to accept that. So, before I write the unit test itself, I want to write a test class that's going to service my client class. Now, later we're going to have different client classes for these repositories. Remember I said that in this architecture you can access a DataRepository from a business engine class or straight from a service class. In either case, they're both going to be clients to the repository classes and are both going to use dependency injection to get to those repository classes, so I'm going to simulate that environment now with a class that I'll call RepositoryTestClass. (Typing) Now, I want the RepositoryTestClass to have the data repositories that it needs injected into it using dependency injection, but in order to do that, I need to first set up my dependency injection container. Now, I'm going to do that from the initializer of my TestClass. MSTest and actually any other good unit testing framework out there such as NUnit, for example, will give you the ability to designate a particular method as the initializer, and that method will get called before every test starts. So, I'll write that method now. (Typing) And what makes this the initializer method for my unit test is going to be the attribute called TestInitialize or should I say TestInitialize. And what I want to do here is go ahead and build my dependency injection container. Now, if you're using the dependency injection container such as Unity or Ninject or Castle Windsor or any of the good ones out there, you would then in your initialization code for setting up your DI container perform all of the registrations, all of the mappings. With MEF we do the discoverability process where we tell MEF go out to the following assemblies and

look for all the classes that have export attributes and perform those associations. Now, rather than do that code here, I'm going to do it in my Bootstrap for that's what I have the Bootstrapper project for. So, if I come here to my Bootstrapper project, you notice I have a MEFLoader class, and in here I'm setting up my MEF catalog, and all I've got to do is add my assemblies here. (Typing) And all I have to do to build an AssemblyCatalog is give it any one of the classes in that Assembly. And that's it. As this application grows, you're going to see me add more entries to this Bootstrapper class here, and that's just going to tell MEF go out and scan more assemblies for dependency injection. So, now I have to go back to my unit tests and actually perform a call to this Init method here, and that I'll do from the Initialize method. Now, I need to hold onto that dependency injection container, and I gave my ObjectBase class, which was, if you recall, the base for my client entities a static property called Container. I could've put this anywhere I wanted. I just chose the ObjectBase class as the place to house it in. (Typing) And that should get me all set up. Now, the reason I needed to store that container where it's accessible to the application anywhere that I want it is because somewhere at the beginning something is going to have to call upon that container. Now, the idea is that you ask the container for one class, and it resolves everything down the chain, but at some point at the beginning you're going to need access to that container. Now, I want the RepositoryTestClass to be my client of the repositories. I need the RepositoryTestClass to actually perform a resolve process because I'm going to give it injections, I'm going to give it dependencies that it's going to need to resolve using the MEF container. What this statement does, it tells MEF go ahead and resolve me. Because the RepositoryTestClass was not exported, it did not participate in the MEF building, in the container building. I am telling it explicitly here you probably have dependencies defined somewhere within the body of your code. Go ahead and resolve them if you can. Such a dependency would be the AccountRepository. Now, I want my RepositoryTestClass to use one of my data repositories. I think I've been saying the AccountRepository this whole time. What I actually want to use here is the CarRepository, so that's the one that I'm going to define, but I'm going to do it using its interface. And now I have to mark this so MEF knows to resolve it. (Typing) The last thing I'm going to do to this class is simply give it a method of usage, something that I'm going to call upon to use the CarRepository. (Typing) Now I've got a class that exposes a public method called GetCars, and this class is using the CarRepository to go out and get the information. Think of this class as a service class where it's exposing a service operation called GetCars, or it may be a business engine whose job it is to get cars and maybe act upon that collection before it sends it up to the top. Either way, it's going to be a client of the CarRepository, but at nowhere within this class do you see me instantiate this. This is classic decoupled coding. This is what you want at minimum in your application because now I can inject the ICarRepository in a production scenario. But if I want I to let it do its job through MEF and inject the actual CarRepository in a production scenario, I may not necessarily want to do that in a testing scenario. So, what I also want to do is give it one constructor here that receives an instance of ICarRepository. That way I can send a mock repository in, in testing scenarios, and you'll see a couple of scenarios in the next clip.

Repository Usage and Mocking

I want to show you two usage scenarios here, and I'm going to use a unit test to demonstrate both of them. The first one is an actual usage test in its production form, meaning it's going to use that RepositoryTestClass that I wrote, in all its MEF splendor, to have it work the way it would in a production scenario, and the other test that I'm going to write is one that is going to mock the repository, and then use the client class in a real unit testing scenario where I'm using a mock implementation of a dependency. So, the first one that I want to write is a unit test called test_repository_usage, (Typing) and this is going to be a TestMethod. (Typing) And what I want to do is instantiate the RepositoryTestClass using the default constructor because remember it's that default constructor right here that is going to tell MEF go ahead and resolve any dependencies that may be declared inside. (Typing) Then I'm simply going to call that public method (Typing) and perform a simple assertion that I didn't receive a null in return. (Typing) Now let's run this test and see if it works. And there we have it. It works fine. So, what I'm going to do is run it again in debug mode and put a breakpoint down here at the client method, and let's see what is getting there. And there's our breakpoint. And if I hover over CarRepository, you'll see that what it resolved to was an instance of CarRental. Data. CarRepository, which is exactly what I wanted because that's the class that got exported through MEF. And now you see how I can use dependency injection to inject my data repositories into any client that needs to use them. Now, in a real testing scenario, this is more of an integration test because I've tested actual database access here, but in a real testing scenario I may want to test the GetCars method without actually hitting the database, and that's what I wouldn't be able to do if I was not using dependency injection. If my method, if my GetCars method used the old fashioned technique of instantiating whatever class it needs to use at that time instead of receiving it as an injected dependency, it would be hard locked into that implementation of any of those classes. But because I'm not doing that, because I'm receiving the CarRepository through dependency injection, I gave it an additional constructor here where I can send any implementation of ICarRepository that I want, and that's what I'm going to write next. So, now I'm going to write a method called test_repository_mocking, (Typing) and this is also a TestMethod. Now, here I've got a little more work to do, but that's very normal when you start using mocking. Now, I'm not going to get into an entire tutorial on how mocking works. I happen to use the Moq framework, but there are a lot of good ones on the market right now. Two other good ones that come to mind are Rhino Mocks, and of course, JustMock from Telerik. And they all work very, very similarly. The idea is to create a test implementation of an interface without actually having to write that class yourself. Now, what I want to do is that I want to implement the ICarRepository's Get operation, and that is designed to retrieve an IEnumerable<Car>. So, the first thing I've got to do is I have to set all that up myself in a simulation-type scenario. So, I'm going to create a collection of cars myself. Then I'm going to create a mock for the ICarRepository. Then I'm going to define that mock and tell it that when you hear a call to the Get method of that mock, I want you to return my test list of cars. So, I'll do that now. So, there's my list of cars. And here I'm creating a Mock of the ICarRepository using the Moq framework. And then I'm telling it when you see a call to Get, return those cars. So, now that I've simulated everything that I want, I can then instantiate my TestClass again, (Typing) but this

time I want to use the constructor overload that receives an instance of an implementation of ICarRepository, which in this case is the Mock that I just created. (Typing) Now I can perform the call to the GetCars method and do my assertion. (Typing) And now I just want to assert that what I'm returning is the same as that mock list that I've set up. (Typing) And that's it. Remember in this test what I actually want to test is the true unit. That's why it's called a unit test. I want to test this unit right here. I want to test that GetCars is returning what it's designed to return despite the dependencies inside of it, and I've mocked that dependency, so I'm not really hitting the database. Let's run that test and see if it works. There we go. And, once again, let's run it with a breakpoint and see what we're getting. Now when I hover over CarRepository I should see not the Car. CarRental. Data. CarRepository class, but here it's a Castle. Proxies. ICarRepositoryProxy. This is the class that the Moq framework created and instantiated for me. And the great thing is that this class is not going to hit the database, so I'm not going to get any incremental IDs or any kind of database operation actually taking place.

Intro to The Data Repository Factory

Okay. Now let me set the scene up for the next clip, because this is going to be important. In a real world scenario where we're using, for example a WCF Service to call some data repositories, I may have a service that has 12 or 15 different operations. Each one of those operations may rely on a different data repository, a different table. Maybe some of the operations rely on more than one. Maybe some of them don't rely on any. The point is that I have four data repositories at my disposal here, and I may end up calling upon a service operation that only uses one of them. So, for the entire length of that one call using the pattern that I'm showing you right here, I would have to import all four repositories just so I can get to one of them from the operation method, and that sounds like a real waste of resources. In fact, it is. So, I want to come up with a different pattern, or should I say a pattern around this, where I only get the CarRepository if that's the one that I need, and I don't have to undergo instantiation of Rental, Reservation, or AccountRepository unless I need them. And if I use this pattern of declaring them right in the class, they're always going to get instantiated even if I only end up using one repository, and that's a real waste. Yet, I don't want to kill the fact that I'm using dependency injection, and I still want my testability, and I still want to have this separation of concerns and these patterns of abstractions. So, for that I'm going to use an abstract factory pattern next.

The Data Repository Factory Abstraction

If you're a pattern junky, then you probably already know that a factory class will retrieve a concrete class when given an abstraction. Well, an abstract factory does the same thing, but actually abstracts the factory out into its own interface so you can have more than one factory, and that's what I've put together here. What you're looking at here is in the Core. Common. Contracts project. It's an interface called IDataRepositoryFactory. And the job of this data repository factory or the class that will implement this is to do exactly what the name implies, to get me a data

repository. Now, as you can see, it defines one method. This method called GetDataRepository<T> is going to return a data repository. How do you know that? Because that's what the constraint says that T is going to be, an implementer of IDataRepository. Now, this is where the IDataRepository class without the generic has come in handy. If you recall, IDataRepository<T> is all I talked about for a long time, but it inherits from IDataRepository without a generic. Now, T here is an IIdentifiableEntity. I don't know what that entity is using the data repository factory, nor do I care. I just care that I return a valid data repository. That's why I needed a common abstraction that was not generic based, and that's what that interface is right here. Now what I need to do is come up with a class that implements this, and that's what I'll do next. So, I'm going to go down here to my CarRental. Data. Contracts, and I'm going to create a new class called DataRepositoryFactory. And this class is going to implement the IDataRepositoryFactory interface that I just showed you. (Typing)

The Data Repository Factory

So, now the job of the DataRepositoryFactory is to go out and find me a data repository. As long as what I'm sending it in T is an IDataRepository implementer, I'm going to be fine. And if you recall, all of my repository interfaces, for example, IAccountRepository, all come from IDataRepository<T>, which in turn comes from IDataRepository. So, in essence, all of my repository interfaces, which are the interfaces that apply to all of my data repositories, all fit the T constraint here. So, the usage of this method would say GetDataRepository of ICarRepository or of IAccountRepository or IRentalRepository. Now, rather than instantiate it manually here, we know that they've been exported through MEF. So, what I can do is that I can go out to the MEF container and obtain it this way. (Typing) GetExportedValue is just MEF's resolve. This is how you resolve something manually through MEF. So, all I've done here is that when something calls upon the GetDataRepository<T>, if T were, for example, IAccountRepository, then I would in turn go out and say go get me whatever class you have in your dependency injection container that is associated with IAccountRepository. And what is that going to be? It's going to be the AccountRepository class. So, now I have my factory class, which is going to serve up any repository that I ask for when I give it the interface of that repository. And because that repository factory is an abstract factory because it has an interface of its own, I can use that independency injection now like this. (Typing) And I just realized I made a mistake and I created the DataRepositoryFactory inside of the CarRental. Data. Contracts project. I really meant to create it in the Data. So, through the magic of Camtasia, let me move that over real quick. There we go. Now it's in the right project. (Typing) And now I'm exporting the DataRepositoryFactory through MEF, and I don't need to worry about my Bootstrapper because my Bootstrapper is already pointing at the CarRental. Data assembly by way of the AccountRpository type. So, it's going to find the DataRepositoryFactory and export that as well. So, now in the next clip I'll show you in my unit test class how I would use the DataRepositoryFactory instead of a straight data repository in a similar example to the previous one.

The Repository Factory Test Client Class

I want to set up another test class here. I'm going to call this one the RepositoryFactoryTestClass, and its basic structure is going to be very similar to this test class, so I'll do that first. There we go. As you can see, this class is pretty similar to the previous one; however, let's add the differences here. First of all, I'm not going to inject the ICarRepository into here. I'm going to inject IDataRepositoryFactory. (Typing) And, as before, for testing purposes later, I'm going to create a constructor that receives an instance of IDataRepositoryFactory that I can mock manually later on. (Typing) Now, the main difference is how I use this. So, whereas before I already had the CarRepository at my disposal, now I'm getting it on demand. Now I have an IDataRepositoryFactory dependency injection, injected the concrete DataRepositoryFactory into it, and now I can use that to go out and get a CarRepository by giving the repository factory the ICarRepository interface. So, I'm still benefiting from dependency injection in all layers here, and I'm benefiting from my interface usage. Think if this as kind of on demand repository obtainment. So, I defined what repository I want, and now I ask the factory to get it for me. (Typing) Just like that. And remember the DataRepositoryFactory that was exported through MEF, the production version of this factory class, is going to go to MEF to obtain the implementer of this interface. (Typing) And I'm done. And now I don't have to worry about injecting one or four or maybe in the future 100 different data repositories into this one client class that may have a bunch of methods where I may only need one or two of them for that specific call. I'd be really wasting a lot of resources implementing or instantiating all of those classes. So, now I just inject one factory class here, and on demand I obtain however many repositories I need. So, now let's write the two tests to actually use this, one of them in a production type of scenario using dependency injection, and the other one will be the mocking one.

Repository Factory Usage and Mocking

So, my first test class is going to be very simple. In fact, it's going to be virtually identical to the top one here. It's going to look like this. As you can see, all I'm doing is that I'm using my RepositoryFactoryTestClass instead of my RepositoryTestClass. Let's see if this works. (Typing) And there we go. Everything works exactly the way it did before, only no necessary instantiation has taken place. Now, in this case, I'm only injecting one repository, so there were no wasted resources, but in a live scenario I may be injecting several of them. Now I only need to inject the one factory class and that's it. Now, for my mocking tests, I have two ways that I can go about doing it. The first one is a pretty straightforward way. I'm basically going to mock the DataRepositoryFactory, and I need to set it up whereas when it receives a call to get a DataRepositoryFactory of ICarRepository and that call also includes a call to the Get method of ICarRepository return the cars list just like my previous example, and I can do that all in one shot. The next one I'll break it out into two steps, which will probably be a little easier to read, but it will be more code. So, let's do the first one first. Notice I'm building my list as before, but this time I am creating a mock of IDataRepositoryFactory, not ICarRepository. And then it's that mock that I need to set up. So, here I'm setting it up, and I'm saying when you call GetDataRepository with this and you're also calling Get off that data repository, so I'm doing it all in one shot here,

return the cars. And then I can create my client class and sent the mock into that overloaded constructor and call upon my operation. Now, the other test is just slightly different. It just does the exact same thing. But instead of doing this setup here in one step, it does things in two steps allowing you a little more flexibility because here I'm creating two mocks. I set up my mock for CarRepository as I did in my very, very first test up above, and I set up that MockCarRepository where when the Get is called, return the cars. Now, after I set up that mock, I set up another one for the RepositoryFactory, and its setup will say when GetDataRepository and ICarRepository is called, return the previously configured mock. Notice I'm not jumping right to the Get of that data repository, I'm saying just return the previous mock and let its setup take over. So, it's the same thing in just two steps, but now you can perform more complicated setups without having it all occur in one line and making it look like real complex code. And the rest of the test looks exactly the same. So, we should be able to run this entire class of tests, and it should pass hopefully. Let's pray to the demo Gods, all together now. And there we go. Alright, so we know how to use data repositories now. This is exactly the way they're going to be used later on from either a business engine or a WCF class, and we'll get into that in a future module. The next that I want to show you are those custom DTOs that you saw that I commented out earlier in this module.

Custom DTOs

A custom DTO or Data Transfer Object is nothing more than a wrapper class for more than one entity. This is so I can do something like this. This is one of the methods in the RentalRepository that I commented out earlier. And what I needed to do is write a method that retrieves information for a customer rental, a current customer rental. Now, that information spans more that one table, thus spanning more than one entity. It includes account information, it includes rental data, and it includes car data. So, I wanted to join all three of these things together, and I didn't want to do it in three different database calls. I could've done that. I could've made three calls from the service or the business engine down to the data repository, three different calls to obtain all this information, then package it all together at the service layer, but then I'd be hitting the database three times. And if I can avoid that I will. And it was simple enough to do it this way where I can just have one custom query that says GetCurrentCustomerRentalInfo and return a combination of the customer, car, and rental entity. And in order to return that information, I needed to create a class that wraps all three of those. So, I created a class called the CustomerRentalInfo. I'll create that now. (Typing) And this class, once again, is just a simple wrapper for three different entities. Now I can actually use CustomerRentalInfo in a Linq-to-Entities query like I do here. And that's really all I'm doing. As you can see, this is just a Linq-to-Entities query that does a couple of joins, and then builds upon the returned results using this class right here, CustomerRentalInfo. Now, if you're a big time user of Entity Framework, you'll probably know of one of its, some people call it advantage, I call it a pain in the butt, but Entity Framework does a lot of lazy loading especially in the area of collections. And lazy loading is meant to be used when you're going to keep the CarRentalContext active. And this is very common when you're using Entity Framework in a business context meaning you're using it directly from your

business classes or your services. I don't want to do that. I wanted to encapsulate Entity Framework usage, all Entity Framework usage in my data access layer so my services know about the data repositories through an abstraction, but they have no knowledge of actual Entity Framework usage. They don't know that it even exists, which means I have to get in and out of my context, my DbContext. I can't keep it alive. And if I try to return a collection through Entity Framework, it tries to do it through lazy loading, which means if I kill the context like I'm doing when it exits out of this using statement here, I'm not going to have access to the rest of that collection. So, I have to override the lazy loading. And to override the lazy loading, you basically have to take the resulting query and cast it to an array by sending it ToArray. And that finishes the loading of that collection and then makes it available so you can close the context variable, and you're not going to lose the collection at all. So, rather than repeat the ToArray, and you can't really do a ToArray from the query because query is an IQueryable, so what I can do is change IQueryable to a list, and then the list can be changed to an array. And so I didn't have to repeat two lists open close parentheses dot ToArray every time that I want to do this. I wrote an extension method in my core framework, and it's an extension method that extends IQueryable, and I call it to fully load it. So, all this is doing is doing a ToList. ToArray so it overrides the lazy loading. Now, if you recall, there was one more commented code piece here in the ReservationRepository. Here's the interface. Here's the one for Rental. And the ReservationRepository had two sections, and they require customer reservation information. So, I'll create that one now. (Typing) And that one is just a wrapper of the Account, the Car, and the Reservation entity. So, now back to my repository, I should be able to uncomment this now. (Typing) And there's that ToFullyLoaded that I explained earlier. And now I have a full data repository here called ReservationRepository that has the standard CRUD operations that come from the DataRepositoryBaseClass, it has all of the custom ones that I added myself, and it has an interface that it implements, which defines everything, the custom ones explicitly here, the common CRUD ones by way of the base interface.

Recap & Conclusion

So, to recap, I have a full-blown data access layer written now. It's got all Entity Framework usage completely encapsulated into the data access layer, so the outside layers don't know anything about it. Everything is abstracted. It only incorporates dependency injection making any client of it fully testable. The fact that I'm using an abstract factory is great because I now have on-demand repository obtainment, which means I don't have to inject a whole bunch of different data repositories into a client. I can just inject a factory and get them all as I need them. Clients only work with these interfaces whether they're using my factory or they may choose to use a straight data repository if they want. They're still using interfaces only, which makes them fully testable. Now, as I promised you, the Fluent API for Entity Framework Code-First is available via this link, and it gives you full documentation on everything the API has to offer, and you'll see that it's a thousand times more than the little examples that I've put on the screen, which is really all I needed to do for my application. So, in the next module, I'm going to be covering services and business engines, and you're going to see more of this data repository usage take place, so stick around.

Services & Business Engines - Part 1

Intro and Highlights

Hi. This is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. This is the first of the two modules that cover the service layer of the car rental system. In this module, I'll be covering the building of all the WCF Services including their service contracts on the business side, any custom data contracts that I might need, and I'll retouch on the business entities that we covered in an earlier module in the course. I'll also be using dependency injection within the WCF Services, so I'll show you how all of that is set up, and I'll be writing some business engines if the need arises. And those business engines will also be incorporated into dependency injection. During the building of the services, you're going to see a variety of different calls. Some of the calls are going to be data center calls meaning the only job of the WCF operation is to just retrieve data, and some of them require more complex behavior, sometimes reusable behavior, and that's where business engines will come into play. I'll also be using some patterns for reusability to take care of some of my common exception handling. And then in the next module I'll secure all the services.

Service Contracts and Services Overview

WCF is a really big topic. There are entire courses dedicated just to WCF. In fact, Pluralsight has such courses. I'm not going to be teaching you WCF entirely from scratch, but I am going to be building things using WCF from scratch, and then along the way I'll show you in detail what I'm doing and why I'm doing it. And if you're relatively new to WCF, I feel confident you'll still walk away with a relatively good understanding on why things happen and how things work. But the first thing I've got to say is that WCF is very explicit in its code meaning everything that you do must be explicitly declared starting with the service contracts. Now, service contracts are what define the API that your services are going to make up. Service contracts are the only thing that the client knows about. The client doesn't know about your services. It doesn't know about how it does things behind the scenes. It just cares about this API, what services are available, and what service operations are available. Now, as I stated during the architecture module, I'm going to have three services here, the Inventory service, the Rental service, and the Account service, and I'm going to have three service contracts, one for each. Now, services can implement more than one contract, but in my case I don't need to, and I want to keep it simple, so we're going to have a one-to-one correspondence on this. Each service contract is going to define all of the operations needed. This is, once again, based on my use cases, based on those actor tasks that I set up. What can a customer do? What can an administrator do? These are the operations that are going to be defined inside my service contract, and this is what is going to define the guts of the

API of my business tier. Now, remember our old friend the business entities? They're going to be used by the service contracts and the services. They're going to be returned. They're going to be subject to serialization and sent up the wire. Now, that's not always going to be the case. There are going to be a few situations where I'm going to have to write a custom data contract, and I will do so when I get to those cases. Now, a custom data contract can aggregate one or more entity, it can intersect one or more entity, or it can just plain build any custom information that I need to send up the wire. Basically whatever I need I create in a custom data contract if an existing entity cannot accommodate my needs. Now, because we have two sets of entities, and these entities, the business and the client entities are going to be used by the service contracts, and in a lot of cases are going to be returned by the service contracts, I need two sets of service contracts, one on the business side, one on the client side. But unlike the entities, they're going to be identical. They're just going to reside in two different sides of the wire, and they're going to be in two different namespaces. Now, once I have my service contracts set up, I can implement them in my services. And I use the term implement literally because service contracts translate to just interfaces in C# code, and services are just classes. And just like a class implements an interface, a service is set to implement a service contract. I am going to refer to it using this lingo though. I want to get you used to the service oriented lingo. So, you're not going to hear me say this class implements this interface in the case of a WCF Service. Instead I'm going to say the service will implement the service contract. Now, as I said, I'm going to have one service per contract, though I'm not locked into that. I can certainly have a service that implements more than one contract, and I can also have two services that implement the same service contract in a totally different way, and there's a lot of different scenarios for this, but I'm not going to be covering those because this system just doesn't call for it. I'm also going to be introducing a common base class. You heard me say earlier when I taught you entities that I'm a big fan of using a base class for most things whether or not I'm going to require something right away because it gives me a good placeholder. Well, in the case of services, I am going to have a use for a base class, and I will show you what I put in there when I get to it. I'm also going to be using dependency injection in my services as well. Now, for dependency injection in WCF I need to do things a little differently. I need to use something called post-construction dependency injection resolve. The reason for that is because my container is not what is going to be serving up my WCF Service. Remember we don't instantiate the WCF Service on our own. WCF's runtime does that. So, we need a way for WCF Services to resolve their dependencies from a DI container, but after they've been constructed meaning after the WCF runtime has already served them to us, and I'll show you how that works. I also need to incorporate fault-handling on every operation. Now, every operation is going to require some basic kind of fault-handling, and some operations will have more extensive fault-handling. But for the basic fault-handling, I want to try to do it in a reusable way, and I'm going to show you a trick using the base class and how I can accomplish this. And then lastly, we need to incorporate some kind of security. I'm going to be combining standard WCF authorization with a custom solution that I came up with or user- based data authorization. Remember that IAccountOwnedEntity interface that I showed you a couple of times? Well, this is where it's going to come into play again. So, next I'm going to start writing my service contracts and services, and I'm going to do so with the inventory service. I'm going to start

with service operations that merely do data access. You saw me put a lot of time into the data access layer. Now we're going to use this from the services to just retrieve and update information. Not a lot of custom behavior here. Later I will cover a more complex scenario, which will add more complex behavior to a service operation and thus incorporate the use of a business engine.

Inventory Service Contract

As far as projects go, I have two new ones here. I have the CarRental. Business. Contracts project where I'm going to store my data contracts and my service contracts, and then I have my CarRental. Business. Managers project where I'm going to store my actual services. Now, let me talk about nomenclature for a second. I've been using the word service over and over again, but you're going to see when it translates to C# code I'm going to name my service contracts using the suffix Service, but I'm going to name my services using the suffix Managers. I do this for two reasons. The first one is because services in essence are just that, they're managers, they're orchestrators. They basically manage all of the down level calls from the service wall all the way down the chain. But I also do it to differentiate between the service contracts and the services because in future cases I may have very good reason for implementing two service contracts from the same service in which case now I may run into a naming confusion here because I'm using the term service for both things, so I felt that having two different suffixes would probably be a smart thing to do. Now, I'm going to start by writing a service contract for my inventory service, so I'm going to call this InventoryService. And because it's going to be an interface, I'm going to use standard interface naming standards. So, I'm going to put those in my ServiceContracts here, so let's add that now. (Typing) Now, as I said earlier, I'm going to try to use WCF terminology here. So, right now what you're looking at is a regular interface. Actually, you're looking at a class. Let's fix that. (Typing) In order to make this a service contract, you have to decorate it as so, and you do that with the ServiceContract attribute that's in the System. ServiceModel namespace and assembly, so I'll add that now. Now, remember I told you earlier that we're going to have two sets of service contracts, so later in the course I'm going to write, or should I say copy over, a copy of this that's going to serve as my client-side service contract, but regarding the client and server-side service contracts, I'm going to run into the same namespace requirement that I had with my client and business entities. The namespaces have to match. And I told you that in the DataContract attribute I can do that with the Namespace property. I can do the same in the ServiceContract. Now, here it's very easy for me to just add it because I'm going to have both service contracts, that on the client side and business side decorated with a ServiceContract attribute. But in order to keep things consistent, I want to use the same trick that I used in the business entities and the client entities, which is the entry in the AssemblyInfo file. So, let me add that now. And there we go. Now, the CarRental. Business. Contracts namespace in C# terms, which is where my service contracts will reside on the business side, will map over to this URI namespace. And then later in the course I'll do the same thing for the client one, and that way they'll both match. Now, the two operations I want to write first for you are going to be very simple data retrieval operations. One of them is going to get a single car based

on the car's I'd. The other one is going to get all of the cars. In fact, you recall that I had something in place in the data access layer to do this. The data access layer has a car repository that had a couple of Get methods, and one of them got a single car based on an identifier. The other one got all of the cars. However, I'm not going to be exposing the data access layer to the outside. There's a service wall in place, so I'm going to basically write a simple service operation that will wrap the call to the data access layer and use the data repositories. But before I write that code, I have to define the API. I'm only in the contract right now, so I have to add a couple of things to the ServiceContract, and these are called operations. (Typing) Now, like the interface-to-service contract relationship, the same naming difference applies here. So far we have a ServiceContract that exposes no operations. In C# terms you can say that you have an interface that has two methods that are defined, but in WCF terms we're not exposing anything. WCF is explicit remember, which means just like I decorated the ServiceContract with the attribute, I have to decorate the operations with an attribute as well. (Typing) The last thing I want to mention here is that I'm not returning an enumerable or a list or anything. I'm returning an array. Now, WCF gives us a lot of flexibility. WCF gives us the luxury of working in C# terms, in. NET terms meaning I can very easily return a list of Car here, which is something everybody is comfortable and familiar with, but I like to be a little more explicit. It's just my own personal choice. Behind the scenes WCF is going to convert that list into something that anything that reads SOAP can understand. And we can't count on the client being a. NET client 100% of the time. And, of course, non-. NET clients don't know what a list of is, especially in terms of a generic. So, an array is what WCF will convert it to, and I just happen to like explicitly declaring it as what I know WCF is going to ultimately convert this to, which is an array of Car. So, that's my ServiceContract. Next, I'm going to write the inventory service that's going to implement the ServiceContract.

Setting up The Inventory Service

The first thing I'm going to do is add a class to my CarRental. Business. Managers project here called InventoryManager. (Typing) Now, before I implement my ServiceContract and start writing all the implementation code, let's set up this class first for things that we anticipate are going to be needed, the first thing being access to our data repositories. Now, if you recall from the data access layer module, I'm not going to be injecting data repositories directly into my client classes, in this case my services. I'm going to be using that abstract factory pattern, which manifests itself in the way of the DataRepositoryFactory, and that is what I want to inject here so then I can access any data repository that I want from any of my operations. Remember, I'm going to have several operations by the time this service is done, and I may require the use of more than one data repository. However, not every operation is going to require every single one, so why force the dependency injection container to instantiate every single one just for the call of one service operation. That's why I have the DataRepositoryFactory. So, I'll add that now. (Typing) And in order for this to be instantiated by MEF, I have to import it. Now, as I told you at the beginning of this module, in order for our WCF Services to incorporate dependency injection, they have to somehow be able to resolve their dependencies. Normally if we ask a MEF container or any other dependency injection container to give

us a class like InventoryManager, as long as it's provided by the dependency injection container, it will automatically resolve recursively all the way down the chain everything that it needs. Now, in the case of WCF, MEF cannot service the InventoryManager class. That's the job of WCF. WCF's runtime will instantiate this service, so I have to tell MEF to resolve this service after it's been constructed by WCF. That's what I meant earlier by post-construction resolve. This is not a feature that is supported by every single dependency injection container out there. MEF is not the only one that supports it, however, but it works nicely in this case. So, what post-construction resolve is the ability to tell MEF after the class has been constructed already now go resolve any dependencies or any imports, to use MEF terms, that you need to. Now, the only constructor that WCF is going to worry about is going to be the default constructor. (Typing) And this is where I can tell MEF to satisfy these imports. Now, if you recall when we did the unit test for the data access layer I did this already, and I think I explained back then. What I want to do is access my container. And if you remember, I have a static variable called Container in my ObjectBase class, and I can tell it satisfy the imports here. (Typing) And now when WCF instantiates the InventoryManager, it's going to go ahead and resolve anything that is tagged with the Import attribute, in this case the DataRepositoryFactory, the same way my unit test did back in the data access layer. Now, like that TestClient class that I wrote in that unit test back in the data access layer, I want to be able to test this InventoryManager class and mock my own version of the DataRepositoryFactory for testing purposes. So, I want to give it another constructor just like that. Now, WCF will not care about this constructor. It doesn't even know it's there. It only cares about the default constructor. This one is merely there so that I can instantiate the InventoryManager class manually from a unit test and send another DataRepositoryFactory, possibly a mock version of one in for the purposes of a unit test. Now, let's got ahead and implement the IInventoryService service contract. (Typing) And there's the implementation of my two operations. Remember I'm using service oriented lingo here. There are operations now. They're not just methods. Next, I'll write the code for the implementation of these operations.

Fully Implementing The First Operation

I'll start with the GetCar operation, and this is very simple, as is the next one, the GetAllCar. They're simply wrappers for what we already have in the data access layer. Remember the CarRepository? It has a couple of Get methods. In fact, they're both called Get because they're overloaded. One of them receives a CarId as an argument and retrieves a Car entity. The other one retrieves an IEnumerable of car entity meaning all cars. So, I'm going to wrap that call to the data access layer in the service. So, the first thing I've got to do here is that I have to obtain my CarRepository, and I do that from the DataRepositoryFactory like this. (Typing) See how handy the DataRepositoryFactory comes in now? We don't have to worry about instantiating it. It's taken care of automatically using dependency injection. We can go ahead and assume that it's there and start using it. Now all I've got to do is get a carEntity like this. (Typing) And it can be as simple as just returning it up the call. (Typing) This can be as easy as this. In fact, I can almost consider this operation complete except for the fact that I don't have any exception

handling whatsoever here, and the problem with a WCF Service is that if it throws an unhandled exception, it's going to fault the proxy, it's going to fault the client, and we don't want that. We want every exception to be properly handled. Now, I'm not going to do the common exception handling just yet. I want to do something a little more specific because what I want to do is test this return carEntity here for null. And if it is null, I don't want to just return a null up the chain. I want to actually throw a specific exception, and I have custom exception already set up called NotFoundException, and that's down in my Core. Common project here. (Typing) One of the rules that we're taught when we're learning C# is don't write a custom exception if an existing exception that the system provides for you will do. This is a decision you have to weigh our yourself, and I found that having a NotFoundException, which I use quite often, actually has some value. So, what I want to do is test this carEntity for null, and if it is null, then I'm going to throw that exception. (Typing) Now, here's where things get a little different from what you may be used to. In the world of WCF, you don't really throw an exception because remember there's a physical separation between the server and the client, so it's not like you can throw an exception that the client will automatically catch because the client is on a different machine on the other side of the world. What you've got to do is wrap this in something called a SOAP vault, which will get transmitted through the SOAP message, and then the client will know how to handle that later. WCF allows us to do this with something called a FaultException of T. What we've got to do is throw a FaultException of T, T being this exception that we're defining here, the NotFoundException. (Typing) Now we're throwing something that WCF is compatible with and that a client will be able to trap specifically and translate back to a NotFoundException. Also, throwing a FaultException of T will not fault the proxy, which means that proxy will still be usable if the need is there. Now, I mentioned earlier that we're going to have some common exception handling or common fault handling to be specific of the terminology. Basically what that means is that I never ever in the WCF operation want an unhandled exception to go up the call because that will fault the proxy, and you'll get undesired results. What that means is that I want to have a try/catch around all of the code in every operation and catch just the exception class so that then I can instead of throwing an unhandled exception, I can throw a specific FaultException. So, what I want to do here is that I want to wrap all of this in a try/catch. (Typing) And now in the catch I can throw something called a FaultException. Now, this is not a FaultException of T. This is just a regular FaultException like this. (Typing) Throwing a fault exception will also prevent the proxy from being put into a faulted state, and it gives us a clean approach to handling false here in a way that's SOAP likes. Now, if you look closely at this code, you may pick out the fact that there's one little flaw in it. If NotFoundException is thrown, it's going to immediately fall into the Exception catch. We don't want that, so I have to add one more little piece of code to this. FaultException of T derives from FaultException. So, I'm going to add one more catch here, and here I'm just going to rethrow it. That way any FaultException of T's that I throw in my code will always be caught here and then rethrown, and they're never fall into this. They'll actually be thrown as a FaultException of T, which is what I want. Now, later I'm going to show you a reusable approach to doing what I just did, but for right now, this operation is pretty much complete, it'll work, and it's also clean and elegant because it does exactly what it's supposed to do, and it does it cleanly. It throws exceptions where it's supposed to throw them, and it doesn't let unhandled exceptions go up the chain, so it's a well written

operation. Next, I'll implement the GetAllCars operation, and then I'll also show you some characteristics that we have to give our service.

Implementing A more Complex Operation

The GetAllCars operation is going to be just a bit more complex, but still pretty simple. But either way, it's going to start with me needing the carRepository, which I'll just grab from here. Now, I'll obtain the list of cars. (Typing) Now, I can just return it just like this. (Typing) But why leave things that simple? We can make it a little more complex without a lot of effort and make it a little more realistic. Now, if you remember, when I showed you the data access layer, I wrote a DbContext class. I'll go to that class now. And I gave it some ORM rules down here. And I told Entity Framework ignore the CurrentlyRented property when dealing with the carEntity. That's because the carEntity itself has a property called CurrentlyRented. (Typing) However, this is not a database column, so I don't want it mapped over using Entity Framework. But now I'm going to show you why I have this property here. What I want to do is obtain a list of all the cars that are currently rented in this operation and then go through all of the cars, and any car that is currently rented, I want to tag this property with a value of true. That way, when the caller receives the result of this operation in a list of all of the cars in the system, he or she will know which of these cars are rented or not, and this will come in handy later on. So, in order to retrieve the list of all the rented cars, I need to obtain my RentalRepository like this. (Typing) Now, I need to obtain the list of rentedCars. (Typing) And as you can see I have a data access layer method already set up for this. Now, I'm going to iterate through each of the cars in the main list. (Typing) And now I'm going to obtain an item in the rentedCars list whose CarId matches the car in the current iteration. (Typing) If rentedCar is not null, I'm going to consider the car in the iteration rented and tag the property appropriately. (Typing) And the last thing I want to do is the same common fault exception handling that I did in the first operation. (Typing) And my operation is complete.

Setting Service Characteristics

So, now I've got a couple of operations here, but I want to show you something else. I want to show you how to set the characteristics of this service to what I like to be my staring point characteristics of every service. Now, by default, WCF when it finds that this service is not decorated with any specific service behavior, is going to instantiate this service in what's called per session mode, which means the lifetime of the service is going to equal the lifetime of the proxy. This is going to be equivalent to conventional object creation, which means so long as a proxy is open, every call to the service happens on the same instance of that service. That way you can maintain class level state. However, it's not the most scalable mode because the instance of the service is left in memory so long as that proxy is still open, and you never know what kind of work the developer is going to do on the client side. They may leave the proxy open for a long time unnecessarily. Even though I will not be doing so in our application, you can't trust that of other

developers. So, what I'd like to do is introduce session to a service only if I need it, and I don't need it here. So I'm going to turn this service into what's called a per call service, which means every call is going to happen on a new instance of the service. And after that call is complete, the service is going to be disposed, which means memory is going to be freed up. I do that with the ServiceBehavior attribute like this. (Typing) The next thing that I want to do is set up concurrency handling for this service. Now, I don't have the need to allow multiple concurrency in this service, and I don't want to introduce the complexity of any kind of locking. I just don't have the necessity here. So, I can leave the concurrency mode as single. However, because we're safe with the instance mode meaning we don't have to worry about concurrency because every call occurs on a different instance of the service, I do want to set the concurrency mode to multiple because if it's left to single, WCF will handle one call at a time even though it's servicing each caller with a different instance. It will not do it concurrently. So, by setting it to multiple, I can service the calls concurrently without worrying about multithreading problems because they're all going to be serviced on different instances anyway. (Typing) The last thing that I have to do is set one more property of the ServiceBehavior attribute called ReleaseServiceInstanceOnTransactionComplete. It defaults to true, and I need to set it to false because of the combination of instance mode and concurrency mode that we have. When I introduce transactions later on in the course, you're going to see that as soon as this service is hosted it's going to give us a warning or should I say an error saying we need to set this property to false. WCF is going to want to equate the transaction boundary to the instance boundary, and it shouldn't matter because we have a per call situation, which means we're going to drop the instance as soon as the call is done, but without setting this flag, it's still going to bark at us. (Typing) And that's it. That's all of the service characteristics that I need to set for now. So, next I'm going to introduce a base class for this service and show you how to offload some of the stuff that you're seeing here into the base class so that every service can benefit from it including default handling that we've done here so that every operation can benefit from it without us writing this much code in every single operation.

The Manager Base Class

There are quite a few things in this service that I'm going to be repeating in another service in this application, and I'd really like to get some reusability here, so I'm going to offload some of this stuff into a base class from which all my services will inherit. Because my services are referred to as managers here, I'm going to refer to this base class as the ManagerBase class, and I'm going to create it here in the same assembly where the services are, so I'll create that now. (Typing) Now, first let's go back to the InventoryManager service and identify some of the things that I think should go in this base class. The first one that immediately pops out to me is this right here. As I explained earlier, this is the line that tells MEF, the Managed Extensibility Framework, to resolve the dependencies for this class after the class has been constructed. Obviously, this is in the constructor of the service. So, WCF will still serve the service up for us, and then MEF will resolve its dependencies; in this case, IDataRepositoryFactory. So, I'm going to offload this to the base class first. (Typing) And just to secure myself against unnecessary errors, I'm going to check this container

for null first. (Typing) And now I can go back to the InventoryManager and inherit from this class. (Typing) The other stuff that I can offload to my base class is my error handling code down here. Now, not all of this error handling code is going to be offloaded to the base class, only the general stuff, this part right here. The specific things to the operation are going to remain in the operation, but I am going to offload most of this to the base class because then what I'm going to do is write a couple of base class methods to which I can send these pieces of code right here in the form of a lambda expression very similar to the lambda expression technique that I showed you during the object walk section earlier in the course. Then I can take this code and invoke it within a try/catch that catches these things here. That way I can reuse that technique for all of my operations, and I won't have to repeat any of this in any of my other operations. Now, I'm going to call my base class method ExecuteFaultHandledOperation, and I'm going to end up with two versions of it, one that returns a type, and one that does not. That's because I have two types of operations. Now, right here I only have an operation that returns a type as I do down here in GetAllCars, but as this service grows, I'm going to have operations that are actually void return type, and I'm going to need to account for them as well. So, I'll put these operations or these methods in my ManagerBase class. (Typing) The argument that I need to receive here is a Func of T because it's going to be a piece of code, a code snippet by way of a lambda expression that's going to return T, and that's what Func does. (Typing) And to actually execute the code coming in and return the type, I simply have to invoke it like this. (Typing) Now, I'll put the error handling code in here in a second, but first let's go back to the InventoryManager and actually make a call to the ExecuteFaultHandledOperation method. So, first I'm going to remove my error handling code from here, and now I'm going to wrap this code in a call to execute FaultHandledOperation. And because it's the version of that method that is expecting a return value, I'm going to return something here. And I don't have to tell it the generic because that can be inferred. Now I can go back to the ManagerBase and wrap this invoke in the try/catches that I need. (Typing) And as you can see, this is the same kind of error handling that I had in the actual service operation back in the InventoryManager class. So, now I can feel perfectly comfortable going back to the InventoryManager service or any other service for that matter and adding as many operations as I want so long as I follow this pattern because my code will be safe from any exceptions that are thrown, at least in a general sense. Any specific exceptions I would still have to handle within the operation just like this. But there's still one piece that's missing in order for me to actually be able to throw FaultException<T> and have it received up to the client, and that's what I'm going to talk about next.

Adding The Fault Contracts to The Operation Contracts

WCF likes explicitness. It likes to be notified in advance of what it's going to need in order to do its job properly, and the raising of false is no exception to this. So, in order for me to be able to raise a fault exception of NotFoundException like I'm doing right here, I need to tell WCF in advance about the NotFoundException. That way it knows how to properly serialize it, and that's done by using the FaultContract attribute in the ServiceContract. So, I'll go there now. And what I need to do is add to the OperationContract any types that are going to need to be

serialized. Now, when dealing with exceptions, this is done with a FaultContract attribute, but in a way it's really the same thing as using the ServiceKnownType attribute when you're dealing with contract inheritance, which we are not in this case. So, I'm just going to add one FaultContract attribute here to notify WCF that the GetCar operation may potentially use a NotFoundException so it knows how to properly serialize it and get it up to the client. (Typing) If I was using any other specific exception type in the InventoryManager's GetCar operation, which as you can see I am not, I would also have to list that out in the ServiceContract, which means you can stack the FaultContract attributes and have as many of them as you need. Now, as you can see, I'm not using any such specific exception in the GetAllCars. I'm just relying on this catch all down here, so I don't need to add any FaultContract because I'm not sending up any specific one as I am here. So, think of the rule of thumb as if you're going to use FaultException<T>, T will always need to be declared in a FaultContract on the ServiceContract side. If you're just using FaultException by itself, then you don't need to declare anything in advance. I'm going to quickly add a couple of more CRUD operations to this InventoryManager service that are going to allow me to update a car and also delete a car. Then afterwards I'm going to add another operation that's going to introduce us to the concept of a business engine.

Adding CRUD Operations and Transactions

Before I add another operation to my service, I have to, of course, make sure that I add the OperationContract to its ServiceContract. The first two operations I want to add are, once again, very simple CRUD operations. They basically are going to wrap some of the CRUD stuff that we have in the data access later because remember a client will never ever see our data access layer. It's always going to go through a service. These two operations are going to be called UpdateCar and DeleteCar. Now, you notice something that these two OperationContract's have that the other two do not, and that is the addition of the TransactionFlow attribute. This is new. Because these are output operations, output in the context of I/O, input/output, they would require transaction handling. Now, to be specific, transaction handling would probably not be required in this case because I'm really only doing one database update. Transactions are important when you're doing more than one database update so that if one piece of it fails, everything is rolled back, and the system remains in a consistent state. However, I like to be safe, and any kind of output operations that I write I introduce transactions. It makes me feel more comfortable because somebody may come later and add something in the data access layer without touching the service, and maybe in the future the task of doing an UpdateCar will involve more than one output operation. Because I designed this system, the chances of that happening are, of course, very minor, but you want to be safe anyway. So, my rule of thumb for transactions is I do them on all non-fetch operations meaning all output operations. Now, it could also be argued that transactions would be needed in Get operations if we were to be doing something like audit trailing or logging or something like that, but that's not the case here so I'm not going to bother with the Get operations. So, there are two attributes that I'm concerned with. The first one is called TransactionFlow, and all this does is that it tells this operation if a transaction happens to exist already meaning it was started before this service call was initiated, in this case

probably by the client, but it also could be by another service, then I want that transaction to be allowed to flow into my UpdateCar or DeleteCar operation. That way that operation can participate in that transaction and vote on it, but that's all this attribute means. I still haven't told the actual UpdateCar operation to use a transaction. I am merely allowing the existence of an existing transaction. So, let me go into the InventoryManager service and write the UpdateCar operation. As you can see, I am following the same practices that I followed earlier. In fact, I never did change this one did I? Let's change it real quick. (Typing) There. Now the GetAllCars is also doing its regular exception handling. So, back to the UpdateCar. As you can see, it's just instantiating an ICarRepository instance from the DataRepositoryFactory, and it's doing an Update or an Add depending on the existence of the CarId property from the incoming entity, so I can leverage this method for both situations. Now, in order to make this code participate in a transaction, whether it is an existing transaction that came in or a new one that this operation will start on its own, I have to decorate it with the OperationBehavior attribute like this, and I have to set the TransactionScopeRequired property to true. And believe it or not, that's it. We are now transaction friendly on this operation, which means if an existing transaction already exists when the call to UpdateCar executes, this method will vote on that transaction, and then, of course, the root that started the transaction in the first place will tally up all the votes. If there is no existing transaction, then the execution of this operation will be done in a new transaction, and it will either commit or rollback depending on the existence of an exception. Now, I'll quickly add the DeleteCar operation, which will also be transaction friendly. Now, you'll notice that this is failing, and that's because I only gave my base class one operation call, ExecuteFaultHandledOperation, or should I say one method in this case. And that's the one that returned a type, a type of T actually. I now have to give it one that does not return anything so it can handle this void. (Typing) And now the InventoryService should compile fine. I have one more operation that I want to add to this service, and that one is called GetAvailableCars, and it looks like this. As you can see, it retrieves the list of available cars based on a specific pickup or return date because a car might be rented or reserved for those dates. So, now this operation is going to have a little more intelligence, and some of that intelligence may be reusable in the future, so that's where business engine is going to come into play next.

Adding A Multi-Data and Behavior Operation

So, let's start writing the implementation for the GetAvailableCars operation by going to the InventoryManager of course, and we'll take this step-by-step. I'm going to have an operation called GetAvailableCars that, of course, returns an array of Car like this. And if you remember from the OperationContract, the arguments were pickupDate and returnDate as a DateTime. Now, following the same pattern that I have above here, I'm going to return a call to the ExecuteFaultHandledOperation, and then put the meat of my code in there. (Typing) So, in order to get an accurate list of the cars that are going to be available for rental between a certain pickup date and a return date, I have to exclude those cars that are rented or reserved for those dates. So, the first thing that I've got to do is retrieve three different pieces of information. I've got to retrieve all my cars because then I'm going to filter that list of allCars

down based on two other lists, the list of rentedCars and the list of reservedCars. And I'll do that first by instantiating three different repositories, the carRepository, the rentalRepository, and the reservationRepository. Now, if you recall, we put together the DataRepositoryFactory specifically for this kind of situation. So, instead of having to import all three data repositories, which are going to be used in their entirety in this operation, but not in any other operation, I've only imported that one factory class, and now I can obtain on demand all three of them like this. Let's switch the screen to full screen so we can see a little better here. There we go. So, first let me get all of my cars using the carRepository. (Typing) Now, let me get the list of all of the currently rentedCars by using the rentalRepository. (Typing) And finally the list of all currently reservedCars, which comes from the reservationRepository. (Typing) Now, I'm going to create an empty list called availableCars, and that's what's going to contain my filtered down information. And in order to filter down the list of allCars, I need to iterate through it of course. (Typing) And finally I'm going to return the availableCars as an array. (Typing) So, it's going to be in here where the magic is actually going to happen, and some of you may already be arguing why am I doing this with three different calls to the database? That's fine. That's a valid argument. You can probably accomplish this very thing in five different ways. I chose to go the simplest route that served the purposes well, and I was able to demo properly here on the screen and also bring in the use of my next piece of technology, which is the business engine. Now, what I have to do to each one of these cars is find out if it is available for rental individually, and that information is going to be based on the pickupDate and the returnDate, and then compared to the list of rentedCars and reservedCars based on those pickupDates and returnDates. Now, when I first wrote this service, I wrote the code that I just described right here between this foreach block, and later I found that it came in handy in a couple of other places. And it's that kind of behavior reusability that leads me to use a business engine, and that's exactly what I'm going to do next.

Intro to Business Engines

Let's talk about what we're going to write next, which is a business engine. These classes are more fine-grained than services are. Services are managers or orchestrators if you will, which means that they wrap more than one area of functionality or area of volatility depending on how you decompose your system. I tend to use business engines when I find myself needing to reuse behavior. Now, you've noticed that in my services most of the operations that we've written so far involve data access and not much more complexity than that. I am going to, however, be adding an operation now that's going to involve a little more complex behavior, and I want to make this behavior reusable, so I'm going to have a business engine for it. I also want to make everything as testable as possible just like I did before. So, just like I was injecting data repositories into my services, I'm going to be injecting business engines into the services as well. So, it's going to be the exact same pattern. Not only am I going to set the business engine up so that it is subject to dependency injection, but I'm also going to set up an abstract factory pattern around its instantiation for the exact same reasons that I did in the data repository. So, enough slides. Let's get back to the code.

Setting up A Business Engine

I basically want to perform a check on each car, and if that check is successful, I'm going to add the car to the list. So, I'm going to write a little pseudocode right here just to put something in place so that I can compile and also explain to you what's going to happen. (Typing) And what I'm actually going to do here is that I am going to use CarId in conjunction with the two dates, as well as the two other lists that I retrieved. And the information here in all five of these different pieces of information combined will determine whether this is car is available for rental or not, and that's why I want to put in another piece of code, something that I can reuse later. A business engine is significantly more fine-grained than a service. A service is called a manager because it is an orchestrator. It is a wrapper of a lot of functionality that need to be handled all together within the context of one call whereas a business engine is very specific to the behavior that it is handling. In this particular case, I need to handle behavior that is very specific to the rental of cars or the information regarding rentals of cars, so I'm going to call this the CarRentalEngine. And where I put my business engines is in the CarRental. Business project here. So, first I'm going to Add a Class here called CarRentalEngine. Now, I'm going to write the skeleton for the method that I want. Notice I'm calling it a method, not an operation. An operation belongs to a service. This is just a business method of a business engine. It's going to be a simple bool, yes or no. I'm asking it a question. And I'm going to pass into it those five pieces of information that I showed you in the service behind the comment. (Typing) I'm obviously going to have a Boolean here. I'll call it available, and, of course, I'm going to return it. (Typing) So, now it's a matter of writing all of the business logic that's involved in using all of this information here in arguments in order to obtain the value of available. I can divide the bulk of this code into two primary chunks. One of them is a check against the reservedCars list, and the other one is a check against the rentedCars list. So, I'll do the reservedCars list first. Basically a Linq query. I'm not using Linq syntax. I'm using extension method syntax here, but it is in essence a Linq query going into the reservedCars list and checking all the information that I need including the specific car in relation to the rental and pickup dates and how they relate to the reservation rental dates and return dates that are on file. Now, I'm going to do something similar with the other list, the rentedCars list. Now, I only need to do the next check if available is still set to true, which means nothing was found here, which means the car is still available. So, now we check it against the rentedCars list. And now we've encapsulated the behavior for checking if a car is available for rental in a business engine. I chose to do it this way because it was simple and because I went ahead and I retrieved those lists from the service and sent them in here as arguments. I could've chosen to retrieve data right from the business engine and made the call from the service simpler. It really is six in one hand, half a dozen in the other. I could actually provide and overload for IsCarAvailableForRental where it does not receive these lists here, and instead it goes to a data repository and obtains the information. In order to use this business engine from the service, we need to do something very similar to what I did with the data repositories, and I'll do that next.

Setting up The Business Engine for DI

Like the case of my data repositories, I'm going to be injecting this CarRentalEngine into anything that uses it, and that's InventoryManager, which means first order of business is to abstract out what I've implemented here, which means just like I have a CarRentalEngine class, I'm going to need an ICarRentalEngine interface, and I'm going to put that in the CarRental. Business. Common project here. (Typing) Now, I simply have to provide a definition for IsCarAvailableForRental in the interface. (Typing) I'm now using CodeRush. I can do that easily without having to type it. And now if I go here to the ICarRentalEngine, you'll see that I have the definition. And now in order to use the CarRentalEngine in dependency injection, I, of course, have to export it through MEF. Remember this is the equivalent of doing an association or a registration using another type of dependency injection container. And remember because MEF defaults all resolving to the singleton pattern, you have to tell it explicitly not to use a singleton pattern. (Typing) And lastly, due to the nature of the discoverability mechanism on which MEF relies on, exporting is not enough. We have to tell MEF to look at the assembly that the business engines reside in. And remember that was done in the Bootstrapper. So, I'm going to load up the MEFLoader here. And as you can see, here I'm telling it look into the assembly in which the AccountRepository resides, and, of course, it's going to find every other repository there. I just needed one class that can point me to that assembly. Now I need the same thing to get MEF to look at the CarRental. Business class. (Typing) And that's it. So, now when my Bootstrapping process takes place and I tell MEF go out and discover anything that has been exported, I'm not only telling it to look into the CarRental. Data assembly, but also the CarRental. Business assembly by way of the CarRentalEngine class. And any future engines that I add that reside in that same assembly, which are also exported and associated to their corresponding interfaces, will be found as well. And there we go. We're ready to use the CarRentalEngine from any class that needs it by injecting it into that class. And, of course, the class that I'm talking about here is going to be my InventoryManager. So, now I need to make a call to the business engine and pass this stuff in, but first I need to instantiate the business engine. And, once again, it's a no-no to instantiate it within this method here because then I'm going to be tied to that concrete implementation, and I'm also tied to that instantiation. And like before, I want to be able to unit test this operation and mock any dependencies. And that includes not only the data repositories, but any business engines as well. Now, going up here to where I do my import, I could chose to do things the more traditional way and import the ICarRentalEngine interface that you just saw me write, but we have the exact same problem. What if in the future we have a large number of business engines that this service needs to use, but not all of them at once. Why go through the process of instantiating all of them just to end up using one or two? So, I'm going to implement the exact same abstract factory pattern that I did before. So, if I go out to my Core. Common. Contracts, you'll see this is where I put the IDataRepositoryFactory interface earlier. I also have an IBusinessEngineFactory interface, and it looks virtually identical to the data repository one. The difference is the constraint is different. The constraint is going to constrain me against only a business engine, which means I need a common interface from which all of my business engine interfaces inherit. So, first I'm going to go back to the ICarRentalEngine interface here and inherit it from IBusinessEngine. Now, I need to write an implementation class of this interface that does the same thing that the DataRepositoryFactory class was doing using MEF to go out and get a business engine from the

container. So, just like earlier in the data access layer module I put the DataRepositoryFactoryConcrete class in my project's data access layer, I'm going to follow that same pattern and go to the CarRental. Business project and add my BusinessEngineFactory there. (Typing) And this class is going to inherit from IBusinessEngineFactory. (Typing) And its job is to, just like its cousin the DataRepositoryFactory, go out to the container and get an exported value. (Typing) Now, I'm going to go back to the InventoryManager and actually use this engine next.

Using The Business Engine from A Service

So, here I am back in the InventoryManager service class, and I need to do an import of that IBusinessEngineFactory in the exact same way that I did the IDataRepositoryFactory import. So, first I'll add it here. (Typing) Then in anticipation of my unit test's needs later, I'm going to add some overloaded constructors. The same way that I added a constructor that accepts a mock of an IDataRepositoryFactory, I'm going to add one that accepts a BusinessEngineFactory as well. (Typing) And now I wouldn't be complete if I didn't have an overloaded constructor that accepts both in the case of a unit test that needs to send into an operation a mock of an IDataRepositoryFactory, as well as a BusinessEngineFactory. (Typing) So, now that I have all this set up, let's go down to our latest operation and actually add the code to call that business engine. So, just like I was obtaining data repositories on the fly here, I need to do the same thing for the business engine like this. (Typing) So, you can see the pattern is exactly the same as the DataRepositoryFactory pattern, and I would repeat this for any business engines that I need if there are more than one. Now, I just have to change my pseudocode here to actually call upon that engine. (Typing) And now I am making a simple call to the business engine to determine the availability of a specific car, and all of the business behavior that determines that availability is now available for reuse in that business engine.

Recap and Summary

So, you saw quite a bit in this module, and there's still more to come regarding services. I covered the building of all the service contracts and services in detail, I covered several different types of calls, and we also built the base class to handle a lot of the common functionality. Some of this common functionality included how to handle dependency injection in WCF. Also, whenever we came across behavior that needed to be reused, we built business engines for it and had those business engines inject it into the WCF Service and also rely on the data repositories. So, all we have left to do now is add some security to this and, of course, do some unit testing, and that's going to come up in the next module.

Services & Business Engines - Part 2

Intro and Highlights

Hi. This is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. In this module, I'll continue working on the service and engine layer by adding security and unit testing. I'm going to be covering security in two different forms. The first one is standard WCF-based security. And then I'm going to take it up a notch and do something very specific called user-data authorization. This is where we're securing the data on an already authenticated user so that they cannot access data that may pertain to another user. After I'm done with security, we're going to get into unit testing, and I'm going to be unit testing both the business engines and the WCF Services.

Security Intro

Security for these services will take two different shapes. The first one is your standard principle and identity-based security. The authentication scheme will be simple, regular Windows-based authentication, which means the client that's calling the service will be authenticated based on the same domain that the host is running under. Now, despite the fact that one of our clients might be a web application, and in fact it will be later, this is all still internet-based security that we can use because in the desktop application we're running a desktop client inside the firewall, and the user of that desktop will already be authenticated onto the domain that the host is running on. And the website IIS is the true client that's the true caller of the services, and that is also inside the firewall. So, in both cases, we're dealing with internet-based security, and we can use Windows users and roles. Now, authorization is going to be just a bit more complex even though it's still going to use Windows group or role authorization. I'm going to be setting up a special Windows group for the admin functions. And the admin functions are those functions that are going to be used from a desktop system. Basically, from a business perspective, it will be the car rental employee that is sitting behind the car rental desk. So, any user of this potential desktop application will be a member of a very specific Windows group, and that's how I'm going to be authorizing certain calls that should not be open to other applications. Non-admin functions will run under a specific user. That way, I can actually assign the web application or any web application to run under a specific user. I can tell IIS run under this user, and then authorize WCF for that specific user only in certain calls. Such a user will not be a member of the special group that I mentioned earlier. That's because admin functions will not be accessible to web users, for example. A web user should not be able to rent a car to another customer or accept a car return. These are all administrative functions that only a car rental employee sitting behind the car rental desk should be allowed to do. So, none of those non-admin functions should

be open to any other application other than an administrative application, whether it's the desktop application that I'm going to write or some other one in the future. Now, for both of these different things, either authorizing against a specific group or authorizing against a specific user, we're able to use the PrincipalPermission attribute, and that's exactly what I'm going to do. So, I'm going to set all this up first in the code, show you how it all works, describe it in detail while you're looking at the code, and then I'm going to get into the other side of security, which is going to be user data authorization later.

Adding Security to A Service

Now, the first thing that I want to bring to your attention is that user and group that I created ahead of time. As you can see, this is just the basic computer management screen in Windows, and when I go to the list of users that I have, I have a user here called CarRentalUser, and when I go to the list of groups, I have a group down here called CarRentalAdmin. So, now I have a specific user setup that I can run applications under, and I also have a specific group set up that I can add users to that would use applications, and these would be admin users. So, now let's go to Visual Studio. As you can see, here I'm in the InventoryManager service, and what I want to do is identify each operation and determine who should have access to that operation. So, for example, in those first two operations that I wrote, the GetCar and the GetCars operations, let's go to them now, I want all my users to have access to these two whether they're coming from an application that is open to non-admins or if they're coming from an application that is open only to administrators. Note my terminology there. I haven't yet determined if I'm going to write a desktop app or a Windows app, a web application, I don't know what kind of technology is going to be used. All I know is that I'm going to have admin scenarios and non-admin scenarios, and I want to accommodate the GetCar for both scenarios. So, anybody that is in the car admin role or car rental admin role should have access to this operation. And if the user is specifically a car rental user, I want to have access to this specifically. Now, in a more complete system, I would probably have yet another role called CarRentalUserRole to make things a little more flexible, but I kind of knew in advance what I was going to get into here because remember the initial design stated that we're going to have a desktop application for the person sitting behind the car rental desk, and we're going to have a public-facing website for the users of the outside world. So, to authorize this GetCar operation and this GetAllCars operation against a specific user or a specific role, I have to use the PrincipalPermission attribute. This is not a WCF attribute. This is an attribute that comes from System. Security namespace, so it can be used down the call chain whether it's from a business engine or any other class that the service might call, but I only need to do it here because this is my point of entry. This is my first line of defense. So, this is the way I would decorate this operation. (Typing) So, now WCF is going to authorize this operation only for users that are in the car rental admin Windows group. And remember we're using Windows authentication here and authorization, which means we're just sending in a security token, a Windows authentication token based on who's logged in on the client side. In the case of a desktop app, that's going to be an actual user inside the firewall. In the case of a web app, that's going to be IIS. Now, I don't really like the idea of putting

hard-coded strings here. I tend to point this to a constant, so I'm going to change it to that now. If you recall when I went over the project list, I had not only core common projects, which are framework level projects that are used by this application and any other in the future, but I also had a set of projects that were common code just for the car rental application, the CarRental. Common and CarRental. Common. Contracts. And inside CarRental. Common I have a Security class where I define these two constants, CarRentalUser and CarRentalAdminRole. So, now I can actually change my attribute decoration to point at this, and that just gives me one central place to manage the string from as opposed to having it all over the place. I just felt it looked a little cleaner this way. (Typing) And because I want to authorize this operation not only against the CarRentalAdmin, but also against any CarRentalUser or any application that is specifically running under the CarRentalUser identity, I repeat another PrincipalPermission attribute because you can stack these as much as you need, but this one is not going to point to a role. It's going to point to a user name. (Typing) And I already have that set up as a constant as well. (Typing) And I want to repeat this for the GetAllCars as well. (Typing) And now all I need to do is go through the rest of the operations and figure out which attribute to attach to them. So, for update and delete, the only one that I want to give access to on that one is going to be a desktop application, or should I say an admin application. Let's not say desktop because now we're isolating a specific type of client, and obviously we don't want to do that. So, anybody in the CarRentalAdminRole should be able to update or delete a car. (Typing) And the last one is my GetAvailableCar, and this is going to be available to everybody. So, I'll take the two attributes from the top and just copy them down. See, security doesn't have to be overly complex. In fact, I like to say that the more complicated your security is, the more likely you are to be insecure. So, WCF allows us to keep it simple, but we're still not done. There are a couple of more things I've got to take care of here.

User-Data Authorization Intro

Now, this standard principle and identity-based security is fine, and it gets us part of the way there, but it's definitely not going to be enough. There are still some other things we have to take care of. For example, an application that is used by users that are non-admin, for example the website, will have access to all of these non-admin operations. Now, the users themselves, the browser users really will not have access to those calls because they're going through the IIS server, and, of course, like I said earlier, the IIS server is sitting behind the firewall. However, in the future, I'm going to be extending the reach of some of these WCF Services using the Web API, which means through a REST call you're going to see that some of these calls are going to be made available to the outside, and we need to be able to secure this. Now, we're going to have foreign-based authorization or foreign-based authentication installed on the website, and our Web API is going to piggyback on that. So, a user will at the very least have to log into the system in order to make a call, make a Web API call. However, once they make that call, who's to say that user A cannot look at car rental history for user B? We don't want this. We don't want user data to be cross accessible by other users, so we need a facility in place that's going to help us with this. Now, here's a problem. In the case of a desktop application,

the caller of that WCF Service is indeed the user of that desktop, and we pretty much got that covered already with that role. However, in the case of a website, we have Windows authentication being used so the caller identity is actually the IIS user. In this case, it's going to be that dedicated user. However, I want the actual user that is logged into the site, meaning the browser user, I want the name of that user to be passed into the service as well because I need to do some kind of filtering or trimming if you will to make sure that one user cannot access another user's data. And I don't want to do this by adding the user name at the beginning of every single operation. That's just a little too much work, and it's also a pretty ugly solution, but I do need to receive the user login name in every operation. So, what I'm going to end up doing is using the SOAP header. And by using the SOAP header, I can automatically at the client side inject the username into the header and have that come into the service call without ever explicitly being declared in every single operation or in any operation for that matter. And now that I have the actual user who is using the website, which in the case of a desktop application will be the same as the user that is making the call, I now have the data that I need to use to perform my user data authorization. And to assist me with this will be that IAccountOwnedEntity interface that you saw earlier in the course, so all things will be made clear with that interface finally. So, let's get to the code, and I'll show you how all of this is going to work.

Setting up The Rental Service

My current InventoryService does not have any operations that call for this user data authorization about which I speak, so I'm going to go ahead and add another service to this. This service is going to encapsulate anything that has to do with renting or reserving a car, all of those operations, and I call this the RentalService. So, I'm going to start by adding the IRentalService ServiceContract and add only one operation to it because it's with that operation that I'm going to demonstrate the user data authorization. So, of course, I'm going to come here to the Service Contracts folder in the CarRental. Business. Contracts project and add my IRentalService interface. (Typing) And this, of course, is a ServiceContract. (Typing) And I'm interested in one operation. And this operation is going to get the rental history for a specific user and the user whose history the service is going to return is going to be an explicit argument in this operation. (Typing) And, of course, I have to decorate this as a WCF operation. So, now I'll add my manager, and the manager is going to be called RentalManager. (Typing) And as before I'm going to be inheriting from the ManagerBase class, and I'm going to be implementing the IRentalService ServiceContract. (Typing) Now, to this service, I need to set up the same pattern that I did in the InventoryService where I have the default constructor, I have the constructor that receives a DataRepositoryFactory, I of course have the declaration of the IDataRepositoryFactory interface and mark it with the import attribute for MEF to be able to inject. I have to do the same thing for the BusinessEngineFactory under the assumption that I'm going to be needing a business engine, but I probably will, and importing the factory is not really painful at all. So, I'm going to go ahead and put all that code in. (Typing) And lastly, I need to add my ServiceBehavior so that I can mark my service as a per call service with multiple concurrency. (Typing) The InventoryManager also had the ReleaseServiceInstanceOnTransactionComplete property

set to false, but I'm not going to add that just yet because in truth I don't need it. In fact, it's going to bark at me if I have it until I have at least one operation that's going to be transaction aware, and right now I don't because the only operation that I have will be this one, the GetRentalHistory. And because it's a fetch operation, I don't have the need to make this transaction aware. So, next I'll provide the implementation for the GetRentalHistory operation.

Implementing The GetRentalHistory Operation

So, the first thing I'm going to do here is go ahead and put in the implementation for GetRentalHistory. I'm simply going to put it all in at once. It's pretty straightforward, and you already saw the way most operations are structured. (Typing) Now, let's go through this quickly. Basically what I have here is the need to obtain rental history for a particular user. Now, I don't know who called this in yet. I don't care about that yet. I haven't gotten there. I just need to obtain rental history for a particular user, and all I have to identify that user is their login. So, I know I'm going to need to make a call to the rental repository's GetRentalHistoryByAccount because that's what I have available to me, and that receives an AccountId as an argument, not a loginEmail. Now, of course I could give you another overload or another method on that repository to obtain this by loginEmail, but that's not what I did. I wanted to demonstrate a few different patterns here. So, I'm going to leave the GetRentalHistoryByAccount to require the AccountId. So, now my first order of business is going to be to convert this loginEmail to an AccountId, and I do that by obtaining an accountRepository and then getting the account record for that loginEmail with the GetByLogin operation. Now, if I can't find one, I'm going to go ahead and throw an exception as you can see here, but if I do find one, then I can use the rentalRepository, which I've got right here, and call upon the GetRentalHistoryByAccount using that AccountId and then return that. Now, here's where the problem lies. What if I am user A and I am initiating a call to GetRentalHistory for user B's loginEmail. I shouldn't be able to do that. And with the current technology we have in place, a user cannot do that. However, later when I start extending some of these operations like I probably will be extending the GetRentalHistory using the Web API because I'm going to need to service Knockout with it, that means I'm going to offer the website a REST URL that will allow any authenticated user into the website to make a call that will land them in this operation, and I don't want them to be able to make a call for a user that is not them. And that's the problem I'm going to address next.

Retrieving Call User from SOAP Header

So, my first order of business in securing this operation here is to give it the traditional attributes so that we're secure against the proper user or the proper role that is able to use an application that addresses our services. But now we have to address the problem that I spoke of previously, and that is I have to make sure that the person that's coming in, the user of the actual application that is accessing the service whether it's a desktop user or a browser user, and notice I'm very specific in saying browser user, not the IIS user or CarRentalUser, but the actual user that logged in

with their user and ID when they went into the website, I have to make sure that user is the same one as this. Otherwise, it's a breach of security. That means that a user is trying to obtain rental history for somebody else. The only one that should be able to do that is that user themselves or somebody that is a member of the admin role, and you'll see how I address both of those issues in one solution. So, first and foremost, I have to now obtain the user of the application that is accessing this service. So, in the case of a desktop application, that's going to be the actual desktop user, and that comes in automatically on the primary caller on the identity. However, the in case of a browser user, the browser user is not the identity that comes into the service as I stated a couple of times already. So, now I need a way to obtain that identity, and we established earlier that I was going to send it into the header, into the SOAP header of the call. So, I'm going to do that later in an upcoming module when I do the client, but right now let's work under the assumption that our header is going to carry a string that is the user or the loginEmail of whoever is currently logged into the application, and I need to pull the information out of the SOAP header. And the best place to do this is going to be my ManagerBase class because I need this to occur for all services, so let's go there. So, I'm going to put something in the base constructor to ensure that it gets called every time the service is used. The code that I'm going to put in here needs to retrieve something from the IncomingMessageHeaders. IncomingMessageHeaders is a variable that we have available to us from every service call, and it's an implicit argument. It's an argument that we have available to us at all times, kind of the same way that we are used to dealing with HttpContext. Current from the context of a web application. There are many things we have always available to us without having to instantiate them, things like the session and the request variable and the application variable. Well, in WCF we have an operation context that works the same way. So, the first thing I'm going to do is establish an instance of the current OperationContext, and then check it for null just to make sure that I cover all ends like this. (Typing) Now, I'm going to go ahead and attempt to bring the content of the IncomingMessageHeaders, which is just a property on my OperationContext, and obtain the loginName. And I'm going under the assumption that loginName is in there and headers are stored in the OperationContext outgoing message headers by the client using a key to address them. In fact, it used two keys. One of them is the type of what you're putting in there. The other one is the namespace of what you're putting in there. (Typing) This is, of course, working under the assumption that my client is somehow going to get the loginName of that user into the header of the SOAP message using the same pair of keys. Now, I really need to span the scope of loginName because it's going to be used outside of this constructor, so let me do that real quick. (Typing) Now, because the only Windows desktops that are going to be accessing these services are applications inside the firewall, I really need to just protect myself against those services whose reach I extend using the Web API, and that's only going to be in the case of web applications. So, what I want to do is check this loginName to see if it's an actual login that came in from a desktop application or one otherwise. And the reason I can tell if it's a desktop application login is because it would have a backslash in it because all login names that authenticate with a Windows token are always prefaced by either the domain or the machine name. So, I'm just going to check for this real quick. And if there is a backslash here, I'm just going to set the loginName to an empty string.

Okay, so we're on the right track. I now have the information that I need. I have the loginName that needs to be checked against the login that may be sent as an explicit argument during the call to obtain information with.

Setting up for Retrieval of Authorization Account

Now, I can do something as simple as just checking this loginName against that loginEmail that you saw came into the operation and call it quits there; however, I want to make sure that I am secure in future scenarios as well. So, I really want to make sure that this loginName is a valid login, and in the future, I may have safeguards in the system where I can perhaps lock a user out or tag a user as temporary suspended account because of nonpayment or for whatever reason. I don't want to just rely on this simple string that can possibly be fudged by a good hacker. So, I want to actually retrieve the account that corresponds to this loginName and make sure it's a valid account. Besides, the facility that I designed for this really wants to work with an AccountId, not a loginName. If you recall, back in the module where I discussed the database, I mentioned that any table that contains data that pertains to a particular user account is going to have an AccountId property in it. That's how I am identifying the account which that data belongs. So, AccountId is really what I need here, not just the loginName, another reason why I want to obtain the account record for this loginName that came into the SOAP header. Now, here's where I have to make a decision. Like I said, I want to obtain the account record for this user right here. If I do that here in the base class, I'm going to need the data repository. And remember the base class does not have the data repository. The derived class does. Now, it's very easy for me to just move that DataRepositoryFactory and indeed the BusinessEngineFactory to becoming protected properties here in the ManagerBase class, and therefore I won't have to import them in every service. And that would probably be one solution and a very good solution. The reason I chose not to do this, and this was just a judgment call, is because there may very well be a service in the future that will not need a data repository that will go through a business engine for everything or vice versa. And once again, I am now needlessly instantiating something that I'm not going to use. That's why I choose to keep the RepositoryFactory and the BusinessEngineFactory in the service itself even if I have to repeat it. But now I have the need to use a data repository in order to retrieve the account information for this login here. So, what I'm going to do is that I'm going to declare an abstract method, or should I say a virtual method here in the base class that then is going to be called in the constructor of the base class, but whose implementation is going to be in the derived class. Let me show you what I mean. (Typing) Now, for my constructor, I'm going to call upon this method. And I need a variable in which to hold my AuthorizationAccount or the account that is going to correspond to this loginName. (Typing) Now, every service is going to call upon this method here. However, the implementation is going to go in the actual service itself, not the base class. And if I don't need to do user data authorization, I don't need to override this and do anything with it. It's simply going to call and return a null. And later you'll see where that's actually being used as in the case of the InventoryManager where nothing there pertains to a particular user, and I don't have to undergo this code needlessly. So, next I'll go back to the RentalManager and provide the implementation code.

Adding The Authorization Check

So, to provide the implementation for this method, let's go into the RentalManager service, and up here I'm going to override that method. And remember what I need to do here is obtain an account entity for this login right here, which is the login of the user that's using the site that's using the application and has made the actual call that wound us up in the service here. And there we go, I'm returning that account. So, here in the base class when that gets called right here, it's being set in AuthorizationAccount. So, now the question that remains is what do we do with this AuthorizationAccount? So, let's go back to the RentalManager class and go down to that operation. Now, here I went through the trouble of obtaining the account entity for this login. Now, this is not the user that is in the site, or hopefully it will be if we're going to be secure, but this is actually the user that we're trying to obtain rentalHistory for, and I obtained the account entity so I can use the AccountId to make a call to GetRentalHistoryByAccount. But now that I have the account entity and back in the base class I have the account entity that corresponds to the actual user that made this call, I can now make a comparison with the two. And what I want to do is put a facility in place that I can do that with any entity that contains data that pertains to a particular account. Now, that description there should sound familiar to you because that's the way I described the AccountId property or the AccountId column that was in the rental and the reservation table, and it was also the account table back in my database. Well, if you recall when we put the entities together, I gave those three entities an interface that implement call IAccountOwnedEntity, and that interface exposed the property that corresponded to the account to which the data in that entity belonged to. So, I have an implementation of IAccountOwnedEntity here, and in future cases I may end up with an entity like rental or reservation, which also contain data that pertains to a particular user. All three of those entities implement this one interface. That means all three of those entities will lead me to the AccountId that represents the account to which their data pertains. So, what I want is a facility that can receive any entity that implements IAccountOwnedEntity so then I check it against the AuthorizationAccount in my base class. So, let's go back to the base class, and let's create a method here called ValidateAuthorization. And it's a protected member because I want to be able to call this from any service that needs it, and I want to be able to send into it any entity whose data I wish to protect meaning an entity that has data that pertains to a particular account meaning an entity that implements the IAccountOwnedEntity interface like this. (Typing) Now, if the user that made this call is a member of the CarRentalAdminRole, I don't want to go through this. And all this is going to do is that it's going to perform a security check, and if the security check fails, I'm going to throw an exception. That's why I'm not returning anything. So, I want to omit that check if the user that is currently logged in is a member of the CarRentalAdminRole because they should have access to do all of this stuff. So, I'm going to case around that first. (Typing) The identity that came into the service, meaning the identity of the desktop user or the user that we set up as the identity that the website was going to run under comes in, and that's the identity that is checked up here in these attributes, PrincipalPermission. But what WCF is actually doing is no magic. It's simply wrapping all of this code in this operation around an IsInRole check that we've been doing for years now off the Thread. CurrentPrincipal, which contains that

same identity, and that's why I'm able to call Thread. CurrentPrincipal. IsInRole CarRentalAdminRole. This is how I do programatic authorization checks. Now, if the loginName is empty, I'm not going to perform any security check. It may not be the safest thing to do, but I'm working under the assumption that if a loginName is empty, it was set up here, which means we're on a desktop application anyway and we're inside the firewall. And now I want to check to see if the incoming entity into this method has an OwnerAccountId property, and it does because it's an implementation of the IAccountOwnedEntity interface, and I want to check that AccountId against the AuthorizationAccount that was loaded when this methods implementation was executed. (Typing) And now in here I'm just going to throw an exception, and I've created a custom exception ahead of time for this. Now, this custom exception is called AuthorizationValidationException, but because I'm inside a service here, I need to do the right thing and wrap it up in a FaultException of T. (Typing) One last thing I need to do is take this exception here and make sure I set up a FaultContract for it here. (Typing) Now all I have left to do is actually use ValidateAuthorization. And for that, I want to go back to my RentalManager, go down here, and simply make a call like this. This is a call that I can now make with any entity that implements the IAccountOwnedEntity. And in this case, that's exactly what I'm doing. So, a simple call to ValidateAuthorization sending this account in is going to ensure that this account is a matching account to the user that is actually sitting behind the browser to that user's account. And I can do this with the rental entity, and I can do this with a reservation entity as well. So, this gives me a nice facility that's clean. It's an elegant pattern. Sure you incur resources. You're making a call like this on every call, but this is security. Security you pay to live and you pay to be secure, and these resources are not that bad. So, next I'll finish up the RentalManager and then write one more service that the application is going to need, the AccountService.

Finishing The Rental Service

Okay, before I wrap up the services, I just noticed that I am throwing an exception here, which of course should be wrapped with a FaultException of T. If one of you caught this, I think I heard you yelling out there. So, now I'm going to add the other operations to the IRentalService ServiceContract, and that's also going to include another FaultContract on this operation indicating NotFoundException, the change that I just had made a second ago. And then I'm also going to add those operations to the RentalManager. And I'm going to do all of that just in the next couple of seconds once again through Camtasia magic, so stand by. Now, this service also introduces the concept of custom data contracts, which I'll fix in a second. As you can see, this is broken because I haven't created these classes yet, but as I said much earlier in the course, not every service operation needs to return a single entity. We're not locked into using these entities as our exclusive data contracts. And, in fact, if you don't like the idea of sending entities up the wire as data contracts, you can use them in any capacity that you want. You can make them stop in the service. In other words, come up through the data access layer, through a business engine if necessary, up to the service, and then in the service, create a DataContract all the time just for the specific reason of sending it up the wire to the client. I chose not to do that because the entities work just fine for me, but there are one or more situations

where I've had to wrap some extra data up alongside an entity or perhaps wrap one or more entities together, or perhaps just send custom data up the wire, and that's where a custom data contract will always work for me. So, in this particular case, I'm going to need a custom data contract called CustomerReservationData, and here's one called CustomerRentalData, so let me add them first. And as you can see, we can compile now, and let's see what these look like. There. This isn't even a wrapper for one or more entities. This is basically the intersect of a couple of entities. Instead of sending two whole entities up the wire, I just designed the contract as I saw fit just with the data that I needed to send up the wire. And the same goes for the CustomerReservationData. And as you can see, my data contracts have a common base class, and this base class looks like this. Look familiar? This is really what EntityBase is doing, but I wanted a different base class just for the purposes of those custom data contracts. So, now let's go out to the RentalManager and provide the full implementation for the RentalManager. Now, you're going to see when I add all of the operations to the RentalManager a couple of them are going to be broken because a couple of them use that rental engine for some behavior reusability, so I'll have to modify that as well. And as you can see, we've got some broken code here. That's because the IsCarCurrentlyRented service operation calls upon the business engine to ask that very question. And the reason it does that is because I had to ask this question from a couple of other operations to perform other tasks, so I refactored the behavior of checking to see if a car is currently rented by its carId out through the business engine. Let's see if there's any other place where the business engine is going to be required. Here's a call to the business engine where I'm calling a RentCarToCustomer. The reason I did that is because there are two operations in this service where I'm going to rent the car to a customer. One of them is a raw rental like if the customer walked up to the reservation desk and said let me rent a car. And the other one is rent a car to a customer based on an existing reservation, which I think is what this operation is, ExecuteRentalFromReservation. So, the actual task of renting the car was behavior that I need some reusability on, and I refactored that into a carRentalEngine. See, here's the call to RentCarToCustomer from a regular RentCarToCustomer operation where there is no existing reservation. And here's an overload to that that takes some other argument. So, now I need to finish up that business engine so this code will compile.

Finishing The Car Rental Engine

So, now I'm going to go out to the interface that defined the rental business engine, and I'm going to finish this up. And as you can see, there are the other three methods. We can't call them operations here because they're not service operations. They're business methods. These are the three other methods that were called from the RentalManager. So, now let's go into that business engine and add their implementation. So, here we only have the IsCarAvailableForRental, but now we have all of them. There's the RentCarToCustomer, IsCarAvailableForRental, and there's the IsCarCurrentlyRented in two different forms, two different overloads. Now, I want to point something out here. Take a look at the RentCarToCustomer method. Notice that here we're not doing things in the WCF way because this is not a WCF Service. This is a reusable business engine, and in the future it may be called from

elsewhere in the system on the backend hopefully, but not from a WCF Service. So, this should not be WCF aware, so it should not throw fault exceptions. It should behave in the traditional way where if you have an exception you throw that exception. And because it's a WCF Service that's going to call upon this method here, it needs to account for the fact that it may receive these exceptions, and it needs to be able to catch those, and, of course, re-throw them while wrapped inside a FaultException. Let's go back to the RentalManager and see if that what's taking place. There's a call to RentCarToCustomer, and as you can see, that call is made inside a try and I'm catching all of the exceptions that I am potentially expecting from that rental engine. And these are not done in the base class because they are specific to this operation. I didn't want to put everything in that base class. So, the base class simply does the catch all, but all of the operation specific exceptions are caught here, and, of course, every one of these exceptions here is declared in a FaultContract in the IRentalServiceContract right here. Okay, so that concludes the rental service. We have one more service to add and we're done with the service layer, and that's the account service coming up next.

Wrapping up with The Account Manager

So, here I have the ServiceContract for the account service, and as you can see, it's pretty simple. You're allowed to GetCustomerAccountInfo based on a loginName or you're allowed to UpdateCustomerAccountInfo given an account entity that was sent in as the data contract. Now, if I go over to the AccountManager, let's see what that looks like. As you can see, the pattern is exactly the same as the two other services. We're inheriting from ManagerBase, we're implementing our service contract, we have a default constructor for WCF usage, and we have a non-default constructor for unit testing purposes. We're not leveraging any business engine here, so I found no need to import the BusinessEngineFactory, but I am using the DataRepositoryFactory. And I have to override the LoadAuthorizationValidationAccount the same way I did in my rental service, though I didn't need to do that in the inventory service. The two operations are here, GetCustomerAccountInfo and UpdateCustomerAccount. Here's my basic principal identity base security checks for authorization. There's my transaction code. Because I set transactions to true on this operation, I had to set my ReleaseServiceInstanceOnTransactionComplete to false. When you forget this, it will bark at you the minute you try to host a service. Notice I said when and not if. I forget it all the time. Now, here I want to make sure that we authorize these the same way we authorized that GetRentalHistory operation in the last service because later I'm going to be able to perform these calls from the website, which means there's probably going to be a Web API wrapper around these because I'm going to be using Knockout, and in Knockout I make REST calls to the Web API. And because I'm extending the reach of these calls, that means that I'm going to be eventually exposing a REST URL that will get me to these two locations. So, I certainly don't want Bob to login with his email address and be able to execute a REST URL to update customer account information for a data contract that he built out of JSON data or another guy named Richard for example. So, I want to make sure that this email address or this login equals the user that is making this call. So, I'm undergoing the exact same thing that I did

back in the GetRentalHistory operation where I'm obtaining my accountEntity based on this guy, and then I'm calling the ValidateAuthorization on that accountEntity because remember it is an implementer of the IAccountOwnedEntity interface. Now, down here I'm receiving an accountEntity so I don't have to get anything. I can just run that through a ValidateAuthorization. See how clean that is. Now, this guy will be validated against the user that is actually making this call, and hopefully they're the same guy. Otherwise, it's going to fail. But because of that little check that we put in there, if it's an administrator, then it doesn't throw an exception and allows the call to go through. Now, we're not totally done with this layer yet because we still haven't unit tested it, so I'm going to show you how you would write a unit test to actually test these services.

Testing The Business Engine

Okay, so we're on a roll here. Let's go over some unit testing, and I'm going to start with the car rental engine tests, and I'm only going to write one. I'm going to write one for one of the methods, and you're going to get the idea on how the others would work. And, of course, the finished code has all of the tests. So, the first thing I've got to do is come down to my CarRental. Business. Tests project, and I'm going to Add a Unit Test here. And I'm going to call it CarRentalEngineTests. Now, I have to make sure that all my references are set, so I'll check that first. And notice I have nothing set. So, let me set the ones that I need. Now, notice I didn't set the CarRental. Data because I'm going to be mocking up the data repositories that go in. I need to set CarRental. Business because that's the class that I'm going to instantiate, and there are its interfaces and of course the entities. But notice I am not setting a reference to anything that is WCF related. We're at the business engine layer, which is one layer below. So, even though we have entities that are climbing up and down the layers on the business tier, we're still following the architectural principle of one-way referencing only. So, our service layer only references the layers beneath it and not the other way around. Our data access layer has no knowledge of any layer above it, and the business engine has no knowledge of the WCF layer. And this is followed in the unit tests as well. I also need to set one assembly that I know I'm going to need here because if you recall, the entities implement IExtensibleDataObject, and that comes from this assembly. Okay, so I want to write a unit test for a business method called IsCarCurrentlyRented. And if you recall, I had two overloads for that, one that accepted a carId, that's the one that I'm going to write the unit test for, and the other one accepted a carId and an accountId. And I really don't have any pattern in place for a naming convention. I told you I use two naming conventions for my tests. One of them is _delimited sentence describing the test, and the other one is Pascal casing, which indicates that this test corresponds to a method of the same name. I think I'm probably just going to break a little bit of the rules here and kind of combine both of those. Alright, so we just set a new convention in place. How about that? We have Pascal casing indicating it's the name of a method followed by our second convention, which gives you more information about the test. Now, let's go over to that method and see what it's doing. As you can see, I'm obtaining the rentalRepository from the incoming DataRepositoryFactory, and I'm making a call to the GetCurrentRentalByCar given the carId that came into the method's argument. And if there's a record found, that

means the car is currently rented. And if there isn't, it is not rented. And this is what I want to simulate to make sure this logic works fine without actually hitting the database. So, I know for a fact I'm going to need a rental entity of some kind, and that rental entity represents a car that is rented, so I'll give it an ID for a car as one of the properties. And it's just a little test entity within the test itself. And then I'm going to need a mock for the rentalRepository, which means of course I'm going to need a mock for the DataRepositoryFactory. So, let's go back and set that up. (Typing) So, this represents a rental entity indicating that car number one is rented. Now, let's do the Mock. (Typing) And remember in my references I didn't reference the data access layer. I referenced the data contracts project because that's all I need. I'm not going to be using the real data access layer implementations. I'm going to be mocking those. (Typing) So, what I want to mock up is when the GetCurrentRentalByCar is called, if it's car number one that is sent in, I'm going to return my test rental class. This of course means if it's not car number one, it's going to return a null, and I'll be able to test one way or the other. Now, I've got to create a Mock for the DataRepositoryFactory. (Typing) And this setup is going to indicate when I ask for a rental repository, I'm going to return the one declared in the previous Mock. (Typing) Now I can instantiate the engine and send my mockRepositoryFactory in. Now I'm going to make a call to IsCarCurrentlyRented, and I'm going to make two calls, one of them with ID 2, carId that is, and the other one with ID 1. One of them should fail. The other one should pass. (Typing) And my assertions. (Typing) Now, let's run this test and see if it passes or fails. And we got a passed test. Now, I can hear the TDD guys out there probably yelling I should've written this first and then go back and write the engine method. It's just not the way I code. I'm a big fan of unit tests. I'm just not a TDD guy. I follow it very loosely. And I wrote the method first, I got the method to work, I got my application working, and, in fact, before I even wrote any clients, I of course wrote my unit tests. But my goal here is that this unit test represents that method. And now if I were to go back and change something here, I can always come back here, run a regression test, and see if it passes or fails. And if it fails, I have to do one of two things. Either I have to fix something here or perhaps what I refactored or what I rewrote in the method really does merit adjusting the test accordingly. You have both sides. There are times where you have to use the test to see if you broke something, and there are times where you purposely rewrote something that is going to make a test that was currently passing fail, and you have to adjust the test. So, now next, I'm going to write a test for the manager.

Testing The Manager

So, my tests for my managers, my services, are going to here, CarRental. Business. Managers. Tests. Now, keep in mind the test that I'm going to be doing is not a test to see if the service can be called in a WCF capacity. That'll come later when I do hosting. I'll write some tests at that point. I'm writing tests that address the manager class, the service class, as a regular class, not a WCF class. So, let's start by writing a test called InventoryManagerTests. (Typing) Now I'm going to write two tests here, and they're going to test the UpdateCar operation. Now, if you recall, the UpdateCar operation worked in two different capacities. Let's go back to that service and see. There's UpdateCar right there. It

has the capability of adding a new car or updating an existing car based on the ID of the incoming car entity. So, as you can see, I'm going to need a DataRepositoryFactory, I'm going to need to obtain a carRepository for this test, and I need to mock up a couple of cars to use for my tests. So, let's go back to the test here, and I'm going to write a test called UpdateCar. And this one is going to add a new car. Now, at the same time, let's do the other tests so you can see how they differ as I write them. Now, I need to represent an incoming entity and one that is the result of the call because back here in InventoryManager I have an incoming entity and then call to Add or Update gives me an updated entity. And, of course, the task of adding a new car will install a new CarId for me. So, let's simulate that by having two cars, one called the newCar, the other one called the addedCar. And then in the other test I'm going to have an existingCar, and the next one is called the updatedCar. (Typing) And in this test I'm going to have an existingCar and an updatedCar. (Typing) Now I need to set up my mocks for the repository. (Typing) And I'm going to take the approach of doing all of this in one line instead of like the previous tests where I did it as two separate mocks. (Typing) And in this one, I have to mock up what happens when the Add is called using the newCar as the incoming argument, and I need to return the addedCar. Now, let me copy this over to the other tests and then alter it accordingly. In this one, I need to mock up not a call to Add, but a call to Update, and I'm sending into it the existingCar, and I'm returning the updatedCar. So, now from both tests I need to instantiate the InventoryManager as if it was a regular class, and I'm going to call the updateCar operation and return a resulting variable. Then I can assert that this return variable is equal to the addedCar. Now, in the case of this test, I still need to instantiate the manager of course, but now I'm going to call the updateCar again, this time sending in the existingCar. And now I'm going to assert that the result is equal to the updatedCar. And I have my two tests. Next, I'll run them because I suspect we're still going to have an issue here.

Adding Needed Credentials for The Manager Test

Now, I'm willing to bet that most of you think that there's nothing wrong with these tests and they're probably going to pass. And, in fact, they should with one exception. Let me run one of them and you'll see what I mean. As you can see, it failed. Now, let's see what the failure is. I can hover over this, and you can see that it says that it threw an exception, a System. Security. SecurityException. The request for principal permission failed. I happen to recognize that error message. That error message comes from this right there. Remember I told you this was not a WCF thing. PrincipalPermission attribute can be used on anything. And what's happening is that we're running from a unit test. The unit test has no security context at all. In fact, if I put a breakpoint in the test and check the Thread. CurrentPrincipal. Identity. Name, it's actually blank. So, we're actually failing security here. Now, the easiest way and probably most encapsulated way of solving this problem is to set up a temporary principal just for the purpose of the test, and we do this right in the code. So, I'm actually going to write the initializer just like I did before. (Typing) And here I need to come up with a generic principal and an identity inside of it. I can call that user whatever I want, but I want to make sure that I can adhere to what that service is expecting. And if you remember, it's expecting me to be in

the CarRentalAdminRole, so I need to be in this role called CarRentalAdmin for the purposes of my test. So, I do that by creating a GenericPrincipal. And in the creation of GenericPrincipal is where you create the GenericIdentity and the roles because there is no public property for exposing the list of roles. All we have from a principal is the capacity to do an IsInRole check, but the creation of the list of roles has to be done at creation time. As you can see from the IntelliSense here, that's exactly what it's asking me for. And I just care that I'm in the CarRentalAdmin group. Now I have a GenericPrincipal, and I can place that in the current thread. Now, let's see if the unit test runs. There we go. Let's run both of them. And now you know how to write a unit test to address a WCF Service. Once again, not in the context of WCF, but as a regular class and mocking all its dependencies. Okay, next I'll do a quick recap on what we learned in this module, and I'll let you know what's in store for you in the next module.

Recap and Summary

So, how was that for a brain dump, huh? Whoever said security couldn't be fun. Well, I sure as hell didn't say it because I think it's a pain in the butt, but unfortunately it's necessary, and we got it covered in a couple of different ways. We covered it using standard principle and identity security, but I also took it a step further and used the custom solution for user data authorization. This is where you saw that I sent the username into the SOAP header, or at least I'm assuming that it's there already. I'll be sending it later when we do the client layer module. But I pulled it out at the services, and then I used it to compare against a user that may be coming in through an operation. This is where the IAccountOwnedEntity interface was finally used, so now you understand why I have that interface. You also noticed that the unit test needed to fake some credentials in order to execute properly. This is because that security principal demand attribute is a. NET thing, not a WCF thing, which means the minute that I place it on a method, that method is going to become secure whether we're in WCF or not. That's why we had to take care of that in the unit test. So, now that we have the services written and secure, we need to host them so we can start calling them in the upcoming modules. So, that's what the next module is going to be on, all about service hosting.

Service Hosting

Intro and Highlights

Hi. This is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. In this module, I'll cover WCF hosting and bring to light those services that we wrote in the previous module by exposing them to clients for consumption. Let's go over some of the highlights that we'll cover here. I'm going to discuss several different types of WCF Service Hosting and then show you how to set one of them up. We'll also cover configuration, of course, which is going to be necessary for WCF hosting, and I'll do some testing on client

consumption using a unit test. In doing so, I'll cover some client configuration and also how to use the WCF channel factory. And lastly, I'm going to set up an unattended process. There's a process that needs to run at a certain interval for this application, and it's not something that a client is going to kick off manually, so I'll show you how to set that up. So, let's get started.

Service Hosting Intro

WCF Service Hosting can take several forms. All of them can be run unattended. However, it's also possible to run hosts that are meant to be monitored by a person and that can actually expose information about the incoming calls on the services that are being hosted. All hosts expose endpoints for clients to consume. These are the actual addresses where clients are going to be able to find endpoints. Some can be accessed only inside a firewall such as ours, and some can be accessed outside a firewall as well. All the endpoint information needs to be fed into the host either programatically or declaratively. In fact, it's actually all done programatically, but if you leave that out and do it declaratively as in a config file, then the programatic part of it is done for you in the background by WCF itself. Endpoints can also be configured if you wanted to change some of the characteristics or limitations such as different timeouts, transaction handling abilities, reliability, things like that, and we're going to be covering these things. I'm going to be writing a host using a Console app. It's easy and it's great during development because you can expose information to the actual console so you can see what's going on during hosting. In production, my preference happens to be to do the exact same thing, but in an NT-Service. Remember those? A good old-fashioned NT-Service. The kind that you have to go into the control panel's admin console and then start and stop. Those are unattended, and they also start up automatically, and I prefer them to IIS. I just find them a little more elegant. And IIS does have a limitation, which we'll talk about later. So, next I'm going to get started by actually writing a Console app that's going to service my host for this application. But keep in mind I can later very easily offload that to an NT-Service, and I'll show you how to get something like that started.

Simple Host Setup of Inventory Service

Hosting services in WCF is a very easy process, and because I'm using a Console app, I have the luxury of outputting stuff on the screen, so I'm going to write some wrapping code that's just going to go around my hosting code just to let me know what's going on. (Typing) And my hosting is actually going to take place right in between here. I, of course, got to do a ReadLine here so it doesn't just shut down the application. It sits there and waits for me to press enter to exit, and then it shuts down the host. So, now I have to actually perform the hosting, and that's actually very easy. I need one instance of the ServiceHost class per service that I'm going to host. The ServiceHost class comes from System. serviceModel, and it can be as easy as doing something like this. Notice I have an alias set up for the System. serviceModel called SM. Now I need to tie the ServiceHost class to the service that I'm going to host, and

that's done in the constructor like this. Now, when I set up this project, I went ahead and I referenced my CarRental. Business. Managers assembly so I have access to all my services. Now, we're talking services here, not service contracts. Now I can do something like this, and I'm ready to host. Now, at this point when I open the host, this is where it's going to look for endpoint information, and I need to give it that information in the config or programatically. I'm going to start by hosting just this one service because then I'm going to spiff up this application, make this module a little more interesting, and also output some interesting information right to the console host. Now, like I said, the host is expecting to find endpoint information for this service, and it wants it one way or another, either programatically or declaratively. I prefer not to do it programatically unless I have a good reason to do so. A reason for doing programatic configuration can be that you don't want to store your configuration information on the app config or web config in the case of an IIS host. You may want to store it in a database or in some other kind of configuration schema. You are responsible for writing your own project to go out and read such a schema and then provide the ServiceHost with that information programatically. If you don't do this and if you don't feed it programatic information directly here, you're going to need to declare it in the config file, which is the way I'm going to do it. So, next I'm going to go into the App. config and set up the configuration for InventoryManager.

Configuring The Inventory Service Host

So, I'm still here in the ServiceHost. Console project, and I'm going to go into the App. config. The first thing you've got to know about the App. config of the host is that this is the App. config that the services are going to read. This is the executable in whose context the services are running. So, any configuration that the service or any of its down-level dependencies are going to need needs to be configured in this App. config. Such an example is my connection string. My services are going to call upon a down-level class such as a business engine or a data access repository, and those repositories are going to require a connection string, so this is where I have to give the host the connection string information. Now, the great thing is that the clients will never need to see this, and their configuration will never have a connection string. But I also need to put the configuration for the services here, for the actual hosting, and all of this occurs inside the system. serviceModel section. Now, this is very simple. I can do two primary things here. I can either expose services or I can consume services, and we're writing a host to expose services, so I'm going to give it a services section. And inside the services section I can have one or more service sections. Now, here I have to give it the name of the service, and this is a little bit of bad nomenclature in my personal opinion. It should have been called type instead of name because name implies that you can name it anything you want, but what it's really looking for here is the fully qualified path of the service that I'm hosting. So, my service is in CarRental. Business. Managers. This is the service that I'm hosting right now, this is the namespace, and it's called InventoryManager. Now, inside the service tag I need to expose one or more endpoints. This is where the service contract comes into play because endpoints are done at a service contract level. And since the InventoryManager is exposing only one service contract or it's implementing one service contract, I need to expose that service contract under an endpoint like this. Now, the

other two pieces of information are the address and the binding. I'm going to be exposing this service internally to my internet. Remember, it's going to be for either desktop apps or websites running in a local IIS server that is inside my firewall. So, the fastest binding that I can use for the inside of my firewall or my internet is TcpBinding. And now I have to give it the address where the clients can locate this service. And because I'm using TcpBinding, I have to use the protocol identifier net. tcp, and I, of course, am hosting on localhost because I'm right here on one machine. And now because it's TCP, I have to give it an available port. Be very careful with the port. If you give it a port that's in use already, especially by something that's really important such as SQL Server, it's going to give you some unpredictable results, but that could also be the subject of some really good practical jokes. The last thing I've got to give it here is an actual path. This can be anything. It can be your name, it can be Mickey Mouse, it can be whatever you want, but the standard that I like to follow is just give the name of the contract without the I. This just makes a little more sense. Now, I can expose another endpoint for this service for the same contract under a different binding if I wanted to, for example, expose it outside the firewall using HTTP or if the service InventoryManager was implementing another contract, another service contract, the client could not see those operations unless I give it another endpoint here exposing that contract. But we only have one, so this is really as simple as it gets right here. I should be able to run the host right now and see if this works. If it gives me no error, that means it found the configuration information successfully, and it's sitting waiting for calls to take place. Now, I'll get this message here one time. This is because it's asking my firewall if it's going to allow that port to expose. And there we go. No errors, so my ServiceHost is just sitting hosting the inventory service through that one endpoint waiting for calls to take place. Now, this was all very simple, but I'm going to fancy it up just a little bit next.

Reporting Hosting to Console

I'd like to give this Console app a little more visual credibility by letting it show information about the endpoints that it's exposing. So, what I'm going to do is change this line here so it calls upon a helper method that starts up at host, but also writes some information out to the console. So, first I'm going to write that method. Now, because we're inside a static method, I have to call upon another static method like this. And this static method is going to receive an instance of the host and a description to show on the screen. So, the first thing it's going to do is actually open the host just like it did up here. Now I can display to the screen the first piece of information that this service actually started. Remember the Console. WriteLine automatically has that string. format built into it so I can use that kind of notation as you can see right there. Now what I'm going to do is iterate through the actual endpoints that the host found in the config file when it opened up. Now, I only had one, but I can have many in the future if I need to. And now I want to display information about each of these endpoints, and I'll give it a little bit of a space down here. And now up here I'm no longer going to call the Open, but I am going to instantiate the host because I need to send that into my helper method. But now I'm going to call it something other than just host because I'm going to be starting up more than one service, so I'll call this one hostInventoryManager, and I'm going to need three more. (Typing) And I

need one for the AccountManager. (Typing) Now that I have the three host classes I can call my helper method three times. I can use any description that I want of course. And one thing that I didn't do down here that I really should is have some proper housekeeping. The host will stop automatically when the Console app ends, but it's always a good idea to do your housekeeping. I'm going to do that with a similar helper method that I'll call StopService. I want to point out one difference between Close and another method that's available on the host called Abort. Close will allow any pending calls to finish up, which is really what I'd like to do. There may be a time where I would need to do an Abort, and that will stop the host abruptly, and any unfinished calls or calls that are still in the middle of doing some work will just stop what they're doing. And if anything is transactional, it will, of course, scroll back to transaction. And now I can call my helper method up here. (Typing) And that's it. The only thing I have left to do is add some configuration information for the other two services, and I'll do that next.

Configuring The Other Services

So, in order to be able to open up the host for these other two services, I have to come here to the config file and add them to this configuration, which I'll do now. As you can see, we get some nice IntelliSense here. (Typing) Now, there's some other configuration information that I'd like to add here, but first I'm just going to test this host to make sure that it can see the services. If you take this one step at a time, then when something doesn't work, you can pretty much rely on the fact that the problem was caused by your recent addition. And once I get to my Press Enter to exit with no exception, that means that my host is now listening on those three endpoints for those different services, and I've got some information of the console to let me know on what it's listening. Next, I want to add some characteristics to these services. So, I'm going to come back out to the config and change some config around.

Binding Configurations

There are a couple of characteristics that I like to give all of my exposed services, and they're at the binding level meaning the transport level, which means they're going to need to go on both sides of the wire. So, what I want to do is add a binding section here and modify the binding characteristics of all three of these endpoints. So, to add binding configurations, you go outside of the services section, but still maintain everything inside the System. serviceModel section, and you add a bindings section like this. Now, the next tag corresponds to the binding that you wish to configure, and there's a tag for each and every binding that WCF offers. So, for example, this binding here, netTcpBinding, tells me that there's actually a tag called netTcpBinding like this. As you can see from the IntelliSense, there are strong-type tags for every single binding. And keep in mind everything that I'm doing declarative here can be done programatically if I choose to go that route. Now, inside the netTcpBinding tag I need to add one or more binding sections, and they define the characteristic that I want to change away from their default setting. Now, I only need one, so I'm going to just put one binding section here, and optionally I can give it a name. If I

give it a name, then I have to proactively target this name from each of the endpoints that I want to configure using this binding section. In. NET 4. 0 they introduced the concept of a default binding configuration, which is what I'm going to use, which means I can leave the name out like this, and now any endpoint that does not have a binding configuration attribute identifying a specific name will use this no name binding configuration that you see here. Now, some of the configuration settings are properties of the binding tag and some are internal to the binding tag. The first one that I always like to set is this one. Because I'm using transactions in some of my services, it's a good idea to turn on transaction flow at the binding level. What this does is allow any transactions that might have been started before the service call to flow into that service call and have that operation become part of that ambient transaction. This is very similar to setting the transaction flow at the contract level like we did earlier in the previous module; however, this is like a master switch at the binding level. So without this, the other one is not going to work properly. The other setting that I like to give it is an increase of the max receive message size. This corresponds to the SOAP message size, and it defaults to 64k. There's really only two times that I've seen the default message size of 64k not be enough, but I like to play it safe. Those two times are when I'm doing big lists or when I'm doing some kind of streaming. And I'm not doing any streaming, nor am I doing big lists in this application, but you never know how big that list of cars can get. And remember, we exposed the GetAllCars operation, which means if we had one or two thousand cars in the database, we may blow that 64k limit out of the water, so I think it's a good idea to increase this, and it doesn't hurt. And I'm going to set it to 2 meg. The last configuration setting that I like to give all my services is the turn on reliability. What this does is that it gives me a retry mechanism so that I can rely on the fact that my message, my SOAP message, will get received by the service properly, and it will not be subject to network hiccups or things like that. And that's it. We didn't break anything, but I think it's always a good idea when I make any kind of change here, I'm just going to go ahead and run again to make sure that everything is still cool. So, next I'm going to test the consumption of these services, or at least the connection to these services, and I'm going to do so from a unit test.

Configuring-the-test-client

In order to test my ability to connect to the hosted services, I need a client. I need to set up some kind of client executable, and I'm going to use a unit test for that. It just makes sense. So, I have a project down here called ServiceHost. Tests, and this project has an App. config because it's going to service my client to the services, which means I'm going to need to configure something here, so I'll go there first. Now, the App. config in the host had a services section, and then inside it had three service sections, and then inside each service section it had an endpoint. While a client does not know about any of the services, the client only cares about the API, and remember the API is defined in the service contracts, so the client doesn't know about the services. It just cares about a list of endpoints no matter how they're configured on the host. So, the first thing I've got to do is give my client a serviceModel section like this. And since now I am consuming and not hosting, I don't put a services section, I put a client section. And the only thing that the client cares about is a big list of endpoints, not a breakdown per service.

Now, I'm just going to steal the endpoint information from my other App. config like this. (Typing) And now I can remove all of the service sections. The endpoints have to match what is exposed by the host. And as you can see, they're identical. Now, for the purposes of this test because I'm not creating any true clients as far as a desktop application or a website is concerned, I'm just reusing my business side service contracts as you can see here, CarRental. Business. Contracts. IInventoryService and Rental and AccountService. And these contracts return the business entities. So, I'm using just the business side of the wire, but this is just for the purposes of testing the connection to the service. In a true client, I'm going to be using my client entities. And in order to receive client entities or to have WCF deserialize into client entities, I need to have a client version of the service contracts, but I'll get into that in the next module. I'm going to keep this site simple for now. I also need to make sure that my binding configurations match on both sides, so I'm going to steal that from the other App. config as well. So, now that my client is configured, I can actually create a unit test class and start writing some test code.

Testing Service Connectivity

To call a WCF Service that's currently being hosted, I normally need to have a proxy class. And in a production environment, that's exactly what I would have because the proxy class wraps everything. It wraps all of the WCF stuff behind the scenes so the developer just simply has to instantiate an object. I'm going to cover that in full in the next module. But for the purposes of unit testing connectivity to the service, I'm not going to have an actual proxy class. Here I don't mind having WCF code in my actual unit test. After all, that is what I'm testing. So, I'm going to use something that WCF offers me called the channel factory. And the channel factory actually allows me to create a proxy on the fly without having an actual wrapper class known as the proxy class. So, to have this virtual proxy, I first need to create an instance of a channel factory, and that's what ties me to the service contract. Now, I'm going to do this in a unit test called test_inventory_manager_as_service. First, I need to create an instance of the ChannelFactory. Now, if you've ever done any WCF, you probably know how to use a proxy class already, and you may also know something that I'm going to teach in detail in the next module, and that is that the proxy is going to have different constructors. The default constructor, which is the main one that I'll be using, tells the proxy to go out to the config file and look for an endpoint that exposes or that tries to access this contract. If it finds only one, use that one. Now, if there's more than one endpoint for one particular contract, they have to be named, and that's where the proxy offers an overload for the constructor that accepts an endpoint name. Well, those overloads are also available at the ChannelFactory level here. You would think that the ChannelFactory works the same way where using the default constructor just uses that one endpoint for the IInventoryService contract, but there's actually a bug. Somebody forgot to make a call from the default constructor to a non-default constructor sending in a blank endpoint name. So, the default constructor in the ChannelFactory is actually completely useless. You have to explicitly give it an endpoint name of nothing like this. So, now that we got around that quirk, my next step is to actually create a proxy using this ChannelFactory, and that return type of that proxy will be the service contract type because that's what I'm going to

be able to make calls on. Now if I wanted to, I can make a call to the service and make an assertion to complete my test, but that would be more of an integration test because now I'm actually making a service call and returning some data. I don't want to depend of the fact that I need to return some data. I may not have any data to return when this system goes online. So what I want to test is just the connectivity, the fact that I could actually communicate with a running service or a hosted service without making an actual call. In the case of a proxy class, that's done with the Open statement, but I don't have a proxy class here, and the methods exposed in my proxy variable are just those declared in the IInventoryService contract. So, what I need to do is make an Open call on my proxy, but that Open comes from the ICommunicationObject, so I can pass my proxy to an ICommunicationObject and then make an Open like this. Now in the same way that I would normally close a proxy, I can close the channelFactory here. Now, let's run the test. But before I do that, I'm going to kick off the host. There we go. My host is running. Now let's run the test. Perfect. As you can see, we passed just fine. Now, let's purposely fail the test to make sure that everything is working the way it's supposed to and that this is indeed a valid test. Host is closed and run the test again. And as you can see, it failed. Now, I didn't want you to fall asleep as you were watching this, so I purposely edited out the 60 seconds that I had to wait for this to actually fail, but it did take 60 seconds because what it does it that it keeps retrying for that amount of time and then determines that an error has taken place. So, now I can actually duplicate this test three times just to test the other three managers and make sure that their endpoints are all set up properly. That is what I am testing here, the fact that I've configured everything correctly on both the service and the client and that I'm able to make calls to each one of those services. (Typing) Now, this one is going to do the exact same thing, but for the IRentalService contract, and this one for the IAccountService contract. (Typing) There. Now, if I run my host, I should be able to run all the tests. Perfect. We're now hosting services successfully, and we're successfully connecting to them. So, next I'm going to return back to this project because I'm going to add an unattended process. I need something to run on a regular interval so long as this host is running.

Unattended Process Intro

The last thing I want to do with my host doesn't really have anything to do with hosting, but it really does fit in this module because it keeps us in the same project, which is the ServiceHost. This is running an unattended process. Basically all I need to do is kick off a process that's going to run at a regular interval and perform some kind of operation. It made sense to me to put this in the ServiceHost because the ServiceHost is going to be running, so it simply made sense, and it was also very easy. I could've made it a little more complex and made it its own application and then I'd be responsible for kicking that application off. If I wanted to truly make it unattended, I would have to account for a system reboot, which means I would have to probably turn the application and do an NT-Service or something like that. And since I'm probably going to do that in production for my ServiceHost anyway, it simply made sense to put it in the same project. So, this unattended process is very simply going to be running on a timer that I'm going to start up on the host when the host starts, and I'm going to set the timer's interval so that it kicks off an

operation or performs a task on a regular basis. Now, what exactly is this task? Why do I need this unattended process? Well, I need to check for dead reservations. I need to check on a regular basis that a reservation is past its pickup date and the car has not been rented by the customer meaning we have to cancel this reservation. Now, our UI on the administration side will have the facility to do this manually, but it'd be nice to have a process that does it automatically on a regular basis so that we're not left with dead reservations out there if a car rental desk employee does not go and proactively perform this process. So, we already have the facility in the managers to take care of this. In the service module I added this, and I'll briefly show it to you again in this module. However, I don't need to access this service in the context of a WCF Service. Remember, my host references the assembly in which the managers reside. So, I can actually just instantiate the service like if it was a regular class. I'm not going to bring WCF into the picture if I really don't need to. Remember, I'm not crossing a machine boundary here, and I'm not even crossing a process boundary. There are some that will argue that even within the same process I can benefit from making a service call to this manager, but in this particular instance my argument is that I simply don't need to. I don't need to overcomplicate things. However, I do have to account for account credentials because remember we're decorated with that PrincipalPermission attribute, and you noticed in the unit test before that I actually had to provide credentials when I'm calling a class that is decorated with that attribute even if it's not by WCF. So, I've got to account for that, and I also have to account for transactions. I still want to be able to make my call transaction aware. Now, I can still do this even though I'm not relying on WCF, and I'm going to show you how next.

Writing The Unattended Process

Here I am in the RentalManager looking an operation called GetDeadReservations. As you can see, it's very simple. It's simply going out to the data access layer and making a call to a data access method called GetReservationsByPickupDate, and it's sending a pickup date of yesterday, which means these are reservations that are still reservations. You see the system is designed to execute a rental whether it's a new rental or one based on a reservation and then eliminate the reservation if there is one associated with it. So, any reservations that are left out there that are not current or going forward in date are considered dead reservations. So, what I'm going to do is write a process to obtain this list and then call upon this operation here, CancelReservation. Now, as you can see, both of those operations have security attached to them, so I'm going to have to set myself up with credentials to actually access them. And the credentials they have in common are CarRentalAdminRole on this one and, I believe, CarRentalAdminRole down here. So, now that I know what I have to call to get this task done, I'm going to go over to the program class where I started up my hosts, and this is where I'm going to kick off this unattended process. Here in the main method, I'm simply going to add some code right after my hosts have started successfully. The code that I want to run needs to happen on a regular basis, which means I'm going to rely on a good old-fashioned timer. Because I am holding in this method after my host starts with a ReadLine here, I'm never going to go out of scope until this method ends or until this application ends, so it's safe for me to go ahead and create the timer inside this

method instead of giving it a class-wide scope. And here's where I choose the interval, and I think I'm just going to go with 10 seconds. Now, this is the event that's going to get fired every 10 seconds, and I'll write that in a few seconds. And now I just need to actually start the timer. And to be consistent with this Console application, I'll write something out to the Console to let you know that this has taken place. Now, in the interest of good housekeeping, after I'm ready to close the host and exit out of this application, I'll stop the timer first, and I'll just write something to the Console here. Now I have to write the OnTimerElasped method. It's a static method of course because it's being called from a static method. Now, here is where I have to put the guts of the unattended process. And all I'm simply going to do is instantiate the RentalManager, and I'm going to do it as a regular class, not as a WCF Service. And even when instantiated as a regular class without the aid of WCF, it's still going to use that default constructor, which means it's still going to call upon the base default constructor, and if you recall, in the ManagerBase class is where I was satisfying my dependencies. So, I'm still going to get all the benefits of all the dependency injection that's taking place. (Typing) Now, I'm simply going to get the list of reservations, the dead reservations. (Typing) Now I'm going to iterate through this list and cancel each one. (Typing) Now, I need to call upon the CancelReservation operation in the RentalManager. Actually, let's call it a method this time because we're not calling it as a WCF Service, which means it's a regular method from a class, not an operation from a service, but I want to do this in the context of a transaction. Even though that cancel is only doing one operation, you never know how that may grow in the future, so I want to make sure I play it safe and do it in a transaction. If I was calling this as a WCF Service, it would handle transactions for me; however, I would then be responsible for creating a proxy using either a ChannelFactory like you saw earlier in this module or a proxy class like I'll show you in the next module. But I'm only calling it as a regular class, not a service. So, now I'm responsible for initiating a transaction, and I don't want to do this outside the loop because I don't want the transaction to encompass all of the reservations that are being cancelled, just one at a time. Now, for this to work, I obviously need to add a reference to the System. Transactions assembly, so I'll do that now. (Typing) And now I have to remember to vote on this transaction, and then when it closes it will commit provided no exceptions took place. The last thing I want to do is go back to the RentalManager and check to see what exceptions might be thrown here. Now, there's nothing being thrown, but remember because of the call to the ExecuteFaultHandledOperation, we may throw a fault exception, not a FaultException of T, but a fault exception if a regular exception gets caught. So, I want to come back out here, and I want to perform this inside a try/catch. Now the level of catching is all going to depend on how much detail I want to report to the client. I could put two catches here, one for the FaultException that I know is explicitly getting thrown by the service and then a catch all with exception. That's only if I really wanted to report the difference between the two. In this particular case, I'm going to keep it simple and just catch Exception, which of course is the class from which all Exception's classes derive, and then I'm just going to report out to the Console that something wrong took place. (Typing) And we're done.

Adding Security Credentials for The Process

Now, if I were to start up the host, the services will start up fine, but then when it runs this code after the timer has elapsed 10 seconds, it would blow up because we're trying to access an operation called GetDeadReservations that's got security put on it. So, I have to do the same thing I did back in the unit test when I was unit testing the services as regular classes, and I have to assign myself credentials to that CarRentalAdmin. And rather than do it in real life, meaning do it by really assigning those credentials to the ID that I'm running under, I want to do it programatically so that you can just run this code without having to add anything to your Windows system when you download it. So, I'm going to come up here, and I'm going to do the same thing that you saw me do from a unit test earlier in the course. (Typing) Here's where I add myself to the CarRentalAdmin. And I'm not really adding myself to it. I'm actually creating a new identity. It just happens to be called Miguel, and I'm already logged in under an identity called Miguel here, but remember the Console app does not have an identity that it's running under by default. So, I'm creating one called Miguel, and I'm adding myself to the CarRentalAdmin. Now I'm going to place that Principal in the thread. And now I should be able to run without a problem.

Bootstrapping MEF and Test Run

Now, before I run this, I just realized by looking at this code that there's one thing that I forgot that's pretty important. When this class runs and this instantiates, it's going to instantiate that base class, ManagerBase. And in the ManagerBase I'm making a call to MEF to resolve dependencies, remember? Well, if you notice up here, I'm not initializing the Bootstrapper. I forgot to do that part and I need to do it. In fact, that's important for even the hosting of the services. So, up here I'm simply going to say (Typing), and that'll call upon the Bootstrapping process that performs the discoverability for everything that was exported. And remember we've exported two assemblies or the types in two assemblies, the assembly that the business engines are in and the assembly that the data access repositories are in because those are the sets of classes that are being exported at this time. And now I should be able to run this. So, as you can see, my services started up. Now, they always started up in the past, but the minute they received an actual call to perform an operation, and if that operation depended on something that needed to be resolved through MEF, we would have run into a problem. But now that we've actually performed the Bootstrapping process, we're okay. Now, as I've been talking, the timer has been elapsing, and 10 seconds has gone by, and you notice that everything has been fine. It did a check for reservations. It didn't find any. If I wanted to be more detailed, I can do something like this. (Typing) There. Now, let's see what happens. So, now my services have started. They are, of course, listening on these endpoints waiting for calls to take place, and the timer is elapsing. And as soon as 10 seconds goes by, we're looking for dead reservations. If we would have found any, you would have seen them listed because of that second Console. WriteLine that I added, but nothing was found because we have no reservations out there. Now, 10 seconds later, it searched again. So, now I'm going to exit the host, and we're done.

Other Hosting Options

Now, as I said at the beginning of this module, I chose to host using a Console app simply because it was just that simple. It was very easy to do, and it also allowed me to write things out to the console so you can see what's going on. So, it makes for a good development experience. I could always offload this information. And when I say this information, I mean the tasks of starting these services here and stopping them and even running the unattended process in a timer to another application. And when you're using ServiceHost, the ServiceHost class, any application can be a host. It can be a Windows app, a WPF app, anything you want. One of my personal preferences is to do an NT-Service as it's still known. I like to still call it NT-Service because it differentiates from WCF's service. But WCF gives us the ability to do that, and I'm going to mock something up quickly just so you can see what the process is like without getting too far into it and digressing too far from this core module. Writing an NT-Service is as simple as adding that type of a project. I believe it's called Windows Service here. There it is right there, and I can actually just create it like any other project. And then in the code I'm given the events that I need to implement. If you've seen Windows Services or NT-Services in the past, and I'm sure you have, they're the ones that are in the administration section of the control panel, you basically start and stop them. You could also pause and resume them. And that's done simply by adding stuff to these events. In fact, I can create a service host instance right here and then open it like I did before. And as long as I have the proper configuration, I will be hosting services, and then I can perform my close on the stop. The important thing to understand here is that I am not going to be holding at a Console. ReadLine, which means that I do not want to create the instance of the service host inside OnStart. I'd want to widen the scope to a class level. And that would really be the only difference. Everything else will be the same. I just wouldn't have the luxury of outputting information, so I wouldn't have all that extra stuff that I showed you with Console. WriteLine. As far as starting up the timer, I would do it in the OnStart and then stop the timer in the OnStop. It really is as simple as that. As far as installing the NT-Service, the Visual Studio documentation has a full walkthrough on how to do that. I personally don't remember the steps by memory, and in fact there are several steps. That's pretty much the only pain in the butt part of an NT-Service. It's not writing it. It's installing it. But once you get it installed by following the walkthrough, it's really easy. All you have to do is go into the control panel, bring up the list of services, find yours there, hit Start, and away you go. And if you set it to start automatically when you reboot, you have an unattended service host that will start automatically if you power down that machine and bring it back up. So, this is my actual preference. Because I am self-hosting, I am still using the ServiceHost class, I'm not hosting through IIS, I can use any binding that I want. And since I typically use netTcpBinding in most of my solutions, this works beautifully. Now, there's a lot of argument for using IIS. I'm personally not a big fan of IIS hosting, but I don't have enough arguments to give you against it to tell you not to do it. IIS hosting is fine. The only limitation is that you're limited to HTTP. There is something on your machine called Windows Process Activation Service, which is turned on in the programs and features of the control panel, and by default it comes turned off on your machine. There's also a script you've got to run that's deep somewhere in Windows system. I never remember exactly where it is, so I hardly ever use was, but was gives IIS the ability to host other protocols besides HTTP, which is fine. I just personally like the idea of having the NT-Service because I have code at my disposal, and I'm kind of a code monkey so I feel like I'm in complete control

of the situation when I'm writing the code for it. So, that's it. Let's do a quick recap, and then I'll tell you what's in store for you in the next module.

Recap and Summary

So, we've written a very simple service host, and in fact service hosts can always remain simple. They don't have to get complex. The most complex part of a service host really is the configuration, and even that is, though verbose, easy to write, and to me it makes a lot of sense. You can host on any other type of app including IIS like I said. My personal preference is an NT-Service, but for development purposes and demo purposes, a console host works just fine. You saw me write some unit tests to actually test the service connectivity. That worked out really well. I didn't have to make a call. I just ensured that I was able to connect to the service, and I confirmed that by actually shutting the host down and then running the test and of course having it fail, so we know that was fine. I also wrote an unattended process to cancel dead reservations. This was not within the scope of WCF, but it was still a process that needed to be done, and I showed you how to do it outside of WCF, but still embed it into an application that I knew was going to be running, which was the service host, so it made sense to put it there. Now, with this service host, I can service any type of client that I want. Now, right now I'm currently configured to service inside the firewall because of the endpoints that I'm exposing, which are TCP endpoints. However, I can expose HTTP endpoints instead or as well and then extend the reach of my services outside my firewall. Now, in this case I'm still dealing with SOAP, and SOAP requires a little bit of pooling, so that's where I say that the interoperability aspects of WCF are not all the way there. There's still a lot to be said for having something as simple as Web API to expose real interoperability, and we'll get into that in a future module. This concludes our business tier. We have the data access layer. We have all the entities. In fact, we have the entities on the client side set up as well. We have my service layer and the business engines that I need. We have the host set up. We're ready to go and service any client that we want, so now we're going to shift our attention to the client side and start writing some code to access all of the stuff that we spent all this time writing.

Client Access

Intro & Hightlights

Hi, this is Miguel Castro and welcome back to Building End-to-End Multi-Client Service Oriented Applications. In this module, we're going to move to the client-side and write the layer that is going to take care of accessing the WCF services. I'm going to go over Client Contracts, which are very similar to the service contracts, and then we're going start talking heavily about WCF Proxies. Now these are the classes that the clients or any UIs are going to use to access the WCF services; these are the classes that are going to provide the layer of abstraction so that the UI

developer doesn't really even need to have WCF knowledge, they basically just need to have a set of objects to access the services with. And, we're going to be talking about SOAP Headers as well. I'm going to bring in Dependency Injection yet again; you're going to see how that's going to incorporate itself into the client-side. And then I'm going to end with the Service Factory - a pattern that I'm applying to the creation of proxies that's going to help out significantly later in the Client.

Client Contracts Intro

We're not going to spend a lot of time discussing client-side Entities. Module 4 covered that in its entirety and they really haven't changed at all since we covered them earlier in the course, but just a quick recap to let you know that these are the entities that the client-side uses and they start out by becoming the target of deserialization or the SOAP message when a call to WCF is made. These client-side Entities, of course, have more intelligence that are used by the client-side, things like property change notification and validation. Now because we have a completely separate set of entities on the client-side that are going to be the target of deserialization, we need to have a set of service contracts on the client-side as well. Reusing the service contracts from the business-side will not work out, because those service contracts return business entities as their actual return type. So we need a set of contracts that are equivalent in naming to return client-side Entities. So for the most part, the client-side service contract is going to look the same as the service contracts on the business-side, but they're instead going to be working with client-side Entities. Now one thing that I'm going to be adding to these client-side contracts, that I may not use right away in this course, are async operations, "Task-based" async operations, and these are not used on the service side, only on the client-side, and they will allow me to make asynchronous service calls if I were to write clients such as Windows, store apps or phone apps. So next, we'll get started writing the client-side contract.

Setting up The Client-Side Contracts

My client contracts will be extremely similar to their counterparts on the business-side, the business contract. In fact, they're so similar that I could actually start by simply copying one into the other, like this. (Typing) And all I would have to change is the reference to. Business and make it. Client. Now I have a client-side ServiceContract that is pointing to the Client. Entities and I can do the same for the other three, or the other two. (Typing) Now for the most part, my contracts are done. One additional thing that I can add to these now would be the async operation. Now, it's not crucial that I do so and, in fact in this course, I'm probably not even going be using them. But in a future part 2 of this course, for example if I write another client that's a mobile app or a tablet app, I would have to use async operations, it would be nice to have them all ready for me and they're not very difficult to add to this. All I've got to do is add one for each of the operations here, actually one for each of any operations that I want to make async, but I think it's a good rule of thumb to just have one for each operation in your ServiceContract. So for example in the UpdateCar

operation here, I would have an UpdateCar async operation that returns a task of Car instead of car, like this. (Typing) And the only attribute that I need to decorate this operation with is the OperationContract attribute. The system will know about the other attributes that it has in its counterpart method, like TransactionFlow or any FaultContract, because it will actually look for an operation of the same name without the async suffix and find this one. So now let me just add the other one (Typing) and now I'll do the same thing for the other contract. (Typing) Now as you can see, this contract actually has some broken code. That's because back in the business-side when I wrote the ServiceContract, I was using custom data contracts in these particular operations, so I now need counterparts to those data contracts on the client-side. I'll create them in just a second. Let me add the async operations first to this and now I just need to add the CustomerReservationData and CustomerRentalData data contracts and I can actually just take them right from the business-side, like this (Typing) and make the same change here. And now I should have no broken code here. (Typing) Now my contracts are all set up and I'm ready to address them with some proxy classes. I want to mention one more thing about those custom data contracts. You noticed that I had two versions of them, one on the business-side and one on the client-side. I do that for consistency; in this particular case, if you notice, those data contracts had their own set of properties. I could have probably put them in the CarRental. Common. Contracts project and then shared them on both sides of the wire. I chose to do it this way because, in the future, I may have a custom data contract that perhaps encapsulates two entities, and in that case, the business data contract is going to encapsulate two business entities, while the client-side would need to encapsulate two client entities, and then, if I have a mix-and-match there, I would end up with some data contracts on the specific side and some data contracts in the common project that are shared and I just didn't want to get to that level of confusion so I made the decision to have two sets of custom data contracts, which right now just happen to be 100% identical. (Typing) Now taking a look at this custom data contract and the fact that it inherits from the common data contract base class reminds me of one more thing. For purposes that I'll describe specifically a little later in this module, I'm going to set up an interface that my client-side service contracts are going to inherit from. And that interface is not going to contain any members, it's just going to be a common place of inheritance from my logical grouping of classes, or in this case, client service contracts. Remember I said earlier in the course, I like to have a base class or a base point of inheritance every time that I have a common set of classes or interfaces that can be logically grouped together. Now I already have this interface set up down here, called IServiceContract, and it's completely empty. So all I'm going to do is go to my client-side contract right here and I only need to do this on the client-side and you'll understand why in one of the upcoming clips, but I'm going to go to the IAccountService and I'm going to inherit from IServiceContract. And now I'm just going to do the same thing with the other two contracts. (Typing) And that's it.

Writing The Basic Client Proxies

A WCF proxy is nothing more than a class that hides all of the low-level code needed to establish communications with the WCF service. This is a luxury we have as. NET developers; this is the SOAP tooling that we get when we work with. NET. A Java developer, for example, should be able to access our WCF proxies as well, but they have to rely on their own tooling and hopefully their tooling has the capacity to generate these kind of things for them. I suspect it does, but I'm not a Java guy so I really don't know for a fact. Now, all I have to do is write a class that hides all of that stuff from the UI developer. This is going to be my proxy class. A naming convention that I like to use, and it's pretty much an industry accepted standard, is giving the class name the same as that of the ServiceContract since there is one proxy per ServiceContract, but without the suffix of service, obviously without the I for Interface, and then with a suffix of client. So the IInventoryService service contract will be proxied by a class called InventoryClient. I have a project set up here called CarRental. Client. Proxies and that's where I'll create my InventoryClient class now. (Typing) Now as far as references go, we're referencing the client-side contracts and client-side entities, any common references that we need, and then we need the System. ServiceModel, which is basically the WCF runtime, and the System. Runtime. Serialization. Now this class needs to simply have methods that correspond to those of the IInventoryService service contract. So it will just implement the IInventoryService service contract in order to get those methods. When the client developer uses this class, it will simply see an object with some methods on it. Now, all of the low-level communication stuff, all of the code that we do not want to write, the stuff that will actually open up a communications channel with a hosted service and establish contact with it and provide the back and forth communication, is done by WCF by way of a base class called ClientBase. I have to inherit my InventoryClient from the ClientBase class and there's a generic argument, which is the type of the ServiceContract. (Typing) Now I have to implement my IInventoryService contract in order to get all the methods or all the operations and expose them to my InventoryClient class. It's important that this implementation is done implicitly, not explicitly - meaning you end up with public methods that are exposed as the actual Interface of the InventoryClient class. Now, each implementation here is very, very simple. All I'm going to be doing is delegating the call out to the Channel property that comes from the ClientBase base class. And because of the generic argument that we sent into the ClientBase class during the inheritance, the Channel property has the same operations that are defined in the IInventoryService ServiceContract and we just pass over the argument. (Typing) And that's it. All I have to do now is repeat the process for all of my other operations. And now I have a fully functional proxy class that a client can simple instantiate and use to access the WCF services. Now I am going to need some configuration to go along with this and I'll do that during the unit testing phase of this module, but right now all I need to do is duplicate this exact same process for the other two service contracts, the IAccountService and the IRentalService service contracts. So I'm going to have two more classes called the AccountClient and the RentalClient. (Typing) There's my AccountClient (Typing) and there's my RentalClient. Now, I want to point out that I'm not going to be adding any constructors here because I'm going to be relying on the default constructor, but if you recall back when I discussed ChannelFactory, I mentioned that a proxy class, just like the ChannelFactory class has several constructor overloads and one of them receives an endpoint name, in the case of more than one endpoint that points to that particular service contract; another one may include

a manual addressing of the binding and the actual address URI. There are several other constructors for other purposes. I don't need to bring that level of complexity into this application because I'm only going to have one endpoint later to point to this ServiceContract. So I can rely on the default constructor and save a little code here. So next I'll cover how I'm going to use dependency injection with these classes.

Making The Client Proxies DI-aware

With all the dependency injection we've done since the beginning of this course using MEF, it should be pretty obvious to you that the first step to setting up any class for dependency injection usage, whether it's with MEF or any other container, is to abstract out its implementation from its definition. So once you have a class separate from an interface, you can export it using the type of that interface, and as long as you tell MEF to go and discover it later, it will make that association to the class from the interface, so later you can use the interface from any other client class and MEF will automatically retrieve an instance of the associated class. Well, if you look closely here with this standard WCF proxy, and I say standard because this is what all WCF proxies look like, you'll see that we're already set up for this. We have this inherent structure set up automatically for dependency injection, just by virtue of the fact that this is a WCF proxy and the way that it is written. So all I have to do for usage of this class, through dependency injection later, is to export it through MEF like this. (Typing) And of course we can't forget the PartCreationPolicy attribute because that's what tells MEF, "do not use the singleton pattern for this. " (Typing) And that's it. Later, I can actually use the RentalClient class independency injection, which means in reality I'll be using this interface for dependency injection, which means just like my data repositories in my Business Engine Interfaces in the previous modules, I'll be able to inject the IRentalService ServiceContract into any client that needs to use this proxy and MEF will take care of instantiating the proxy for me. So now I just need to add this to the other two classes. (Typing) And now, all three of my proxies are set up for MEF exporting and it can take advantage of dependency injection.

Soap Header Intro

One thing I want to do from the client-side is access the SOAP Header. Remember, when I wrote the service layer earlier, I was accessing the SOAP Header in every service call at the service base class, the manager base class, and I did this under the assumption that my client will be placing a username in the SOAP Header. I explained this in detail in the module that covered services. Now I am here at the client-side where I need to actually place something in the SOAP Header. In order to place something into the SOAP Header while you're making a proxy call, I have to access a collection called the "OutgoingMessageHeaders" collection. This is very similar to when my services accessed the "IncomingMessageHeaders" collection and pulled a username out of there. Now I'm going to be doing the reverse. I want to actually put the identity's Name into the SOAP header. Now I'm going to obtain this identity from the thread's current principal. In a desktop app, this identity will be that of a desktop user. In a web application, it will be

established through the authentication scheme, meaning if Windows authentication is used in the website, it's going to be the identity of the desktop user. If forms authentication is going to be used, then it's going to be the custom user that logged into the app. And that's pretty much the authentication scheme that I'm going to be using later when I write the user interface on the website. Now I want to pass this in to every call so the service can pull it out later, but I don't want to explicitly declare it in every operation. That defeats the whole purpose of simplicity. And I don't want the client to have to do it manually. I don't want the client to have to access any SOAP-type code or any WCF code for that matter. I just want the client to instantiate a proxy and use it and that's it. So what's the answer? Custom base class, of course, and that's what I'll show you next.

Adding User Name to The Soap Header in A Proxy Class

I'm going to start by working here in the RentalClient class and then I'll offload the stuff to a base class in a few minutes. So the first thing I'm going to do is add a default constructor, because that's the only constructor that I need; it's the only constructor that this proxy is going to use for my purposes. (Typing) Now the first thing I need to do is obtain the information that I want to place inside the SOAP header. And that's going to be the username that's in the current identity, in the current principal of the thread, like this. (Typing) Remember, in some cases, this will be the same identity sent over to the service as the primary identity in standard WCF base security, but in the case of web applications, it will be different, because when I'm using forms authentication from a Web application or any kind of authentication from a website for that matter, the Thread. CurrentPrincipal. Identity is based on the user that is using the site. Whereas the caller to the WCF service, that the service will receive as its primary identity, is going to be that of the actual caller, which will actually be IIS. So in some cases, the username will be different and that's why it's so important for me to actually send over to the service the actual username of whoever is logged in to the system or database authorization, and I covered that in its entirety back in the Services module. So now I need to place this username inside a MessageHeader class like this. (Typing) Now in order for me to access the outgoing message header's collection, in which I have to insert this information, I have to spin up something called an OperationContextScope and I have to send into the constructor of the OperationContextScope, the Channel class that is actually used in my proxy, actually called InnerChannel in this case. (Typing) Now if I was letting the client take care of all this code from the outside, from outside of this proxy, it's the InnerChannel that's actually exposed publicly, but I want to do it all inside the class; I don't want the client to actually have to see or do any of this stuff. Now the last thing I need to do is insert the MessageHeader that I just created into the outgoing MessageHeader's collection of the OperationContext. Now if you recall back in the service, I actually accessed OperationContextScope. Current and accessed its incoming MessageHeader. This is essentially the same, only we have to spin it up manually here because without me doing this, it will not be done automatically for me until the time of the call, and I need to do it a little earlier. The rest of this is very similar to what you saw in the service class, where I was pulling something out of the incoming MessageHeader's collection using its pair of keys, the Namespace and the Type, String, and System. Here I

am using those pair of keys to insert something into the outgoing collection called the OutgoingMessageHeader, like this. (Typing) Without me doing the line above this, Current would be null right now. (Typing) Now you notice that the MessageHeader class is actually a typed class. We have MessageHeader of string, but what needs to go into the collection is an untyped version of this class, but we don't interact with the untyped version ourselves; we use the GetUntypedHeader method to actually perform the conversion for us. (Typing) And that's it. Now I would normally just need to repeat this process in every other proxy, but it's so much easier if I ad it to a base class, so I'll take care of that next.

Refactoring to A Proxy Base Class

So as I said, rather than repeat all of this code from every proxy that I write, it'll be much easier if I offer a base class, so any proxy that would like to can inherit from it. And I'm going to do that down here in the Core. Common. ServiceModel Project. This is in my Core Framework stuff (Typing) and I have a class called UserClientBase, where I believe this is already set up. There we go. As you can see, UserClientBase inherits from ClientBase of T, it provides a T argument itself, there's your constraint, and it does exactly the same thing that you just saw me do inside the actual proxy. So now I can come back into the proxy, I can remove all of this code, and simply inherit from UserClientBase. And that's all I have to do. Now I just have to repeat the process for my other two proxies. (Typing) And now any client can instantiate any of these proxies and automatically that identity's name will be sent into the SOAP header and be made available to the service, and if you recall, our service was certainly using it. So next, I'll get into the topic of what I call ServiceFactory.

Service Factory Intro

Proxy classes can also benefit from the abstract factory pattern that you've seen a couple of times already. You've seen it in the case of data repositories with the data repository factory and you've seen it in the case of business engines with the business engine factory. Now you're going to see it with WCF service proxies, by way of something called a service factory and the pattern is exactly the same. It is the abstract factory pattern in almost an identical form that you've seen already. Now what this pattern will allow me to do with service proxies is similar to the case of business engines and data repositories; it allows me to receive a proxy on-demand. Here's what I mean by this. We set the proxies up for dependency injection usage, which is going to be great, because we can actually inject the proxies by injecting the ServiceContract into any client that wants to use them. However, proxies should be in and out, meaning they should remain as stateless as possible just for the purpose of accomplishing a specific task. In the case of MVC controller-actions, those are stateless already, meaning every time that you issue a web request to a controller in ASP. NET MVC, it instantiates an instance of the controller, executes the action, and then disposes the controller. So as soon as you issue another request, you get a whole new instance of the controller again. Now later, in

a future module, you're going to see how I'm going to actually inject proxy classes into my MVC controllers using the service contract and I'm not going to have to worry about instantiating proxies from neither MVC or Web API Controllers, because they can be injected using dependency injection. Now, it's great in that case because, once again, I state those controllers are stateless, meaning the proxies will be disposed, they will not be held on to. Now if I were to do the same thing in a WPF ViewModel, for example, the ViewModel is stateful in its nature, meaning its lifetime is the lifetime of the actual view that you have in use at that moment in time. Unlike a web page, whereas once you have that web page rendered, the service stuff is gone already; it's already been disposed. In the case of a desktop application, as long as you're viewing a Window, that ViewModel is still in memory. So if I were to inject a proxy into a ViewModel, I will be leaving a proxy open unnecessarily as long as that view remains open, and who's to say that that user can't open up a view and then walk away and go to lunch. Now I have to accommodate timeouts and take into consideration the fact that if the user comes back from lunch, the Window is still open, they go to initiate a retrieve or an update or something caused by an action, like a button click, and they may have a proxy that will now fail because it reached an inactivity timeout back on the service. So I want to be able to accomplish a proxy on-demand pattern in any situation that I feel I may need it, and in the case of desktop applications, I feel like I'm going to need it. And that's where the proxy factory is going to come in handy. So I'm going to be able to inject a service factory into my client instead of the actual service contract and then use that factory to obtain a proxy whenever I want it and then use it and get rid of it or close it up. And the only thing that remains open or available for use is the Factory class, which will allow me to obtain another proxy on-demand at any point in time, despite the lifetime of the Window that is currently left open. And once again I state this is the same exact pattern used in the data repository factory and the business engine factory. Even the usage is the same; it's meant to obtain an object on-demand and I'm going to still take advantage of the fact that the proxies use dependency injection, the same way that my data repositories and business engines use dependency injection and then their corresponding factories grab their instance from the dependency injection container.

Writing The Service Factory

Remember, this is an abstract factory pattern, which means we start out with an interface, just like we did with the previous factory, and I have that down here in the same place, Core. Common. Contracts. Now if you recall, this is where I have the IDataRepositoryFactory interface and all this was doing was obtaining a data repository, and one that inherits from IDataRepository, there's my constraint. I have the same pattern for the IBusinessEngineFactory and that constraint is IBusinessEngine, meaning the job of this Factory class is to obtain a business engine that inherits ultimately from IBusinessEngine. Well, there's the IServiceContract that you saw that I wrote earlier in the course, from which I inherited all my client-side service contracts, these three up here. So I have an IServiceFactory Interface, which obtains a service proxy, and there's the constraint, IServiceContract. So the job of this ServiceFactory is going to be to obtain instances of proxies that ultimately derive from my ServiceContract. As you

recall, a proxy will inherit from a ServiceContract. Now I just have to write the implementation of the IServiceFactory interface and I'm going to do that right from the CarRental. Client. Proxies project. (Typing) And that's going to inherit from the IServiceFactory interface. (Typing) And remember, the job of this ServiceFactory is to obtain the client proxy from the MEF container. In the future I can actually write another implementation of IServiceFactory that obtains a client proxy from somewhere else or some other container. (Typing) And now, because I want to be able to inject this ServiceFactory using the IServiceFactory Interface, I need to export it through MEF. (Typing) So now I have a working ServiceFactory that I can inject into any client and use the instance that is injected, the instance of ServiceFactory, to actually obtain a client proxy by calling upon the CreateClient method, so long as what I'm sending into T is a derivative of IServiceContract, which is one of my service contracts, one of my client-side service contracts. And remember, the reason MEF is able to find our client proxies is because we exported them by associating them with their appropriate service contract. So coming up next, let's write some unit tests.

Setting up The Client Bootstrapper

I want to bring your attention to the Bootstrapper MEFLoader class that I've been using from my business chair. If you recall, my hosts actually called upon MEFLoader. Init and so did my unit test for that matter and this actually kicked off the discovery process for MEF. So basically what I told it to do was go out to whatever assembly the AccountRepository class comes from and discover everything in that assembly and then do the same thing for the assembly that the CarRentalEngine class comes from. I only have to give it one type in that assembly just to get to the assembly, so it does the actual discoverability. And then with that discoverability having taken place, I can build a container like this. I need one of these on my client-side now and it's going to be called from my client's, which is my running executable on the Client tier, just like my service host is the running executable on my Business tier. Now I already have one set up here, but I want to show you how it differs just a little bit. Notice it's in CarRental. Client. Bootstrapper, it's also called MEFLoader, and this one is just slightly different. It still builds a catalog, as you can see instantiating your catalog, adding the assemblies, creating the container and returning it, just like its counterpart on the business-side. And in this case, I'm telling it "discover everything in the Assembly to which the InventoryClient class belongs, " but I'm also doing a couple of other things here, because you see, this going to be kicked off from either a desktop app or a web app, those are my two clients that I'm going to write in the upcoming modules, and those projects themselves will actually have classes that need to be discovered by MEF. So when I kick off the process from the Client project, that project itself needs to be included in that discoverability process. So, the way I'm going to do it is that I am going to allow the MEFLoader to be called, just like the business-side one was right here, but I'm also going to allow it to be called in a way that it can receive a catalog of parts or a catalog of assemblies from the caller, in this case the Client, meaning the client, let's take for example a WPF app, can set up its own catalog collection of what classes or what assemblies need to be discovered, send it into here, and then here this Bootstrapper will add to it anything that I put here, in this case only the assembly in which InventoryClient resides,

and then it will also add to that catalog that it's building the incoming catalog, meaning all of the other assemblies that the Client has added on its own, and then build a container out of that aggregate list. So it's just a slightly more complex MEFLoader so that it can receive an incoming list of assemblies, as well as the one that it is in charge of building on its own, unlike the business-side one, which only built assemblies on its own. And the reason for doing it this way is because, remember, the client is actually referencing the Bootstrapper. If I were to have the Bootstrapper build everything from here, including the stuff that's inside the actual client project, it would need to reference the client project and, of course, we'll run into a circular reference, which we cannot have. In the case of the Business tier, we didn't have that situation. Now, if the actual service host itself were to have types that need to be discovered by MEF, then we would run into the exact same problem and I'd have to adjust the MEFLoader on the Business tier to look just like this. So now that we have the MEFLoader in place, we can then start writing a client, which in this case will be a unit test.

Proxy Obtainment Unit Tests

So before I jump to writing the unit test, I just wanted to point out that, in case you did catch it, while I didn't earlier, I moved the three Client proxies into the Service Proxies folder, which is not where I created them. So I'm now going to move down here into a project that I have called CarRental. Client. Proxies. Tests and I have a class set up called ProxyObtainmentTests. What I'm going to test with this class is the actual obtainment of a proxy, either through the dependency injection container or through a factory. Now before I'm able to obtain anything from the container, I need to set up the container, and since I just finished setting up the MEFLoader in the ClientBootstrapper project, I'll call that from the TestInitialize method. So I'm going to write three tests and the first one is going to say "obtain a proxy from the container using a service contract. " And I'm going to be obtaining a proxy from the MEF container and then asserting that it's the proper proxy. Now, when I ask MEF for a proxy class, MEF is actually going to instantiate that class for me and the second it instantiates the class, the proxy class is going to look for endpoint information. So I need to actually add endpoint information to this unit test. And the endpoint information is going to be the Client section of the ServiceModel section in an app. config. And if you recall, I did that back in the host module. So I'm going to grab the config from there and make whatever modifications I need to. (Typing) Now this unit test project is referencing CarRental. Client. Contracts, not. Business. Contracts. So now I can write my test. (Typing) And now that I've asked MEF for this proxy based on the ServiceContract, I need to assert that it's the right class that it returned to me. (Typing) Now I can run this test. (Typing) There we go. The next test is going to obtain a proxy from a ServiceFactory and I'm going to provide the instance of that ServiceFactory myself. (Typing) Now you'll notice that I am defining the variable as the interface type, but instantiating the concrete type. That's because the concrete type implemented the interface explicitly, which means I would need to cast this variable to the interface type in order to access any of its members anyway. (Typing) And now I can make the same assertion. (Typing) The last test is going to obtain a ServiceFactory from the container and then try to obtain a proxy from that. (Typing) And the rest of the code

is just like the previous test. (Typing) Now let's just run the whole set of tests and see what happens. (Typing) And all three passed. So the next unit test that I'm going to write is actually going to test the proxy class to see if it can call a running host, which means that I'm going to need to kick off the host.

Service Connection Unit Tests

These last set of tests will test if the proxies can connect to a running host, to a running set of services. This is very similar to the last module when I wrote the unit test to test the connection to the host using Channel factory, only this time I'm going to use an actual client proxy. So I have a class here called ServiceAccessTests and I'm going to write three tests, one for each proxy. (Typing) Now I don't need to write a test initializer or initialize the Bootstrapper, the MEFLoader or anything like that, I'm simply going to instantiate the proxy. We already know we can obtain one from dependency injection, we tested that in the previous unit test. (Typing) Now if you recall, when I wrote the ChannelFactory based tests in the service host module earlier, I actually had to grab a proxy from a ChannelFactory and then cast it to an ICommunicationObject interface in order to call its Open and test that connectivity to the service. But here, I have a proxy already and it itself is a derivative of ClientBase, which is already an implementer of ICommunicationObject, so all I have to do is call its Open. (Typing) And now I'm going to run my host first, and now run this test, and it successfully connected. How do we know that? Let's purposely fail it by shutting the host down and running the test again. Now remember, it's going to wait 60 seconds before it reports a message to me, but luckily you don't have to wait the 60 seconds, only I do. (Typing) And as you can see, the test failed. And if I look at the message, it tells me that it could not connect to a certain endpoint. So now I'll write two other tests quickly. (Typing) And now I'll kick off my host again and run all three tests together. (Typing) There we go. So let's do a quick recap and then I'll let you know what's in store for the upcoming modules.

Recap & Summary

Well this concludes the first module on the client-side, the client access module. And as you saw, we have a really nice encapsulated client tier now and it includes the three primary projects - the contracts, the entities, and the proxies, and we're completely set up for DI. We know everything works because our unit tests confirmed that we can access things from the DI container and that we're also able to instantiate proxies and connect to a running host. So we know we're going to be okay for service calls later. The beauty of this is that now any client can actually reference these project and access our services and, of course, as I stated a couple of times already, the clients will never have access to anything behind the wire. They will only access the services through the proxy layer, using all the contracts on the client-side, and they'll never be subjected or exposed to any of the service code on the business-side or any of the down-level code or any of the connection string information. All of that will be completely abstracted away. Now the remaining modules in the course are going to cover the building of the two UIs, the two clients. The next

module is going to build a website, and the module after that, the desktop application using WPF. The next module is going to cover a lot of information and I'm going to use a multitude of technologies, including MVC, Knockout, and a few other things. So stick around, we're going to have a lot of fun with the next one.

Car Rental Website - Part 1

Intro & Highlights

Hi, this is Miguel Castro and welcome back to Building End-to-End Multi-Client Service Oriented Applications. This module is the first of three modules that will cover the building of the Car Rental Web Site application. In this module, I'm going to cover all of the infrastructure and layout of the site. And also, start working with Knockout as I start building the site's security. So I'm going to cover in detail how the Layout view is set up and what components are used; how the MEF bootstrapping works because, of course, we are going to be using MEF for all the dependency injection; because of that, I'm going to have to write a custom Dependency Resolver; and I also have a Security Adapter in place and you'll see what this is for. This is going to wrap all of the security access to make our controllers very easily testable. I also have some infrastructure to discuss regarding Knockout. This site is going to use heavy Knockout-JS, which means we're going to be doing things in an SPA or or Single Page Application Style, and in order to not repeat a lot of the code that I'm going to be using, I set up a ViewModelHelper class in JavaScript that I'll be going over in detail. And also, every view that uses Knockout is going to have a very specific view structure, which I'll talk about in detail. I've incorporated quite a few third-party components here and I'm going to show you what they are and who wrote them and where you can get more information about them. And then we're going to talk about Single Page Application patterns or SPA patterns. There are several ways of doing SPA. I took the easy approach for the purpose of this course and I'll be discussing in detail, how I'm accomplishing everything that's taken place on the site. And then, after all the infrastructure and layout stuff is done, we're going to cover the implementation of all the security, using Forms-Authentication of course. So first, let's discuss the infrastructure on how the MVC project is set up.

Infrastructure Description

For starters, all of the controllers in this site, including both MVC and Web API controllers, are going to be served by MEF, which means they're going to take part in dependency injection. This means that I can inject all my controllers with any dependencies that I need; specifically, the ones to access the services or the service-proxies, as you know them. Now in order for the MVC and Web API controllers to be served by MEF, we have to write a custom dependency resolver, which I've already done, which will override the default way of instantiation of the controller

classes and ask MEF to resolve them instead. Now in the interest of following good practices and making sure that any service proxies that we use in our controllers are properly disposed, I've implemented a pattern to help me follow some standards when I write controllers. And all of these controllers will inherit from a custom base class, which among another things, will allow me to dispose of my WCF proxies properly. Now all the views in this site are going to use a regular layout view, a master Layout view, and if you've done any MVC, which I'm sure most of you have, you know what a Layout view is, so I won't be going over the details on how you set up a Layout view, but I will be going over my specific Layout view and what it does for you, what it contains. I'm also going to be talking about some third-party components that I'm using in this site. The first one is the Attribute Routing component. Now, this was a NuGet component written by a community influencer and it's a fantastic component. I've been using it for a long time. As we speak, the new bits that are being released for ASP. NET MVC are including this functionality built in, but at the time of the building of this course, it was not released yet. So I'm still using a third-party, which is perfectly fine, it works beautifully and you'll see what that does when I show you in code. Now, for CSS styling, I'm really lazy. I'm very CSS challenged. I'm not the best CSS designer in the world, so I take the easy way out and I use Bootstrap. It allows you to apply some nice and professional looking styling to your site with very, very little effort. In fact, Microsoft's incorporating the Bootstraps CSS framework as their standard in the upcoming MVC project template. I'm also using a library for date parsing, which I'll get into when I get to that part of the code called Moment. JS and here's the link where you can find more information about it. So next, I'm going to bring up the site in Visual Studio and show you how I've set up all the infrastructure and all the stuff that's in place to help us with the patterns and to help us with MEF and the dependency resolver and the layout and all of that stuff. And then afterwards, we'll get into some Knockout details.

MEF Bootstrapping and The Dependency Resolver

So for starters, here's the site's Global. asax and, as you can see, it looks like a conventional Global. asax for any new ASP. NET MVC site, with a couple of additions. I've added the MEF stuff down here. Now if you recall, when I showed you the client-side bootstrapper in the previous module, I showed you that the MEF loader's Init method was designed to receive a list of catalogs from the client itself. So, here's where you see it in use. So as you can see, I'm not just telling the MEF loader to Init itself and return the composition container, but I'm also sending to it an AssemblyCatalog to add to the stuff that it needs to discover and in this case, it's the current assembly, which means the assembly of the CarRental. Web project. That's because there's classes within this project itself that need to be discovered, classes that are being exported. Now, some of those classes are actual controllers. Let me show you the HomeController, which is the one that everybody's very familiar with, that comes with the new MVC Template Project out-of-the-box. Now as you can see, this looks like a regular home controller that you're used to seeing when you create a new MVC project, with the addition of the Export that I've added up here. Now, by exporting the controller, I'm able to tell MVC now, I want you to go out and get the Controller class from MEF as opposed to instantiating it in

the traditional way, after it figures out what Controller class to use from the route. So in order to intersect this process of creating the controller and give it over to MEF so it can resolve it, I have to install a dependency resolver here. Now, I wrote one called MefDependencyResolver, let me go to that class and this is a very simple class. It implements the IDependencyResolver and basically, by default, a DependencyResolver in MVC will instantiate a class that it is asked for, so it receives in the either GetService or GetServices method, a type, and then it just instantiates that's type. Well what I'm doing with a custom dependency resolver is intercepting that process so that instead of receiving the type that it needs to instantiate and simply instantiating it, I'm going to MEF to get it. One thing I want to point out here, MEF by default does not offer the ability to get a type from its repository based on a type as T Y P E. Notice, in previous examples when I said GetExportedByValue, I've given it a generic, but here we don't have a generic, here we have an actual type as an argument. So what I did was write an extension method that extends the CompositionContainer with a method called GetExportedValueByType, that's sitting in the core project, up here, right in there. And I'm going through the list of types that MEF has discovered and when I find the one that equals this one, I'm returning it. So, that's the nice thing about MEF is that you're able to write these extension methods for it as well, though I have no doubt, with a little creativity you can probably accomplish something similar for other dependency injection containers. So, now that the dependency resolver is written, it needs to be installed with a simple line, SetResolver. And now, anytime that MVC figures out what controller class it needs, instead of going to a conventional resolver to instantiate that controller class, it goes to the MefDependencyResolver and tells MEF, get me this controller. And what we accomplish by this is that now my controllers can have dependencies injected into it, that are also exported because, since it's being resolved by MEF, MEF will now resolve all its dependencies and you'll see that when I start writing controllers. Now, unfortunately, writing one DependencyResolver is not enough if we're going to do the same thing for the Web API, which we are, which is why you see that I have a MefApiDependencyResolver, and I have to install that on its own. So, it's just the same thing, but two different classes they inherit from two different interfaces that look exactly the same.

Describing The Layout View

There's a lot of core classes that I've written ahead of time here and I'll be describing them as we approach their usage. You see that there's a controller base class from my Web API controllers, as well as one from my MVC controllers. There's also a filter that I had to write, that the Web API controllers are going to use and I'll go over that in detail when I start writing all of this stuff. And when I get into implementing all the security, you'll understand what this SecurityAdapter class does, as well as the actual implementation right here. And then there's also a UIHelper to help me do a couple of different things in the UI. And I'll be covering all of these as I get to their usage, instead of linearly right now. So right now, I want to go over the Layout page and a couple of other things regarding views, so you can see how all that is set up. My Layout page is not too different from the Layout page that you would get in a conventional MVC new project, with some additions, as you can see it's just a little longer. So, I want to describe to

you what it is that I added here so you're familiar with it because you'll see it on and off as we develop this site over the next couple of modules. I am using Knockout, of course, I think I stated that several times already. I'm also using the knockout. mapping plugin and also the knockout. validation plugin written by Eric Bernard. He's done a fantastic job on this plugin and I'll be showing you how it works and how you write knockout. validation-aware JavaScript models, later on. There's the moment, JavaScript library, and here's the toastr little alert box JavaScript Library, that John Papa wrote, nice job he did with that. And as you can see, just standard script tags for all of this, same thing for bundling up here. None of this is anything out of the ordinary. Now here's the main area of my page, the body. And of course, here in the RenderBody area is where the actual view is going to be rendering when we write a new view that uses this Layout page. Now, I want to show you something that I've done here because for those of you that don't know, and I'm sure a lot of you probably do, you can use Layout pages for rendering all sorts of different sections, and these sections don't necessarily have to contain HTML. They can contain JavaScript as well and some JavaScript needs to be rendered at certain points in time. And that's why you see that I have a RenderSection up here, for a section called head_scripts, and one down here for a section called scripts. And that's because later my views are going to have these sections and they're going to have JavaScript definition in there or linking the JavaScript files. And I want to have that done at this point or this point. Another reason for doing it this way is that, for example, when my views are going to render their JavaScript view models or bring in their JavaScript view models, I don't want to bring in every single view model in the entire site for every single view. Obviously, if I have 25 views in my site, each one is going to have a different JavaScript view model, I'm not going to bring in all 25 of them just for the purposes of rendering that one view. So, that's why I have a RenderSection in the Layout page here because then, when I show you later when I write a view that brings in JavaScript view model, it's going to bring in the script only for that particular view and it's going to do so in whatever section I define it in. So you'll see examples of all of this when I start writing the actual views later on. Now, something else that you may notice here is the TopBar, which is just a common partial control that's going to go at the top of every view, I'm going to show you that in a second. But before I leave this Layout page, I want to talk about this down here. This piece right here is basically so I can resolve the rootpath no matter where my site it hosted; whether it's hosted in local host or hosted on an actual website, you don't want to start doing the calculations as to what's the actual path of the site. You want to let ASP. NET do that for you, since it knows how to do that. The trick here is that ASP has the capacity to do this at server side with the Url. Content and a little squiggly mark and that's how it resolves what the rootPath of the site is. I need to convert that to JavaScript and the easiest way to do it is simply to set up a JavaScript variable called rootPath and as you can see, I'm making it equal to this razor call, but in-between these single quotes. I'm kind of mixing a little JavaScript with server side here. Now in a minute I'm going to show you the app. js file where I define the rootPath and couple of other things. Let me discuss this real quick. And this is the document-ready jQuery event. This is where you would put any code, any JavaScript code that you want executed, once this document has rendered and is ready to go. And this is where I'm rendering a section call ko_apply. Later, when I show you the Knockout stuff, you'll see that the Knockout apply bindings should be done at a certain point in time and it should be done in the document ready and that's how I'm accomplishing

this. I am putting a place holder here so later the view can define the Knockout apply bindings and have it done at this specific point in time. So next, let me show you the app. js file, so you can see some of the JavaScript that I'll be using throughout the site. And then I'll show you the TopBar.

General Javascript Definitions and Their Location

This is my app. js file. As you can see, it's in the Scripts folder and this is where I define all of the JavaScript stuff outside of the context of models and view models that I'm going to be using throughout the site. And first and foremost, this is where I'm defining a JavaScript namespace up here and everything that I write, all variables and all functions used throughout my site, are all going to go into the CarRental namespace. This is so it doesn't pollute the Global namespace, the Global JavaScript namespace. And I'll talk about these other variables here, initialId and initialState, as their usage shows up later during development. Here's that rootPath I showed you. So back in the Layout page, you notice how it was setting CarRental. rootPath =. There's the definition of rootPath right there and as you can see, it's being put into the CarRental namespace. So this little syntax usage that you're seeing repeated here is so that whatever variable I'm defining, falls under the CarRental namespace. I'll be going over the other stuff in this file, throughout the rest of the next couple of modules. This, for example, is a knockout. validation custom rule. This is the viewModelHelper function that I'll be using significantly in all my view models later. And here's a datepicker custom binding (Typing) and this is a loadingwhen custom binding. And I'll be talking about this in detail later on in this module when I go over the Knockout details. So let me show you the TopBar partial view so you can see how that works.

Top-Bar View Description

This view, believe it or not, is not is not as complex as it may look to you. In fact, it's become quite cookie-cutter for me because, since I use this as a reference app from which I start a lot of other projects, I tend to use this whole concept of a TopBar, which is a top navigation bar, in most of the projects that I work on or at least as a starting point. And they all look pretty much the same. All of these classes here, this is all Bootstrap stuff. And this is so that the TopBar actually shows you, not only highlighted area, but also dropdowns and that's what these dropdown classes are here. Now what I'm doing here is showing three primary sections - a Home link, and LogIn link, and a CustomerActions link. There's some security around it. So as you can see, the LogIn section, this part is rendered if you're unauthenticated and this part is rendered if you are authenticated, as this clearly indicates here. So, it's not that difficult to read. You can see that if you're not authenticated, you're going to get Login, a Register, and a ForgotPassword link. If you are logged in, you're going to get a MyAccount, a ChangePassword, and a Logout link, and this is all done with Bootstrap-type dropdowns. Same thing in the CustomerActions. Obviously, that's only going to show up if you are authenticated because if you're not, you shouldn't be able to reserve a car or look at your current reservations or anything else like

that. Now this other stuff here, I'll show you later when I start writing views, that all my views are going to have this ViewBag. Name at the top. And basically, whatever of these top three categories, either Home, LogIn or CustomerActions, a view falls under or is navigated from, that's the category that that view will actually define as it's ViewBag. Name. That way, whenever you get to that view and this page here, this TopBar re-renders; it knows whether to make that top level link in the TopBar itself, active or not. And by active, I mean literally visually active where it shows highlighting. So what I mean by this is that if you were to be sitting at the Home page, the Home will visually show as the active link; it'll be a different shade on the TopBar, but the second that I go into Register, for example, my Register view is going to set the ViewBag. Name at the top to LogIn. And by doing so, when this page re-renders, when the whole TopBar re-renders, it's going to highlight the LogIn link at the top instead of the Home link. That way, you know what section of the site you're in. The rest of this is just a combination of Bootstrap styles, HTML, and of course, some Razor stuff here, URL. Action, which I'm sure everybody's very familiar with already. Nothing else out of the ordinary here. And remember, this will show itself at the top of every single view because it's part of a layout right here. In fact, there's one that's at the bottom also, but that one's very simple. It just shows a copyright statement and a company name and stuff like that. So next, let me give you a little bit of an intro into how we're going to be using Knockout here. And then we're going to jump right in and start writing some code and probably remain the entire rest of the module in the code, where I'll think you'll enjoy it a lot more than PowerPoint slides.

Knockout Intro and Component Credits

Now, for those of you that have never used Knockout before, it's basically a JavaScript-based MVVM framework. MVVM meaning, Model View ViewModel. If anybody out there is a XAML developer, then this pattern should be second nature to you, at least I hope it will be, and I'm also a WPF developer. I spent the last couple years doing pretty much nothing but WPF before the beginning of this year and when I got back into doing web apps, and I'm an old ASP. NET guy so I've kind of fluctuated back and forth, but I stayed out of the web for about two years and when I got back into it, I was introduced to this framework and it just felt right. Coming from the world of MVVM, where the whole concept of bindings and notification and change notification, things like that just made sense to me. All of a sudden, seeing that on the web, was almost a lifesaver because, to be honest with you, I dreaded a little bit getting back into the web side because I remember what it was like and it was cool when things work, but sometimes you have to jump through some hoops, in order to get something very simple to work just right. And now, with the demand for more rich UIs and for more client-centric interaction, meaning leave things on the client and not make that many server trips and having a much more richer client experience with things dynamically changing on the screen, I dreaded as to how that was actually going to work with just jQuery and especially me, I'm rather JavaScript challenged. So, I learned Knockout and it just completely changed the way I write code on the web now. Now, Knockout's not the only MVVM framework out there, it's the one that I happen to like, primarily because it's the one that I know best, but there's also Angular, which is an excellent framework and there's a couple of others out there.

There's one called Backbone, there's several JavaScript libraries that do very similar things. I happen to like the syntax that Knockout uses and in a way it just reminds me of how things work in the XAML world and that's why it just feels right. Writing a very, very rich, client side, friendly web application now is almost a no-brainer to me and I hope you feel the same way if you've never done Knockout before. I'm going to be using it throughout this entire site and I've really gotten to the point where I don't write MVC sites unless they're written in this way so I hope you enjoy learning it. Now, all of my Knockout-aware views are going to follow a very specific layout pattern. I briefly touched base on this earlier, where I talked about what a view will be set up like, regarding different sections, where sections will not only render HTML, but some sections will also render JavaScript. Well, this is going to be a very important pattern to follow in the Knockout-aware views and pretty much every view that I have is going to Knockout-aware. So you'll see, coming up shortly, we're going to start writing views and you'll see how these patterns manifest themselves. You'll also be seeing throughout this course, you'll be seeing a couple of custom components that I grabbed from the web here and there and I really do have to give credit where credit is due for some of these. There's a very, very simple binding handler called Loading-When or the loading placeholder, I believe he calls it, that a guy named Steve Greatrex wrote, and it's very, very cool, very, very simple, and I really like what it does, and I'll show you how it works and what it does, like everything else, as I approach its usage. Same thing for the Custom date-picker that I'm going to be using and also for Knockout-Validation and we're going to be using Knockout-Validation heavily because I showed you earlier in the course, back on one of the earlier modules, how I set up validation rules in my client-side entities. However, for the web, we're going to need some validation done immediately while still on the client. And remember on the web, when we're on the client, we're on the HTML side. So we're not even looking at those client-side entities; that's all sever stuff. So we need to have some good client-side validation that tells our clients something is wrong, right away. And that's what Knockout-Validations going to be for. So enough slides, let's get into the code and start writing some views. We're going to jump right in to developing the security for the application and that's how I'm going to show you how the views are set up, how the controllers are set up, how the API controllers are set up, and also how I handle SPA or Single Page Application. Now, we're going to start by writing the Login page, the simplest one, and then work from there on complexity.

The Security Adapter Class

Now, I'm going to start by writing an account controller class and work from there, but before I do, I need to bring your attention to this SecurityAdapter class. Now, I'm going to being using the simple membership provider because just that, it's simple and it works fine. So, this simple membership provider uses this webSecurity static class to do everything regarding security, like the initialization of the DatabaseConnection, the creation of these fields if they're not already there in the table, which in my case they are and I showed you this in the Database module earlier in the course, the creation of a UserAccount, the Login, the changing of a Password, the determination of whether a UserExists or not; these are all things done by this static class, but this static class is designed to go to a database to

obtain this information, and what database does it go to? Well I tell it that up here. So what I need to do is that I need to give my controller the ability to do all these things, but, I want to provide a layer of indirection between that because I want to make my controller testable, which means I don't want to lock in my controller to using the webSecurity class only. Because then if I instantiate that controller from a unit test and test some of its actions that may call upon webSecurity, then I'm going to be hitting the database and actually doing security stuff or database security stuff during a unit test and I want to have the option on whether I want to do this or not. So, this is very similar to the pattern that I put in place earlier in the course regarding Business Engines and Data Repositories, where everything was abstracted out to an interface. As you can see, this SecurityAdapter implements that ISecurityAdapter interface and all the SecurityAdapter does is wrap the common things that I'm going to be calling upon the webSecurity static class to do. It just provides a wrapper for them. As you can see, the WebSecurity. CreateUserAccount, which uses the LoginEmail, password and some propertyValues, is wrapped by the Register method, which provides the LoginEmail, password and propertyValues. Same thing for Login, and ChangePassword, UserExists. And the ISecurityAdapter interface, as you can see, is just the abstraction of this class. That way, I could now inject, because as you can see, I'm exporting this class through MEF, which means I'm going to able to inject this interface into any controller that needs to use it. And of course in production, the controller will receive an instance of SecurityAdapter and do all this stuff, but if I instantiate that controller in a unit test later on, I can mock the ISecurityAdapter and send that in instead and never hit the database for the purpose of my unit tests. So, that's the SecurityAdapter description. Next I'm going to be creating a controller called the AccountController and we're going to get started from there.

The Account Controller Setup

So as you can see, I have two subfolders in my Controllers folder, API and MVC. I like to separate the controllers out. So first, I'm going to create a controller call AccountController. And the first thing I'm going to do is tell it to inherit from my custom base class, which is called ViewControllerBase, right here, and this class inherits from the Controller class so I'm ultimately still inheriting from MVC's controller class. I'm also going to export my controller so that MEF knows how to find it later on. And I'm going to remove all of the default stuff. This controller is, of course, going to have an action called Login and normally this would just return a View, like this, but I'm going to be spiffing it up just a bit. So first of all, let's define how we get to the AccountController and Login action because conventionally, you'd rely on the routing table and if you rely on the default routing, you know that there's a pattern in place where it's part of the URL for the controller and part of the URL for the action, which means that the URL of Account/Login would get you here. Now, I can leave that alone and it would still work, but I'm going to be explicit in my routing declaration because later on throughout the site, I may have actions whose naming does not particularly match the route that it took to get there and I want to maintain consistency across the board for all of my actions and all my controllers. So, this is where I'm going to bring in the attribute routing plugin. Now, the attribute routing plugin allows me to, instead of

going out to my routing table, which was out here and adding another route, telling it what URL gets me to the AccountController Login action, I can just find it up here using an attribute, and it makes itself documentable because if I were to use the routing table like I used to in the past, I always ended up putting a sample of that route, like this, at the top of the action. That way when me or any other programmer is looking at that action, we know specifically what route it took to get there, in case it's not obvious by its naming. So with the attribute routing plugin, what I get to do is the following: Now notice there's two attribute routing plugins, one for MVC and one for the Web API. They do the same thing, but they're two different sets of classes so you've got to make sure you include the right one here and here I give it the route. Now, I also like to explicitly declare the HTTP verb that this action is assigned to. Now, in the implementation of this attribute routing that Microsoft is working on, that they're coming out with in the next version of MVC, these two attributes are going to be combined into one, so you're going to be able to say, HttpGet and then give it the route in the constructor instead of having two separate attributes like this. Now, the other thing I want to do from this Login action is call upon the SecurityAdapter's initialize, which is going to set up the DatabaseConnection if it's not already set up. As you can see, there's a static property that gets set automatically so we don't hit this code here more than once, but I need to call upon the SecurityAdapter. Initialize from here. Now I already told you, I don't want to make a call to the webSecurity static class from here because then the class is going to become untestable, the AccountController class will be untestable. So what I'm going to do is that I'm going to inject into it, the interface that my SecurityAdapter inherited from, like this. (Typing) And in order to tell MEF, use constructor injection this time instead of property injection, which is what you've seen in the past, I'm going to decorate the constructor with ImportingConstructor. This is actually quite a nice feature that MEF offers because it avoids confusion when you have multiple constructors. Now this attribute tells MEF specifically what constructor is going to be involved in dependency injection. So now, back to my Login action, I can do this. And later, when I instantiate the AccountController class from a unit test, I can provide this constructor with a mock implementation of the ISecurityAdapter and I'll never hit the LiveSecurity method. Now, the view that I'm going to put behind this action is actually going to be a Knockout-aware view and that Knockout-aware view is going to call upon a Login view model class that I'll write in a few minutes, and that Login ViewModel class will use a JavaScript base model class to store all the information that I'm collecting about the Login task, meaning the Email, the Password, a RememberMe Boolean, but I also need a server-side version of this because I eventually need to bring this from JavaScript back into C#, which will mean deserializing a JSON stream into a C# object and then I'm going to be doing something with that C# object. So, the task of me writing a conventional mode class to apply to that view, that task of providing that class has not gone away, I still need it, and the reasons will become apparent in just a few minutes. So I'm going to come out here to the Models folder and I'm going to create a class called AccountLoginModel. And as you can see, it includes the LoginEmail, the Password, a RememberMe Boolean, and it also includes a ReturnUrl. I'm going to show you what this is used for. Back here in the controller, we're going to go to a view and I want to send that view any URL that might have come into the LogIn action. Now, what kind of URL would come into the LogIn action? Right now we're not defining anything, well what if I do, like this? Now what happens is there that there are cases where the

LogIn action will receive this and it relies on this. Let's go out to the Web. config real quick and you'll notice that in my Authentication section, we have traditional forms authentication and we've been seeing this since ASP. NET 1. 0. And here's my logInUrl path. This means that when I reach an authorized-only controller or controller action, one that's decorated with the Authorize attribute, meaning an action or a controller that's not going to allow access by an unauthenticated user, this tells ASP. NET automatically route me here, which means it's going to get me here. And whatever URL I try to access in an unauthenticated state, that URL will be automatically sent into this argument. Now what I got to do is that I got to send this to my view, because later my view is going to need to send it to my ViewModel and I'll show you how I do that a little later, but I need to send this to the view by binding this model class right here to my view. Now I'm not going to use this model binding for creating text fields on the screen like you've seen in traditional MVC apps, but it's still going to be model binding. So, I'm going to come back out here to my controller and I'm going to change it to look like this. (Typing). And the only property I'm interested in setting is the ReturnUrl because I need to get this over to my view because later, my view's going to need to get it into my JavaScript ViewModels. I'm not interested in the other ones, LoginEmail, the Password, and RememberMe because they're not going to be used for binding onto the view; you'll see what they're used for later on. So now that I have my Controller action set up, I need to write my view. So the first thing I'm going to do is do it the easy way by simply right-clicking here, saying 'Add View', Use layout or master page, and when I hit Add, you see that it creates my Login. cshtml inside my Account folder under the Views, just like I expect it too. So next, I'll show you how this view is set up.

Setting up The Login View and The Knockout Model

Now I'm going to modify my Title here to say Account LogIn and I'm also going to add this line here. (Typing) And if you recall, this was so the TopBar right here, when it finds that the ViewBag. Name is LogIn, it knows what link at the TopBar to set active. So now I'm going to set up my model binding definition up here (Typing) and now let me show you how this view's going to be laid out because this is pretty much going to be the standard for every view going forward. So not only do I have this here, if I had a server-side model that I wish to bind to this view, for whatever purpose, and in this case I do because I need that ReturnUrl, and then the ViewBag. Title and Name are set there. My next section here is call script. (Typing). This is where I'm going to bring in any JavaScript or CSS that is specific to this view only because if you recall back here in the Layout, I'm rendering scripts at a specific point right here, but this is so I don't have to declare any JavaScript that belongs only to this view or any CSS that belongs only to this view up in the Layout page, where it'll be rendered no matter what view I'm in. This is for the view-specific stuff. So I'll get to the inside of this script section in a second, but it is going to be Knockout stuff that I'm going to be bringing there. And since I'm using Knockout, I'm going to need to apply the Knockout binding at some point and I do that at a very specific point and, once again, back in the Layout page, that's done inside the document ready. So I need to define the section called ko_apply, that's inside document ready here. And the rest of this is going to be the actual HTML. Now my HTML is going to be very simple here. It's simply going to have a textbox for LoginEmail, of course, one for

Password, and a checkbox for RememberMe. Now because I'm using Knockout, these three elements are going to need to be bound to properties of a ViewModel. Now, rather than put the properties in a ViewModel themselves, I like to actually have the properties in a separate model class, but nonetheless we're still talking about JavaScript here and I put all that stuff in my Binding folder and as you can see, I have a folder set up for Models and View Models. So the first thing I'm going to do is add a JavaScript file here called AccountLoginModel. (Typing) Now I'll code this one step by step and then the other models, going forward in the application, are going to follow the same patterns, but I'll make them appear a lot faster. Normally, I would start out with an AccountModel class like this, (typing), but as I stated earlier, I don't want to pollute the Global namespace so I want to make sure that this is inside my CarRental namespace So I have to add this around it. (Typing) Now with this syntax, my AccountLoginModel JavaScript function is going to need to be referred to throughout the rest of the project as CarRental. AccountLoginModel. Now, one of the first things that we typically do is something like this. That way we can differentiate this in this particular context from any other context that this may represent in this function, even though there will be no other context; it's a good standard to get into. Now, I'm going to have three properties here and I'm just going to declare them quickly in the basic JavaScript Knockout way and then I'm going to add something to them. (Typing) Now I have three properties and I can use this in my HTML to bind to and the beauty of this is that changing something in the HTML controls will automatically update the properties in this model and, of course, the other way around also is true, whereas changing something in the properties will update the view and that's the whole beauty of MVVM, but, I want to add some validation to this and I'm going to do that using the Knockout-Validation plugin and this plugin is done with the extend method on the Knockout observable and this is what it looks like. (Typing) Now I want to add two rules to the LoginEmail property -- I want it to be a required field and I want it be an actual email and there's already an email validation rule built in to Knockout-Validation. So let me show you what this syntax looks like (Typing) As you can see, I've extended the Knockout observable property with two custom properties here, required and email, and this is all part of Knockout-Validation. And then you tell it what the message is in case it fails that validation rule. There's already a validation rule call required and there's already one call email. And you can write your own too and I'll do that later on when we do the registration view, but this is all it takes to add validation rules to this. Now, just as important adding validation rules is how we check them and how we report on them and you'll see that coming up, but right now, let me add the password validation rule as well. As you can see, I'm going to make the password required and it has be at least six characters long and I'm not going to have any validation rule attached to the RememberMe. Now that I have my JavaScript model set up, I need to set up a JavaScript ViewModel to service Knockout and that's going to be bound to the actual view that we were in the middle of right here. So next I'm going to write a class and I'm going to put that class here in ViewModels and it's going to be called AccountLoginViewModel. (Typing)

The Rest of The Login View

So the syntax that I'm going to use to declare the AccountLoginViewModel function is the same as what you saw in the LoginModel, where I have to put it inside the CarRental namespace like this. (Typing) So now we're ready to start coding inside of it. Now the first thing I'm going to do is add a property here that represents that AccountModel that you saw that I created right here. And I'm going to put that property right here because my view is going to bind to this. (Typing) And as you can see, I have to address it using the namespace. So now that my ViewModel is somewhat configured and has an AccountModel property in it and we know that the AccountModel represents this class, which has a LogIn email, a Password, and a RememberMe, I can continue setting up my view and working under the assumption that I'm going to be okay because I'm binding to a ViewModel called AccountLoginViewModel that has a property call AccountModels. So I'm going by that information right now and I'll continue building upon it. So I'm going to come out here and I'm going to start putting some HTML in here. Now, my entire form that constitutes the Login form is going to go inside this div tag and this div tag is set up with an ID. The reason for that is because in Knockout you can say apply bindings to an entire view, but you could also apply a ViewModel binding specifically to a section so that the contents of that ViewModel, the properties of that ViewModel and any model classes within it, are bindable and usable only by that tag that you gave it. In this case, I have a tag here call Login and I like to limit that kind of scope so there's no confusion and a programmer doesn't think that that ViewModel is used by the entire view. It's very explicit on what's it's used for. First thing I've got to do is bring in these two classes, the AccountLoginModel and AccountLoginViewModel, into here and I need to set that up in my script section here. Now the next thing that I want to do is then I want to instantiate my ViewModel class itself. (Typing) Now one thing that I want to add is I want to take that ReturnUrl, remember we got to this view from the action, the Login action, that Login action sent an instance of AccountLoginModel with a possible value in the ReturnUrl property. Well, that ReturnUrl property came into here, which means it's available if we access the model property of this view and, of course, the ReturnUrl property of the Model property. And what I want to do is convert that to JavaScript and I can do that easily like this. Now notice I'm inside quotes here, but now I'm going to access Razor syntax. Now, I have a JavaScript property that includes the value that came into this property here and what I want to do is get that to my ViewModel, which means I'm going to send it into here. And if I'm going to send it into there, I, of course, need to declare it here. (Typing) Now next, I'm going to take this class instance viewModel and bind it to this section right here so that now any HTML in here can benefit from the properties exposed in this JavaScript viewModel. Now normally I can do something as simple as this (typing) and that will bind the viewModel to this entire view, but I want to isolate it further like this. (Typing) Now what this will do is bind the viewModel only to that tag right there. Now, I'm going to use a different syntax here and this syntax is going to look a little complex and this syntax is because I'm using Knockout-Validation. So you have to bind it in a specific way with this WithValidation if you're going to be using Knockout-Validation so you can report on those errors later on. So think of this as a cookie-cutter pattern; you're going to be seeing it across the board in all of my views. Now, I just need to build the HTML to show the Login form. (Typing) Now this HTML is not as complex as it may look. It's regular HTML, is using Knockout binding, which means for this div tag I'm saying change the context to the accountModel of the viewModel you're bound to. Now remember, in the LoginViewModel, we

have this accountModel property that we're exposing, so that now becomes my context, which means anything inside this div tag is going to refer to properties of accountModel. And as you can see, this input textbox here is bound to the LoginEmail, this one is bound to the Password, this checkbox is bound to RememberMe, and those are the properties of LoginModel. Now what's the rest of this stuff here? This is all Bootstrap stuff. These are all just styles that I've grown fond of and accustomed to using so that my forms look somewhat elegant and professional, but you can make this form look like whatever is that you want. This is just regular CSS styling. And like I told you, I'm very CSS challenged, so if I can find something that's cookie-cutter that I can just reuse from now until the day of Armageddon, I'm going to do that. Now, this down here, let me explain what this is. This is going to be a button that provides the Login. Now this button is obviously going to need to bind to a function. So let me go back to the ViewModel and fulfill that by writing that function. (Typing) Now I have a function exposed by the ViewModel call login so let's go back to the view and I'll further explain. I'm binding the button to the Login of the ViewModel. Now the reason I have the parent here is because, remember, as long as I'm inside this div tag, my context is set to AccountModel. So I am inside this property right now. I need to come up one to the parent so I can access Login and that's why you see '$parent'. Now, here is that loading placeholder I told you about that was written by Steve Greatrex and he did a great job with this and I'll show you the ViewModelHelper a little later, but basically what this is going to do is that it's going to depend the content of this div tag on a property of my ViewModel that you haven't seen yet; that is actually in the ViewModelHelper, it's call isLoading. And if the isLoading property is set to true, then all of the content in here, meaning all the contents of this div tag, is going to be replaced in its entirety with a loading indicator, a little graphic that shows you a little animation, and only when the isLoading is turned back off to false, will it actually show this. And you're going to see that somewhere in the Login function, I'm going to set the isLoading to true, which means once you press the button and you're performing a login, the button's going to disappear while the task of that login is taking place so that you don't run the danger of pressing the button again. And that's the beauty of the isLoading indicator and it goes hand in hand with a custom binding handler that Steve wrote, call LoadingWhen. So now that we have form set up and it's bound to this ViewModel and clicking on that button will actually cause this event to fire, we need to fill in more stuff on this event and that's what I'll do next.

Checking Validation Rules and Reporting Them

So I'm going to write the contents of this function line by line, explaining to you in detail what's going on. And then, as I said before, all the future stuff going forward is going to follow the same pattern so I'm not going to have to re-explain it again. So the first thing I want to explain to you is where this is coming from right here, the model property. If you remember, back in the view, we're inside the context of this accountModel and that's where the click is taking place, which means when we get back here to the ViewModel, because of that click having taken place, the modified property accountModel, and I mean modified because remember the accountModel contains the LoginEmail, the Password, and the RememberMe, so whatever the user sets this to in the user interface will automatically update that

accountModel property and that updated property is what comes in right here. So the first thing I need to do is rely on the Knockout-Validation and figure out if the contents of that model object, if the contents of those properties meet the validation requirements and I do that like this. (Typing) Now if errors contains anything and it's an array, that means that one of these validation rules here is broken. This means that I really want to continue here only if errors does not contain anything. (Typing) So now the question is what if errors does contain something? What do we do? And obviously, we want to establish some kind of pattern, some kind of standard to use on how to report errors to the screen. So let me go back out to the Login view here and let me add a piece of code right here, which brings in a partial view call ValidationErrors and let me show you what ValidationErrors is. ValidationErrors is a simple div tag, whose visibility is dependent on a regular Boolean that happens to be in the ViewModelHelper, which I haven't gotten to just yet. So whatever this value is, it's either going to be true or false, that's going to determine the visibility of this entire thing. ModelIsValid, remember that. And then inside of it, it's simply going to list out the contents of an array call modelErrors. You may see where I'm headed with this already. One Boolean to determine visibility or invisibility of an Error panel, in an array, to list out any errors. So now let's go back to the ViewModel and the first thing I'm going to do is that I'm going to instantiate my ViewModelHelper up here. (Typing) Now ViewModelHelper contains a whole bunch of stuff that I use on a very regular basis. I'm starting to introduce you to the ViewModelHelper by showing you the error validation stuff, but it's going to be used in just a few minutes for a couple of other things. So the first thing I want to do down here is set that modelIsValid to either a true or a false based on the errors that are returned from the line above it and I can do that with the isValid property that's automatically extended on the model object. (Typing) Now based on this execution right here, not only are we going to get a return array back that's either empty or has errors in it, but the isValid property, extension property, of this variable will be set and I can use that since it's just a regular Boolean, I can use that to put it into my modelIsValid. Now modelIsValid is a Knockout property, as I'll show you in a couple of minutes. So it's a Knockout property, which means it's a Knockout observable, very similar to these here, which means you have to set it using this syntax. You don't say modelIsValid equals because in reality a Knockout observable is a function, so you set it by sending something into its argument. And setting that property will automatically refresh anything in the view that depends on it. And once again, back here to the ValidationErrors, the visibility of this div depends on the value of modelIsValid. So now, the only thing I have left is if there are errors, meaning in the else clause, I'm going to set modelErrors property like this. (Typing) And by setting the ViewModelHelper's ModelErrors array to the array that was a return of the validation check, here in the ValidationErrors panel, you're going to see that I'm depending on the content of modelErrors and I'm just doing a Knockout for each and listing all those errors out there. And this is the pattern you're going to see going forward for checking an incoming model into a click action or a user action in a viewModel, checking it for validation rules, and then determining whether we have validation rules or not, if we're going to continue, and if not, how do we report on those validation rules? Okay, so now that we're in here and we don't have any errors, we could now proceed with the actual login process.

Test run for Validation Display

Now before I continue writing the rest of this code here, let's test what we have so far to make sure things are working fine and I just noticed, I wonder if one of you caught this, that I'm running into a little bit of a stack overflow potential here because I'm instantiating AccountLoginViewModel inside AccountLoginViewModel. This was a typo, this was supposed to be AccountLoginModel. So now that I have that done, let's do one more thing here just for tests. I'm going to put an alert box right here (Typing) and now let's run the application and see what happens. There's my services. And here's the site. And as you can see, the Layout has taken care of making sure that everything looks fine; the body here, there's the TopBar, there's the BottomBar, here's my login part of the TopBar, those are my options. Now I'm going to click on login and as you can see, I'm using a Bootstrap metro type of plugin that I found, it gives us a little bit of that Windows modern app-type look regarding fonts and square boxes and things like that. So, let's go over to the Login page and see what happens here. And there's the login. So now, I'm going to test the validation. Let's just go ahead and press the Login and see what happens. There you go. So we got to the click event, we pulled out the ModelErrors and the modelIsValid, the ValidationErrors panel showed itself based on those values, and now, let's see if we can perform something that's valid. (Typing) And we should see that textbox now or the alert box. There we go. Okay, so in the next module, I'm going to start writing the code that goes inside here, in this ViewModel, which will also bring us into working with the Web API and setting up the API controllers. And then the next part of the site we're going to be building SPA or Single Page Application. So there's some really cool stuff coming up next.

Car Rental Website - Part 2

Intro & Highlights

Hi, this is Miguel Castro and welcome back to Building End-To-End Multi-Client Service Oriented Applications. This is the second of the three modules that cover the building of the Car Rental Web Site. In this one, I'll be bringing Web API into the mix and also covering Single-Page-Applications, which is the style in which we're going to write all the upcoming views after the Login View. Let's go over some of the highlights of this module. I'll be showing you everything you need to know about using the Web API, including writing a custom base controller, which I already have; how to use attribute routing, which you learned already doing MVC controllers, we'll be doing that with the Web API controllers as well. I also have a pattern in place that all my operations or my Web API actions are going to follow; the same as I had a pattern for my WCF service methods. Now you're going to notice that Web API is going to offer the embracement of HTTP as opposed to the abstraction. When we use WCF, we abstract everything regarding the transport away so we don't have to deal with it. Web API requires a little more embracement of HTTP, of the actual transport itself. So you're going to be learning about request messages and response messages and how to

properly do error handling in the way Web API wants it, which is using HTTP status codes. I'm also going to be showing you what a Single-Page-Application is or SPA, which is the style of development that I'm going be using going forward after we're done with the Login View. This has to do with one delivered view, one physical view from MVC, that's separated of the sub-logical views or sub-sections of HTML within it, and these sections can flip themselves off and on, all using client-side. This is without a server trip except to go out and get data, in which we're going to be using the Web API of course. So this is going to offer the richest user experience. Think Gmail; Gmail doesn't really go to the server to redo its screen or to render its screen, it just goes to get data, but everything stays on the client side. Now there's many techniques for doing SPAs. I'm going to be discussing that and I'll show you my technique for it, which is simple, but it's still elegant and it works very well. So, for the remainder of this entire module, I'm going to stay in the code, you're not going to see anymore slides, and I'll get started by completing that login ViewModel, and we're going to need Web API to do this.

Setting up An API Controller

Now the task of actually logging the user in using the information grabbed from this model argument is going to fall upon C# or the server-side, because remember it's going to use that adaptor. So the first thing that I've got do is figure out a way to get there and the way it's traditionally done with Knockout is you issue a jQuery REST call to a Web API or some kind of REST service. In this case, I'm going to use the Web API of course, I'm already in a. NET application here. So I want to actually send the information inside this model variable into a REST call, into a Web API call. So, in pseudo-code it would be something like this; I'm going to make a call to a REST URL called api/account/login and I'm going to send to it the information in this model. Now I'll show you how to do that in just a second, but first let's start out by setting up a Web API controller that can receive this URL into one of its actions. So I'm going to do that up here in the Controller section, I'm going to Add an API controller and I'll call this one AccountApiController and, like an MVC Controller, I have a base class for this one as well, it's called ApiControllerBase, and I'm also going to export it just like I do any MVC controller. Now, this controller is also going to receive the ISecurityAdapter interface as an injection, just like the AccountController in MVC here, which I used to call the Initialize. So I'm going to add a constructor to this Controller, inject ISecurityAdapter into it, and of course widen its scope just like I did in MVC. (Typing) And now I'll tell MEF to use this constructor for dependency injection. And remember, all of the controllers served up my MVC, by ASP. NET MVC, whether it's a controller for an MVC view or a controller for the Web API, they're both served up by ASP. NET MVC. They're all going to be routed through the dependency resolver, which in turn is going to intercept the process of controller instantiation and grab them from the MEF container. And since they were all exported, thanks to this, it's going to find them, and when it resolves them, it'll resolve all their dependencies. Now I stated earlier in this comment here that I want a REST call to go to a URL called api/accounts/login. So that's what I'm going to give my controller and I'm going to do it using a regular action called Login, like this. Now I'm going to change the arguments and the return type of this action in just a second, but

first, let's tell it what verb to use to get here. Well, that all depends on how the call's going to be made. So, here I'm not requesting information, I actually want to take the data inside this argument and post it to this URL. So that's going to be a POST verb that I'm going to initiate and it's going to be done with a jQuery POST action, which you'll see in a few minutes, but first I have to tell this action here that it's going to be an HttpPost. As always, I'm a big fan of explicitness. Whether it's a GET or a POST or any other verb, I'll always put the attribute on there and I also really dislike the use of the convention over configuration on the Web API. In other words, prefacing the action names with the word GET or the word POST and relying on that. I like to see things explicitly and there's a lot of people out there that disagree with me, I honestly don't care. I like writing just a little bit more code and eliminating all guesswork out of the equation. And the last bit of guesswork elimination is setting up the route that will get me to this action and that route was going to be api/account/login. So using the AttributeRouting plugin, I used the POST attribute like this, and this one is going to use the web. http namespace, not the MVC one, because now we're in a Web API controller, and this is where the route's going to go. Tell me that's not a lot cleaner than having an action method called POST or POSTLogin or something like that. This is much cleaner, it eliminates the guesswork, it's self-documenting, and you don't have to think too much to figure out what's going here. Alright, so here's what's going to happen now - there's different ways of doing Web API actions. I have my way, a lot of people agree, a lot of people disagree, but I like to make everything consistent and follow a specific pattern. And the pattern I want to put in place here is a very, very RESTful pattern, meaning I am not letting Web API abstract anything out for me; to me that defeats the whole purpose of Web API. And one of the reasons they wrote it in favor of WCF REST extensions is that Web API should embrace HTTP and not abstract it and one of the things that I see in most API usage out there is that it doesn't embrace it enough when it comes to an HTTP Request and Response message. So I like to explicitly, always return an HttpResponseMessage. Even though a lot of Web API examples will show you that you can return a model class right here, a model class is going to be wrapped in an HttpResponseMessage and at your client, where you read this information from, you're going to need to pull it out of there. And I like to be explicit and actually return an HttpResponseMessage right here from my Web API action, like this. The other part of my pattern is that the first argument of every Web API action method is always going to be the HttpRequestMessage. And yes, Web API accommodates this and it's prepared for this. So, just because you see a lot of Web API demos out there where the action method starts having some regular arguments in here that perhaps are sent in the query string or something, if the first argument is an HttpRequestMessage, not Header, Web API will detect that and it will accommodate it and I'm going to be using this argument in just a couple of minutes. Now, the model information that I'm receiving as a POST instead of being my first argument in the action, it's going to be the second one. Now remember that model object that I set up earlier, this one here, that's what I'm going to be converting, because remember, in my JavaScript Knockout model, the Knockout JavaScript model right here, I have properties called Login, Email, Password, and RememberMe. Now what I'm going to do is that I'm going to take those properties, I'm going to take this object that comes in to the ViewModel, and I'm going to turn it into JSON and send it through a Web API call, and automatically the Web API can receive it inside an argument here and as long as it contains properties of the same name, meaning it is an equivalent contract,

it will deserialize it into that object. So I can actually say, and now tell it, this is going to be the body of the POST and now I can make this call. Now I'm going to return a null here just so I don't break compilation and I'll fill it in later, but now I can go back here and work under the assumption that I have a functional REST API call listening and waiting to receive calls and I'm going to call it from here, sending this information in. That's what I'll do next.

Finishing up The Login Viewmodel Function

So, though I haven't finished the meat of the that API controller action, I can still work under the assumption that I have it finished because the controller's already set up and the routing is set up. So, now I can actually call it or show you how I can call it. And the first thing I've got to do is take this argument here, which if you recall is the Knockout model, this class here, and I have to send that over to a Web API call, but I can't do that as easily as just sending this, because this is actually a pretty complex structure because it's a Knockout observable model. So it's made up of a bunch of observable objects and I really need to flatten this out, I need to un-map it, if you will, from a Knockout observable state to a regular JSON state and I can do that with the Knockout mapping plugin, like this. (Typing) Now I have a variable called unmappedModel, which is just a JSON representation of what was inside this model argument, but without all of the Knockout stuff, with all that observable stuff, and there's also a mapping from JS, which allows me to take an incoming model that is a result of an actual REST query, a GET if you will, and turn it into a bunch of Knockout observables, something that I can use for Knockout binding, and we'll probably do that in a future view, but now I need to go the opposite direction so I'm mapping out of an observable state into a regular JSON state. And now I'm going to make an API POST. Now normally I can do this with just a $. post in jQuery, but I'm going to call upon my friend, ViewModelHelper, and I'm going to make a call to a helper function that I have and I'll show you this function now. So here's the ViewModelHelper and here is the modelValid, Errors, isLoading, all that stuff. Here are these helper functions I'm talking about - apiGet and apiPost. And what they do is that they wrap the $. get and the $. post and do all the common stuff that I find myself doing during every jQuery REST operation or REST call. So, for example, the first thing that I do is that I always have to figure out what the actual path to that REST URL is by prefacing it with the actual server name, but remember, we figured out how to do that earlier in the Layout page by setting this rootPath variable. So I'm incorporating the rootPath variable here with the incoming URL so I don't have to repeat this all the time. So, the API, well this is the GET, but here's the post, but it's almost the same thing. So in the case of the post, I'm sending the URL and the data over and here I'm calling a regular jQuery post to that URL with the data as that second argument. Now a jQuery post has a done, a fail, and an always callback and I give you the opportunity to send those callbacks here. Now you don't have to send all of them because as you can see, if you didn't send failure, which means a failure is null, then I'm going to do some common failure stuff here, but if you do send a failure callback in, then I'll run that one instead. Same thing for the always one. And then the success one is called on the done callback. So this gives me the ability to do my validation reporting, for example, in a way that I don't have to repeat it in every call, as you can see. I can now make a call to apiPost like this, watch. Now here I can just send the route to get to that

API controller action without worrying about the server name or anything like that. Now I can send the unmapped model in and now here, I'm going to send a callback. And what's this callback going to do? Well, let's go over to the App. js and, as you can see, the third argument is a success. So it's just going to execute that callback when I'm done. If I failed, it's not going to execute that callback and since I'm not sending a failure callback, as you can see, well I haven't finished it up, but I'm not going to send a failure callback, it's going to execute this here, which means depending on the result of that API call, it's going to look at that HttpResponseMessage and if it's something other than a 200, which means it got to the failure state because it was something other than a 200, it's going to report on it using my modelErrors. Remember modelErrors was turning on that little panel that displayed the list of errors, so I'm leveraging all that stuff right here. So now let's send something in for the success callback. Now, what is it that I want to do here? Well, if I came back from here successfully, that means that I logged in successfully so now I want to do one of two things. I want to either go to the Homepage or if we landed at the Login page as a result of an attempt to navigate to another unauthorized page, that means that the Login viewAction received the returnUrl, which I eventually took out in the View and brought into the ViewModel here. So I'm going to go to one of two places - either the Homepage or the returnUrl. (Typing) See how handy it is to have that CarRental. rootPath set up at the beginning of the site and not have to worry about it anymore? So now as you can see, if a returnUrl came in, I'm going there, and if not, I'm just going to the rootPath. And whatever is set up as the default, usually its home/index, that's where I'm going to end up going. So now I need to come back into the AccountController here and actually take the information that came in and use it to log the user in, to actually try an authentication, to attempt an authentication.

Completing The Login REST API Call

The actual core code in attempting to log the user in is very easy, it's going to look like this. Remember, I'm leveraging the SecurityAdapter and its built-in functionality. Now as you can see, I'm returning an HttpResponseMessage or at least expecting to return that. So I'm not just going to return to success, I need to actually return a response message with a success code, which would be a 200, but there's an enum to help us not worry about the response numbers, the code numbers, or something other than a 200. So here's what I'm going to do. I'm going to first create an HttpResponseMessage variable, set it to null, and now I'm going to set it equal to one of two things based on the success of this variable. So, if I successfully logged in, I'm going to set it to a code 200, the HTTP code 200, which is an enum of status code OK, like this. (Typing) And this enum is great because, as you can see, the IntelliSense will tell me what the status codes are, watch. See 200, NotModified is 304, NotFound is 404, and so on. So this is going to return a successful response message and when this gets back to the ViewModel here, and this call of course comes from the ViewModelHelper here, we're going to wind up in this callback, but anything other than a 200 will automatically wind us up here, where we can then report on the error. And those errors are coming out of that HTTP response because jQuery knows how read that from that result argument here. As you can see, result. status, statusText, responseText. So now I need an else. And here I'm going to create an unauthorized status code response, which I believe is 401, I'm

not really sure - I'll tell you in a second. That's the beauty of having the enums, you don't have to worry about remembering this. Yes I was right, it's 401. So there is Unauthorized and now I can give it a help text message to help me along and now return that response. Now I can consider my job done here and this will actually work just fine, but I put a pattern in place here that I want to follow even though I don't need it in this specific example because I'm not expecting to throw any other exceptions by this Login. There's going to come a time where I'm making API calls that are going to call WCF services and, if you remember, those WCF services are returning potential fault exceptions of a bunch of different types of exceptions. And I need to account for these and in the Web API I don't want to raise those exceptions up the chain, I want to actually create a response message that contains a status code of a particular type based on what exception came back from the service call. Now I have to make that determination myself, I have to map that. So for example, remember that NotFound exception that I was throwing from the service? That's going to come back as a Fault exception of NotFound exception and my job now is to turn it into an HttpStatusCode of 404. So, in order to do this in a pattern where I don't have to have a bunch of tricatches in here, I'm going to do something very similar to what I did in the Manager base class. I'm going to call upon a lambda expression based helper function that's in the base class of the controller. Now let me go to the controller here, the controller base class And show you this GetHttpResponse. As you can see, it returns an HttpResponseMessage and in its arguments, it receives the request and then it receives the code that it's going to execute, in a func of T just like you saw in the past, and then I'm executing or I'm invoking that code, but I'm invoking it inside a tricatch. This pattern should look familiar to you because we did this in the Manager Base Class, but here I'm testing for whatever I think the common exceptions are that I'd like every action, every API action to be tested for and I'm turning them into responses, HTTP responses. So as you can see, a SecurityException gets turned into a response of HttpStatusCode. Unauthorized. A regular FaultException gets turned into a 500, which is an InternalServerError, as does the catch-all of the Exception class. Any other specific things I'm going to take care of in the actual controller itself, because they're going to be specific to that operation or to that controller action. So it's the same philosophy that I use when I was writing the Manager base class. So, now all I've got to do is wrap this code up. (Typing) And this is the pattern that I'm going to use for all of my controller actions, they're all going to look the same and there may be other patterns that work similarly, for example, I know some people like to actually throw HttpResponseException instead of creating an Error response as an HttpResponseMessage. I prefer to create the HttpResponseMessage myself as opposed to throwing an HttpResponseException, which will then create this for me; I just like to be really explicit in what's going on. Once again, I'm going under the philosophy of the embracement of HTTP as opposed to the abstraction of it, which is my understanding is the whole purpose of Web API, to fully embrace HTTP. So that's it for this controller action. We should be okay, so we should be able to run this and see if it works.

Testing The Login Section of The Site

So before I run this, I want to show you one more thing. Back in the ViewModel, when the ViewModelHelper calls the apiPost, I want to show you one more thing that's happening here. As part of the general procedure for calling in a POST, which is typically done by pressing a button in one of my screens, notice the first thing that I'm doing is that I'm setting the self. IsLoading(true). This is the property that the View is depending on right here - see viewModelHelper. isLoading, so by setting that isLoading to true, the loadingwhen custom binding handler is going to change the entire contents of this div tag, meaning all of this to a little loading indicator, a loading placeholder, one of those animation indicators telling you something's going on. And then after it's done, after I do everything, whether it's a success or failure, I'm going to set it back. As you can see, in the always, I'm setting it back to false. So, what you're going to see when I run this now is you should see me log in, and as soon as I press that Login button while the Login process is taking place, meaning while all this code here is running, you should not see the Login button anymore, but instead you'll see a wait indicator giving you a lot of extra safety so that you don't double-click on a button. So let's run this, see what happens. So I know for a fact that my email is in the system already so I'll do that. Now watch carefully what happens when I press the Login button. Here's the indicator. And now, we went back to the Homepage, but look what's up here. When it re-rendered that top bar, it saw that the request was now authenticated and it displayed something different. In this case, my Login with a different menu, so we know things worked. Now let's do one more thing; let's run this again and try to log in with a user that does not exist and see what happens. Okay, I'll try my fearless editor who I know is not on here because he's a lowly pee-on. Now, I could have chosen to give you a little prettier of an Error message, but I left it pretty plain. Remember I was telling the box here to just show whatever came back in the HTTP response. In this case, 401:Unauthorized. And how did we get there? I'm glad you asked. When this call here failed, we wound up here, and we set a code to Unauthorized. Now once this is not okay, anything other than okay, automatically we drop in to here and this is where I'm pulling that information out of the result and all I'm doing to show it on my standard error panel is cramming it into the modelErrors. Remember, its modelErrors and modelIsValid that that validation Errors partial control is using. So, all I'm doing is cramming the information into here, which is expecting an array. So that's way you see that I'm giving it a little array indicators at the beginning and in the end. If I wanted to adjust the way this looks, this would be the area that I would adjust, right here. Okay, perfect! So, we have a Login page, we're able to authenticate now, and we've got good validation. So we got a nice, clean, elegant solution, pretty simple. We're going to do the account registration next and this one's going to be a little different because this is going to be a multi-step process, meaning it's going to be a 2 or 3-step process of collecting information from the user, so it's going to be using SPA style or Single-Page-Application, meaning before we actually perform the registration, we're going to go through kind of a wizard-type feel, if you will.

Setting up The Registration Models and Initial View

So first I'm going to go back to the AccountController, the MVC controller where I had that Login action that we originally browsed to, and I need to add the Register one here. And you know what, before I add the Register one,

let's add one other one here that I know we're going to need - And that's the Logout one. There we go. So now, when we do account/logout, we're going to go to the SecurityAdapter and do a Logout and then just go back to the Homepage. So now I need yet another one that's going to be account/register and this one's simply going to initialize the SecurityAdapter or initialize the Web Security simple membership provider in the background and then just go to a View of the same name as the action, which is going to be Register. So now when we browse to account/register, we're going to go to a view called Register, so let's create that view. And, of course, this view is also going to have a ViewModel and it's going to have a JavaScript model behind it and all of the same kind of stuff and follow the same pattern that we saw in the Login view. So first, let me define those classes that I'm going to need, or should I day the code files, to store my account/register model in JavaScript and the account/register ViewModel in JavaScript. I'm also going to need a C# model class for the same reasons that I needed one in the Login, because that's going to eventually use as a point of deserialization from the JavaScript stuff and then push into the Web API to perform the actual account registration. So let's create that class first, it's going to be down here in Models. Now this is all the information that I'm going to be collecting from the user and I'm going to do it in more than one screen, or should I say, in one screen, but more than one sub-view within that screen, this is where SPA is going to come into play, but I want you to take note of all of these properties -- the Name, the Address, City, ZipCode, State, the CreditCard number and ExpDate, the LoginEmail, and the Password. This is all information that I'm going to collect and I'm going to need that information on the JavaScript side, but I'm going to break it down into different JavaScript models because each one is going to correspond to a section of the View. So first, let me create the code files that I'm going to need for that. (Typing) And now let me set up this model file here. Now, notice I said file and not class because it's going to contain more than one JavaScript function. Now here's the way that I do this and there's many ways of doing SPA. What I'm going to do is render an MVC view and then logically break that view up into different sub-views. And more than just logically, almost physically, and you'll see what I mean when we get to the View, but each one of the pieces of the View, think of it as a wizard, each one of the steps in the wizard is going to have its own model that is going to represent on the screen. So, I'm going to create a JavaScript model to represent each one of those and I'll call them AccountRegisterModel Step 1, Step 2, and Step 3. And each one is simply going to be a JavaScript function that contains a bunch of Knockout observable properties, just like the Login model that we created earlier here, and uses Knockout validation to validate them. So, I'm not going to explain to you in detail or write it line-by-line because you saw that already taking place in the AccountLoginModel. So here is the first step. As you can see, it's called AccountRegisterModelsStep1 and there's all the properties that are going to be in that step of the wizard with its validation. Now here's Step2. That one has the Login information with two Password fields - One for the Password and one for Password confirmation. And as you can see, look at the validator extension property here from ko or ko. validation. It's calling upon CarRental must equal, this is a custom rule that I wrote that's declared over in the app. js file. And now I need Step3. And that one's going to have the CreditCard and the ExpDate. So now that I have the models set up, let's go out to the View and set up the View the way I'm supposed to. Remember, the Register link is underneath the Login section in that top bar. So by setting ViewBag. Name to Login, it's going to highlight that

particular link. So here's the script section. There's the AccountRegisterModel. js file and AccountRegisterViewModel. js file that I just created here. Now I'm going to have a div tag that contains the meat of all the HTML here. That one's called registration, which means my ko_apply MVC section is going to bind my ViewModel after it instantiates it to that registration tag, and there's the instantiation of the ViewModel. So we have everything in place. Next, I'm going to write the HTML for that entire view and like I told you, it's going to be split into three different wizard steps. Now, once again I stress, there's many ways of doing SPA and one of the preferred ways of doing SPA when the sub-views that make up a large view are just that, are very large and heavy, you want to bring them on demand from the server and there's techniques for this and there's actually Frameworks that allow you to do this or give you a lot of help in doing this. I'm not going to use that technique. I think it adds complexity just for the purposes of this course. Now, I will tell you, it's not a technique that I advise against. In fact, I strongly recommend that you use some kind of go-to-the-server technique and perhaps in Part2 of this course, I'll change it to do just that because I want to show you some new stuff. Besides, notice I'm SPAing, for lack of a better term, parts of the site, not the entire site. There's different types of SPAs, like I said, and one like Gmail, for example, the entire application is literally a Single Page Application and when you have one actual rendered page that is that is that large, you do want to pull its sections from the server as you need them only, but when you have a situation like mine, which is the way I like to do SPA, kind of a Hybrid approach, where you have sections that are done in single-page fashion, and you notice that's exactly what we have. You have different areas of my top menu that gets you to a view and then everything within that view flips around an SPA style, just for the section of that site, but if you navigate away to a different section, you're in a totally different view and the process starts all over again. There, I find it more acceptable to just bring in all the sections at once because, once again, they are smaller.

Writing The Registration Ciew Using SPA-Style

Now before I add all this HTML, let me explain to you the general structure on how this will work. This is the primary section of the view right here, this is what's going to render and determine what's going to be in view at any point in time. The sub-parts, in other words, the sub-views or the wizard steps if you will, are not inside div; they're actually inside other tags that look like this. I'll do an example of one first, and I'm actually going to have three of these, like this. (Typing) And these three templates will include the HTML of each particular step or each particular view that I'm going to flip between in my SPA HTML View here. And then up here in the main div tag, I'm going to determine which one of these scripts is in view and I'm going to do that with simple div tags whose visibility is determined by a property in the ViewModel like this, which means I'm going to have a property in the ViewModel, in the AccountRegisterViewModel, called viewMode and viewMode will be set to either register1 or register2 or 3 or something else. And each div tag that may be visible or invisible, and remember only one is going to be visible at any point in time because it's visibility depends on the viewMode property of the ViewModel, each div tag is going to be responsible for pointing to one of these script template sections that you just saw me write here. (Typing) And now I

need two others. So depending on the value of this property, one of these three is going to show. Now, despite the fact that there are three wizard steps that allow the user to enter information, there's going to be two additional steps that look like this. One of them is a page letting them confirm all the information they entered and then, if they were successful in registration, there'll be a Welcome section, and each one of those will have its own template. So I'll need two more of these. (Typing) And now all I've got to do is fill in the remaining HTML in there. So here's the first step and, as you can see, it's just straight HTML and it looks very similar to that Login view. As you can see, I'm depending on a property that's going to be the context, so this is the model property, and when we get to the ViewModel you'll see that, and that model property is going to have properties called FirstName, LastName, Address, which you already saw me write, right there. And there is that ValidationErrors panel and there's that same kind of button technique. However, look at this button; this button is going to a function called nextStep and you'll see that, because I'll have to write it in the ViewModel, but I'm using the same isLoading technique that you saw earlier in the Login. So now I'll add the other steps. Here is Step2, here is Step3. As you can see, they're all just HTML forms. Now look at this one - Step2. It provides a button to go to nextStep or a function called previousStep that you'll soon see. Same thing for Step3 - nextStep or previousStep as we're doing this in a wizard form and there's a little typo. So now I'll add the Confirm template and the Welcome template and, as you can see, the Confirm template is showing a bunch of information based on some properties. Now notice Step2 is doing a 'with', a Knockout With on accountModelStep3, which is in the ViewModel. It's going to be a property and it's going be a property that probably is instantiated to this function right here and there'll be a property to represent this one and there'll be a property to represent this one and it's being used here as you can see, but then the confirm template doesn't have any 'with', because it's showing you data that's coming from all three of those steps. Then it's giving you a prevStep button and then a button that goes to a function called createAccount and then if it is successful, we should hopefully end up at this template where we just say Welcome to CarRental. So next I'll get into the ViewModel and I'll define all of this stuff that you just saw the view represent here.

Setting up Viewmodel and Providing Previous-Step Functionality

So here's the general structure or skeleton for the AccountRegisterViewModel function and if you recall back in the View, we were referring to properties called accountModelStep1, Step2, and Step3. We also had a property called viewMode that was determining which sub-view was showing up here. There is viewMode and there is the accountModelStep1 and, of course, there is Step2 and Step3. And then I know that my ViewModel is going to require a ViewModelHelper property as well, so I'm going to need one of those so let me add those three things first. I use this technique of viewMode quite often and I always like to show a comment after, where showing you the possible values of viewMode, because remember, they're not stored anywhere. Now if you recall, we had buttons in that View and they pointed to functions called nextStep, prevStep or previous, and createAccounts. So, I'm going to create those empty functions right now, just to get them out of the way. And of course, they're the result of a button click, which

means they're going to bring into the function the current context that they're in with that With statement, so I'm going to need a model argument here, and then I want to instantiate these three properties as Types of their corresponding JavaScript models here and I'm going to do those when the ViewModel is first loading up, but rather than put them right in the ViewModel code, I like to put them in an initialize function that then I can call from down here like this, just a practice that I've gotten into. And to initialize those three properties to three instances of those JavaScript functions, I used this syntax. And now the last thing that I want to do is set the viewMode to my initial state and that will determine which one of those templates will show first. Now I'm going to write the prevStep function first for reasons that I think will become obvious. It is a lot simpler than the nextStep because the job of this function will simply be to test what the current viewMode is and set it to something else. That's all it is. I'm also going to start by setting the modelIsValid property of my ViewModelHelper to true. In case there were any views or any errors in display at the time, I can just clear that panel. Now I'm simply going to check for the different viewModes like this and obviously I start with 2 because there's nothing before 1 and remember, when checking Knockout observables for a value, you have to use function notation because that's what Knockout observable properties are; they're actually JavaScript functions so you have to have that open/close parenthesis, you're looking at an object here. (Typing) And that's it. Now, depending on what modem and step of the wizard I'm in, when I press the prevStep button and I come to this function, I check to see what step I'm in and I set it to prevStep. That's all I've got to do; I could have probably done this in a switch statement. I can never remember if JavaScript has a switch statement or not. So now I have to write the nextStep property and this one's a little more complex and you'll see why next.

Client-Validating All Three Steps

The nextStep function is a tad more complicated. The reason for that is because you notice that I'm doing validation at the JavaScript level on all those objects, on the properties of those objects. And I'm testing them for things like the proper format and email format or requirement, things like that, but there are going to be some validation things that are going to need to be done that are going to be at the server level. For example, let's go out to the model so I can look at the properties and tell you what I mean. In the case of Step1, all of these can be tested using JavaScript; however, here I want to make sure that the state that they entered, because I happen to give this as a free form text field, I could have gotten it, I could have made it a little more complex and given them a drop-down with the states, but I presented it as a free form text field, which means now I want to validate to make sure they entered a valid state. And I'm going to do that on the server-side because that's where I'm storing that stuff. If I come here to Step2, the Login email, one thing that I need to do on the server-side is test to see whether this email has already been taken, which means I've got to make a database call to check if it's an email that's in use already. On Step3, I may want to run the CreditCard number through some kind of mod check. You know those checks that you can do on a credit card number to determine whether it's a valid number even without authorizing the card. I could never remember what those checks are, there's a bunch of formulas for them on the Internet. I'm just going to do a simple check to see if it's

16 numbers in length, but either way, it's going to be a check that I'm going to be doing on the server-side. So each one of these steps here - 1, 2, and 3, on each one of these three model classes is going to need to undergo additional validation using a REST call and that's why nextStep is going to be significantly more complex. And the very first thing that I've got to do is unmap this model that's coming in, which is going to be either modelStep1, Step2 or Step3, and I've got to un-map it into JSON so I can send it into a REST call just like you saw in the Login example. Now I know I'm going to be doing three different checks, three different calls, depending on the mode - register1, register2, registar3, so I'll put an if statement in place to accommodate that. (Typing) Now I need to validate the incoming model here and for some reason it wouldn't let me do a ko. validation group like I did here in the Login one, just on the incoming model. Something about the way this incoming model is structured as instances of either Step1, Step2 or Step3 and then everything inside, it just wouldn't let me do it, I had to play a little bit of a game. And after searching around the Internet, I encountered other people having a similar problem and I couldn't remember exactly what the pattern was, but this is what I ended up having to do; as ugly as it may look, it did work. I had to create a new object that includes the model in just the regular property, not a ko. observable, not a Knockout observable like this, call that property anything I want, and point straight to the AccountModelStep1 or Step2 or Step3. Now I can check for the errors and let me show you something else that you can add here. This ensures that you're checking validation throughout the entire model object as deep as it goes and now I'm going to set my ViewModelHelper modelIsValid to either true or false based on that valModel. So it can display the error panel or not display it, one or the other. (Typing) And now, if I have no errors, I can continue. And if I do have errors, I want set my ViewModelHelper's modelErrors array to the errors so it can display them properly. Now this is going to take care of the JavaScript errors only, but once I pass that test, which means I'm inside here, now I have to do the server-side validation, but before I do that, let me go ahead and duplicate what I just did on the other two steps real quick. And I'm sure with a little creativity I could probably come up with some kind of routine where this code can just be reused somehow, but I'll leave it like this for now. So now that I'm checking JavaScript validation on all three steps, I need to check the server-side stuff in case I did pass validation in Step1, right here; I pass validation in Step2, right here; and Step3, right here. So now I'm going to make server-side calls and they're going to be made to Web API functions and the Web API calls, the Web API URLs, are going to look like this and I'm going to have one for each. So next I'll have to go back out to my AccountApiController and add the actions that correspond to these URLs that receive the unmapped model, which is going to be an unmapped version of AccountModelStep1 or Step2 or Step3, and perform any server validation that I need to perform.

Servre-Validating All Three Steps

So here I am back out in my AccountApiController and the first thing that I've got to do is that I have to add three methods that correspond to those three API calls that you just saw me make or at least document in the ViewModel. So I'll need three actions that look like this and I'll just return a null for right now. As you can see,

ValidationRegistrationStep1 gets access by this URL and it receives the AccountRegisterModel. Now, if you remember, this is the server-side version of the model, looks like this that contains all the information. Now in some cases, if URL was called from the ViewModel in the context of Step1 using unmappedModel when Model is Step1, then the server-side model is only going to contain values inside the properties that were deserialized from the JavaScript model representing Step1. So it wouldn't contain CreditCard or Login or Email, but if it was sent by de-serializing the JavaScript model for Step2 into this, then it wouldn't contain these - it would contain this instead. And then in Step3, it would contain this. Now I could've made it one-to-one and written three different server-side model classes, but I didn't want to - one served its purpose just fine. So now I go back out to the controller and, as you can see, I'm receiving the HTTP request, the accountModel in the body. It's a POST, there's my URL, and I just need three now or two more that look exactly the same, but with a different name. Now the format of these is going to be the same as the Login one that you saw up here; it's going to use the GetHttpResponse helper method, it's going to perform some code, and it's going to build a response. What it does is going to be different depending on the step. It's simply going to take the data coming into here and do some kind of validation with it and depending on the results of that validation, however shape it takes, I'm going to return either an HttpStatusCode of OK or some kind of Error. So I'm just going to move the code for all three steps in one at a time and let you see it. So there's Step1 where I'm actually getting a list of states that I have from a UIHelper and checking to see if the state that came into accountModel is in that list and if it's not, I'm building a collection. Notice this collection, which is a list of string here, and I'm just adding an error to the collection, because I may have more than one problem because now I'm checking to see if the ZipCode is in an Invalid format using a regular expression or like my good friend Mark Dunn likes to say, cartoons characters cursing. And then if I have nothing in the Errors collection, I'm building a StatusCode of OK. If I do have something, then I'm returning that as an array of string, there it is right there, using a BadRequest StatusCode. That way, I can return the entire array and actually display the list of errors that was built here out on the JavaScript-side. So now let's validate Step2. And Step2 was the account login information so I wanted to use the SecurityAdapter to see if that user exists and if it does, then I want to make a BadRequest StatusCode and send an error message in there. And I'm still using the same format of returning an array of strings. So here, even though I'm concerned about only one error, I'm returning it as a new list of string with that one item in there. That way I have the consistency and I can check for just that one format of a return type back on the JavaScript-side. So now the last one is Step3. And this one is checking the CreditCard. Length and it's checking the ExpDate, which is also a free form text field, but I'm checking it again as a regular expression to make sure that you entered a date in a proper format and then just returning the Error list or StatusCode of OK. So all three validations are encapsulated in these three calls here. And even though these are open Web API REST calls, they don't need to be secured because they're not doing anything that is insecure at all. So now, back to here, we have to make those calls. So here I'm going to make a call, an API POST to the URL account/register/validate1, send to it the unmapped model, and then if the result is successful, I'm going to change the viewMode to register2, effectively doing what nextStep was supposed to do to begin with, going to Step2 or going to the nextStep, but if the result is not successful, I'm going to let the

ViewModelHelper take the data out of that response stream and display it in the error box the way it was designed to do, the way you saw it do in the Login View. So there's the call to register/validate1, and if it's successful, I'm just changing the Mode. There's the call to validate2, and if it's successful, I'm just changing the Mode. And there's the same thing for the call to validate3. Now if that is successful, I'm changing the Mode to confirm. And by changing the mode to confirm, I'm going to be displaying this view down here. Now this view lets you look at all the information, and if you like what you see, you can press a button that goes to createAccount, which means in the ViewModel, there should be a createAccount function and there it is and that's what I'll write next.

Completing The Account Creation Process

Now the createAccount receives in the model argument absolutely nothing because it doesn't have a context of anything except the entire ViewModel; however, we already have all three steps and we got this far and we're confirming it now. So what I need to do is that I need to merge all three JavaScript models into one JavaScript function, one JavaScript object, so then I can unmap that one and send it over to a REST call that will be received in the server-side version, my AccountRegisterModel object, this one right here. And remember this is the combination of all three JavaScript models, but right now I have the three JavaScript models as separate. So I need to merge them together so I can send them all as one object that can then be deserialized into that server-side object, like this. Now this is just using some jQuery magic here to create a variable called unmappedModel, unmapping using the ko. mapping. toJS, unmapping each separate step into a separate variable and then adding each one of those steps into the unmappedModel, effectively turning out one JavaScript object that contains a mash-up of all three of those other ones. And now I can make the call to a Web API REST call that actually performs the user creation, the accountCreation itself, and if successful, changes the View mode to Welcome, like this. So now, obviously I need a function or an action in my controller to represent the URL or API accountRegister. So I'll switch over to the AccountApiController and add that at the bottom here and I can add it in its entirety, I think you get the picture on how these actions are all set up kind of uniformly. Now look what I'm doing here, I'm calling upon the previous three. I know it's kind of redundant, but I'm doing this just as a security check against hacks because this is a REST URL, which means it's going to be exposed. So, it's not going to take a lot of effort for somebody that knows what they're doing to build a model that looks like this in JSON and use something like Fiddler to post it to this URL and create a user account. So I want to make sure that if that takes place, if a hack like that occurs, I'm going through the actual validation as if they use the website and step through the wizard step-by-step. And only if they pass those validation steps do I then want to call the SecurityAdapters. Register with all this information and then after I register successfully, if an exception was not thrown, I'm going to call Login. And as long as an exception was not thrown, I am going to send an HttpStatusCode of OK and everything should be fine. So next we'll test this.

Test Run and Recap

Okay, let's run this through a quick test, see if it works. By the way, I cleaned up the view a little bit there while I was off-screen. There was some excess BRs that were not necessary there, I'm sure some of you in the audience were probably screaming bloody murder when they saw that earlier. Anyway, let's go ahead and run this. (Typing) Now my services are running, but they're not used in this part of the site. I chose not to go the service-oriented route for authentication simply because it was not necessary. I didn't feel that everything had to be put on the server-side of the wire. I'm still accessing things securely so all this is happening from my IIS, but these are parts of the application that only the website's going to need, it's not something that's exposed to the outside world or exposed to the desktop app or any future mobile app or anything like that. This is specific form-based authentication for the purpose of this website. So using the simple membership provider was easy and it worked and I just found no reason to put this behind a service wall and bring WCF into the picture. So now, we know that Login works, I'm going to go to Register. Okay, first of all, let's see if we can go to the nextStep and test client-side validation. There we go, it reports all our errors. So now I'm going to quickly fill this in. Yes, that is where I live. Now let's try something here. Let's enter an invalid State which we know is validated on the server-side and now try to go to the next step. As you can see, it made a REST call, came back with the fact that it's an invalid State. So now, I'll go to the next step and there's our next step. First of all, let's see if the client-side validation is working, and it is. So now, I'm going to try to enter my Email address, which I know is already in the system. Now incidentally, if I make a difference here on the Confirm Password like this, it should remain in client-side validation and tell me the Passwords do not match, perfect. So now they do match and now let's see what happens. The account is already registered. So I did a server trip just as an API call, checked this Email, found out it's already there, and came back with the problem. So now, let's add something different, let's add my friend Jason again. He's my editor at Pluralsight and I like to have a little fun at his expense here, and let's add him. And we didn't add him yet, but it confirmed that he's not in the system so it let me move onto the next step. Now, watch this, I can go Previous here and come back out. Have you noticed that we're not doing a server trip between Previous and Next? I can navigate back and forth. This is what an SPA is - a Single Page Application, watch. And all of those variables are still left in their JavaScript models so when they're rebound to the View, the View is showing their values as it's supposed to, but there's no server trip. The whole screen is changing, this section of the screen is changing just by flipping those templates around. This is how an SPA works at its essence. The technique used to retrieve each part of the view can differ from SPA style to style. Like I said, I chose to just bring it all in at the beginning, but in a production app you may choose to bring it in on demand from the server and there's various techniques for this and like I said, there's frameworks that help you with this stuff, but the concept is the same. You're just rendering one initial view and then flipping the screen around, flipping parts of the screen around. This is essentially how Gmail works. If you look at the Gmail screen, you'll notice it's one view, but it never goes back out to the server and re-renders itself in whole. It just changes parts of the view based on actions by you, the user. So now let's enter 16 numbers here and let's enter an Expiration date and hit Next. There's my conformation, as ugly as it looks, and I'm going to say Create Account. And it says welcome to carrental! If I click on Home, as you can see I'm logged in, jason@gmail. com, and everything is fine. And remember we put that Logout in place, we should be able to

say log out and there we go. Now we're not logged in again and now I should be able to log in as Jason. Alright, so we got security built into the site now. The only thing that we would need to add and I'm not going to do this in front of you because the concepts are still the same, the patterns are still the same, it's just more work for you to sit back and watch, and that is add the Change Password part, add the Account Profile maintenance, and of course, in the case of not logged in, add the part for forgot password. So you've seen quite a bit in this module so far, as far as building the website is concerned, doing it in SPA style, using the simple membership provider. You've seen some pretty cool patterns on the Knockout-side with the ViewModelHelper; you've seen some patterns on the server-side with the helper class, the helper methods in the controller base classes; you saw that we're using dependency injection. Consider that a recap, of course. And now in the next module, we're going to finish up this website by adding a couple of more screens to it and I'm going to start out by enhancing the registration screen that we built in this module so that is accounts for one thing. You see, if in the middle of that wizard, let's say Step2 or Step3, the user would hit the Back button, guess, where you would go? You'd go all the way back to the point that you were in before you went to Step1, probably the Home screen. So we need to incorporate what's called the HTML5 History API so that the Back button can actually Back navigate us among the steps. And it's very slick the way this works. I'll show you that in the next module.

Car Rental Website - Part 3

Intro & Highlights

Hi, this is Miguel Castro, and welcome back to Building End-to-End Multi-Client Service Oriented Applications. This module is going to continue where the last one left off, we're going to finish The Car Rental Web Site. So let's go over some of the highlights. The first thing I'm going to talk about is HTML 5 Push-State, this is otherwise known as back-button history. And I'm going to teach you how to track it and how to push stuff into it to give your application a really final polish. Then I'm going to write some of the other controllers and views, not all of them, but a couple of more to show you some of the other patterns that I've put in place and to also show you how to access services from the site. Then I'm going to add some user-data authorization; this is very similar to what you saw me do in WCF services and it's certainly done for the same reason. We don't want a user to try to access data that belongs to another user. So I'm going to show you how to secure against that in a Web API fashion now. Then I'll get into Unit Testing the controllers, both the MVC and the Web API controllers.

Intro to Push-State

So let's start by talking about what's known as HTML5 Push-State. Push-State is about pushing history onto the browser. The browser tries to track history on its own, but it tracks it based on server trips, server calls, which means every time you do a request to the server to bring something back, automatically that request, the history of that request gets pushed onto the browser. The reason for this is so that when you press the back-button, it knows what the previous thing was. Well push-state is about making the back-button think it has a place to go, when in reality it may not. And this is crucial in the case of single page applications. So without it, pressing a back-button will navigate to the point before the server request took place. So take this for example - we're currently sitting on the Home Page, so we click on the Register button, and of course Step 1 of that three step wizard comes up, and we fill out the form, then we press Next. Now when we press Next, what you saw was Step 2 showing, now Step 2 was shown, if you recall, simply by changing the View mode property on the ViewModel, and then the view itself knows how to react to that through its bindings and turn off one div section and turn on another div section. But in reality, we have not gone to the server so no history has been pushed onto the browser. So at this point in Step 2, we press the back-button, and guess where we go - right back to the Home Page. What we would like in an ideal situation is to press the back-button and react the same way as if the user would have pressed the previous-button, meaning go back to Step 1, the previous step. So in order to do this, we need to manually push the state of the view onto the browser history every time we switch steps; that way when we press the back-button, we can tell it where do we want to go in more of a manual way. Now this is not 100% necessary for an SPA; in fact, you don't need it for an SPA to work properly, but it does act as the final polishing touch in a single page application, because you're going to have users that just don't understand that when you're in a situation where the page is changing, but not going to the server, that hitting the back-button may not be the wisest thing to do. You have to account for the fact that they're going to hit the back-button and you have to account for it in a nice and elegant way so that the reaction that the website has is as if they would have navigated to the previous point manually. Now this is really a big topic and to explain it thoroughly in a lot of detail would take a couple of hours at minimum; in fact, I can do a whole course on this, perhaps one day I will. So the solution and pattern that I put in place is designed to work with the way I do SPAs, using the techniques and the patterns that I showed you in the last module, where we changed this one property that the view reacts to and then flips the different script templates around and the pattern that I put in place using the ViewModelHelper is there to assist me in pushing HTML5 push-state and working with the way I do SPA. There are frameworks out there that allow you to do this in an easy fashion, they work with any pattern you may put in place. I didn't want to introduce yet another library into this course and take even more time, so I did it my way, I think it's a simple way, and I'm going to show you how I do it in my view models and how you actually implement the pattern, how you use it, because I'm going to use it in any ViewModel that uses SPA, that uses a Single Page Application Style. So once I have this reusable pattern in place, my goal is to just make it easy enough to code it and implement it so you can move on to the next task. And that's exactly the way I'm going to teach it to you.

Initial Setup for Push-State

So here we are back in the AccountRegisterViewModel and this is the ViewModel we're going to spiff-up with some Push-state stuff. That way, we can traverse the Wizard Steps, but then rely on pressing the back-button to get us to the previous step that we were on. Now the pattern that I've put in place starts by requiring the ViewModel to have an initial state. This is all based on the fact that I'm dealing here with a ViewModel that's going to have a viewMode property, which can be set to one of different settings that back in the view determine what divs are going to be turned off and on, very simple SPA style. Now, the initial view that you see here in my initialize is register1, this is the view that I am setting it when the ViewModel first initializes, but there can be other complex situations where depending on the URL, the starting viewMode can be different. Now I didn't design this wizard to be that way because it didn't make any sense. Think about it, you shouldn't directly navigate to any step in a registration wizard. Your navigation should say, "account/register" and then go directly to Step 1, but there may be other single page scenarios where a different URL takes you to a different mode of that single view. Now I'm not getting that fancy here because, like I said, I'm relying on only one URL, the account/register, but I do need an initial state. Now my initial state is register1, but due to the pattern that I put in place to help me with push-state, I need to store that in a variable called initialState, so I'm going to set that up here. (Typing) And in fact, now that I have it in a variable, I can change this down here. (Typing) Now I'm going to need the initialState in a variable for reasons that'll become apparent in a couple of minutes. Now I need to determine, when do I need to push-state? And remember, I'm relying on a property here called viewMode. Any time that this property changes, the view's mode is going to change; so even though I'm still in the same CSHTML view, I'm going to change the internals of that view and that's what gives the appearance of a Single Page Application. So, if I want to rely on doing some code whenever the viewMode changes because I'm working with Knockout here, I can go to the bottom of my ViewModel here and I can actually subscribe to whenever the viewMode object changes. So anything up here, like for example this code right here that changes the value of viewMode will call this function that I'm about to write. (Typing) Now I can write code in here that will get executed anytime anything changes viewMode and the code that I want to put in here is actually quite simple. Like I said, I have a pattern in place already, I'm just going to use my pattern. I'm going to make a call out to my viewModelHelper and I have a function there called pushUrlState and I need to send it some information. The first piece of information that I need to send it is a code. Now this is important, it's actually the most important piece of information, because this is what's going to get sent back at an event that I'm going to listen to later on, that I haven't written yet, and based on that code, I need to put my view back into a specific mode. Remember, the system doesn't know how to put the view back in the mode that you want it; all it knows is that you pushed a specific code and this code can be something meaningful or just a letter or whatever - the system knows about the code that you pushed onto it and it must be unique per mode of that view, and later, it's going to be able to send that code back to you saying, "Guess what, the back-button was just pressed, here's the last code that you gave me, do something with it. " Now, it almost makes sense that I used the viewMode for this, because that's my unique code that identifies every sub-mode of this

particular view. So I'm going to send into this first argument the viewMode. (Typing) Now the next two are not as important, in fact I'm only going to send a null into there. The first one is a title for the Browser Bar in case you need for it to change and the next one's an ID and this was just because of the pattern I put in place, I designed this so that not only can you enter a ViewModel in a specific state, meaning in this case, I set the initialState right there. However, in other view models, the initialState might of entered from the outside, meaning you browsed to a specific URL and then I parsed that URL and determined what the initial state of the accompanying ViewModel should be, but in this case it's hard-coded into here because we only have one initialState and we're only going to have one URL. Now with that explanation in mind, the idea I was just referring to follows the same pattern as the initialState. I want to be able to in the future to write views where I can design a URL to access a view and put it, not only in an initialState, but perhaps that state has to do with displaying some kind of data record and perhaps that data record has to be identified using an ID of some kind. So that's what the ID argument here is for, but we're not going to use it in this course. The next one we are going to use, and this is a URL. Now in this particular example, I'm always pushing to the browser history the URL account/register. What's important is this code here, this is what has to be unique, because later when it pops back out it's going to tell me what the codes are so I can put it back into a certain view, but in some situations, depending on the view that I'm in, I may want to display a different URL on the Browser Bar. In this case I don't, but that's what this argument is for. So in the case where I would have a three-step wizard and each step would have its own URL that can directly address it from any other part of the site, instead of having this all in one line, I would probably have a switch statement here, switching on viewMode and each line would be exactly the same except the URL, which would be slightly different, but in this case I've left it simple and it's a single URL. And the next line I've got to put here looks like this. (Typing) In the next clip, I'll show you the code for pushUrlState and handleUrlState and you'll see what it does. But this is where I'm sending that initialState in and then coming out with either an initialState or an empty string. And this is used kind of as a flag, so things in the background do not happen more than they need to.

Finishing The Pattern Implementation

So here we are back in the App. js file and there's my pushUrl and handleUrlState, now notice pushUrlState is just a simple helper function, so I don't have to repeat this code in the ViewModel; it's simply receiving those four pieces of information and building an object with them. Now, the important part in this object is this State property here, which includes the code and the ID. Now the handleUrlState is doing either a replace or a push based on that initialState. The reason is when we first enter the ViewModel from the outside and view renders for the first time, I want to override the browser history item that was automatically put in place with my own, with my own definition that contains my viewMode and my URL information and any custom information, yet going forward every time the viewMode changes, I want to do a pushState. So I'm basing which call I'm doing, replace or pushState, and by the way the history here was given to me, this is part of HTML5 history API, I'm basing which one of these two methods I'm

calling on this initialState, but once I'm past the initialState, notice I'm setting it to blank, and then I'm returning it, so the next time it comes in here I know what it is. That's why you saw that this line of code here looked a little weird, where it said initialState equal a call to this method with the initialState, but that's so I'll only do this one time that first time and then going forward I always do pushState. Now statePopped will make sense in just a couple of minutes. I'm going to come back out here to the RegisterViewModel because I've taken care of pushing onto the browser history, but now I need to wire-in the event that does the popping, meaning the event that would automatically fire whenever the user presses the back-button. So first I need to make sure we have this capability and I incorporate the help of Modernizr by checking for the history object in Modernizr, and if it's found, then I have this capability; if not, I'm not going to have this capability, I'm not going to worry over it. Now the event that I want to wire in to is called "onpopstate" and it's part of the Window object. (Typing) Now the arg argument that I receive here is actually based on the information that's here, that I pushed or replaced the browser state with. Now this arg argument has a property called "state, " which I'm going to check for null, now this is state in the lowercase form like this (Typing). Now this value right here corresponds to the first argument here, so whatever I push into this argument, replaceState or pushState is what that arg dot lowercase state is, as opposed to my uppercase State. So that arg dot state in the onpopstate event is going to contain a code or ID, because that's what I'm pushing into it here. So we'll come back out here and now that we confirm that it's not null, I've got to do something with it. Now the first thing I'm going to do is set the statePopped flag to true and then I'll explain why. (Typing) So if I come back out here to the App. js, you'll notice that when handleUrl is called, I'm checking to make sure self. statePopped is false, and only if it's false do I want to replace or push the state. Remember I'm only replacing once initially and then every other time after that I'm pushing, but I only want to do it if statePopped is set to false; if it's set to true, I want to bypass all this and come here and then set it to false. Now why would I land in the handleUrlState function when all you saw is me calling it here manually? That's because of the next thing that I've got to do. Now remember, we're getting arg. state comes through here and the arg. state. code, as defined right here, was my viewMode. So now I'm given information about the last viewMode because the user just hit the back-button. So now I'm going to set the viewMode to arg. state. code and the second I do this, what I'm trying to tell it to do is to revert back to the previous viewMode, which is what's coming into here, but the second I hit this line, I trigger this because I just changed viewMode and remember I'm subscribing to its changes. So that's why I changed statePopped to true, so when I hit this function again, I can tell if I'm here because of a manual call by me or as a result of the state being Popped. And that's it, that's the implementation for my pattern for pushState. Believe me, doing this without the helper is significantly more complex and I know this was complex enough. So next I'm going to go ahead and run this through a test and let's see what the results are.

Testing The Push-State

So here we are at the site and I'm going to go to the Register page, and I'm going to go ahead and fill out the form. (Typing) Now I'm going to hit Next and I'm going to enter Login and I've got to make sure that I enter something that

hasn't been used, now I'm going to hit Next again. Now I realize I've made a mistake, so instead of hitting Previous, I'm going to hit the back-button - watch what happens. And I'm back to here and notice it wasn't a server trip. Now the URL is still the same and that's what that last argument was in my ViewModel. If I would have put a switch statement there and switching on the different view modes and then each time the viewMode changed I pushed a different URL, the browser history will remember that and every time that I changed the code or it pops back to the previous code, it'll pop back the previous URL as well, but I wasn't interested in that for this example. So I'm going to hit the Back again and now I go to the first step of the wizard and now if I hit the back-button, do I go back to the Home Page. Without these modifications, I could have gone through the third step of the wizard, hit the back-button, and I would have been right back here. So now we know it works. So now let's get started writing some more stuff for the site.

Intro to The Rest of The Site

There are three other sections to this site, one of which is the one that I'm going to write for you. The first one is the My Account - this is where you could manage your account profile, which is basically all the information you entered at registration time. The next one is the Reserve a Car section and this is the one that I'm going to write. And then there's the Current Reservations section, where a user will be able to see a list of their current reservations. Now all three of these, which constitutes the remainder of the site, will be accessing the WCF backend that we spend so much time writing. And of course it's going to use the proxy classes to do those. Now the proxy classes, like I said a couple of times, because of the fact that they're already abstracted using the service contract, they can be injected into the controllers. Now because they're going to be injected, we're not going to create them manually, meaning we're not going to put them around a using statement, we need to remember to dispose them properly or close them properly. And there's a good reason for this and I'll describe it in detail when I get there and I'll also describe the pattern that I've put in place to allow me to remember to do this properly, cleanly, and elegantly. Now each section of the remaining three would be an exposed SPA-style section; like I said, not the entire site is done in an SPA fashion, just its separate sections are, which once again is my preference for sites. To me, a hybrid SPA/non-SPA type of site like this one is my kind of compromise for having a good performing site, something that performs well to the user and that offers a rich experience, but at the same time lets you perform well as a developer. If we were doing the entire site 100% SPA-style, such as Gmail for example, it would be very cool to use and look at, but it would also take a significant more amount of time and effort to develop. Now I'm not saying that you shouldn't do that, I'm just saying every site calls for different things and this was the decision I made when I wrote this one. I needed to write the site as quickly as possible, yet still do it where it delivers a nice, clean, and rich user experience. Now the Reserve a Car section, which is the one that I'm going to write, is going to have three sub-views in its main HTML-delivered view. These Views are going to be three steps. The first step is where you choose the reservation dates, the second step is where you choose an available car based on those reservation dates, and the next one is the confirmation of the

reservation. Now if you recall, we're already set up completely on the backend for this. We have service set up to accommodate this, we have operations in this service to get us all the information we want, so we're ready to go. All we've got to do is write the front-end for it. So the rest of this module is going to be spent in Visual Studio. You're not going to see any more slides, just a whole lot of code.

Setting up The Files for Car Reservations

I decided to put all of my actions that deal with customer interaction in a controller called "Customer. " And as you can see, here in my top bar cshtml file, I have three links set up - one to Reserve a Car, one to View Customer Reservations, and one to see the Rental History, and they all point to the CustomerController with different actions. So the first thing I need to do is set up a controller call customer and give it an action call reverse car and that's the view that I'm going to start with. (Typing) Now like before, first thing I'm going to do is change the base class to point to my common base class for all my MVC controllers. And now I'm going to export this controller so that MEF can discover it later and the dependency resolver can resolve it for us. (Typing) The other attribute I need to put on here is the Authorize attribute. This is because this controller and its associated actions are going to be subject to authentication, meaning if you are an unauthenicated or anonymous user, you haven't logged into the system yet and you try to get to any route that reaches this controller and its actions, you're going to be automatically routed to the Login Page and be forced to authenticate first. So by the time we get into this controller, we should already be authenticated because we're going to be forced to, which means we're going to have an actual user identity in play already. Now that's going to be important later on and you also need to know that this Authorize is also used in Web API Controller; this is how I'm going to be authorizing or authenticating or should I say forcing authentication onto usage of the Web API Controllers later. Now let me write an action here called ReserveCar (Typing) and this action is simply going to return a view of the same name and this is where the SPA-style will kick in, with the rendering of this one initial view. Now I just need to put the route in here. (Typing) That's it. We got the controller set up, let's go ahead and add the view here. And this view is going to use Knockout, of course, and it's going to use a Model class and a View Model class, which I'm going to put here in the Binding. So I'm going to add a Model class, which is a JavaScript file called ReserveCarModel (Typing) and I'm going to add a ViewModel class called ReserveCarViewModel. Now back in my view, let me follow the pattern we've been following in the last couple of views, this is where we're bringing in our JavaScript files, and now of course I'm going to have a div tag down here that contains the beginning of the meat of my HTML and I need my ko. apply section that actually binds my ViewModel or instantiates the ViewModel, then binds it to this div tag. And of course I haven't written any code in the JavaScript yet, I'll get to that in a few minutes, and that's it. Next, we'll start adding stuff to this view and of course adding stuff to the Model and the ViewModel and so forth.

Skeletal Code For JS Files Ad CSHTML View

Now I've gone ahead off screen and added this skeleton structure for my Model class and my ViewModel class, the JavaScript files ahead of time, simply because they follow the same pattern that I follow with any new view that I'm going to write. Now as you can see, I have a Model class here and I haven't written any properties in it, but I've set up the namespace and the var self, the class is ready to go and have properties added to it. And then the ViewModel class follows the same structure that I set up all my View Models with. Now you saw this before. I know that I'm going to have three different sections, I know this ahead of time. When I designed this view, I decided that this view is going to allow me three different steps and I mentioned these in the slide. The first one is choose the dates, the next one is going to be choose a car, and the next one is going to be the confirmation. So I have these three different steps, I'm calling them reserve, carlist, and success. So I'm going to have a viewMode and this is once again the pattern that I put in place that you saw me use in the Registration screen, I'm using the exact same pattern here. There's my ViewModelHelper. Remember, for HTML 5 pushState, the way I designed my pattern is that I'm going need an initialState. If the ViewModel does not allow an initialState to come in from the outside so the view can be placed in any state using a URL, I'm going to just hardcode the initialState here to the first view that I'm showing, which is the Reserve. So there's my initialState variable, I'm instantiating my reservationModel, which I'll fill in later on, there's my declaration, and I'm setting the viewMode to the initialState. And now to handle my pushState for the different wizard steps so I can take the back-button into account, this is the exact same pattern you saw at the beginning of this module when I added this to the Customer Registration. I'm basically doing a pushUrlState of that viewMode in the subscription of the viewMode, so this fires every time viewMode changes. I'm doing it in one line instead of three different one for the three different steps because I'm going to use the same URL. Once again, I'm not changing the URL in this case. There's my onpopstate event, which is going to fire every time the back-button is pressed. The rest is exactly the same as you saw at the beginning of this module when I did this in the Registration screen. Now all I've got to do is keep adding things to this ViewModel. I'm going to come back out here to the view and set up those three steps. The three steps are going to be: reserve, carlist, and success, and they're of course going to point to three different templates, just like you saw in the previous example. In fact, I just noticed that this says Reserve, and this says reservation. Now I just need to add the three different templates (Typing) and all my skeletal code is done and I'm ready to now design the three different parts of the screen here and, of course, keep building the Model and the ViewModel as I need to. And, of course like before, there's going to be calls made from this ViewModel to the Web API using REST URLs, so I'm going to need an API Controller as well, but I'll write that one when I get to it.

Writing The First View-Mode

I'm going to go to the ReserveCar view here and write each one of the sub-views one at a time and then address their necessities in the Model and ViewModel classes one at a time. As I already stated, this first mode in the view, the reserve mode, this is for selecting the dates, the pickup and the return date for the reservation. Now you already know about the validationErrors panel, this is just a partial control that I put up here that's designed to read upon the

properties of the ViewModelHelper whenever it needs to report on validationErrors. Well, I'm going to use that exact same technique here. Now, if you remember back in the ViewModel when I initialized everything, I set a property here called reservationModel and in my Initializer, I instantiated reservationModel as an instance of the ReserveCar model, which has no properties right now. Well, I'm going to be using the ReserveCar model for all of the fields in this entire set of View Modes, so this entire View as a whole. So as you can see, I'm changing the context here to a reservationModel so that I can add my first two textboxes. Now these textboxes are going to be for dates, but they're going to done a little slicker and I'm going to explain to you in detail in just a minute. The first thing I want to explain is that I am binding to two properties called PickupDate and ReturnDate of the reservationModel, so obviously I need to add those properties here. So there's the PickupDate and the ReturnDate, and as you can see, I've added the validation requirements, the validation rules to them, so the only validation rule that each requires is that it is a required field. Now there's no validation here that this is a valid date, that I'm going to handle here in the view. Now as you can see in my data-bind, I'm binding this textbox instead of with the text binding, I'm binding it with a special binding called datepicker. Now this is not a special built-in binding, this is a binding that I added and I can't take full credit for writing it, I took bits and pieces from a couple of sources that I found on the web. My goal was to create a custom binding that combined itself with the popping-up of a calendar using the jQuery UI Date Picker control and that's what I came up with and it's a binding called datepicker. And the way it's going to work visually is that when that control receives a focus, when that textbox receives focus from the user, it's going to pop up a little calendar and it's going to let the user enter the date that way. Now, this part here is just like you saw in the other two screens - the Login screen and the Registration screen. I'm simply showing a button and binding it to a function called availableCars. This is kind of like the Next button, but I'm specifically saying "select an available car" just to be more descriptive with what the next step is going to be. So now obviously I'm going to need a function in the ViewModel called availableCars. (Typing) Now remember, when that button is clicked and this function is executed, the current context that we're currently in, which as you can see is reservationModel, the contents of that object are sent over in the Model argument and we can do with it whatever we want. So, following the same pattern and standards that we put in place in our first couple of views, the first thing I'm going to do is that I'm going to validate the model, check it for its validation error using KO Validation, and then I'm going to set the modelIsValid and the modelErrors on my viewModelHelper to the appropriate responses so that they either not show or show any errors that need to be displayed to the user. (Typing) This should start to look very repetitive to you because this is the exact same pattern that I've been using this entire time with the other views as well. Now my next step here is to go and make a REST call to the Web API so I can obtain a list of cars and send to that API call, the dates that the user just chose. So I can obtain a list of available cars that are truly available in-between those dates, meaning they're not reserved for somebody else or they're not going to be currently rented at that moment in time. And then I'm going to return that list and show it in another viewMode, so that's what I'll hook up next.

Making A 'Get' Call to The API

So here's what I know I need to do - I need to make a call out to the Web API. So let me document that first. (Typing) So this is what the call is going to look like and I'm going to send to this Web API call, the ReturnDate and the PickupDate that I just entered, which as you know is stored in the reservationModel here, which came into here, and then I'm going to return a list of cars. So of course I'm going to make a call to the viewModelHelper, which has a helper method called API GET. Now you already saw an earlier version of a helper class called API POST, well this one works very similarly. (Typing) And remember, just like the API POST, the API GET will automatically insert the server name in front of my URL and it will take care of all error handling automatically for me, so it does quite a bit. Now the next argument that I've got to send into the API GET is the data that I'm sending to that REST call. And rather than unmap the model and convert it JSON, it's very easy for me to do something like this. (Typing) As you can see in JavaScript, due to the fact that it's a dynamic language, I'm able to build an object on the fly and that's exactly what I'm doing here, though you can do that with C# now as well, it's called Anonymous Objects. And don't forget the model that came in contains observable properties, so they have to be addressed in object format because that's what they are, they're actually objects. So now the next argument is the Callback function that is called if we have a successful result. (Typing) So now we make a call to this API, send this information through, and receive this back. So let's skip over to the API controller now and write the call behind the scenes and then we'll come back and finish this up.

Setting up The API Operation

I'm going to write a new ApiController called ReservationApiController and of course I'm going to put this in my API subfolder of the Controllers folder here. (Typing) And let's set up the standard pattern first, my base class will be ApiControllerBase and, of course, I'll be exporting this and this is going to be a secure Controller, so I need to authorize it. Now I'll write the method that's going to represent that REST call that you saw me make in the JavaScript ViewModel. Remember, the pattern calls for returning HttpResponseMessage on all API actions (Typing) and receiving a HttpRequestMessage as the first argument of every action. (Typing) And then what follows this are the other arguments that I sent over during the call. And back here in the ViewModel, I'm sending over PickupDate and ReturnDate. (Typing) And the Serializr will automatically take care of converting them to DateTime for me. (Typing) Now I'm going to add the attribute that tells it what HTTP verb it's going to use. Once again, I like to be explicit with this and not rely on any guesswork and I'm going to add the routing. (Typing) Now note that there's two. I showed you this before, but I'm going to repeat it because it is important and it's a mistake that is very likely that you're going to make, because I still make it myself. We're in a Web API Controller, not an MVC Controller, so we need to use the right namespace for the AttributeRouting, that's Web. Http. (Typing) Now here I'm faced with a choice that can really go six-in-one and half-a-dozen in the other. I have to determine whether this action here should truly be secure or not. And if it is, I can leave it alone. However, in the future I may want to expose a public API; in fact, that's one of the things that I have planned for Part 2 of this course, is to expose more Web API stuff to give it a public facing API so other

developers are encouraged to write applications against my API, perhaps mobile application. So, if this REST call here, api/reservation/availablecars, does not truly need to be secure, if there's no harm in turning it on for everybody, and there really is no harm because all I'm doing is receiving a couple of dates and sending you back a list of available cars, there's no updates to the database taking place. I can add this (Typing) and now making this API call is not going to route the user to a Login Page or require any other security credentials. So now let's quickly finish up the pattern before we write the meat of the code. (Typing) So now I need to make a call to a WCF service and if you remember, my Inventory Service has a function or an operation called GetAvailableCars, which when given a PickupDate and a ReturnDate, returns an array of the car client-side entity. And that's the call that I need to make, which means that I need to get the proxy for the inventory service into here and I'm going to depend on MEF to do it.

Injecting and Using A WCF Proxy from The Controller

Just like earlier when the AccountController, the AccountApiController, used constructor injection to receive that SecurityAdapter, I'm going to use the same technique to receive a WCF proxy. So I'm going to create a constructor first and that constructor is going to receive an argument of type of the service contract for the InventoryService (Typing) and there it is in CarRental. Client. Contracts and now I'm going widen the scope of this argument. (Typing) And now I can go ahead and use it down here. But before I do, let me remind you that we are a client to WCF right now, so of course we're going to need the Web. config, which is all set up already, just like I showed you back in the hosting module how to set up a Web. config or an App. config for a client in WCF. There's my system. serviceModel section with the client subsection and, of course, all the endpoints to the services that I want to access. So now I'll go back here and now I can finish writing this operation. (Typing) And provided no exceptions were thrown, I can go ahead and return an HttpStatusCode of OK with the return type. Now you haven't seen this yet. The last couple of API calls we made were POST calls, this is a GET call. Now watch what the difference is. (Typing) Notice I have the CreateResponse and I can give it a generic of what type I want to serialize that's going to become the content that gets returned back up to the caller of this API call. In this case, my content is going to be an array of Car. Now I can give it the StatusCode, like you already saw in the past, and the next argument, as you can see in the IntelliSense, is data that adheres to that generic type. And this will be the data that gets returned to the ViewModel right there in the result. Now I don't need to check this for any other exceptions, except the conventional SecurityException and Fault exception, and remember my GetHttpResponse is taking care of that for me by wrapping and invoking all of this code inside a tri-catch. And if that tri-catch does catch an exception, it will create the appropriate error HTTP response. Now though we can almost consider this complete and it will work, it's still not properly house-kept, and what I mean by that is that my InventoryService proxy is not properly closed. So I have to take care of that and there's a good reason why I have to do that and I'm going to explain that next.

Pattern for Closing Proxies After Use

So let me explain to you what's going on here. When this controller gets instantiated by MVC for the purpose of calling upon the GetAvailableCars action, MEF is going to instantiate the controller and it's going to resolve these dependencies. In fact, it's not going do that unless I do this first. Then after it calls upon the GetAvailableCars with now the instantiated proxy, it's going to release this controller after it's done. Remember, MVC or ASP. NET by its nature is stateless, which means for every API call you're making, you're instantiating the controller, calling the action, and then getting rid of the controller; it's not staying in memory, it's completely stateless. Now what's happening here is that the InventoryService is being used and it's not being closed or disposed properly. And until this service is properly closed, until this proxy is properly closed, what happens is that it's still subject to the WCF throttling limits and a throttling thresholds, and more often than not, it's not going to make a difference to you, you're not going to notice anything. However, if you were to ever hook up a load tester or any kind of load testing software and instantiate many, many instances of this ApiController and run many concurrent calls to get availableCars for the purpose of load testing, you're going to notice some degradation in performance and that's because the proxy is not being disposed properly and this actually bit me in the ass once and it happened to be at a client's site, which was a little embarrassing. And all I had to do to fix the problem was properly close the proxy after I used it. Now, it's very easy for me to just cast this to disposable and then call the Dispose, which will in turn, close the proxy; however, that's something that I have to remember to do on a per action basis and I'd rather come up with something a little slicker, something to standardize the pattern. So let me show you what I did and then I'll explain to you how it works. The ApiControllerBase class provides the virtual method called RegisterServices that looks like this. (Typing) Now here what I've got to do is just add any service proxies that come in that need to be disposed to this incoming list, disposableServices, like this (Typing) and that's it; I don't have to do anything else. After this action gets executed, the base class will take care of disposing this. Now the way it does this should be simpler than what it turned out to be. In an MVC Controller, you inherit from a class called Controller and that class provides two virtual methods for you called OnActionExecuting and OnActionExecuted, which give you before and after the execution of an action and they're great. Unfortunately, for the reasons that I have yet to understand, the developers in the API Team did not provide the same virtual methods to the ApiControllerBase class. So instead, I have to jump through a couple of hoops and write a custom filter, which looks like this (Typing) by way of an attribute, which I'll have to decorate my controller with in a second. And then in that custom filter, I can then override the OnActionExecuting and OnActionExecuted and go through the list of those services and dispose them properly. So all I have left to do is to decorate my controller like this. (Typing) If this were an MVC Controller, I could have done it a little simpler with just the override of OnActionExecuting and Executed in the base class, but on the ApiController I have to put it as a custom filter, but it's not that big of deal, but now I don't have any more work to do. All I have to do is remember to add my services here and every time that you're given a virtual method, it's almost like a fill-in-the-blanks situation - you automatically know what to do because of that extra help that you get, so I don't have to worry about doing anything down here in the action, it will get disposed of for me. So now I can come back to the ViewModel and finish up working with the results, which is that list of cars.

Returning The List of Cars and Proceeding with The View

Now that I have a list of cars that got returned here, my next step is actually quite simple - I just need to convert this list, which comes across as just regular JSON results, I need to turn it into a Knockout observable that is then going to be bound to the next mode in this View, which is the carlist Mode. So the first thing I'm going to do is setup a Knockout observable property to hold the list of cars like this. Now this is a Knockout observable array and this is the equivalent of a regular array, but it's observable in Knockout context, which means I can use it using Knockout syntax from a View and bind it to a list. And that's exactly what I'm going to do - I'm going to build sort of a grid in the next part of the View. Now I need to convert this result to that Car List and rather than scroll through or iterate through the results and do it one at a time, which I certainly can do, I'm going to incorporate the use of the Knockout mapping plugin, like this. (Typing) And now that I've converted the incoming JSON to a Knockout-absorbable array, I'm going to set the viewMode to carlist, like this. (Typing) Now back out here in the view, if the viewMode is carlist, then this is the template that I'm going to show, which means this one's going to be hidden or turned off, and this one is going to be turned on. So now I just need to build the HTML here to display the carlist.

Showing The List of Available Cars

So before I continue with this view, let's test what we have so far to make sure that everything's cool. So I'm already logged in, here's my Home Screen, I'm going to go to customer actions, reserve a car, there's the first step of the reservation process, now watch the datepicker I was talking about - so I'm going to pick a date, I'm going to pick another date here, I'm going to click on Select an available car, and I haven't written that part of the view yet. So what's probably going to happen is that we're just going to see this disappear and nothing show in its place, and there we go. So now we have to fill in this empty space with the list of cars. Now to display a list of cars, it's very easy. All I have to do is write a table, I can do it in a div, I can do it in a list, in any kind of iterative-type HTML, but I'm just going to use a good old-fashioned table because it's nice and simple and it lets me show you kind of a grid-type structure without a lot of extra CSS stuff. So I'm going to write a table and that table is going to be bound to that cars collection that you saw me declare in the ViewModel and it's going to be bound using the foreach binding like this. (Typing) Now here's the heart of what I was talking about - as you can see it's a table, this is just some Bootstrap styling to give it a little niceness to look at, and I'm declaring the heading section first. So I'm going to show you four columns - the Year, the Make / Model, the Color, and of course the daily rental price. And then in the body of the table is where I'm going to iterate through the cars collection that was declared in my ViewModel. Now inside here, I'm in the context of a single car. That's going to be important, because notice I'm addressing each property of the car. Now, remember my ApiController returned an array of the car entity right there and then all I did was take that array here and convert it to a Knockout Observable. So I didn't manually build a model in JavaScript, I used the mapping plugin to create one dynamically for me based on my server-side car entity and that's why I'm able to address the car entity properties here of the car's JavaScript collection, because they were created for me during that mapping process. So this is

going to build the actual table. Now, for each one of these rows, it's going to be a different car, and for each one, I'm in the context of that particular car, which is why you see that the first column has a link on there called select that is bound to a function in my ViewModel called selectCar. And when I write that function, which is what I'm going to do right now, the Model object that gets sent into that function is what my current context is, and in this case, it's a single car. So I'm going to come back out here to my ViewModel and I'm going to write a function called selectCar. (Typing) And the model that's coming in is that single car that I just selected. Now I'll come back in here in a second. Let's go back out to the HTML. The rest of this is just to polish it up a little bit. As you can see, I'm taking this entire table and I'm wrapping it in a div tag, whose visibility is based on there being something in that collection of cars, because if there is nothing in the collection of cars, then all of this is going to be rendered invisible and instead this is going to appear. And then this last part is the equivalent of a Previous button, which basically is going to set the viewMode back to the previous mode and that's going to be done in a function called reservationDates, so let me add that now. (Typing) And here I just need to set my viewMode back. (Typing) And I'm also going to clear the cars collection, (Typing), make it an empty array just so we can start the process over since we're going to ask for another set of dates. So now I just have to write this function, which is essentially going to take the PickupDate and the ReturnDate, which if you recall were bound to this reservationModel over here, see. So we had two places to grab the information from; when we pressed availableCars, that reservationModel came into here, but we could have also obtained it just by grabbing this property, either one, but it's good that we have it globalized at the ViewModel level because I'm going to need it down here. And I'm going to take that PickupDate and the ReturnDate and I'm going to take the ID of the car that was selected and I'm going to make another Web API call to actually perform the reservation process. And then I'll finish by setting the viewMode to success, which is going to show the last step in the View.

Making The Actual Reservation

So of course I'm going to be making an API call from this function here and I need to send to the API call a model, some kind of model that is going to contain all the information regarding my reservation request, including the two dates and, of course, the ID of the car. So I'm going to create that Model first. (Typing) And I can create this just as a Dynamic object, which means created on the fly without having to build an object ahead of time, once again the power of JavaScript. (Typing) There, now that I have a model built, I'm going to send this in to an API call. Now obviously when I get to writing that API action, I'm going to need to receive that in a C# version of this object here that I just created dynamically in JavaScript, which means I'm going to have to create a C# model object to receive that information. (Typing) And I'll fill in the success callback in just a few minutes. Let's go over and create a REST API action that can respond to this URL right here. Now first, let me create that C# model object I told you about and that's going to go down here in my Models folder. (Typing) And, of course, it needs to look exactly like that dynamically-built JavaScript object that you just saw in my ViewModel: PickupDate, ReturnDate and Car. So now that we got that set up, go to the ApiController, and I'm going to create my API action. Let's put the skeletal code in place

first. Now as you can see, it's called ReserveCar, but the name here is irrelevant of course; what's important is the routing and this route is what we called from the ViewModel and there's the reservationModel that I am receiving this information in. So now I need to execute the reservation. Now I know my service layer and you'd have to go back to the service layer and look to see what I'm talking about, but I know that I have a MakeReservation operation available in my service layer in order to execute the actual reserving of a car, but it belongs to the RentalService, not the InventoryService. And as you can see up here, we've brought in the InventoryService, so I need to add another service to this. (Typing) And, of course, now that I have this pattern in place, I can add it here and it will get disposed of properly when the time comes. Now here's where a decision had to be made and I just wanted to show you how both sides work, because here I'm actually doing something that I suggested against when I wrote the services earlier. If you recall, we put abstract factory patterns in place so that we wouldn't have something like this. Back in the services, I use a Data Repository Factory or a Business Engine Factory so I can just import one thing, the factory, and then grab anything that I need from it on demand in each operation. If I wanted to do this 100% efficiently, I would probably import my Service Factory here and then instantiate the service proxy that I want on the fly. I just chose not to do it that way to show you the variation in code. Now the reason it's a little inefficient is because this particular operation is using one service and this particular operation is going to use the RentalService, yet for both situations I'm instantiating both services, which is a little wasteful, but I wanted to show you how it can work both ways. So now in here, I need to make a call to the MakeReservation operation of the RentalService like this. (Typing) Now look at the IntelliSense, it's asking me for a loginEmail, this is the user to which this reservation is going to be applied, then it's asking me for the carId, the rentalDate, and the returnDate. Now let's talk about that user for a second before we fill this in. Rather than give the URL of the REST call the ability to receive a user, because if I do that I'll be opening it up so that anybody that's authenticated can then make a reservation for another user, because it now becomes part of the URL. I don't want to do that, I want to secure it a little better, and the easiest way to secure it is simply to grab the authenticated user, the user that is currently logged into the system, like this. (Typing) And this user property here is the same good old-fashioned user property that we use to have in ASP. NET web form code-behind classes, which is actually a short cut to Thread. CurrentPrincipal. So there's the Identity. Name and I'm making a call to the MakeReservation and finish this operation up. (Typing) And provided no error took place, I can now create a success response, and return the reservation. (Typing) Now, why am I returning the reservation? Because the return type of the MakeReservation operation is a reservation entity which contains a newly created reservation record, including an ID, that reservation ID, that's what I'm going to use for my confirmation number. So I'm going to return that reservation entity back up to the ViewModel here, it's going to be returned in the result, and now I just need to show it and maybe change the view to the Thank You and the Confirmation screen.

Displaying Reservation Results

So now that I have come here and I have my reservation result, it contains a ReservationId Property, I want to show that to the user. So in order to show that to the user, since I'm using MVVM here, all I need to do is expose it in the ViewModel somehow, so that the View can react to it. So I'm going to come up here and add another property to my ViewModel called reservationNumber. (Typing) Then I'm going to come down here and set reservationNumber to the ReservationId that came into the result. (Typing) And now I'm just going to change the View to its success mode. (Typing) Now that the viewMode is success, I of course have to account for that in my View here. (Typing) I'm going to come down here to the success template and add some code "Thank you for reserving your car" and then showing you what you're reservationNumber is and there's the binding. So now we should be able to run this. Now I'm going to magically log in here, watch - like that. Now let's go to reserve a car and let's choose August 18th to the 23rd, there's my list of cars. As you can see, a link right next to each one. So I'm going to reserve this Mustang here, "thank you for reserving your car! your reservation number is 14. " And we have a record in the database. So every other screen that I would write for this website would follow the same kind of pattern so I 'm not going to do it all step-by-step. I'm pretty much done here except for one more thing that I want to show you. You can see that the patterns are pretty standard. I follow the same thing over and over again, I'm very consistent with that, and it's simple, it's elegant, it works nice, and we're using all the latest and greatest technologies here. So I want to show you one more thing regarding user data authorization in another controller that I'm going to write. Now I don't have any views set up right now to address this controller. I do in the final version of the code, but not in the one that's onscreen now, but I do want to put this controller onscreen to show you a particular action and I'll explain what I mean in just a second.

User-Data Authorization

The last thing that I want to show you is something that's in the CustomerApiController. Now you haven't seen this controller yet and it addresses views that are not built yet in this version of the application, but I'm going to go ahead and create the controller, in its entirety as a matter of fact, to show you what it is that I want to explain. (Typing) Here's the CustomerApiController in its finished form. Now, this is the controller that's used for the MyAccount section of the site, when you go and maintain your user profile. As you can see, I'm following the exact same patterns that you've seen a couple of times already. I'm bringing in the AccountService in here, here's where I'm registering it, here's the API call to a obtain CustomerAccountInfo - it's secure because it's using the actual user that's logged in - you can't obtain CustomerAccountInfo by sending a query string. I could write the operation to work that way, but I didn't do that. I'm locking you into the currently authenticated user only; however, when we go to UpdateCustomerAccountInfo, things change a little bit, because here I'm doing a post and here's the payload that I'm posting and this payload is going to contain the login email, the user ID or the username should I say, of the user whose record I wanted to update. So if User A is already logged in and authenticated and User A is a developer who knows what he's doing regarding REST and using tools like Fiddler and stuff like that, it doesn't take a lot of effort for User A to actually write custom payload that includes information for User B, including User B's LoginEmail, and then

make a REST call to this REST URL. Now remember that User A is already authenticated so they're going to be able to get in here, but now the payload is going to try to update a user for somebody else. So you can see, what I do to secure against that is I take the LoginEmail that came in from that payload and I call a base operation called validateAuthorizedUser; this is a very similar process to what I was doing at the service layer earlier. So if I go to validateAuthorizedUser now, you'll see us doing something very simple. I'm taking that user and I'm comparing it to the user that's currently logged in to make sure they're the same person. And if they're not, I'm going to throw a SecurityException. That way, a Power user or a smart developer can't come into the CustomerApiController or can't get here through this REST call and send custom payload that pertains to another user that's not that person that's currently authenticated, and that's how I'm accomplishing user data authorization. So next, I'll write some unit test for the controllers and then we're pretty much done with the site.

Unit Testing MVC Controllers

Before I close the door on this website, I have to make sure that I have some unit tests here and I'm unit testing the controllers in this case, both the MVC Controllers and the API Controllers. So I'm going to write one test for each type of controller as an example and you'll get the idea. Now I'm going to test in the AccountController, the Login action here, and what I'm testing, remember this is a unit test, so what I'm testing is that I'm getting the proper response here. So in this case, I want to mock up the SecurityAdapter, so the initialize just doesn't do anything, and I'm going to return a result. And I want to test that result to make sure that it's a ViewResult, which is the derivative of ActionResult, and that it contains a model called AccountLoginModel and that a property in that AccountLoginModel called returnUrl is set to whatever I send in to the action. So let's come out here to the CarRental. Web. Tests and I am going to go to that AccountControllerTest. And now I have a method here called Login, because it's going to correspond to the Login action in the AccountController. So the first thing that I'm going to do is that I'm going to mock-up my SecurityAdapter, like this. (Typing). And the Initialize is not going to return anything, so I really don't need to do anything. If I wanted to make things explicit just so a user looking at this code knows what's going on, I can do something like this (Typing) and just not do anything with it. Now I'm going to set a return variable here. This can be anything I want (Typing) and now I'm going to instantiate the controller. (Typing) And now I'm going to call upon the Login and receive an action result as a response. (Typing) And I'm, of course, sending the returnUrl into the Login action. Now I'm going to assert a couple of things. The first thing that I'm going to assert is that the result is indeed a ViewResult. (Typing) Now I'm going to cast the result to a ViewResult because I need to grab the Model from it. (Typing) Now I'm going to assert that the Model is indeed an AccountLoginModel. (Typing) And now I'm going to cast the Model to an AccountLoginModel and test the returnUrl. (Typing) And that's it; let's see if this runs now, hopefully successfully. And we got a passed test. The other MVC Controller test would be very similar. So then the next test that I've got to write is going to be one for an ApiController.

Unit Testing API Controllers

Now this next test is going to test the GetAvailableCars action on my ReservationApiController. Now this one's going to be just a tad more complex because, as you can see, I'm expecting an HttpResponseMessage as a result and I am expecting a request message as the first argument. So I need to be able to reproduce those things and I've already taken care of that in my ReservationApiControllerTest. As you can see here, I have couple of helper methods called GetRequest and then GetResponseData from the response message and I can use these to test any ApiController action; in fact, it's probably better that I refactor these out to another class where they can be reused across a bunch of other test classes, but for now I'll just leave them in here. And then the Initializer and in the Initializer I'm obtaining a request, an HTTP request, so this is how I'm simulating all this. So now it's a matter of writing the actual test for GetAvailableCars. Now if you recall, I have two services that I have to mock up and send into the Controller class. (Typing) Now, here in my ReservationApiController, what I am expecting is when a call to GetAvailableCars is made, I'm going to return an array of cars. And I just want to make sure that that unit works, that that action works, not that the actual service behind the wire here works. So of course I'm mocking the InventoryService and that's the service that I have to set up. So in order to set up the InventoryService's GetAvailableCars to return a car array, I have to create a car array. (Typing) And now I can mock the Services method. (Typing) I really don't care what dates come in here. (Typing) Now I can instantiate my controller and send the two mock objects in. (Typing) And now I can make a call to GetAvailableCars. (Typing) Now I'm going to pull the actual data that came back from the response. (Typing) Now before I do that, I'm actually going to assert that the response was okay. (Typing) And now that I have the data, I can test it to see if it's the actual two cars that I mocked earlier. (Typing) And I should be done. Let's run this test now. We've got a passing test. The rest of the ApiControllerTests would follow a very, very similar pattern. So next I'll do a quick recap and then tell you what's in store.

Recap & Summary

So we've completed the first of the two user interfaces for this system and this is actually the more complex one. The next one's going to be the desktop app, which we're going to use WPF for. Now, on this one it was a website and we used a multitude of technologies. You saw MVC in combination with Knockout, in combination with the Web API, had some Bootstrap thrown in there. Of course, jQuery was used to access the REST calls from the Knockout View Models. It was a hybrid SPA/Non-SPA approach. Now I want to describe one more time exactly what that means, because you hear a lot about SPAs, SPAs, SPAs, everybody want to do things the SPA way, and in reality an SPA is fantastic and it makes for an awesome user experience, but it's not without its share of work. So a lot of times coming up with a hybrid compromise, like I did here, where you have sections of the site that are SPA-style, but not the entire site, is not a bad thing. It's a good compromise because you've got to take your own performance as a developer into consideration here. Now you also saw that we use a SimpleMembershipProvider for authentication, because it is just that - it is simple and it works and I had no other reason to overcomplicate things. In part 2 of this course, I'm going to

enhance this to use OpenID and OAuth. Now, we had base classes for both controller types and they provided some common functionality here. They also assisted me with a pattern for proxy disposability and they also assisted me in error catching on the API Controller side. And last but not least, for some nice polish, I gave some Push-State functionality to all the View Models using the HTML5 History API so we have some back-button support. And in any of the Views that are going to be multi-mode views, I'll be able to travel back and forth using the back-button and even the forward-button as well. So, I hope you enjoyed this module. Coming up next, we're going to write a desktop application using WPF and the MVVM pattern.

Car Rental Desktop - Part 1

Intro & Highlights

Hi, this is Miguel Castro and welcome back to Building End-to-End Multi-Client Service Oriented Applications. This is the first of the two modules where I'll be building the car rental desktop application using WPF. Let's go over some of the highlights. In this module, I'll be covering all of the application infrastructure, how the entire app is laid out, what all the code files are, what the relationship of the views to the viewmodels is, and the app. xaml file, what role it's going to play in the application. I'll also be quickly covering the configuration and some new base classes that I've introduced in the core framework. Then I'll be talking about the themes and controls that I'll be applying. Afterward, I'll be giving you an entire overview of all of the views in the application and in fact, we're going to setup the skeletal structure of all of the views and the view modules, and you'll be seeing the relationship from one view to the next. Then, I'm going to be describing some common problems that every WPF application has and most developers don't even know they have them. Then in the next module, where I'm going to build an entire view and a viewmodel for the application, I'm going to solve the problems I'm going to discuss here in detail. So let's get started.

Application Infrastructure Intro

This app will be built using the MVVM, or Model View ViewModel pattern. In fact, it's my belief that every WPF app, or any XAML app for that matter should always incorporate the MVVM pattern. Now I follow this pattern very strictly, probably stricter than a lot of other developers. I'm very strict about every single view having a corresponding viewmodel, no exception to that except one, and that's the main window view that I'll talk about in detail a little bit later. Now remember what a viewmodel is. A viewmodel is a logical representation of the view. By being a logical representation of the view, we're able to test it, unlike a standard view with its code behind. A viewmodel contains all of the state and behavior for its corresponding view. Now, while many developers believe that you should never, ever, ever have a code behind class in a WPF application once you start using viewmodels, I believe that there is a place for

the code behind for very, very specific reasons, and I'm going to show you what those reasons are because I do have code behind in some of my views. Now, there's no logic in the code behind regarding the state or the behavior for the view, that's where I am very strict that I put all that in the viewmodel. But, things like showing message boxes or navigating to other windows, these are things that have to be done from the view itself because the viewmodel does not have any knowledge of the views, it doesn't go in that direction, only the views have knowledge of the viewmodels, so there's still a place for code behind in very specific scenarios and not 100% of the time. Now, many WPF applications tend to use containment. This over inheritance, which WPF does not support at the visual level, unlike its cousin Windows Forms. So what I mean by containment is that it will have hierarchies of views. And when I say hierarchies of views, this also means hierarchies of viewmodels. So we'll have viewmodels contained within others and the physical representation of that will be views contained within others. And view switching will occur using a combination of a content control and some data templates, and I'm going to cover all of that in the code. I'm also going to be using MEF to serve up most of the viewmodels. In fact, all of them but one, and you'll see that when I get to it. Now because MEF is going to be serving up my viewmodels, I'm going to be able to benefit from injecting dependencies into my viewmodels very similarly to that way that I did in the controllers back in the MVC application. And as far as styling is concerned, I'm not going to spend a lot of time on it. I actually use the NuGet package out there that's called MahApps Metro to give this application a little bit of a Windows Modern App look and feel to it, and I'll go over the details of that as I approach it in the code without diving too deep into it. It's actually very, very easy to use and most of the work is done for you and with very little effort using this NuGet plugin, you could actually add Metro or Windows Modern Apps type look and feel to a WPF application. So next, let's switch over to Visual Studio and I'll go over all of the infrastructure and how all of this is setup in the actual code.

Basic Structure and Main View/App Config and MEF Bootstrapping

So here is the CarRental. Admin project, which is the WPF application that makes up the desktop app. Remember, this is the app that the car rental employee uses while sitting behind the rental desk. Now, like I said earlier, every view is going to have a viewmodel. In fact, I have ViewModel and View folder setup already here. Now as you can see I only have one of each and I'll describe this in a second, but every WPF app always starts of course with an App. xaml and a MainWindow, usually kicked off by something in the App. xaml. So let me show you what the main window looks like first. Now this is the startup window of the application. Now this stuff here I'm not going to talk about in detail or spend a lot of time on, this is the actual MahApps Metro plugin that I told you about that adds all the styling and I'm adding this at the main window level because this is the primary container window. Now, I always start out with a primary view that has a corresponding viewmodel, but I never make that view my main window, I try to make my main window just the root of the application that has some resources in it and then kicks off the primary view. And this is where the guts of the application actually start. So as you can see, I have a view here called MainView. This is the MainView. xaml that you see right here in the Solution Explorer. Now let's switch over to the main window XAML code behind file so you

can see what's going on there. As you can see, there's nothing here except one line in the constructor, and that's when I'm setting the DataContext of that control I just showed you. In case you missed it, that control had a name of Main and it is in there that I'm setting the DataContext to its corresponding viewmodel. Now I'm not just instantiating the MainViewModel class, I'm obtaining it from MEF because it was exported as I'll show you in a second. By obtaining it from MEF, I'm letting MEF kick off a results process and if there's any kind of dependencies that are in that viewmodel, they'll get resolved automatically. Now the MEF bootstrapping will take place in the App. xaml right in its code behind, right here. And as you can see, it looks just like the bootstrapping that was done in the web applications. I'm calling upon the MEFLoader. Init and I'm sending to it a prebuilt list of composeable parts consisting right now of just the ExecutingAssembly of this WPF application. And of course, the MEFLoader class will take care of doing the discovery on all the proxy classes and anything else that the client is going to be dependent on and to all of that, the WPF application is going to add its own ExecutingAssembly, adding to the mix all of the viewmodel classes that are being exported. So this is the MainViewModel class here. As you can see, it's empty. The only thing that you see is that it's exported in MEF, like all my viewmodels will, and it inherits from a class that I'll go over in detail as we approach more code, and that's the ViewModelBase class. The only thing I'll tell you about the ViewModelBase class right now is that it inherits from ObjectModelBase and if you recall, in the module where I covered client and business entities, the ObjectBase class is the base class for all client-side entities and those client-side entities offer property change notification. So all the property change notification stuff is in the ObjectBase class and the ViewModelBase deriving from it also may offer property change notification to its viewmodels. Now this is the viewmodel that's going to correspond to the main view. And this is the main view and I'll go over the view hierarchies later on, but this is the view that's going to contain one of other views, other sub-views, and this is where all the view flipping is going to take place. And all that view flipping is actually going to be done by flipping viewmodels because there's going to be data templates that resolve a view from a viewmodel, and you'll see that later on. So basically right now all we have is the application will run and MainWindow is going to display first. MainWindow is going to show the MainView within it. The MainView simply looks like this, pretty empty right now with the exception of that png file. And the MainView is going to be bound to the MainViewModel when the DataContext is set here in the code behind of MainWindow. Now this is the only time you're going to see me set the DataContext explicitly. The rest of the time it's going to be done automatically and you'll see how later on. So next I'm going to show you the hierarchy of views, what are the other views in the system, what does it look like, and we are actually going to build those views, at least their skeletal structure very quickly because we're going to keep them empty, but I want to put all this in place because right now if I run this, this is what happens. Now remember the services run first and there's the application. As you can see, it's just an empty window, that's it. Now I want to show you one more thing and that's the App. config file, because the App. config file just has our WCF stuff here. That way we're ready to access our services. So next, let's go over the rest of the views.

View Hierarchy Description

So you saw what the main window looked like and you also saw that it contained a user control in it called MainView. Now what I'm going to build next is the skeletal structure for four other views that are going to be inside the MainView. The first view is called the DashboardView and it's kind of an at a glance as to what the current fleet of cars is doing at the moment as far as rental or reservation state. Then there's a MaintainCarsView. This is where the inventory of cars is maintained. Now this is the view that I'm going to build and there's a reason for it. The other two views are the ReservationsView and the RentalsView, shows you the current reservations that are currently outstanding and of course the current rentals and allows you to execute a return on a rental. Now the reason that I'm going to build the MaintainCarsView is because it's the more complex view. It contains another view called EditCarView for a specific car. And using this view I'm going to be able to show you not only how a view and a viewmodel are built for this application, how services are accessed, all the other patterns that I have in place to solve a couple of problems that I've yet to go into, but I'm also going to show you how validation takes place when we get into the EditCarView. So next, I'm going to put the structure together for all of these views, that's DashboardView, MaintainCars, Reservations, Rentals, and I'm probably not going to touch the EditCarView, so just four of them, but you'll see how the structure looks and how they relate to the MainView, how they're going to be loaded up by the MainView. And when we get done, we'll be able to at least show, even though they'll all be empty, the MainView with its four sub-views and you'll see how you toggle between them.

Creating The First View and Setting up Its Base

So as I said, I'm going to be creating four views and four viewmodels, but they're going to be completely empty for starters. So let's create the DashboardView first so you can see what the structure of every view is going to start to look like because they're all going to follow the exact same pattern. So I'm going to come here to the Views folder and I'm going to create a new view called DashboardView. Now this view is going to be a user control, not a window, because in fact, all of my views are user controls, the only window that I have is the MainWindow and that's it. Now, that's not the case in every single WPF app of course, because I'm using full containment, which means I have one main window and every sub-view is going to be displayed by view flipping or view switching, almost the same way that you saw that the view modes were flipped in the single page application style we used on the website. But other WPF applications may actually navigate from one window to a brand new window that completely overlaps it in the traditional sense, in which case you're going to have another window, but all of mine are going to be user controls. I tend to work only in XAML view as opposed to design view, I'm not much into drag and drop in WPF, I like to do it all from the code, but to each his own of course. Now, the only thing that I'm going to do to modify this DashboardView is change the base class. I have a base class from which all of my user controls inherit from. Now I'm going to show you this base class a little later because there's going to be code in there that's going to be very relevant to a few problems that I haven't described yet, but the base class is called UserControlViewBase and I have to inherit from it.

So in order to inherit from the base class, I have to do a couple of things, one to the XAML view and the other one to the code behind view. So I'm going to start by going into the code behind view and changing the base class to UserControlViewBase, which as you can see is in the Core. Common. UI. Core namespace. Now once I change this, I have to change something out here. I have to change this right here and I have to change it to UserControlViewBase, however, I have to first define a namespace where UserControlViewBase resides. So let me put that out here. Now notice this namespace takes the form of a URL as opposed to an actual assembly or CLR type namespace. That's because up here in the Core. Common. UI where that base class resides, and there's the base class right here, I've defined in the AssemblyInfo file a pointer. And this attribute here, xmlnsDefinition, try saying that five times fast, basically is mapping any namespace that I define, any CLR namespace to a URL namespace. In a way, this is very similar to that contract namespace mapping that you saw that we did in the data contracts and the service contracts. So now I can define as many as I want here and now I can refer to them in the view simply under that URL. Now once I do that, I can now change this and of course to closing tag. And now I have a view and its code behind all set up and I'm going to define three more views like this.

Creating The Skeleton for The Other Three Views

Now following this exact same pattern, I'm going to quickly create the skeletal structure for three more views, the MaintainCarsView, the RentalsView, and the ReservationsView. And now, I need to change the base class for all three of them. And now of course I have to change their XAML appropriately. It makes a lot more sense to just use clipboard inheritance here, right? And that's it, I have all four views created. Now I need to create their viewmodels and then figure out how am I going to show these views, what's a technique that I am going to use to flip between all four of these.

Creating The Skeleton for The Four Viewmodels

So now I have to create a viewmodel for each of the four views that I created, the DashboardView, the MaintainCarsView, the RentalsView, and the ReservationsView. And I'm only going to setup the skeleton for this, but every viewmodel I create is going to follow the exact same pattern. So I'll create the DashboardView model first, and I'm going to put that right here in my ViewModels folder. And as you can see, the naming convention that I follow is exactly the same as on the view, only with the suffix viewmodel instead. So setting up this ViewModel class is very easy. The first thing I'm going to do is inherit from a common base class, and that's going to be my ViewModelBase, which as you can see is right here in the Common. UI project. The next thing I'm going to do is that I'm going to export this viewmodel. All my viewmodels are going to be exported because MEF is going to take care of importing them later. Now just like my controller classes, they're not associated to any interface even though you can come up with an exporting pattern where you can associate a viewmodel to an interface for whatever reasons you may have. In

this case, I'm not going to, I'm just going to do a simple export on its own. Now the last thing that I want to do to this viewmodel is regarding data. All four viewmodels that I'm going to write are all going to bring in data of some kind, they're all going to grab information from a WCF service and then fill themselves so they can bind to a view and that view can manifest that data on the screen. So all four viewmodels are going to use WCF services in one way or another. Now, unlike the controller classes in the MVC app, or the API controllers where I was importing, or should I say injecting my controllers with the service proxies that I needed to use for those particular controllers, I'm going to inject my viewmodels with something a little different, like this. (Typing) I'm injecting all of them with the ServiceFactory interface, which is my ServiceFactory class following the abstract factory pattern that we created in one of the previous modules. Now let me explain the reason why I'm doing this and I'm not injecting the actual service contract in here. Unlike controllers, controllers in MVC or the Web API are stateless as I stated back in one of those modules. They're stateless meaning when you navigate to a URL and wind up in a controller class, MVC will instantiate a controller for you for the purpose of executing an action and after that action is executed, it'll dispose that controller and you're left on the browser with nothing going on the server, so everything's very stateless. The next time that you initiate a call to the server, you'll get a new instance of that controller. So you're only there for the purpose of that one call. Now viewmodels by their nature are very different. They're stateful in the context on the view, meaning however long that view is being held in memory or being viewed by a user, the viewmodel is alive. And in our case, the viewmodel is alive even longer than that, something that I'll explain a little later. So because the viewmodel will only get instantiated typically once and then held in memory, I don't want to hang on to a service proxy that gets injected here in the constructor because so long as that view remains open, that service proxy will also remain open, and I don't like to keep my service proxies open for any longer than necessary. So what I'm doing instead is injecting the service factory into my viewmodels because later when I need to grab data, I'll instantiate a service proxy on demand using this service factory. And then after I'm done using the service proxy, I'll dispose of it and the viewmodel can remain active for the remainder of the day and the only thing that will stay open is a service factory, which is really not doing anything by staying open, it's not connected to anything, it's just sitting there waiting for you to use it to instantiate a proxy once again. So now I'll just create the other three viewmodels quickly. Actually, the easier way of doing them is just like this. (Typing) That's it. And I'll just refactor this out to their own file. And now I've got four views and four viewmodels that correspond to each other. So next, I'm going to go into the MainViewModel here and show you how this MainViewModel, and if you recall from the diagram in the slides, the MainViewModel encapsulates the other four, and it's going to allow the switching between them visually, and I'll show you where those viewmodels go inside here.

Completing The Main Viewmodel

When writing WPF applications, there's many, many ways of structuring views and viewmodels. Now, when using containment, which is what I'm using here, which is a viewmodel that contains other viewmodels and of course in turn

a view that contains other views, even then, there are a lot of choices at your disposal. A very common choice is to have a collection of viewmodels stored inside a parent view model and then have one property exposed and something within the view changes that property so that it becomes one viewmodel or another. Now, in our case here, I'm not going to do that. I'm going to keep it a little simpler and what I'm going to have in the MainViewModel is basically nothing more than four properties, each one is of type of the other viewmodels, the Dashboard, the MaintainCars, the Rentals, and the Reservations. And the reason I'm doing it this way is because of the type of view that I'm going to be binding to. I'm not going to be binding to a view that's going to display or bind to only one child view model at a time because I'm using a TabControl and a TabControl consists of sub-items called tab items and each one of them can bind to a property in a viewmodel. Now, let me explain to you what I mean in detail as I write code. So the first thing I'm going to do is add a property to the MainViewModel here representing the DashboardViewModel like this. Now I need to expose a public getter because the view is going to be binding to this property, but I can leave the setter private because nothing external is going to be setting this property. Now I'm going to do the exact same thing for the other three. (Typing) Now if you recall, back in the MainWindow code behind, and remember the MainWindow is the window that first comes up when the application is executed. So here in this code behind, I'm setting the DataContext of the main control, and that main control was the MainView, and I'm setting that DataContext to the MainViewModel, which is right here, but I'm doing it using MEF, which means its MEF that is going to resolve this class for me. So when MEF resolves this class, I want it to resolve everything else automatically so I don't have to worry about ever instantiating these properties here, all I have to worry about is one instantiation, and even that instantiation is done through MEF. So I'm telling it, go and get me a resolved instance of MainViewModel and resolve anything that's marked with the import, which I'll do now. And of course because dependency injections resolve process is recursive in nature, the second that MEF instantiates this class and tries to resolve DashboardViewModel, it sees that DashboardViewModel has something to resolve itself. In fact, I don't, I forgot about that part, didn't I? There we go, now I do. So let's put these out here real quick. And now when each one of these is resolved by MEF because of the resulting of the MainViewModel, the service factory is going to be resolved as well. Okay, so now that we have the MainViewModel done with its properties, we need to add something to the MainView that is going to manifest visual representations of these viewmodels, which are of course their corresponding views.

Setting up The Tab Panel

So here I am in the MainView. xaml file, which is the view that corresponds to this MainViewModel. Now keep in mind, MainViewModel is bound to this view because in the code behind here, we told it set that viewmodel to that view's DataContext of course. So now, all of these properties of MainViewModel are going to be available for binding by anything inside this view. So what I'm going to add to this view is very simple. I'm going to keep it as simple as possible and just put a standard TabControl in there. Now, normally it would look like a standard TabControl, but because I'm

using the MahApps Metro styling, it actually overrides the styling for the TabControl and gives it a nicer look and feel, something that looks more to the style of a Windows Modern App. Alright, so I'm going to add a TabControl here and I'm going to set it to the second row of this grid. As you can see, I have two RowDefinitions setup. The image is on row 0, so this is going to be on row 1. And now a TabControl consists of one or more tab items like this. Now the TabItem has two properties that I'm concerned with, as obviously many as you can see by the IntelliSense, but I'm concerned with two, the header of the tab and the content of the tab. Now, before I set the header, let's stop here for a second. In fact, just so we can compile, let's close this up. And I'm going to come out here to my DashboardViewModel, because remember there's four sub-view models of MainView and of course there's going to be four TabItems. So because my DashboardViewModel inherits from ViewModelBase, I gave it the ability to do this. Most of the time my views have some kind of title. As a matter of fact, at least 90% of the time, if not more, especially when these views are user controls. So because of that, rather than expose a property because I like to expose everything from the viewmodel because then I can localize it easier in the future too, instead of exposing a property in every viewmodel, I just made it part of the ViewModelBase. So I'm just going to add the same kind of override to the other viewmodels. Now if I go back here, I can show you the Header property and instead of hard coding it to something, even though I can because I'm going to have four distinct ones, but I'd rather grab that information from the viewmodel and since I know that this view is bound to the MainViewModel and the MainViewModel contains those four properties, I can access those properties now from binding, like this. Remember, DashboardViewModel was not only the type, but it was also the name of the property itself and that viewmodel has a property called ViewTitle. And now I can add three more (Typing). And the next thing I need is the Content property. Now in any XAML control when you have a Content property you could either set it to another control or you can set it to an object, in this case a viewmodel, and then the view that corresponds to that viewmodel will get resolved in a technique that I'll show you in an upcoming clip. So I'm going to set the Content property here to the actual DashboardViewModel property. And remember, we're bound to the MainViewModel here and we have a DashboardViewModel property, which is of type of this class right here. So that's what I want to set the Content property to. And now I've got to do the same thing for the other three. Now all I need to give the application is some information that tells it whenever you come across any Content tag or Content property that is bound to a viewmodel, this is what I want you to show instead because without me giving it additional information, what's physically going to show on the screen is going to be the type of this viewmodel. Let me show you what I mean by running this in its present state. Now as you can see, we have our four tabs and see what the MahApps Metro styling gives us, it gives us some nice looking tabs, they kind of have that Windows Modern App look to it. But look what the content looks like. The content is just the actual type of the object, not anything that looks like a view. Watch what happens if I press this. Now see we get that nice animation there that comes with the styling, but once again the type of a view, and that's it. So next, I'll show you how we can change this.

Setting up The Data Templates

My next step is to access the App. xaml file here and as you can see, this is some of the styling that I put in here from the MahApps Metro plugin and here I've got to add what's called a data template, or should I say more than one data template because this is where the mapping has to go. You saw when I ran the demo in the last clip that all we saw is the content of each tab was the actual ToString representation of the object to which we bound it. And in this case, I bound it to a property called the appropriate viewmodel, so all we saw was the ToString representation of this type. Well I want to map it, somehow telling it whenever you run across this I want you to show me this instead, the DashboardView, and that's going to be here in the App. xaml. Let me do the first one for DashboardViewModel. Now here I've got to tell it what the viewmodel is. Now up here, I already have namespaces setup, VM for the viewmodel's namespace and V for the view's namespace. Now what I'm saying is whenever you run across this viewmodel, this is what I want to show instead and I want to do the exact same thing for the MaintainCarsViewModel. So you can see, whenever it finds the MaintainCarsViewModel, show this instead. And this is only in the context of that content tag that I showed you, and now I need the other two, Rentals and Reservations. And just so we have a little credibility to the views, I think they're empty right now, let's just put something in there, some kind of label. And I'll put something in the other three. Now I'm going to run the application again and let's see what happens. As you can see, there's my text, which tells you that it's actually showing the actual view now, not a ToString representation of the ViewModel class. There's Maintain Cars, Reservations, and Rentals. So now I'm ready to actually start writing better views instead of just these simple views with the label in them. Now the view that I've chosen to write is the Maintain Cars and that's going to be written in the next module, but before I tackle that view and all its splendor, I need to describe a few problems that we're going to come across, problems that all views are going to have, and in fact these are problems that a lot of WPF applications have and they don't even know they have them. And I've come up with a solution for some of these problems and that's what I'm going to talk about next.

Upcoming Problems to Be Solved

In the next module, I'm going to be developing one of the views and viewmodels, the MaintainCars one in full. But in doing so, I'm going to be coming across a couple of problems that I'm going to need to solve, and these are problems that are actually out there and I've seen in many WPF applications and lot of times they go unsolved. The first one is one of deterministic viewmodel initialization. And what this means is that depending on how you setup your views and your viewmodels, I'd like to refresh the data every time the view comes into view. Now, the problem with this is that the viewmodel behind a view is instantiated only once. Remember, I have the main viewmodel whose properties correspond to the four other viewmodels, but that's instantiated only once and it's always kept in memory, unlike its corresponding views. Every time I tab out of one of those views and tab back into it, I'm hitting the view constructor again, as I'll prove to you later. So the view gets reinitialized every time, but the viewmodel does not. So what I want to do is give the viewmodel a way to reinitialize itself every time that the view comes into play, or comes into view while still maintaining itself in memory. Now the second problem is kind of related. Because of the fact that the viewmodels

get instantiated once and stay in memory, anything that I inject into them will remain open, and if I were to inject the service proxies like I did in my MVC controllers and my Web API controllers, they're going to remain open as well, and I don't want to do that, I want to close them up as soon as possible once I no longer need them. Now of course, if I was using a proxy that is expecting a callback or a proxy that would instantiate sessionful services, I would need that proxy to remain open during the period of that task, but these are not the kind of tasks that I'm going to be using. I'm using per call services and services that operate at a more stateless manner. So, I'd like to get the proxy in and out as quickly as possible and I don't want to keep it open, but if I inject the proxy into the viewmodel, it's going to remain open during the life of that viewmodel and of course if the view goes away and comes back in, it's reusing an existing viewmodel, it's not going to reinject a new proxy. So I don't want to close the proxy after usage because I'm not going to get a new one when the view comes back into play, so I'll show you how I solve that problem. The next problem is an age-old problem and it has to do with the fact that in some cases, my views are going to wire some viewmodel events. This is for the purpose of displaying message boxes, telling you that something has taken place. That's not something I want to do from a viewmodel, I want to do that from a view because that's visual, that's a UI thing. But if I wire an event from a view to a viewmodel, well when that view goes out of scope, remember the next time the view comes back in we're getting a new instance of it, we're still going to have a reference to that old view from the remaining viewmodel that's still in memory, and that reference is by way of the event. This is an age-old problem where you think an object goes out of scope, but it really doesn't, there's a reference left out there because of an event wiring. So what I need to do is when the view goes out of scope, I need to figure out a way to unwire any events that I've already wired so they can be rewired from the beginning when a view or a new instance of the view comes back into scope again. And I'm going to show you an elegant way of solving this problem. So in the next module I'm going to build the MaintainCars view and viewmodel and I'm going to keep it interesting by bringing these three problems into the mix and showing you how I'm going to solve them. Another interesting aspect of the view that I'm going to be building is that it's going to also incorporate a sub-view and of course a sub-viewmodel to allow the editing of a single car. So it's going to be a pretty interesting task.

Car Rental Desktop - Part 2

Intro & Highlights

Hi, this is Miguel Castro and welcome back to Building End-to-End Multi-Client Service Oriented Applications. This last major module for this course will complete the building of the Car Rental Desktop application, and as it's the last major module, it's going to be very code heavy, not a lot of slides here. In fact, let's go over the highlights, which constitute the only slide you're going to see here. In this module, I'm going to build in its entirety the Maintain Cars

view and viewmodel. Now I'm going to start with the viewmodel because I'm taking the viewmodel-first approach, which means I'm going to concentrate heavily on the logical representation of the view, what's going to happen regarding state and behavior, and I'm going to do all this in the viewmodel. Then later, the view should come together rather quickly with some bindings. Now I'm going to start by solving one of those problems I spoke about in the last module, the deterministic initialization. This is where I'm going to show you how I'm able to reinitialize the viewmodel even though I'm dealing with only one instance of the viewmodel that gets instantiated once and will be reused. I'm also going to incorporate the ServiceFactory in order to obtain proxies on demand and I'll show you a slick pattern for instantiating proxies using the ServiceFactory when we get there. As I also stated earlier, I'm going to be firing messages to the view, which means the view is going to wire into some events that are declared in the viewmodel and I'm going to show you a pattern for wiring and unwiring to make sure we don't suffer from any potential memory leaks. Now what makes this viewmodel a little more special than the other viewmodels is that it incorporates a child viewmodel as well, which means of course the counterpart view will have a child view. Remember, this viewmodel is for maintaining cars, which means it's going to start by displaying a list of cars, or representing a list of cars, and then depending on what car you wanted to edit, or perhaps add a new car, there's going to be a separate viewmodel to represent that single car. And of course, like most major sections in this course, I finish by unit testing things. And in this case, I'm going to be unit testing my viewmodels. So let's get started with the code, enough of the slides, and finish up the Car Rental Desktop.

Demonstrating View and Viewmodel Initialization Points

I'm sitting here in the MaintainCarsViewModel and I'm going to run the application because I want to quickly show you the order of certain operations. And this is what I mean by deterministic initialization. As I stated, because of the way this application is setup, the MainViewModel initializes through MEF and then of course initializes the four sub-viewmodels, of which the MaintainCarsViewModel is one. Now this occurs one time, which means this constructor right here will get hit only one time. However, the view is part of a TabControl. Remember back here in the main view we're using a tab to go across the four different subsections. Now the view is going to get disposed and initialized every time we switch tabs, unlike its counterpart viewmodel. Now, this can pose a problem. Let me show you what I mean. I'm going to come into the MaintainCarsView code behind and I'm going to put a breakpoint right here in the constructor. I'm also going to do a breakpoint in the viewmodel constructor. Now I'm going to run that app and I want to show you what happens. Now as you can see, we just broke out in the MaintainCarsViewModel. That's because as soon as the MainViewModel gets initialized, it instantiates the sub-viewmodels, thus hitting the constructor of the MaintainCarsViewModel. Now, I'm going to continue the app and as you can see we have not broken out in the constructor of the Maintain Cars view, however, if I tab over to that view, now I hit that constructor. So now I'm going to continue and I'm going to tab back out to Dashboard and tab over to Maintain Cars again. And as you can see, I hit the constructor again. I'm getting a new instance of the view, but notice I can do this as many times as I want and I

don't get a new instance of the MaintainCarsViewModel. Like I said, this only causes a problem depending on how you design your app, but what I'm seeing in a lot of WPF apps is that anytime a viewmodel loads up, the developer puts the data loading part of the viewmodel, meaning that part where the viewmodel loads itself up with data right in a constructor. The problem is that you saw that we're hitting the constructor only once, which means if we tab away and tab back in, we're hitting the same instance, which means any data that we loaded this viewmodel with is going to remain the same. So we would either have to manually provide a refresh mechanism, meaning some kind of refresh button, or, and/or should I say, a way for every time the view gets reset or the view gets reloaded, something triggers in the viewmodel to reload the data, thus keeping the viewmodel completely fresh with fresh data, and that's what I want. Now that was a decision that I made. I could have just kept the data stale and let the users themselves hit a refresh button to reload the data. That would have been fine, but I wanted to do something a little more automatic where every time you hit that tab, you have fresh data to look at in case another user at another desktop application added more cars. So that's what I'm going to show you next, is how to actually make the viewmodel's initialization more deterministic.

Solving Deterministic Viewmodel Initialization

Now, I toyed around with different techniques for accomplishing this and I came up with something that I think is rather clean. Now if you recall, the view has a base class from which it inherits, UserControlViewBase, there it is in the XAML and there it is in the code behind. Now I want to show you this base class quickly. Now keep in mind, I've had complete control of this system meaning I wrote the framework, I wrote the application. And by writing the framework, I developed the base class called UserControlViewBase, which serves as the base class for my user controls and I also developed the base class called ViewModelBase, which serves as the base for my viewmodels. And these two are going to kind of work together because what I'm doing here in the constructor of UserControlViewBase is declaring and binding a property. Now, this declaration right here simply adds a property to the view called ViewLoaded. So every view that I have, every user control-based view that I have has a property built into it called ViewLoaded. Now what I'm doing with the ViewLoaded property is that in the constructor of UserControlViewBase, I am binding the ViewLoadedProperty to a viewmodel property called ViewLoaded. Now I've ensured that all my views have a ViewLoadedProperty and because all my viewmodels inherit from a common base class called a ViewModelBase, I am also ensuring that they all have a ViewLoadedProperty. So keep that in mind, there's an additional property that's kind of invisible to the developer, they won't even know it's there. There's a property that every view receives and then there's going to be a property in the ViewModelBase that every viewmodel receives. Now let me go out to the ViewModelBase and there is a property called ViewLoaded. What's going on here is that ViewLoaded only has a getter. The reason it only has a getter is because that's all it needs. You see, in the UserControlViewBase, because of this binding, what happens is when the view first comes into scope, meaning the view is first instantiated, meaning of course we hit this constructor, and this binding gets initialized and

performed, it's immediately going to trigger a call to the getter of ViewLoaded. And in there, I'm making a call to a virtual method called OnViewLoaded, which I defined right here. And this is just a virtual method. So this enables me to do the following. I can now come into here and say override on ViewLoaded. Now I'm going to perform the same execution that I did before. Watch this. I'm going to put a breakpoint in the constructor of the viewmodel and I'm going to put a breakpoint in the OnViewLoaded now, and then I'm going to put my breakpoint back here in the constructor of the view. Now let's run the app again. There's that initializer, there's the constructor of the viewmodel that like you saw before gets hit only one time. Now if I continue the running, I'm going to go over to the Maintain Cars tab bringing that view into scope, there's the constructor of the view, and now watch what happens. Now the OnViewLoaded gets hit. Now I'm back out to the view, I'm going to go to the Dashboard view, I can go to the Reservations view, I can go back to the Maintain Cars view, there's the constructor for the new view and that new view causes the binding to take place on the ViewLoadedProperty causing the getter to get hit in the ViewModelBase class, causing the OnViewLoaded in the viewmodel to get hit, as we shall see now. And there it is, OnViewLoaded, and we got hit again. So now we have deterministic initialization. We have a method that gets hit every time this view comes into scope without having to reinitialize the entire viewmodel. Now I'm going to put my data retrieval stuff right in here. Now I can certainly put some kind of mechanism where the data retrieval happens only once or it happens every so many minutes, or something fancy if I want it to, but I'm going to do it every time the view comes into scope, meaning every time the OnViewLoaded gets hit. And that's how I'm accomplishing deterministic initialization at the viewmodel level without the developer needing to do anything different except inherit the viewmodels from one base class and inherit the views from another base class. And that's it.

Setting up The Cars Observable

Now my first order of business is for this viewmodel to collect the collection of existing cars. And in order to do that, I'm going to need some properties in which to hold those cars. So, because this is a viewmodel and we're now at the UI level, we're not dealing with just arrays or list ofs or anything, we're dealing now with more observable variables and we need to use the ObservableCollection because this is the kind of variable that is going to be bound to a WPF view later and being observable, it will raise the proper events to the view telling it to refresh itself. So I'm going to create a member variable here called Cars of type ObservableCollection of Car. Now the Car that I'm speaking of is the Car client entity that comes back from our service call. Now I'm going to write a public property to actually expose this member variable. And I'm going to follow the standard observable type pattern that you saw that I followed in my entities where if the value of the property changes, if it truly changes, I am going to fire an OnPropertyChanged on it. If you recall, back when we talked about the core framework, I have this second argument here in the OnPropertyChanged where I can toggle the make dirty to true or false, and because I don't want to deal with dirtiness right now, I'm just going to send a false here. Now that I have a place to store my retrieved car list, I'm going to go out and get it. And I do that from the OnViewLoaded here. Now what I want to do here is obtain an inventory

proxy, an inventory service proxy and then make a call to the GetAllCars operation. It's that simple. Now remember, I'm going to be retrieving an array of car back, so I need to convert that array over to an ObservableCollection because if I bind that array, it'll bind once, but now as I add things to the collection, the view won't receive notification that something was added and it won't refresh its visible list. That's why I want to deal with an ObservableCollection. So I'm going to retrieve the results from the service call and then I'm going to build the Cars collection here from those results. Now the first thing I have to do here is obviously grab a proxy somehow. And remember, I did not inject the proxy into the viewmodel, I injected at the ServiceFactory. So next, I'll show you how to obtain a proxy using the ServiceFactory.

Obtaining Service Proxy and Getting Car List

Now the task of obtaining a proxy to the InventoryService is actually quite simply using the factory. As a matter of fact, it can be as simple as doing something like this. If this looks familiar to you, it should, because this is exactly the same abstract factory pattern that we used in data repositories and business engines. So as you can see, I'm just telling the ServiceFactory go out and get me an object of type IInventoryService. And remember, the implementation that I'm using for the ServiceFactory is the one that goes out to MEF to get it. However, what's different here is that I must close this proxy as I told you, as I explained back in the website development. So, it wold be easy enough for me to cast this variable to an IDisposable and call Dispose on it, that would be fine, but it gives me one more thing to remember to do and I'd rather establish a pattern that I can reuse. So, I came up with something that I think is a little slicker, and it looks like this. (Typing) The WithClient is nothing more than a fancy helper method that's in my base class, ViewModelBase. I'll show it to you now, there it is right there. It uses one of my favorite techniques that you've seen over and over again where I have an action or a funk that I can then invoke and do something before, after or around that invocation. And as you can see, all I'm doing is that I'm evoking the code that you send into here and then I'm casting the proxy to IDisposable and disposing it afterwards. So it's just a fancy littler helper method so that with very little effort, I can now put some code in here and automatically I will obtain a proxy and dispose it after I'm done with this code. That's all that is. So now I'm just going to get the list of cars and now if I return something, I'm going to build the ObservableCollection from this list. (Typing) Now when I write some XAML and I bind some kind of list, some kind of iterative control to this Cars collection, it'll contain what's in my GetAllCars. And every time that the view goes out of scope and comes back in, the OnViewLoaded will get hit again and we'll get a nice fresh list of cars. Now, if you want some kind of caching ability or timing ability where it doesn't do it every single time, you can certainly put something like that in. So next, I'll go out to the view and put some XAML together real quick so we can see this work.

Setting up More XAML and Running The App So Far

Now, so not to bore you with writing a lot of XAML and XML tags in front of you, I've taken the liberty of putting a simple view together ahead of time here and I'll just quickly describe it. So, all I have here is a simple grid, it's got two rows in it. The first row is going to define all of the column headers because that's going to be static, so I'm using SharedSizeGroup here to make sure that any subsequent grids under it share these same names, thus the columns remain the same width, and as you can see I'm using the styles from the MahApps plugin and then the second row is nothing more than an ItemsControl and what I'm doing is that I'm binding the ItemsSource collection of that ItemsControl to the cars. Now if you recall back in my viewmodel, the Cars is right here and we're building it here. And by the way, I couldn't use a. union or something like that because I'm dealing with an ObservableCollection as opposed to a standard IEnumerable and if I try to use a union here, the compiler would just bark at me. So back here, I'm defining the one item of the ItemsControl, what's going to constitute one row of information. And each row is this grid and I'm making sure that it stays in sync with the column widths by sharing the names on the SharedSizeGroup here. And row is going to define the following five columns. I'm going to have one column dedicated to having my maintenance links. Now right now I'm anticipating using an Edit Car link and a Delete Car link, and I'll go over the details of those a little later, but after those links, I'm going to have a column for a description of the car, another one for the color, one for the year, and one for the daily rental price. And here's where those columns are being defined. And as you can see, I'm binding to the description, color, year, and rental price. Now remember, because the ItemsControl is bound to the Cars collection, the internal items, each individual item in the collection is a Car entity, and that's where these properties are coming from. So that's the whole beauty of XAML binding. Simply by retrieving and building this one collection, I can put together some XAML that through the magic of XAML binding will display what it is that I want to display. Now I'll run this so you can see it in action. There's my Dashboard and when I go over to Maintain Cars, there's my cars. I can come out here and now I get a fresh list. I can come out here, and once again I get a fresh list. Every time that this view is reinitialized, I'm hitting that OnViewLoaded virtual method in my viewmodel. So the presence of these two links here are going to segue me into the next section because obviously, showing you a list is only the first part of this, I need to be able to edit or delete, in other words maintain this list. That's what I'll do next.

Writing The First Viewmodel Command

The first test that I'm going to take care of here in my viewmodel is going to be the ability to edit cars, to edit an existing car. Now that's going to be done on a line item basis and if you saw the grid, you saw that I had a car per row and then I had an edit link in the first column next to each car. Well I want to do something in my viewmodel here that's going to become the behavior behind that edit link, and that's going to manifest itself by way of a command, a WPF command. So the first thing I'm going to do is that I'm going to declare a command. Now, I'm using the DelegateCommand pattern here. I set that up in my core framework and this is the pattern that's used heavily by other WPF frameworks or MVVM frameworks out there. In fact, the DelegateCommand is used with the same name

in the Prism framework that's become quite popular, while other frameworks use terminology like RelayCommand for example, but they're all the same thing. The whole idea of a DelegateCommand is that instead of having an external command class, you're incorporating the execution and the can execution of a command inside the actual viewmodel. Now let me set this up and I'll show you what I mean. So the first thing I'm going to do is declare a public property called EditCarCommand and this is the property that's going to be exposed by the viewmodel and that the view is going to somehow bind to. Now the argument in the generic is going to be the item that the command is going to act upon. The command is going to be exposed publicly for the purpose of binding, but it's going to be initialized privately within this viewmodel itself, which is why I have a public getter and a private setter. Now in the constructor, because this only needs to be done one time, I'm going to initialize the command. Now the constructor of the command is going to take a function pointer, because it is a DelegateCommand after all, to the function within this viewmodel that is going to be executed when that command gets executed. And I'm going to call this function OnEditCarCommand. Now this function doesn't exist yet, that's why that's breaking, but I'm going to write it right now. Now this is going to be a private function because the function itself is not going to be exposed publicly, only the command itself will be exposed. But when the XAML that is bound to this command initiates the command by way of probably a clicking of some kind, then the execution is going to take place in this function. Now the signature of this function is going to correspond to the type that's defined in the generic of the command. In this case, it's a Car item. Now I have to wire this command up in my XAML so I can test it to make sure it's being fired properly. Now that's already done. If I come out here to my XAML, you saw the EditCarCommand here. Now next, I'll explain to you in detail what this means, how it's setup, and then we'll run it to see if the command is actually working.

Writing and Testing The Edit Command

Now more often than not, command binding in XAML is actually quite simple. There's a reason that this looks a little more complicated. Usually, when the viewmodel is set as the DataContext to a view, as it is in this case, you can just refer to the command like if it was a regular property. Here it looks a little more complex. In this XAML right here, we're not in the context of the actual viewmodel in its root form, we're in the context of an individual car. Remember, once we entered this ItemControl, the ItemSource became the Cars collection, which means each item that's rendered by the ItemSource control constitutes a single car. So right here, the binding would correspond to a Car entity, just like it does down here, Description and Color. But obviously, the Car entity does not contain a command called EditCarCommand, so what I had to do was go up one to the parent. And I do that using the RelativeSource. So I'm telling it go up until it finds the ancestor of type UserControlViewBase and take its DataContext right here and that DataContext of the view is going to be the MaintainCarsViewModel where the EditCarCommand resides, and that's how I can put a command that is sitting in the actual viewmodel inside a subsection of the XAML where the context might have changed. When the command gets fired, I want to send to the command argument, or the command parameter the individual car currently in the iteration, and that's done in another binding here called the

CommandParameter. And by simply sending the word Binding in, I'm sending whatever the current binding context is. And remember, for each item in the row, the binding context is the car in question, the car that that row belongs to. And that's why out in my viewmodel I'm able to receive that car here. I'm going to put a breakpoint right here and now I'm going to run the application. Now I'm going to click edit on the first car here and we should have that breakpoint hit. And the car in question should be the Mustang. There you go. Now let's try one more. Let's edit the Tesla Roadster. Now that we know we have the right car coming into the command, we can do whatever we need to in order to properly edit that car.

Setting up The Edit-Car Viewmodel

I'm going to need a viewmodel to represent an individual car, because later in the view I need that to manifest itself the same way, meaning I'm going to have a part of the screen that's going to be dedicated to the editing of a single car. So I'm going to create another viewmodel down here and I'll refactor it out to its own code file later. And this viewmodel is going to be called EditCarViewModel. Now this viewmodel is going to be instantiated from within the MaintainCarsViewModel. So I don't need for it to be served up by MEF, but I do need for it to receive the ServiceFactory and an individual car, and I'll do that to the constructor. (Typing) Now here, things are going to be just a tad different. I still need to widen the scope of Car here and I'm going to set the ServiceFactory appropriately, but I'm not just going to widen the scope of the incoming car. You see the incoming car came from that collection, and that collection is bound to the view. I want to separate these two things, I need to decouple them because I need to be able to edit the car without affecting the car that's visibly showing on the screen at the moment. I don't want that to be a live edit, I want it to be a separate form where you can perform an edit and then choose to either apply the edits or cancel them. So I need to decouple the car that the EditCarViewModel contains from the one that came in. And now I can just map the properties over. Now because I just created a new Car object and the properties of that Car entity perform not only property change notification, but they also set the dirty flag as you saw when we did the core framework earlier in the course, I'm going to go ahead and make sure that the Car object is totally clean and it's not in a dirty state. And I'll need that dirty state set properly for later on. This viewmodel is going to represent a single car and once we're here, we're going to need to either apply changes or we're going to need to cancel changes. So I know I'm going to need a couple of commands, one of them to save and the other one to cancel. And I'm going to set those up to point to a couple of methods within this viewmodel. Now I'm going to create these methods now, but notice something a little different from the last command I created. These not only have, well this command here not only has an Execution method, but also a method called CanExecute. Now you'll see what this method is in a second. Now the signature for the CanExecute method looks a little different. In a command, the CanExecute method is what determines whether whatever control, whether it's a button or a link that is bound to that command is enabled or not. So you get to programatically or logically determine whether this command can legally execute or not, and

the only time that I want to execute the SaveCommand is if the Car that's in question is in a dirty state. Now I can put the code for cancellation and saving right here.

Calling upon The Edit-Car Viewmodel From Its Parent

Now you didn't miss anything. I still haven't showed you how we got into this viewmodel here. So first let me refactor this out to its own code file and now let's go back into the OnEditCarCommand in the MaintainCarsViewModel. And here, I'm going to check to see if the incoming car is not null, which it probably is not going to be because we got here by way of clicking one of the roll links, and then I'm just going to instantiate the EditCarViewModel sending the necessary arguments into the constructor, like this. Now I'm not going to leave this code just like this. Here's what's going to happen. Remember, we're dealing with viewmodels here and when you place a viewmodel into the Content property of a ContentControl somewhere in the XAML, it's going to defer to hopefully an existing data template to determine what view corresponds to that viewmodel. Well, inside the MaintainCarsViewModel, we're instantiating the EditCarViewModel, which means logic dictates that we're going to have some kind of EditCarView that's going to be the XAML representation of this viewmodel. So what I want to do is expose this viewmodel, the instance of this viewmodel as a public property in the MaintainCarsViewModel, and that public property will either be a null or it will be a valid EditCarViewModel that represents a single car. And the view will know how to manifest that property into either a visible part of the view that shows the specific car, or a hidden part of the view that's not showing any specific car. So I'm going to widen the scope. Instead of saying var viewmodel here, I'm actually going to setup a property called CurrentCarViewModel and set that to this instantiation. First, let me go up here and create the member variable. Now I'm going to come here and create the property that's going to get exposed publicly. There we go, call it CurrentCarViewModel, there's the type. And now I can replace this. We have a little more work to do inside this method here, but we'll come back to that. Let's go back out to the EditCarViewModel.

Validating and Saving A Car

The job of the SaveCommandExecute is going to be very simple. It's going to take the car that's in the _Car variable and it's going to save it back out to the database. In order to do so, we of course need to instantiate the InventoryService proxy, like this. (Typing) And of course in order for this to work, we need to make sure that our viewmodel inherits from a ViewModelBase. Now I'm simply going to be calling the UpdateCar operation of that service. Now if you recall, when I created this UpdateCar operation, it served two purposes, the updating of an existing car and the insertion of a new car, and it knew how to do each based on the ID of the incoming car. Now that being said, you can almost guess that I'm going to be leveraging the EditCarViewModel for both additions to the inventory or maintenance of the inventory to existing cars. So, this will serve well in both situations because _Car will either be a new instance, meaning the ID will be 0, or an existing instance. Now I haven't shown you what's going to

happen after UpdateCar yet because I need to show you something that has to happen before UpdateCar. I need to do validation. Now, I'm already setup in the Car entity with the validation rules. If you recall earlier, we used the FluentValidation product in order to perform validation routines on the Car entity. So the Car entity knows how to validate itself. Now because of routines that I've put into the ViewModelBase, validation is as easy as doing this. Now obviously the IsValid that I'm referring to here is the one that belongs to this viewmodel. So now the question is, how does this viewmodel's IsValid know that any model objects contained inside of it are valid or invalid. In fact, how does it know that there are any model objects to begin with. Well that's the piece of the puzzle that's missing here. And one thing that I forgot to do is add a property that's going to be exposing the _Car variable publicly so that the view can bind to it. Now what I've done here is provided my ViewModelBase with a way of finding out what models or what properties inside the viewmodel constitute that models that it's wrapping. Now it can 1 or it can be 100, it can be all sorts of different ones. And here is where I can just list them all. So here I'm telling the viewmodel you have one model within you called a Car. So then when I call ValidateModel here, it performs validation on all of the models that I've registered here, and based on the result of that validation, it'll calculate the IsValid property of the viewmodel and that's why I'm able to test it here, just like that. So next, I've got to find out what to do after I save the Car.

Letting The Parent Viewmodel Know About The Update

Now remember, we got into the EditCarViewModel from its parent, the MaintainCarsViewModel. And it's the MaintainCarsViewModel that is bound to the view, so all communication to the view should be from the MaintainCarsViewModel. So the EditCarViewModel should simply be sending up a message to its parent saying something succeeded or something failed, and that's exactly what I'm going to do. I'm simply going to declare an event here and I'm going to call this event CarUpdated. And I already have an EventArgs variable setup for this. And now I'm just going to raise the event down here. Of course, provided the SavedCar was successful, meaning was it null or not. Now the CarEventArgs is expecting the actual car to be sent in like this, the car that was saved, remember, and then a flag telling it is it a new car or not. So I'll just create a variable here, and now I'll create that variable up here. And now, all I have left to do is actually wire this event in, and I do that back here in the MaintainCarsViewModel when I created the instance of CurrentCarViewModel, I can also wire the event in here like this. And now this is the method that will get called when the CarUpdated event is fired from the child viewmodel.

Finishing up The Edit-Car Viewmodel with Cancellation

So before I continue on the MaintainCarsViewModel here, why don't we close the door on the EditCarViewModel by finishing that up. The only thing that's really left here is to do something on this CancelCommand down here. And this is actually going to be quite simple. All I'm going to do is shoot up a message to the parent telling it that the car has been cancelled, or the editing of the car has been cancelled. Now, it's going to work the same way that the

CarUpdated message worked by way of an event, but with a different event, and this event is going to be called CancelCommand. Now as you can see, it's significantly simpler, it doesn't even carry an EventArgs with it. And now my EditCarViewModel is pretty much complete. It represents a single car, it isolates that car away from the car that came into it, so it's a different object, it allows the saving of that car both in a new car capacity or an existing car capacity, it signals the parent that such save has taken place, and it also allows for the cancellation of the editing signaling the parent that the cancellation has taken place. So now, I'll go back to the MaintainCarsViewModel.

Updating The Parent Viewmodel with Updated Car

Now when that car that's being edited gets updated, because of the wiring of this event, this method will fire here. Now once again, that's going to be both in an existing car or a new car capacity, but right now the MaintainCarsViewModel has nothing here to allow it to add a new car. So I'm going to add that quickly by adding a command called AddCarCommand. There's the public property that'll get bound to the view and then here, I'm going to wire it in. And of course, I'm going to need a method called OnAddCarCommand, which I'll add down here. I'll add it right here in fact. Now watch closely as to what the difference is in what I'm doing in this method as opposed to the OnEditCarCommand. First of all, there is no incoming car that gets sent into the command because we're adding a new one, so it's just the regular object, and the way my DelegateCommand class works is that if you're not sending any CommandParameter, you're still required an argument, so I just use an object and not doing anything with that object. Now here I'm also going to instantiate the CurrentCarViewModel, but I'm not going to send into it an existing car item, I'm going to send into it a new car item, like this. Remember, the default ID here is going to be 0. As you can see, the rest of it looks exactly the same. In fact, I can copy the event wiring from down here. Now remember, we added another event to the EditCarViewModel, we added the Cancel event. So, I'm going to add the wiring to that as well. (Typing) Now we have the two events, CarUpdated and CancelEditCar, both are going to be called, or fired, from the EditCarViewModel in either an AddCar or an EditCar capacity. Either way, we end up here or here. So now I can go ahead and continue what happens when the car updating in the child viewmodel takes place what's going to happen back up here in the parent. So, the first thing I want to do is that I want to differentiate between an addition of a car or the updating of an existing car, and remember we have that IsNew flag that the EventArg sent in, so I can case on that like this. Now I can do something if the car is not new. And what I want to do is that I want to locate the car that was updated, which came into the EventArgs, I want to locate it in my existing Cars collection right there, and when I find it, I want to update its properties with the ones that came in from the car that was just updated. And then and only then should the grid visually reflect that change, because until that moment in time, they're two distinct objects. So I'm just going to use a link query to find the car in the collection and then when I find it, if I find it, I'm going to update the properties. And if the car is new, then I want to add the incoming car to the existing collection. And remember, since the Cars collection or _Cars is an ObservableCollection, the act of adding an item to it will be reflected in the view because that's what the view is bound to. So either changing an existing car or adding a

new car to that collection will reflect in the view right away. The last thing that I want to do here is now set my CurrentCarViewModel property, which is the publicly exposed property, to null. That way, if anything is bound to it and it's showing something regarding the editing of a car, I want that something to now disappear. And if the editing or addition of a car was simply cancelled, then I want to do nothing except set the CurrentCarViewModel to null. So next, I'm going to alter the XAML so that the XAML now takes this property into consideration and shows something that will allow the user to actually edit the car, thus effecting and interacting with the EditCarViewModel.

Updating Rhe XAML to Account for Car Editing

Let me describe to you what I'm going to do and then I'm just going to modify the code instantly so you won't see me write a bunch of XAML that will bore you to tears. What I'm going to do is basically take everything that we have on the screen right now and move it over so that it becomes the first of two possible columns in a surrounding grid, a grid that you have not seen yet. And then second column of that grid is what's going to show the child viewmodel in the ContentControl and of course that ContentControl will be bound to the CurrentCarViewModel property of the MaintainCarsViewModel and then of course using a data template it will resolve into a view, which I'll show you in a few minutes. So I'm just going to modify this grid so that it becomes a two column grid. Everything that you're seeing here will be the first column and then the second column will appear or disappear based on the content of the CurrentCarViewModel property. And that appearance and disappearing will be done thanks to a converter that I'll put in place, because remember, back here in the viewmodel, the CurrentCarViewModel property is either going to be null or an instance of EditCarViewModel, but here at the view, I need that to manifest itself as a visibility or an invisibility based on the contents of that property. So let me modify the XAML, make all this go away, and make the new one appear. Now everything looks pretty much the same as what you just saw, with the exception that it's now all surrounded by this other grid here that's two columns. Now the first column is going to contain whatever space is left if there is something in the second column or not. Now the contents of this first column is everything that you just saw in the previous version of this view. The second column is this border down here with the ContentControl. And notice that the Content property of this ContentControl is bound to the CurrentCarViewModel property and we know that this property is either going to be a CurrentCarViewModel instance or it's going to be a null. But, divisibility of the surrounding property, the border here, the surrounding tag, is also bound to CurrentCarViewModel, but using the help of this converter called ViewModelNullToVisibilityConvertor, which is defined up here, and that converter is designed to bring in the binding that you bind it to, in this case the CurrentCarViewModel, and test it to see if it's null or not, and if it's not null, then the visibility is set to visible and if it is null, the visibility to set to collapse. So the end result here is that setting CurrentCarViewModel to either null or something will not only set this property to something, but it will also make this entire border section here either visible or invisible, and depending on whether it's visible or invisible, the 0 column here, which is the rest of the view, will stretch out to fill the whole screen or a little less to make space for that second column.

Test Run Showing The Empty Edit Panel

One other thing that I added to this view is the AddCarCommand up here. As you can see, there's going to be a button sitting atop the grid that's bound to the AddCarCommand. So now let's run this. And there's my list of cars. So now I'm going to click edit and watch what happens. The panel shows. Now we haven't created a view yet and we haven't created a view mapping in the App. xaml either in a data template, so all it's going to show in that content is the actual ToString representation of that viewmodel. So now I need to create a view that's going to show right here. And part of that view will have a save and a cancel button, which will either save or cancel the editing and of course set the viewmodel back to null back in the event firing, thus causing this panel to disappear.

The Edit-Car View

The EditCarView is actually going to be very, very simple, and I'm not going to write everything in front of you, I'm just going to import the finished copy. It's going to be a simple form that has some textboxes in it to edit the car, and it's going to be bound to the EditCarViewModel. Now I will go over it, but I'm not going to write it step by step. And let me update the code behind with the base class. Now like I said, it's a very simple control and all it does is show a grid setup to be a data entry form. There's the labels for Description, Color, Year, and Per-Day Price and there's the textboxes, they're bound to the Car property dot its property, so Car. Description, Car. Color, and remember the EditCarViewModel is exposing a Car property. Now by turning on ValidateOnDataErrors equals true, I'm going to get some validation that's based on the implementation of the IDataErrorInfo in the ViewModelBase, or should I say the ObjectBase base class, which is the base of the ViewModelBase, so validation is automatically wired into this thing. The last thing that I'm going to show you is a ValidationHeader control that I wrote that is placed up here. And this control is designed to interact with properties in the ViewModelBase. Now, I'm not going to go over all of them in detail because that would just take too long and drag out the length of this module, but this is designed to work with ViewModelBase so when you see that I'm doing a binding here, I'm binding it to whatever the viewmodel is in this view. And by doing so, the ValidationHeader is going to be able to interact with properties that are in the base class, in the ViewModelBase class. And depending on validation status and validation situations, there're properties that will cause things in the ValidationHeader to appear or disappear and show you the results of validation that has taken place, and you'll see that in play in just a second. There's the buttons for Save a Cancel. Now all I have to do is map this view to the viewmodel as a data template in the App. xaml. And now I should be able to run and see that actual form and perform editing and perform all the actions that I designed.

Testing Car-Editing

So let's take the application so far for a test drive and see what happens. And I'm going to click on edit the Ford Fusion and as you can see the panel appears. And I could hit Cancel and it disappears. Click edit again and it appears. And

notice the Save button, which is dependent on the IsDirty, it's disabled right now, but the second that I dirty the underlying property like this, the button appears. Now as soon as the property is rendered invalid, I'll know about it right here. And if I try to save, the ValidationHeader kicks in like this. And this will show me all of the validation rules and it will not disappear until I make this valid. Now notice this down here is not changing, it's a totally separate copy of the object. That was an important piece that we did. Now when I press Save, it gets updated to blue. And now if I hit edit, it is now blue. So we know things work. And Add New will allow me to create a new car. So now the only piece of functionality left to wire is the Delete.

Finishing The Viewmodel with Delete and Exception Handling

Now I'm going to add the deletion code without walking you through it because it's going to be the exact same process that you've seen in the past, the addition of a command, the addition of the method that that command executes, and of course the instantiation of a proxy, or the obtainment of a proxy, and then the calling of a Delete operation on the service actually deleting the car. What I want to concentrate on is what's going to happen after the car is deleted. So let me add the delete code first and then I'll go over what's going to come after. So there's my OnDeleteCarCommand and as you can see, I'm doing things just a little bit more complex. Instead of just simply deleting the car, I'm checking to see if it's actually currently rented, because if the car is currently rented, I don't want to delete it. But now I want to put some code around this. Number one, I want to ask for a confirmation from the user on whether or not they want to actually delete this car or not. And number two, I want to report on any error, on any exception that might have possibly taken place. So I'm going to be adding two events to this viewmodel and of course because the user of this viewmodel is the actual view, it's the view that's going to wire into these events later. So I'm going to put these events up here. And as you can see, one event is called ConfirmDelete, the other one is called ErrorOccurred. Now the ErrorOccurred has its own EventArgs that it carries. The ConfirmDelete is of type CancelEventHandler. This is an event handler that carries a Boolean and this is going to allow me to send an event to the view and then bring back the result of that Boolean and decide whether to continue or not. And I'm going to do that down here in the UndeleteCarCommand method. What I want to do is actually very simple. The first thing I'm going to do is instantiate the CancelEventArgs, which is the argument that's carried by the CancelEventHandler. Then I'm going to raise the event. And then if the Cancel property of the args came back with a false, then I want to continue with the delete. Now the other piece of code that I want to add here is if the deletion attempt occurs on a car that is currently rented, I don't want to just allow nothing to happen and not let the user know about why the car wasn't allowed to be deleted, I want to actually tell them something, like this. As you can see if the car is currently rented and a delete attempt was made, I'm going to raise the ErrorOccurred. And of course, now that I have a way to get errors up to the view, I can also use the ErrorOccurred event to test this for any errors that might have occurred. Remember the FaultExceptions or FaultException of Ts or anything that the service may be throwing, I can catch for that now like this. (Typing) And obviously if I wanted to be a lot more specific, I could also catch specific exceptions

and do this for as many other exceptions as I need. And of course, I should also be doing this exact same thing anywhere that I'm making service calls. So up here in the call the GetAllCars, I should wrap all this code here and do all my exception handling just like in the Delete method. So now all I've got to do is take these events that I have here and wire them from the view, but do it in such a way that we don't run the risk of memory leaks.

Wiring and Unwiring Viewmodel Events

The place that I need to go to in order to wire the ConfirmDelete and ErrorOccurred event is of course the view, the MaintainCarsView, and I'm going to do it from its code behind. Now I want to say again, just because you're using MVVM doesn't mean that the code behind is rendered 100% useless and I find too many developers that are very dogmatic about saying don't ever, ever have a code behind. There are things to use the code behind for. What you don't want to do is use it to handle state or behavior. But, I'm going to use it for what I think are its proper purposes, which is to relay information back up to the user because that you have to do from the view and that's where I am right now. Now this brings me to the last problem in that list of problems to be solved, and that is wiring an event to a viewmodel. Now it sounds like this would be an easy thing and that I could just go into the constructor here, access the viewmodel from the DataContext property, and wire an event. The problem is that this view is going to go out of scope, and when it goes out of scope I need to be able to unwire those events because if I don't, the next time the view comes into scope, it doesn't come in from the same instance. Remember we established much earlier in this module that we get a new instance of the MaintainCarsView every time we tab into that view. However, the MaintainCarsViewModel remains the same, so now we have a viewmodel that still has event references to a view that no longer exists. So I have to wire and unwire properly. So in order to that, I have to find out what gets fired when the view goes out of scope and comes back into scope. Well, we know that the constructor gets fired when it comes into scope. However, what gets fired when the view comes out of scope? And this just took a little bit of trial and error on my part and a little bit of research. And what I found out was in the actual view, there's an event called DataContextChanged, and that event gets fired every time that the view comes into scope or leaves scope, whether it's leaving memory or not. And I have to test the DataContext in this event and determine whether this is a new instance of the view or an existing instance. Now I know I'm always getting a new instance in the way I've designed this, but I've still got to test for this because I may use a different design in the future where a view will actually stay in memory even though it's being focused out and then back in. So what I did was tap into the DataContextChanged event, but I did it in a way that the view can just override certain methods. I did most of the hard plumbing work in the base class as I always do. So I'll go to the base class quickly to show you, this is where I tap into OnDataContextChanged and I look at the old and new values coming in, which is the DataContext, to determine whether I should be wiring and unwiring events. And then I'm just calling some virtual methods, which I define here. By me doing this plumbing in the base class, I really simplify what I need to do here in the actual view's code behind. All I have to do is wire in the OnUnwire or OnWire events like this. And now in here, I'm going to access the

viewmodel, which came into here, and wire up the OnConfirmedDelete and the ErrorOccurred event or unwire them. And of course, now I have to provide the methods for OnConfirmedDelete and OnErrorOccured. And now I have a message box that pops up when the user presses the delete link causing the viewmodel to raise the ConfirmedDelete event, asking for a confirmation, returning that confirmation back in the EventArgs, allowing the deletion to go through or not go through based on that confirmation, and if a delete attempt was made on a car that is currently rented, the viewmodel will raise the ErrorOccurred event, thus causing this method here to fire and showing a message box to the user. And when you tab out of the view and go to another view, the DataContextChanged is going to get fired and because of the logic that I have in there, it's going to fire the OnUnwireViewModelEvents letting me properly unwire the events so that before this view goes completely out of scope, there's no references left out there and the garbage collector can do its job. And without this, I will have the view left out there even though I think I don't, and this is a very common problem that I've seen in a lot of WPF apps out there. Okay, so next we need to unit test the viewmodels and that's it, we're done.

Unit Testing The Viewmodel

My WPF applications always have a unit test for every viewmodel that I write. Now I'm only going to show you one viewmodel test here, the MaintainCarsViewModel, and even then, I'm only going to write three tests for it, there's just too many of them. Now remember, a viewmodel will contain state and behavior that represents a view. So I like to test any state changes that are important that are going to reflect in that view, and of course any behavior. So the three tests that I'm going to write are to test how the view loading works, remember that ViewLoadedOverride method that I have in my viewmodel, I want to make sure that works fine, and then I'm going to test the current car to see if that's being set correctly. And then I'm going to test the EditCarCommand. And the remaining tests include the DeleteCarCommand of course, the DeleteCarCommand with a cancel situation, without a cancel situation, with an error, without an error, there's a lot more detail that you can get into and of course make your test class grow significantly larger than the class that it's trying to test. In fact, that's a very common thing. So I'm going to first write a test called TestViewLoaded, then a test called TestCurrentCarSetting, and then a test called TestEditCarCommand. So I'm going to put those methods together right now. (Typing) Now if you recall, the way the ViewLoaded worked is that it worked with a cooperation between the base class for the view and the base class for the viewmodel. And the base class for the view added a ViewLoaded property to every view that inherit from it. And then the ViewModelBase had a property that was bound to that. And when that property is queried, then the ViewLoaded is fired. So all I've got to do is query that property. So the MaintainCarsViewModel in the ViewLoaded or in the OnViewLoadedOverride was actually loading in the list of cars. So I'm going to use as my test assertion a list of cars to see if it exists versus a blank list. So let me set that up first. So there's my mock up list of cars and there's my mock of the IServiceFactory and the mock implementation of the GetAllCars, which is going to return that data that I just created up here. So now I'm going to create the viewmodel and I'm going to first assert that the viewmodel. Cars collection is null because that's

what it should be when the viewmodel is first created. Now I'm simply going to simulate a view loading up. And the way I'm going to simulate that is simply by querying the viewmodel's ViewLoaded property because that's what takes place when a view is loaded up and that binding occurs, the ViewLoaded property and the ViewModelBase class is queried and the getter gets hit. So all I've got to do to simulate a view is just have something access the viewmodel's ViewLoaded property, like this and nothing more. And now I just need to assert to see if the Cars collection in the viewmodel is the same as the one that I created up here. Now I can't just check the Cars collection to be equal to data because one's an observable, one's an array, so I'll just do things a little different. I'm going to check it for not null, make sure the count is the same, and then just check that first item and that should confirm that the two are the same. So now I'm going to run this test and see what happens. And we got a pass test. So the next test here is called TestCurrentCarSetting. Now what I want to test here is that the setting of the CurrentCarViewModel property is done properly. So I'm going to mock up a Car entity and then I'm going to mock up my ServiceFactory of course, instantiate the viewmodel, and then I'm simply going to call upon the EditCarCommandExecute and send into it the car that I'm mocking up. And because of the way the viewmodel is setup, the CurrentCarViewModel should be equal to that incoming car. So let me add the code and you'll see what I mean. There, I'm creating a mock car, there's my mockServiceFactory, the instantiation of the viewmodel, and then I'm asserting that the CurrentCarViewModel property is null, calling upon the EditCarCommandExecute, which simulates the clicking of that edit link, and then asserting that the CurrentCarViewModel has been set to the same as the car. This last test is going to test the actual process of editing a car, which means I'm going to create a car, set some properties on it, simulate the execution of an edit command, then I'm going to change the properties of that car as they would be reflected in the EditCarViewModel, which remember is accessible from the CurrentCarViewModel property, and by changing those properties, I can then execute a SaveCommand and then check the original car that came back to see if that save took place. So first, let's setup the objects that I need. There's my car, it's going to be ID 1, Color white. I'm mocking up my factory of course, instantiating my viewmodel. I am adding that car to the Cars. I'm not going through the ViewLoaded here, I'm just manually putting it into the collection. And then I'm asserting that the properties are okay, the color is white and so far the CurrentCarViewModel is null the way it should. Now I'm going to simulate the execution of EditCar like this, sending into it this car. Now I'm going to assert that the CurrentCarViewModel is no longer null and I'm also going to setup my mock, which I could have done on top, I chose to do it here. So now I can mock up what happens when UpdateCar is called and when it's called, it's going to return the car inside the CurrentCarViewModel, that's the one that's represented by the EditCarViewModel, remember. So now I am going to access the car that's inside the EditCarViewModel, like this, change those properties. I'm not changing the properties of the original car, the one that's inside CurrentCarViewModel that it's currently being edited. Then I'm hitting the SaveCommand, and remember that's going to fire an event back up to the parent viewmodel and now I can assert on the original car to see if it reflects the color change. And now, before I run this test, let me fix one thing that I know is going to fail. We're not going to be able to save this car unless it's actually valid and right now as it stands it's going to be invalid because there's some required properties that are going to fail validation, which means the save is not

going to take place and this assertion down here is going to fail. So let me add the other properties (Typing) and now I should be able to run the test. So this concludes the building of the desktop application. Obviously there's some views and models that we left out, they're all included in the complete version of the code. Let's do a quick recap and then I'll tell you what's in store for the last module.

Recap & Summary

So, we finished building the desktop application. We had a pretty simple and elegant and cleanly laid out WPF application. We had one rule that I'm very, very strict about, and that is every view had a corresponding viewmodel. I follow the MVVM pattern very closely. Now that being said, I'm not dogmatic about it. I don't remove the code behind 100% if I feel it's going to come in handy for certain things. As long as I stay away from state handling or behavior handling in the code behind, which is the important thing, that's what the viewmodel is for, but you saw that in the MaintainCarsView, I was using the code behind to fire up messages to the user. And in another view, I may be using the code behind to receive an event from the viewmodel and then navigate away and instantiate another view. So there are uses for the code behind, they're just very specific uses. Now we also solved the problem of deterministic initialization. Remember, we have one instance of the viewmodel because of the way the system is setup, but we get a new instance of the view every time we navigate to it, every time we tab over to it, the viewmodel doesn't reset, but I figured out a way to actually determine initialization in the viewmodel by offering some help from the ViewModelBase class by way of a virtual method called the ViewLoaded, and that worked out really well. You'll also notice that I didn't inject WCF proxies into the viewmodel. Instead, I injected the ServiceFactory and allowed the proxies to be obtained on-demand. This way, a view can come into scope, obtain some data, and then the user can walk away to lunch, there's no proxy left open. And also, if while the user was away at lunch, the server went down and then up again and the user came back and on that same screen performed some kind of user interaction that caused a service call to be made, the proxies obtain on-demand, so the service call is going to work just fine. If that service proxy would have come in through injection, it would have been left open. If the user would have walked away and the server host would have gone down and then up again, when the user comes back and tries to initiate a call to the server with that proxy that they assume is already open, it would have failed of course. I also added a facility of the view base class for the easy wiring and unwiring of the viewmodel events. And this eliminated the problem of a potential memory leak by a viewmodel holding onto a reference to a view through an event and that view no longer existing and of course would never get disposed of. So all of the other views and viewmodels in this system use identical patterns, it does not change. In fact, I created the most complex of the viewmodel and views. Everything else is significantly simpler and of course the finished code includes all of them. So next, I'm going to do a quick conclusion and talk to you about what I have in mind for version two of this course, which I think you're going to be excited about. Stick around. The next and last module of the course is very, very short.

Conclusion & vNext

Course Recap with vNext Summary and Farewell Surprise

Wow, it's been a hell of a ride, huh? Let's see, what have we covered in this course? Well, we went through a complete design and development of a full application, a full service-oriented application. I mean, we're talking soup to nuts, end-to-end, just like the title states. And in the process I used a whole lot of different technologies. We went through Entity Framework, Windows Communication Foundation, Managed Extensibility Framework, the Mock Fugue mocking framework, Windows Presentation Foundation, MVC and Web API together, we brought Knockout into the mix, in fact, Knockout not only in a regular form, but using SPA approach, single page application. So we did quite a lot and along the way I'd like to think that I showed you a lot of tips and tricks also, some good problem solving, some unique and sometimes rather unorthodox ways of solving problems that we have in software development on a daily basis on just about every level. I showed you how to solve some problems and some common gotchas in the service side and then also on the client side in the MVC application, as well as the WPF application. So I hope you enjoyed yourself. I had a great time writing this course, and let me tell you what's in store for version two. I'm already planning a version two, it's already in development. Now version two will enhance and continue to build upon the very site that I wrote in this course. I'm going to start by adding TypeScript to the mix. So I'm going to be redoing the knockout viewmodels using TypeScript and make them a little more object oriented. You're going to see the viewmodel helper instead of being contained, it will be in a ViewModelBase class, so things are going to be a lot cooler once I bring TypeScript into the mix. I'm also going to be adding a mobile site to the MCV app, which means we're going to have mobile-specific views that a customer is going to use in order to access the car rental system and make a car reservation. I'm going to of course write a Windows 8 Modern App, what course today would be complete without one, right? So you're going to see that takes place, also from a customer standpoint. I'm going to add server-side validation to the entity layer. Now this goes hand in hand with the fact that I'm going to be exposing a public API. I want to expose a Web API layer that will actually encourage developers to write applications against my back-end, but in doing so, I now have to validate the back-end a little better, and if you recall I wasn't doing any validation on the server side in this version of the app. I'm also going to be introducing concurrency management. I'm going to be showing you some techniques for having record locking and things like that take place. And then I'm going to be introducing one of my favorite features of WCF, and that's discoverability. Now that's the ability for the client to actually discover things about the services including if they're running or not. Now this is great because right now, if the host is not running and any of the clients start making calls, all hell is going to break loose and things are going to start blowing up all over the place. But I could actually test from the client to see if the hosts are available and then act accordingly on the client side. And I've got a lot more stuff planned. I'm going to be adding HTTPS capability to the site. I'm going to be adding more framework things. Also some advanced security scenarios that the new clients

are going to need. So, all sorts of cool things I've got in store. So believe it or not, that's it, that's the end of the course. This has been a long course, I really hope that you enjoyed watching it. I hope to hear some feedback from you. You know, you've sat around for a long time watching this course. I think it's only right that I come out and thank you in person. Alright, this a little better huh? This is the way we should end a course. How about this, I love this. Watch this, I can bring Visual Studio back just like this. Is this awesome? Talk about being on top of your code. You can scroll up, scroll down, up. Wow, I'm going to do a whole course like this one day. Anyway, let's go back to the slides. Alright, so I wanted to come out in person and say thank you very much for sticking around for such a long course, probably one of the longest that you've seen in the Pluralsight Catalog. I hope it's been worth it, I hope it's been informative, and I hope you had a great time watching it. I certainly had a great time making it. So I'm looking forward to showing you version two, which is going to come out by the end of the year and I really want to hear back from you, whether the feedback is negative, positive, I want to know what your least favorite, most favorite parts of the courses were, let me know what you think. I want to make the next one even better than this one. So, come back to Pluralsight. com, check out my name, my courses, look at all the other great courses by a lot of great authors out there, and we're going to see each other again real soon and hopefully have as much fun as we had this time. So take care and I'll see you again, bye bye.

Course author

Miguel Castro

Whether playing on the local Radio Shack's TRS-80 or designing systems for clients around the globe, Miguel has been writing software since he was 12 years old. He insists on staying heavily...

Course info

Level	Intermediate
Rating	★★★★☆ (1152)
My rating	★★★★★
Duration	11h 43m
Released	30 Aug 2013

Share course