[How to build a microservices based app using .Net, NextJS, IdentityServer, RabbitMQ running on Docker and Kubernetes](https://www.udemy.com/course/build-a-microservices-app-with-dotnet-and-nextjs-from-scratch/)

<https://github.com/trycatchlearn/carsties>

Contents

[Section 1: Introduction 2](#_Toc176187160)

[1.Introduction 2](#_Toc176187161)

[2. Course pre-requisites 5](#_Toc176187162)

[3. Setting up the developer environment 6](#_Toc176187163)

[4. Course assets and source code 8](#_Toc176187164)

[5. VS Code troubleshooting and tips 11](#_Toc176187165)

[6. Microservices - the big picture 13](#_Toc176187166)

[Section 2: Creating the first microservice 16](#_Toc176187167)

[7. Introduction to section 2 16](#_Toc176187168)

[8. Creating the first micro service 18](#_Toc176187169)

[9. Reviewing and simplifying the project 22](#_Toc176187170)

[10. Adding the entity classes 28](#_Toc176187171)

[11. Adding the Database context class 31](#_Toc176187172)

[12. Adding a Postgres database server 37](#_Toc176187173)

[13. Creating seed data 42](#_Toc176187174)

[14. Shaping the data to return 46](#_Toc176187175)

[15. Adding an API controller 49](#_Toc176187176)

[16. Adding a postman collection 53](#_Toc176187177)

[17. Adding the create auction endpoint 56](#_Toc176187178)

[18. Adding the update auction endpoint 59](#_Toc176187179)

[19. Adding the delete auction endpoint 62](#_Toc176187180)

[20. Saving changes into source control 63](#_Toc176187181)

## Section 1: Introduction

### 1.Introduction

Hello and welcome to my course Building a microservices app with Dotnet and Next.js from scratch. My name is Neil Cummings. I'll be your instructor and mentor through this journey to build such an application.

A screenshot of a computer

Description automatically generated

And let's take a look at what we are building on this course, first of all. So we are going to be building a demonstration, **proof of concept auction application** for cars and users will be able to **advertise their cars** on our auction site and other users will be able to **place bids on those cars**.

And this client side application is built using **next.js and tailwind CSS** to provide the **styling**.

We've got various **filters** and we can take a look at **completed auctions, live auctions, auctions** **ending soon**. We can **order** them and we can **adjust the page size** here as well. And we can also **search for specific makes models or colors** of the cars as well.

A close-up of a sign

Description automatically generated

Now, if a user does want to **place a bid** on a car, they will **need to log in** and to provide the login functionality.

A screenshot of a login box

Description automatically generatedWe're actually going to be using **identity server** to provide a secure method that the users can log in. So we're actually redirected to identity server at this point. And I'll just enter the password for Bob so that I can log in and we're taken back to the app.

A screenshot of a phone

Description automatically generated

And now **Bob** has the ability to do things. He can look at the auctions, he's won the auctions, he's selling, he can sell another car and we can also go and place bids on auctions.

A close-up of a sign

Description automatically generated

And if we look at this auction for this Audi TT, Bob can decide to **place another bid** because the **Reserve** has not been met. And Bob adds his new bid and finds again the Reserve has not been met.

But we're also hooking this up to **signalR.** So anybody else **connected** to this auction would **see** these bids **coming in live** and then they can make competitive bids as well.

And we can see things like the **time remaining** in this auction.

So this is what we're aiming to build in this training course and all the functionality for this auction sites is made up of **several microservices** that all do different **jobs** and provide different **elements** to what we see here and what we are using to build.

A group of logos with text

Description automatically generated

This is in the main part for all our backend services. We're going to be using **.Net** to do that and the client side application is going to be built using **Next.js**, which is an excellent **framework** that is **react based**, but it's high **performance** and it gives us a lot of cool **functionality**, especially with the newest features that they have added to that.

We'll be running our services over **Docker** **in development**. We'll be effectively **publishing** our application locally using **Docker**, but for real publishing and this is kind of **optional** because there's **no free** option, we'll be using **Kubernetes** to do so.

The service to service communication is going to be provided by **Rabbitmq**, and for authentication and identity purposes, we're going to be using **Duende Identity Server** to do so.

So what you need to participate in this course?

A group of logos with text

Description automatically generated

Well, you do need an operating system, of course, that runs Windows, Mac or Linux and a code editor or IDE of some description.

All of the demos in this training course, I'll be using **Visual Studio code** to demonstrate, but that doesn't mean you have to do so.

And you also need a **computer that's capable of running Docker** on your machine. And I would encourage you to check out the **Prerequisites video** I've made available in the previews just to make sure that you have what you need to participate in this training course. And I'd also encourage you to check out the preview videos that I've made available for this training course and check out the reviews of course, from this and any of my other courses.

### 2. Course pre-requisites

In this lesson, I just want to run through the **prerequisites** for you to be able to participate in this course. And regarding your experience, please note this is **not a beginner course**. I am expecting you to have some coding experience and please do not use this course as your very first coding adventure. It will, I promise you, be far **too challenging** as your first coding experience.

Now we are using. Net and we're going to be using next.js which is react based and I wouldn't think you need much experience in either of those so long as you've got **experience in other languages**. I would think that the code that we write on this course, you will be able to pick it up even if you've not used C-sharp before. As long as you've got experience in other languages, I'm pretty sure you'd be able to relate to what we're doing inside this course and understand it.

Also, for this type, of course, we do need to think about the **operating system and hardware** **requirements**. There is a minimum that you need to reach and we'll just review that by taking a look at the **Docker requirements**. We'll be using Docker significantly inside this course to run the containers that are used for our different services that we build on this training course. Please come to this website **docker.com. Get started**, learn how to install Docker and please pick your operating system and see what the **minimum requirements** are for yourselves.

<https://docs.docker.com/desktop/install/windows-install/>

So if I take **Windows** for example, then it does tell us what the **system requirements** are, including the hardware requirements inside here as well. On Windows, there's even **Bios level hardware** **virtualization** that has to be enabled in the Bios settings. So this is something you would need to review to see if you can indeed participate in this course, because Docker is going to be a first class citizen in this training course and we do rely on it significantly to run our different services.

So please note the **hardware requirements** see this as an **absolute minimum and 4 gigs**. I don't know. I think if you're running all the services we create on this course with four gigs of Ram, I think your computer is going to be screaming at you if you've only just meet the minimum requirements. Also, please note the **operating system requirements** as well. If you're on Windows, then you do have to have at least Windows 10 64 bit or Windows 11 to run Docker on your computer. If you're still using an older version of Windows, then apologies. But again, this is something that would prevent you from installing Docker and participating in this training course.

So please do come to this page and make sure you meet the minimum requirements for Docker, because that's the thing that's probably going to be the thing that causes you problems if you don't have enough hardware to run what we're doing on this course, you can also test the ability to run the app by coming to the course GitHub repository.

<https://github.com/trycatchlearn/carsties>

Inside here there are some **instructions** on how to run the app locally on your computer. Nine steps and you'll get to see the app running locally on your machine.

But please bear in mind this part here. Step four Building the **services locally running on your computer may take several minutes to complete**, and if you've got just the minimum requirements, your computer might not be happy about you doing this.

As this is quite intensive work to build all of these different services. So I'll leave that up to you if you want to test to see if you can run the app locally, then this is an option as well for you to take a look at.

### 3. Setting up the developer environment

In this lesson, I'm just going to run through setting up the **developer environment** and really just to let you know what you need to **download and install** for what we're going to be using in this training course to build the application.

So first of all, we need a **code editor** or IDE. It's not a beginner course list, so I suspect you already have a preference of what you typically use to write code. Now, because this is a net course. We do need something that's compatible with **.net** and can **compile and run C-sharp code**. The one that I will be using to demonstrate everything is **Visual Studio code**.

You do not have to use this, but I have to use something to demonstrate and I pick Visual Studio code because it's free to use. It's very lightweight, it's pretty fast and it's pretty good, but it's not perfect. And there is a lesson coming up that will show you how to deal with the **common issues** you might face with Visual Studio Code. But if you don't like VS code, you don't have to use it. You can of course use **Visual Studio** if you prefer to use that. I find it a bit heavyweight and I don't use list to demonstrate because it's a very **Windows specific** thing and as you can probably see that I use a mac anyway.

And the **Visual Studio version for** **Mac** is very different to what you guys on Windows have. The one for Mac is a vastly **inferior product** compared to the version that Windows has for Visual Studio. So I stick with **Visual Studio code** for development.

What I typically use in my daily life though is **Rider**. This is excellent, it is **cross-platform**, but it's not free to use. So I do not use this for demonstrations on this course, but it is an excellent tool for C-sharp development. But please just bear in mind it's **not free** and you do need a subscription to actually make use of that.

Something else that we need to install and we have on our computers is **Docker**, and you will need to go ahead and install this on your computer to participate in this course. And please do take a look at their **installation** **guides** inside the **docs** and getting started and how to install Docker. You'll find there's a guide for each different operating system and inside the installed **Docker** **desktop**. Please do take a look at this, especially on Windows because there's different options depending on whether or not you're on Windows Home Edition or Windows Pro. You do have to have Windows 11 or Windows 10 64 bit. And please bear in mind the minimum requirements to get this stuff running. It does say 4 gigabytes of system Ram, but please see that as the bare minimum that you can use to run this as your operating system is going to consume a chunk of that Ram. So is Docker and the services that we build and run. You might find your computer screaming at you if you do have the bare minimum that's required to just get this thing installed.

We'll also need **NodeJS** because the client side app that we're going to be creating does run on top of Node. And I've used version 16 because that's what I already had installed on my computer. But please go ahead and download a version of Node for your machine. The **long term support** option is normally the good one to use for this as the latest and greatest versions of Node. Sometimes you'll run into issues, but the long term support option typically works just fine.

Because we're developing a net application, you will need to get the **.net SDK** and the current version and the version I've used on this course is **.Net7** and please go ahead and install this for your computer. If Dotnet eight is out of preview, then you might want to use that instead. I've taken a look at the upcoming features that are introduced in Dotnet eight. I do not envisage any issues because nothing they're adding to Dotnet8 is conflicting with what we're using in .Net7 so please grab the appropriate version for your operating system.

Also on this course, we're going to use a tool called **Postman** and we're going to be using this extensively to **test our APIs as we build them**. I do have a pre-built **postman collection** that will save us a lot of time as we go about building and testing our application. So even if you do have a preference for another API testing tool, please do download and install Postman onto your machine as you will find it helpful as we build this application.

We will be **saving our code into source control** at the end of every section. I presume you already have this on your computer, but if you do not, then we're going to be using **Git** to do that.

And also we're going to be using **GitHub** to save our changes into a public source repository. We're going to be using GitHub because that provides us with some **continuous integration** **features** that we can add later on in this training course. So please do get yourself a GitHub account and sign up for this if you do not already have one, of course.

<https://github.com/onlytafkas>

### 4. Course assets and source code

In this lesson, I just want to point out where you can get the **course assets and the source code** from that we're going to be using in this training course.

So first of all, attached to this lesson, if you take a look at the course outline, you'll see attached to this lesson there is a **resources button** you can click on. And that will give you access to download the course assets zip file. Once you have this and you open it up, then what you'll find inside here is a few folders. One of them is **postman** that contains the postman collection that we'll use at various points in this course. I'll demonstrate what we need to do with this as we get to the part of the course that involves it. Also, there's a bunch of **snippets** in here. We do write almost all the code manually, but there are some parts where it's a bit too onerous and unnecessary for us to write out the code manually, and I've supplied some snippets where necessary on this course. On a few occasions we'll use them, and inside here there's various **spec files** that you can take a look at.

A diagram of a software application

Description automatically generated with medium confidence

And one of them might be the **architecture**, for example, that I refer to on many occasions during this course that you can have locally and refer to whenever you need. Also each of our **services,** I've got a **spec sheet** for that. You can also take a look at.

A screenshot of a computer program

Description automatically generated

And just in case you want to know what I'm using inside **VS code**. I've also got the **VS code config** that I've used and the extensions, all of the **extensions listed out** here that I add at various points during this course.

You don't need to install them now. I'll cover that as and when we need them.

So please do have this course assets folder available and just store it on your desktop somewhere, as I'll refer to this at various points in the course.

Also, what you can get access to is the **source code** for the course at <https://github.com/trycatchlearn/carsties>

A blue sign with white text

Description automatically generated

All of the source code is contained inside this **repository**. What I would suggest actually, because microservices, we write a fair bit of configuration and all of those configuration files have the potential for you to make typos in what I would really suggest to help you out on this is just **grab a** **local copy** of this by using the button here, and copy this address into your clipboard and just find a spare **folder** on your computer and just **clone** it locally into your computer.

So you've got a copy of this source code that you can refer to at various points. And if I just clone this repository into this empty folder, and then we can CD into the carsties directory. And this gives us a copy of the code we're going to be writing during this course.

Now each section has a commit.

A screen shot of a computer

Description automatically generated

And we can check the **git log** inside here. And we can see all of the commits that have been added at various points. This view is a bit messy. So what you might want to do if you do want to grab a local copy of the source code is just use **git log –oneline**. And this gives us a cleaner view of the commit itself.

And then we can grab - let's say we wanted to see the code because we were in section 6 - and we wanted to see the code that was being used there. We can just **grab this short version of the hash** and copy that into our clipboard and just say git checkout and paste in the commits, and this gives us the **code as at the end of section 6**.



So if you do need to refer to the code then this is a good way to do it. And we can get back to the main branch by saying **git checkout main**.

And now we're back to the original main branch.



That command that I used to get the cleaner view of the logs is a bit long. So what we can do is we can copy this part of the command. We can add an alias into our git config. This configures an alias for us. So then we can just say git adog. And we get the same command and the output and a clean view of the git commits.

### 5. VS Code troubleshooting and tips

In this lesson I just want to run through some vs code **tips and troubleshooting** options. If you run into issues with this tool that I'm going to use to demonstrate in all the lessons.

Now, first of all, it is not a requirement for you to use **Visual Studio Code** to write this application. I'm expecting you to have some coding experience to participate in this course and therefore I expect you have a code editor or IDE preference already. And as long as that IDE is capable of compiling and running C-sharp code such as Visual Studio, Rider and vs code, of course, then you can use whatever you prefer to use and I'm quite a big fan of VS code.

I don't use it every day. Admittedly, I do use rider because I have a subscription to that, but I do use vs code for demonstration because it's **compatible** with Windows, Mac and Linux and it's pretty much the same **experience** on all of those different operating systems. But if you do use VS code and run into problems, then please take a look at what I'm going to demonstrate here to help you resolve that problem because it is **not perfect** and I have run into several issues whilst recording the lessons in this course.

So I just want to mention a few things that I am going to be using and demonstrating. Don't worry about installing what I have that I'm going to display here and now I'm going to introduce these in the lessons where we actually need them. I want this to be a place you can come back to if you do run into issues with VS code, because one of the extensions that I've used to build this application is this new thing at the time of recording called the **C sharp dev kit**, it is still in pre-release and preview mode, which means it's not perfect, but I found it useful enough to use and include in this training course for developing this application.

And what this gives us is some Visual Studio like functionality and it gives us C sharp project and solution management via a solution Explorer native testing, a language service and **AI assisted** **development**. Don't worry, we're still writing the code ourselves and you'll see what I mean by this. It can be useful, it can be annoying. This AI assisted development feature with something they call **IntelliCode**. However, what I do want to mention is that on occasions when I was writing the code, sometimes in the project I would open up one of the files, for example, and the **using statements** would be **grayed out** even though they're needed. And I couldn't hover over the classes and get information about what's going on.



When that happens. And you kind of got yourself in a bit of a freeze situation, then what I would suggest is you use the **show all commands**

A screen shot of a computer

Description automatically generatedand inside here we've got a **developer reload window** command. Please use this as the first troubleshooting thing that you do inside vs code. This does solve 99% of the problems that you might face.

However, in this c sharp dev kit, this is a special case and sometimes even that would not solve the problem. And I found myself using the **gear button and switching between the release and pre-release versions** again to try and resolve that problem. And I would take that action. It would go ahead and **reinstall** the c sharp dev kit and then I would try and reload the window after that. Again, this might have to happen 2 or 3 times for me to get Visual Studio code back up to where it was before. This can be annoying.

A black background with white text

Description automatically generated

Also inside vs code, if we open up the terminal and take a look at the **integrated terminal**, then on here as well, we have an **output window** where we can get further assistance and information with what is going on with particularly C Sharp.

A screen shot of a computer

Description automatically generatedThis was the one I had most issues with and I don't want to say that I had lots of issues all the time. It was every now and again, typically after leaving my computer alone for a while, then possibly it would run into issues. But you can get further assistance with what's going on in the output windows with these different services or extensions that you're running on VS code. So do please take a look at this **output** window for the **logs** and see if that can give you more **information** about whatever problem you are having.

But like I say, most problems can be resolved by simply **reloading** the window. And students in the past have often written a question in the Q and A about VS code problems they're having. And really the best I can offer is to suggest that you reload the window, reinstall the extension, check the logs.

What you see in this training course has of course been edited and I don't show you me reloading the window whenever it gets stuck. And I'll just go ahead and do that to fix whatever problem I have. And like I say, you don't have to use Vs code. If you do find it frustrating, then please do use Visual Studio or please do use Rider as an alternative to this. I just use this to demonstrate because it's **free** to use and it's the same **experience** across the **different operating systems** out there.

**To be able to make screenshots**

Google – Settings – System -> disable graphics acceleration



### 6. Microservices - the big picture

So before we begin writing any code, let's take a look at the **big picture** stuff around microservices and what microservice architecture enables us to do.

A diagram of a computer network

Description automatically generatedAnd it effectively enables large teams to **build scalable** **applications** that are composed of many **loosely coupled services**.

And what does this architecture look like and when should we use it?

So our services are loosely coupled. **Each service** handles a **dedicated function** inside a large scale application.

A search engine optimization process

Description automatically generated with medium confidenceSo, for example, in our case we will have an **auction service** that handles rest based API crud operations, a **search service** which obviously handles search functionality and a **bid service** to handle the bids for an auction.

We will have more services, but each of these services all can be **individual microservices** and the **idea** is that each microservice **does one thing** and it does it well.

Also, each microservice is going to be **responsible for its own data.** We do not share databases between our different microservices, but these are separate services.

They're **not separate applications** that don't have any reliance on the data that's contained with a different service. So that means they need to communicate with each other and there are approaches to doing so. They can either **communicate** directly with each other via **protocols** such as **Http**, which I'm sure we're all familiar with, or more commonly for **direct** **synchronous** communication between services, then we would use something like a remote procedure call or **gRPC**, which is **fast** communication between our services.

But one of the goals of a well-designed microservice is that it **should be** **autonomous**. And if we **lost** service A and service B should really be **unaffected** by that loss of that service and it should still be able to **operate** and **respond** to any **requests** that it receives.

A diagram of a service bus

Description automatically generated

So whilst we can use that approach to communicating between our services, more typically we would use something like a **service bus** which allows us to make **fire and forget Asynchronous** **communication** between our two different services.

Microservices is **not** an **easy** way to build an application. So we need to focus on the **benefits** it gives us. And one of the benefits is that we can **survive a failure** **of one of our services** and it should not impact the rest of the services. So we want to aim to design our application in such a way that our services can communicate with each other. But **a failure of one does not impact the rest** of our application.

And by **designing** it that way and to have our services as these **small, loosely coupled services** that are easy to reason about because they **do one thing** and one thing, well then this gives us peace of mind to make changes to our individual service as a team of developers and deploy and **deploy often independently** of the other services that make up the application as a whole.

So another goal of a microservice architecture is we should be able to **deploy each service** **independently** of any other services. We do not need to deploy our entire application any time a change has made in a single service.

A screenshot of a computer

Description automatically generatedAnd another reason for microservice popularity nowadays, and one of the reasons it's become a trendy buzzword and the thing to go to when creating a new large application is its ability for us to be able to **scale each service independently** of other services.

If our search service was particularly busy, we could scale that independently of the other services, or if one auction or an auction was having thousands of thousands of bids at the same time, then perhaps we'd want to **scale** that one **individually** so that it can cope with its load without necessarily needing to scale the other services.

And that's one of the **downsides to a monolithic approach**, is you do not have that flexibility. You can only **scale horizontally**, which means you make copies of the entire application and run them on different servers or you **scale vertically**, which means you make your server much, much bigger, which isn't really practical.

So it all sounds good, but then we need to think about what the **difference** is **between this and a** **monolith.**

A orange cylinder with black text

Description automatically generatedNow, when a **monolith** we typically have **one** **single database** and that database would be responsible for maintaining **foreign key relationships** between our different tables in that database. And it would be responsible for **enforcing referential integrity** between those functions or units of our application.

A orange cylinder with black text

Description automatically generatedBut in a **microservices architecture**, **each** service has to be **responsible** for its **own** **data**. And now the burden of **maintaining** that data **integrity** is now shifted onto the **application** **layer** rather than the database layer.

So that is something else that we need to think about when it comes to building our services. How do we maintain **data consistency and integrity** between them? And that's the part where **microservices** becomes significantly **more complex** than a monolithic architecture.

A diagram of a computer network

Description automatically generatedA couple of **other components** I want to mention that make up a **microservice** based application that you would not see in a monolithic application is we would need something like a **gateway** that **receives** **requests** from our client applications and is **responsible for forwarding and transforming** those requests to the different services inside our app.

And we'd also have something like an **identity provider** that sits **outside** of our microservices **application** and our gateway, could check with the identity provider to make sure that the **request** is **valid** and **authenticated** before forwarding it on to each individual service.

Another downside to microservices is that it is **not the cheapest** **way** to build an application. Each one of these individual services and their corresponding databases does cost money and more money than it would be to build a monolithic application.

So this approach is not really suitable for small teams. It would **cost** **more money** and it would take longer to build the application this way. We're really looking at something here. Let's designed for **large teams** so that they can **operate independently** of each other and by building these services **independently and small** **well designed**, **do one thing well kind of service**. Then each individual team can operate simultaneously or concurrently on those different services and it can enable the application to be built faster so long as each team can operate independently of each other and not rely on another team's functionality before they can get on with building theirs.

So **good for big teams and big organizations**, not so good for small teams and small organizations. But anyway, that's just a taster of microservices that I wanted to introduce. And over the coming lessons, we're going to implement this kind of functionality to build our app.

## Section 2: Creating the first microservice

### 7. Introduction to section 2

The first coding section in this course where we go ahead and **create our first microservice** in our microservices application. And in this section, we're going to start off by creating a **simple API for** **the auctions**, and it's going to be a simple **crud** **application**. Now the complexity in this application is going to be the microservices themselves and not the code internally inside those services. We're going to keep them as simple as realistically we can and we're going to start off by building a crud application.

We need our users to have the ability to **create, read, update and delete auctions** in our auction style application, and that's what we're going to create for our first service.

So we're going to take a look at things like **adding those API endpoints** and we're going to be **seeding some sample data** into our database, and the database technology that we're going to be using on this course is Postgres or in this service, it's going to be **Postgres**.

We're also going to be taking a look at **Docker compose** to get our Postgres database up and running. Docker is a requirement for this course. You will need to have Docker installed running and available pretty much all the way through this course and that is a requirement for it as well.

We have to use this even though Docker is not microservices and microservices is not Docker, it would be highly unusual to create this type of application without taking advantage of the **functionality** that Docker gives us to **Containerize** or **Dockerize** our different **services** and have an easy way to get them started, stops, whatever we're doing. So we are going to heavily use Docker in this course as well and we're going to start off by getting Postgres up and running and we'll be using a tool from Docker to do that called **Compose**, and that's how we're going to be running our services whilst we're in development mode.

Also, **Postman** is something else that we're going to be using extensively in this course. So please do have this installed and available as well. And even though there's other ways to test your API, maybe you like to use **swagger**, maybe you've got a different tool in mind for testing API endpoints.

Please do use Postman because I have provided a **pre-built postman** collection that contains various **tests inside it** and also checkpoints as well. So as we go about building our whole application at certain phases, there'll be checkpoints just to reassure ourselves that what we're doing is correct as we go along, so that we don't run into a problem later on related to something we do early on. We'll know that if our **checkpoints** pass, then we're on the right track and the later functionality that we also add will work as well.

A white background with black text

Description automatically generated

A diagram of a software flowchart

Description automatically generated

Now just to highlight the overall picture and I'll reintroduce this slides as we go along just so that we know where we stand as we build the application. We're **starting here with the auction service** and we're going to have an **auction service web application** or a **Web API with Dotnet**, and it's going to have a **Postgres database** to persist the data.

And obviously we've got a lot more to build, but this is where we're starting and I'll refer back to this diagram quite often as we build our different elements that we're creating for the microservice application as a whole and also for each service or net service that we're building.

I've provided a **specification** so that we're aware of the different types of things that these services are going to support. And you get this from the course assets folder. If you do not have that already, please go ahead and download that. With this, this gives us this course assets folder.

### 8. Creating the first micro service

let's begin and start writing the code for our application. Now in this application, we're going to build an auction site. And just to let you know where I'm starting from. Well, it's an empty directory in a folder called demo. And I feel I need to give you a fictional scenario to give you the motivation to build what we're about to build.

And let's pretend that you're an employee working for a very famous auction site Christie's and Christie's have decided to branch out into selling things other than fine arts and jewelry and vases or whatever else they sell, and they've decided they're going to enter the online auction for Cars Marketplace, and they've tasked you as one of their developers, to build part of the online auction sites that they have in mind. Now they don't want it called **Christie's cars**. So you've suggested the name for the site should be **Carsties**. And that's what we're going to utilize for the name of our project.

So I'm just going to make a make directory and say carsties as the name of our project and inside carsties. Let's just cd into this. And naming things always is difficult. By the way, not a great name, and please use whatever name you feel is more appropriate for your particular site. But this is the one that I am going for.

Now we're going to **start with the auction service**, which effectively is just going to be a **crud** application; create, read, update, delete of auctions and auctions in our case is going to be auction **specifically for cars**.

And if we go and take a look at the [**specification**](04%20CourseAssets/CourseAssets/specs/auctionSvcSpec.pdf) that's been provided for this particular service inside the course assets folder. Now inside here we've got some **details about the infrastructure** for this project. It's going to be a .Net web API is going to use a Postgres database, and we're going to use Entity Framework to connect to that database. A bit later on, we'll take a look at asynchronous communication between our services with the service bus. But we're not covering that in this section. Our goals are quite simple actually for this section. And that's just to create a Crud application.

Now you might have done this a hundred times before. And if that's the case, then watching me go through this step by step to build a Crud application isn't something you want to spend your time doing. If that's the case, that's fine, and you can go ahead and just follow the spec sheet and create all of the different elements to get a Crud application for our auction service up and running.

I'll leave that up to you, but everything's included in the spec sheet that tells you which **API** **endpoints** we need to create, what the auction **entities** are, or the **models** are going to be, and also what the **Dtos** are going to be, how the **data** is going to be **shaped** when we return it.

In this section we're going to do things as simple as possible just to get this up and running, so that we've got data that we can query from our database we're creating for this specific service. So that's what's coming up in this particular section.

There is a **postman collection** that we will be using to test our work. So by the time we get to the end of this section, you'll be able to confirm that you've done what's required.



First of all, let's just double check what we have available regarding the dotnet SDK is we can do that by typing **dotnet --info**. And this will give us a list of the different **SDKs we have installed** and the **runtimes** for dotnet that we have installed. We can have **multiple** of these installed side by side. Make sure you have this available.

Once you've confirmed that we can move forwards and I'm going to **create a web API type of application**.

If you type in **dotnet new list**, then this will give you a list of things that you can create with the dotnet SDK. Now I'm going to be using the **dotnet CLI**, the command line interface to create these projects. If you are using something like Visual Studio or Rider, of course you have the facility in there to do this with a graphical user interface, entirely up to you how you want to go about creating these. I'm going to demonstrate it using the command line interface, because that's the method that's compatible with windows, Mac and Linux. And Visual Studio, for example, is only available for windows or at least Visual Studio 2022. The version for windows is different to other versions, so I'm going to be using the .Net CLI for things like project creation. I'll be using VS Code for code editing and the IDE.

So we need a **web API**. So this is what we're going to create. But we're also going to store our projects inside a **solution file** because this makes it more convenient to work with when we're creating our different projects. Now in **real world microservices scenario**, you might be working on one solution as part of the entire application. Another **team** might be working on something else. Another team might be working on something else. They may or may not be .NET, so it's **not typical** that you would have all of your projects neatly lined up inside the **same solution file**.



But because we know we're creating .Net type of projects, we're going to do this for our convenience because it's easier to manage. We'll create a **solution file** first of all and we'll say **dotnet new sln** which takes the name of the **folder** and uses that **name** **for the solution**.

If we list the contents we can see that has been created.



Next thing we'll do is we'll create the web API project. So I'm going to say **dotnet new webapi**. And we're going to give this an **output directory**. And we want our services stored inside a source folder. we'll specify **src**. And then we give it the **name of the service** which in this case is going to be auction service and press return.



And this goes ahead and does its thing. And if I list the contents I can see that I've got the source directory.



And if I list the contents of source, then we can see that we've got the auction service now available inside there.

Now, because we are using a **solution file**, we need to **put our project inside that solution** and through the **dotnet CLI**.



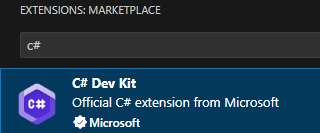
The way that we can do that is by saying **dotnet sln add**, and then we can specify the **name of our** **project** which is auction service and press return.



And with that in place, hopefully you've got your code editor or IDE installed and available. I've got VS Code installed and I can open up the folder that the code is contained in using the shortcut key or command **code**, followed by a **space** followed by a **period**. I'll press return and this should open up the solution inside this folder.

Currently this is like a new installation of VS Code. I don't have any extensions installed, so it's really just a colorful text editor at the moment. And we can see the files that are contained inside this template.

Now we are going to **need some extensions** to make VS code. If you're using that, operate like an IDE.

We need **C-sharp support** for our application. I'm just going to **look for C# in the extensions** marketplace. It includes the **language support for C-sharp** as well. We just need to install this one. This gives us some pretty cool features that I'll explore with you as we build this application.



If you ever need to **give VS Code a kick** – btw - if you take a look at the **background**, we've got an option to show all commands and you can use the shortcut key to open lists. And then if you search for **Developer Reload window**, just use this and it gives VS Code a bit of a kick.

If it does cause you any issues, then that's always the **first troubleshooting step**.

A black background with white text

Description automatically generated

There's one other thing I want to install at this stage. And that's just **material icon theme**. Just to make the icons a bit **better presented** inside Visual Studio Code.

A screenshot of a computer

Description automatically generatedSo I'll install that and I'll select the material icon theme as well. And what we should find now is that we have the **icons available**.

What we also have inside here now that we did not have before, but this dev kit gives us, it is a **solution explorer** similar to how it works in Visual Studio, where we can dive into our solution and project **properties** from this window **rather than just seeing the** **file system**.

If I click on the auction service, I can see the auction service **csproj**, because now this is operating as solution explorer **like it** **would be in** **Visual Studio**.

### 9. Reviewing and simplifying the project

Okay, now that we've got our shiny new project to work with, let's take a look at what's inside it. And we're going to **strip any of the demo code** that's provided with the template away and simplify things and get it down to its bare bones. The minimum we need to run a web API type of project inside dot net.

let's start off at the top and each project comes with its **own project file** and this includes **settings** for the project as well as any **packages** that are installed or any **references** to any other projects.



And it tells us which is the **target framework**. It's going to use dotnet7 in this case, and that's the current stable version of dot net at the time of recording.

And Microsoft releases a new version of the dot net framework or a new major version of the net framework every November. if you're watching this in November, then dotnet8 could well be released. And I've taken a look at what's coming up for dot net eight. And there's no breaking changes between what we're doing here for dotnet 7 and what's coming up in dotnet 8. if dotnet 8 is released, then please feel free to use that.

That will work just fine, but I would not recommend using a preview or pre-release version for what we're doing because that may give you problems later on. I would recommend **just sticking with the** **stable released versions** of dotnet.



Also in here we've got this flag for **Nullable**. Now what is this one doing? Well, **typically** in the dotnet world, **strings are nullable** and if we do not provide a value for a string, that's fine because they are considered **nullable properties**.

But if we go to this example class, this weather forecast, then we can see that they've got a property called summary and it's got the **question mark** here, which means this is **explicitly set to be** **optional**, just like a normal string would be treated. But if we take away the question mark, then we get a warning here saying that the **non nullable property** summary must contain a non null value when exiting constructor.



I'm not a big fan and I prefer strings to operate how they always used to do, whereas we can **declare** **the property without explicitly saying it's optional** because it is just optional. what I like to do and what I would recommend you do is you **disable this flag for nullable**.

And what we'll find then is that inside this weather forecast class now, we don't have the warning and the summary property here is optional, just as it always has been. And that's the way I'm going to build these services and that's **how** **I'm going to** **expect the strings to work**.



The **implicit usings** is helpful. I'm going to leave that enabled. It's useful and it **removes the amount** **of boilerplate** in our code because if we take a look at the weather forecast controller, we have a single using statement at the top here, and that's because we're using something in here that comes from a different namespace in the framework.

Now if we were to **disable** **implicit usings**, for example. Then we'll start to see **errors** in our application because we would **then need to explicitly import the different using statements** for the things that are being used inside this class.

A screen shot of a computer

Description automatically generated

So for example, the IEnumerable, if I click my mouse cursor inside where the error is, then it tells us that we're missing a namespace.



And if we use the **light bulb** on the left, then it tells us this is **coming from System.Collections.Generic**.

Now if we **disable implicit usings**, then we need to **explicitly** add using statements for some of the **commonly used namespaces in the framework**. Obviously that's going to then **increase the** **boilerplate** **in our code** because now we've got extra using statements.

It's a handy new feature that came out with dotnet6 that is very useful.

A blue rectangle with orange text

Description automatically generated

We've also got a couple of packages that Microsoft has deemed necessary to add to the template for the webapi, but I'm actually going to **remove** these because **Swashbuckle** ASP.Net Core is for swagger. It is a useful tool, but it will add noise inside our application and we're going to use a **different tool for testing our API endpoints** and we're not going to use **Swagger** to do so. We're going to use a tool called **Postman**. So I'm going to **remove** the extra noise in the application and any extra boilerplate for code that we're not actually going to be using.

let's work our way down to see what else we have in this project and things that we can change or improve.

If I go into this **properties folder**, we've got some settings to deal with how our application runs. When we either hit the **play button** or we use **Dotnet run** or **Dotnet watch run** from the command line. Now we've got a whole bunch of **different profiles** in here about which API URLs or **which URLs** our application is going to be launched on. If it's going to launch the URL in a browser at the same time when we start our application, we've got a profile for Http and a profile for Https, and we've got one for IIS as well.

We don't need any of this stuff in here. **We need one profile** and it's **helpful** when building a microservices application, to have control over what the **URL** is and the **port** that we start our application on.

A computer screen shot of a code

Description automatically generatedSo I'm going to **remove the iisSettings,** we're not going to use is for this. We want our application to start in a very **specific port,** so we know what it is, and when we create other services, they are also given ports that we can easily remember and use.

We're **not** **going to have a launchUrl**

because we're going to **turn off the browser launching**. We're not creating a web application here. We're creating a web API. So we don't need our browser to test our API.

The application URL I'm going to set to http localhost **7001**.

Please do not use Https for what we're developing at this stage. I'll explain why as we build our application, but we really do want **everything to run over Http** **for the most part**. And as we delve into the architecture, I'll explain why that is and we'll also **take away the Https profile**. let's **remove** that and we'll **take away the IIS express profile** as well.

Please don't worry about the usage of Http. It will become clear later on about how we handle using Https for our application, but we don't need to worry about services that we're creating now and we're just developing these at the moment. So nobody's going to break into our computer unless they're physically here or on our network to go and access this stuff. But it's localhost as well. So this is just running on our machine.

Next up, we'll go to our controllers and we're just going to **delete the weather forecast controller**. We're not going to keep any of the demo code and just hit the delete button and remove that from our application.



Now we've got appsettings.json and appsettings.development.json. These provide **configuration** **settings** for our **project.**

**appsettings.json** is loaded **whatever environment** we're running in, whether we're running in development or production or something else, this configuration file is always loaded and read from.

then we have some **environment specific configuration files**. And this one, **appsettings.development.json** is for **development** and is **only read** from when we're running our application in **development mode**.



Now typically I like to get a bit more information when running in development mode, so I'll swap the warning here for **information**. this is helpful because it gives us **details about the database** **queries** that we're making when we use **entity framework** to make a query to our database.

finally, we get to our program class. we've got a few errors here because we removed Swagger, but we're going to remove this configuration from here anyway.



we've got our **web application creation** at the top here. This gives us a **builder**.

And then we've got a couple of sections. One is to **add services** and the other is to **configure the** **Http request pipeline**.



any **services** we add are going to go in this particular section, although like I said, I'm going to remove the swagger configuration. We need to keep the **AddControllers** because this provides the **services** that we need when creating a **web API controller**



and then we go ahead and **build our application**.

Now, when a request comes into one of our API controllers, it goes through something called an **Http request pipeline**. This gives us the option to **add software or** **middleware**, which can then affect or do something with that **request** in some way **as it comes in** to our API server **and goes out** of our API server.

Typically we'd use this for **authentication and authorization middleware**, we would add so it can **inspect** **the** **request** to see if it meets our authorization requirement.

identity is coming up further down the line in this course. We're not worry about identity for now, but we will remove the swagger configuration inside here.

And because we're running our app over **Http**, we do not need https redirection either. So we'll remove that as well.

A black background with white text

Description automatically generatedAnd whilst we're not touching authorization yet, we will do later. I'm just going to leave that in place as it is.

Then we've got the **middleware to map the controllers**. Each one of our API controllers is going to have a **route** and this middleware allows the framework to **direct the Http request to the correct** **API endpoint**.



And then **finally** we **run** our application.

I'm just going to delete the weather forecast. We do not need that either.

So that can go and now we're in a position to **run our application**

We can open up the terminal and I use the shortcut key to do so.

I like to use the terminal to run the application, but there are other options.

A screenshot of a computer program

Description automatically generatedNow that we've got this Solution Explorer view, we could technically **right click**, say, **debug and** **start without debugging** as an example or **start a new instance with debugging** I presume, but I don't like to use that.

I prefer to just use the **terminal**.

A black background with white text

Description automatically generated

you'll often see me coming into the terminal. I'm just going to cd into the source and the auction service and I'm just going to execute **dotnet watch** and this is going to **start our application** and also **attach a file watcher**.

So it's going to **pick up any changes** **we make into our code** and then either try and **hot reload** our application, which works sometimes, or typically it's going to ask us if we want to restart our application because it's made we've made a change which hot reload does not support.

we've got our **application started**. We can see it's **listening on localhost 7001**. We can see we're **running in development mode** and we can see that we can press control and C to shut down our application and everything looks good.

A screenshot of a computer screen

Description automatically generated

Don't worry about this **warning** **about missing application parts**. We don't have an API controller and we've got the API controller middleware loaded in our application. We're going to be adding controllers soon. As long as you see this, we're good to move forward. And what we'll take a look at next is creating the entity classes we need for our application.

### 10. Adding the entity classes

In this lesson, we're going to take a look at the entities we need to create for our application. Obviously it's based on **auctions**. So we're going to have some **auction type of entities** that we're going to create, and listed in the [**spec**](04%20CourseAssets/CourseAssets/specs/auctionSvcSpec.pdf) for this, we've got our **auction class**, the **item class** with its properties and the status, which is just an enumerable of a few different statuses are auction can be at. And the item is going to be **related** to the auction. we're going to have a **relationship between** **the auction and the item:** a **1 to 1 type of relationship**.



And that's what we're going to create now. So inside VS code inside the auction service I'm just going to add a **new** **folder** and call this **entities**. Sometimes this is referred to as entities. Sometimes it's models. I use entities because we're using **Entity Framework**.

an **entity** represents a **table** inside a database and each of the **entity** **properties** represents a **column** inside that database.

A blue and black box with white text

Description automatically generated

A blue and white rectangle with white text

Description automatically generated

A black box with white text

Description automatically generated

we're going to say add **new file**. And from the **C# dev kit** we then get presented with a number of **templates** to choose from. And I'm going to select the **class**. And then we give our class a name. And the first one is going to be **Auction**.

A screen shot of a computer

Description automatically generatedAnd this **creates the boilerplate** that we need for a very simple **class** inside **dotnet**.

What you typically do is you'd give the namespace the same name as the physical location, even though it doesn't have to be.

I will stick with this actually, because that's what I consider to be normal practice is to just give it the same namespace as the folder name, but it doesn't have to be that way. If you want to keep it simple and just go for auction service, that would be fine as well. But I'm going to update these namespaces to reflect the physical location.

let's go ahead and **create the properties** that we need inside this class. **each entity** needs to have an **Id property.** I'm going to use **guid’s** for this, and it's going to have a guid which is named Id.

Entity Framework is convention based. when we call a field **property Id** then this is what it will use as **default for the primary key of that table**.

And then we'll just create the other properties we need.

A screen shot of a computer program

Description automatically generatedAnd next we'll have a property that's for a string of the seller. And this is optional. We won't give it a default value. But once we have identity, we'll be able to retrieve the information, a username from their user that they authenticated with. Then we'll have a string for the winner, and we'll have an int that's optional for the sold amounts. We'll have another int again optional for the current high bids. We'll have a date time property. This is going to be for the created at. And we'll give this a default property of date time and not. Now we're going to go for **UTC now**. And the reason for UTC is it's a **standard international time format**. And UTC is the same anywhere in the world. And the second reason is the database server we're using is **Postgres**. And that insists that we use UTC values for the date time. We'll also have a date time property for updated at. And this again is going to be initialized to date time and UTC. Now and we'll also have another date time property. And this is going to be for the auction end. And we'll also have a **property for the status** called status that we do not have yet, but we'll add it anyway. And then we'll also have the **related property for the item**. Again we don't have yet but we'll add that in here as well.

A screen shot of a computer program

Description automatically generateda couple more things to create. Go back to Solution Explorer inside entities I'm going to create a new file for the **status**. And I need to say first of all **select enum** and then call it status. And we'll just have a **few statuses** inside here. The auctions can be **live**. They can be **finished** or they can be over, but the reserve price or **reserve not met**. So these are the only statuses are auction could be. And obviously when an auction is live that's when it can accept bids. But when it's finished or reserve has not been met, then the auction is no longer available to accept bids on.

A screen shot of a computer program

Description automatically generatedAnd then we'll create another file. And this one is going to be a **class** and we'll call it **item**. And inside here we'll have **item properties related to a car**. Now the item is going to have its **own table** in the database. So that means it needs an Id property. Again we're going to use a Guid and call it Id. We'll have a prop of string for the make. We'll have a prop of string for the model. We'll have a prop that's an int for the year of manufacture of that car. And we'll have a prop that's a string of color. We'll have an int for the mileage, and we'll also have a string for the image URL.

And we're going to add some **properties to define the relationship between our item and its auction**. Now we've got a definition for the item here.

A screen shot of a computer program

Description automatically generatedEntity framework is **convention based**. So it's going to **recognize** that if we've got a **property called item** then this is **related to the entity called item**.

But if we want to **fully define this relationship**, this 1 to 1 relationship, then we also **need to provide the** **related property from the other way as well**, so that we fully define this 1 to 1 relationship.

Now we can do this **via configuration**, but we can also do this **via convention**.

A screen shot of a computer program

Description automatically generated

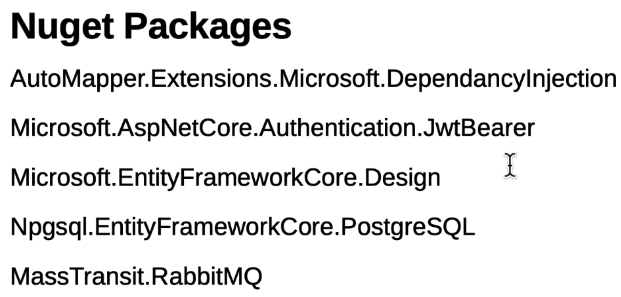
So I'm just going to put in **nav properties**. That's what these are referred to as. And we're going to have a **property for the auction and call it auction**. And we're also going to specify the **Guid**; that is the **AuctionId**. And this is just a **related property or a navigation property**.

So the Entity Framework, when it configures the **schema** for our database based on the **code** that we're writing, it's going to **set up this relationship** as we want it.

So now that we have our entities created, what we're going to take a look at next is creating the **Entity Framework** class that's going to allow us to make a **session** between our **code** and the **database**, and we'll take a look at that next.

### 11. Adding the Database context class

Okay, now that we've got our entities in place, the next thing to do is to create the **Dbcontext** **class** that's going to present a **session** to us so that we can **interact with our database, query it, and update it**.



But we do need to get some extra packages to help us with this. And we need effectively these two the **Microsoft.EntityFrameworkCore.Design**. And because we're going to be using Postgres as our database server, we also need that package as well. Specific to the database server we're going to use. I know you windows guys sometimes prefer to use SQL server, but because we're going to be having a database per service, what I'd really recommend is you stick with what I'm doing on this course, because **Postgres** is kind of a **lightweight** **database**, and it's very **easy** to get up and running with **Docker**, which we're going to be using a lot inside our application. I use Postgres because it's the same thing whether you're on windows, Linux or Mac, and SQL server is - I'm aware there's a Linux version that we can run through Docker as or SQL server - but it's easier if we use something like Postgres. Certainly, when it comes to what we're doing now and for any future deployment we might do. I'm going to be installing these two packages.

And another thing that we'll need in this particular section is **AutoMapper** so that we can shape the data we're returning. I'm just going to go ahead and **install** these three packages.

Now VS Code doesn't come with a NuGet package manager. I'm going to head over to the **extensions**.

A black background with white text

Description automatically generated

Once again search for **NuGet**. And I'm going to install **NuGet Gallery**, which is probably the best one at the moment to install NuGet packages with.



If you open up your terminal, then you've got a **tab** on here for **NuGet** and you can use this to go and **search for packages**. You'll still see me using the older version of NuGet Gallery, but obviously you'll use this much better version of NuGet gallery whenever you need to install a package.

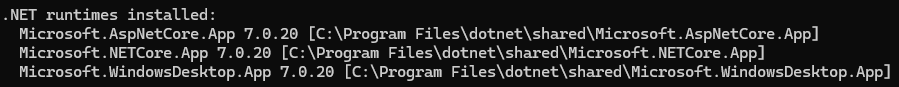
And I'm just going to add the packages we need.

A screenshot of a computer

Description automatically generated

So we need **Microsoft.EntityFrameworkCore.Design**. This one allows us to create **migrations**, which effectively allows Entity Framework to **create the database schema via the code** that we've written.





So we're going to install this a couple of laws around Entity Framework packages, please make sure it **matches the runtime version of DotNet** that you have installed.

It doesn't have to be exact just the **major version**. So as long as it's seven something that's fine. I'm going to go ahead and **install (check version and click the plus icon)** this.

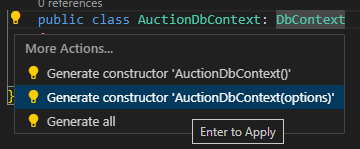


We also need the **Postgres database provider**. I'll just search for Postgres inside here. And we need **Npgsql.EntityFrameworkCore.PostgreSQL**.



And since we're here, I'm just going to install **AutoMapper** just to get ahead of what I know we're going to need. AutoMapper 13+ includes the DI extensions with the main package.

Now that we have these, we can go ahead and we can create our **DbContext class**. So we'll go to Solution Explorer. And inside the AuctionService I'm going to add a **new folder** called **Data**. And this is going to contain the classes that provide us with **data access functionality**. I'll add a new file which is going to be a **class**. And I'm going to call this auction **Dbcontext.cs**.



This class is going to **derive from an Entity framework class** called **DbContext**. And I'm going to click on the light bulb on the left of the class name. I'm going to say **generate constructor** AuctionDbcontext **with options**.

A screen shot of a computer

Description automatically generated

This completes the constructor we need for us to provide the options to the DbContext base class, which we do here.

And when it comes to the **options** that we need for this. We just need to provide it with the **database provider** that we're using and the **connection string**. But we'll do that in the **program** **class**. The **startup** **class** when we get to it.

A screen shot of a computer

Description automatically generated

And if we **hover** over **DbContext**, we can get a description about what this DbContext class is. And this represents a **session with the database** and can be used to **query** **and save instances** **of our entities**. And the Dbcontext is a **combination of the units of work and repository patterns**.

And inside this very simple Crud service*, we're not going to be creating a repository pattern and a unit of work pattern when we don't need to*, because the *DbContext itself is already using those patterns.*

**

Inside this DbContext class, we need to tell it about the entities that we have in our project. We're going to have a **DbSet** as that is what we use to tell Entity Framework. We just need a DbSet for the **Auction**. And we're going to call it Auctions.

As table names in databases are typically **pluralized**, we don't need to specify the item inside here, because our **item is related to the auction**. Entity framework is going to **create the table for the** **Item as well** as the auctions because they're **related**. What it won't do is give it a **pluralized name** for the item.

A black background with white text

Description automatically generatedWe will go back in and go to the Item class. And just above the class name, we're just going to specify **Table** and specify the **name of the table** we want it to be called. And I'm just going to call it items, as conventionally. That's how table names are typically written.

By doing this Entity Framework is going to pick that up and it's going to say, aha, I see you want a **table name called Items** and that's what it will do.

A few more steps to go. We're going to go to now we have a DbContext class. We're going to go across to our **program class**.



By the way **go to file** is the fastest way to locate a class you're looking for inside your project.

You can of course use Solution Explorer and open it up this way, but it does take a bit more time and effort, especially if you've got your solution explorer hidden as I normally would do. It's just way faster to go.

Go to file and then open up the class you want to open. We're going to **add a service** **to our** **container** for the DbContext. This particular class, the DbContext, is something that we're going to want to **inject into classes** that need to get access to something in our database.

We add this **as a service**, and then we use **dependency injection** in order to make use of that service.

A screen shot of a computer code

Description automatically generated

So we're going to add it here and we're going to say **builder dot Services**, and we're going to use **AddDbContext** and we're going to specify the **name** of the **DbContext**. We've created the auction DbContext. And then we're going to specify the **options** here.

And we'll open up the arrow. And then we can specify **opt**. And we want to **UseNpgsql**.

If the auto import didn't work for this one, use the quick fix so that we can bring in the using statement we need. I use the shortcut for that, which in my case is command and period, then we get to see the using statement that we need. You can also use the light bulb and the left. This will give us the quick fix option to bring in using Microsoft EntityFrameworkCore, which is where we get the import for this particular method that we're about to use.

Now this takes a **connection string** as its **argument** and we'll specify **builder dot configuration**. And we've got an option here to go and get the connection string. And we're going to call our connection string **default connection**. And then close off these parentheses with semicolons.

A computer screen with text

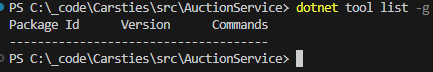
Description automatically generated

We're going to go up to our configuration and get our **connection string**. Which means we need to **define** that **inside our configuration**. Now I mentioned earlier on we've got a couple of configuration files we can use for this purpose. And since we're running in **development mode** we can add our connection string information into **appsettings.Development.json**.

I'm just going to add a comma after the logging. And when I open the quotes one of the options it auto lets us know about is connection strings. And inside the curly brackets I'm going to say **DefaultConnection**. And then we need to supply a **Postgres server connection string** which looks like this. So we're going to say **server** equals localhost. That's what it's going to be running on **port** number **5432**, which is the default port for Postgres, we're going to have a **user ID** that's equal to just **postgres**. That is the default user that's created when we create a Postgres database server. We're going to give it a **password** of postgrespw. And I would highly recommend in development you stick to simple passwords as this is not going out onto the internet; this is just running on your local machine. I don't know how many development hours have been lost by developers typing in wrong passwords because they've given themselves a difficult, complex password to remember. Then following this, we add another semicolon and we give the **database** a **name**. And this is going to be called Auctions.

And now we have the connection string in place. We're in a position now where we can actually go ahead and **create our migration**.

Let's go to the **terminal** once again. And we want to actually stop our application when we do this. And we also need to be inside the **auction service folder**. Now we need a **tool** for this called **dotnet** **ef**.



And you may not have it installed already. But we can check this by saying **dotnet tool list -g** for global tools and press return.

7.0.20 is the runtime version (dotnet --info)

A screen shot of a computer program

Description automatically generated

Once we have that in place, we can then execute the commands to **create a new migration**, which is going to take a look at our code and what we've written - the DbContext class - and it's going to create a migration that will set up the database schema based on the code that we've written.

A black background with white text

Description automatically generated

I'm going to say dotnet **ef migrations add** and I'll just give the migration a name of initial create. And I'm going to output the migration to a folder inside data folder and specify data forward slash migrations and press return.

And as long as there's no problems in our code, then this should work just fine. If you do get any problems at this stage, then please do double check your **connection string**. This is important that the **syntax** is accurate. If you've got a comma instead of a semicolon, then that's going to fail. If you're missing any semicolons, that's going to fail.

And another thing to check is inside the **program class** that your connection string Is free from typos and make sure that you've got this that **matches** exactly what you have inside here.

And the other place to look, of course, is the AuctionDbContext. And make sure that there are no mistakes inside this class. You're deriving from DbContext. You've got the constructor set up properly and you've got your DbSet set up.

A black screen with green text

Description automatically generated

And inside your **Data** **folder** inside **Migrations**. Then you've got **three files** in here. But this is the one we're interested in. And this tells us what is going to happen when we **update** our **database**. It's going to **create** a **table** called **Auctions** with our auction properties and a **table** called **Items** with our item properties.

But of course we do not have a database server at this moment in time. So what we'll take a look at next is **adding Postgres into Docker** so that we can actually use this migration to update a database.

### 12. Adding a Postgres database server

Okay, we need a database server. And the way that we're going to **run our database server** is **via** **Docker**. If you do not have this installed on your computer yet, this is something that you'll need. Docker is going to be a first class citizen in our application and **each service is going to be running as a Docker container**.

But first of all, we need a database server and the fastest, quickest, easiest way to get one up and running is via Docker, because Postgres, like all other database server providers, do provide a **Docker image** that makes it super easy and fast to get one up and running on our machines.

<https://docs.docker.com/get-started/get-docker/>

Now you do need to install Docker and each operating system has a slightly different way of installing it. And if you're on Windows, then please do take a look at the **documentation** and take a look at **download** and install because for Windows you need to take a different step depending on whether you're not you're on a home version of Windows or the professional version of Windows.

So do come in here, take a look and please follow these installation instructions to get this up and running. It should just work fine, but just in case, do take the precaution to take a look at this. Make sure your **operating system meets the requirements**. And if you do need to do something at the **Bios** **level**, then please take a look at this specifically. On a **mac**, it's pretty straightforward. You just need to run the installer. But **Windows**, I do believe you want to take a bit of care about what you're doing here and follow these instructions.

A screenshot of a computer

Description automatically generatedBut once you do have Docker installed and up and running, then you should be able to **open** up the **Docker desktop** and see something like this.

Now inside here, you've got the ability to change a few **settings** and you can control **how much** **resources this is using** on your computer. On **Windows** this is done in a different way.

Let's go and get a Postgres server up and running on our computer. And the way that we're going to approach this is to **use a Docker compose file** that tells Docker what it is we want to run. Inside the solution and in fact, I'm going to go to the file system view here because we want our docker compose in the **root of our solution**. It's going to have more than one service that's running inside as a container.

So we'll do this at the **solution level** and we'll **create a new file**. And this is going to be called **docker-compose.yml**. Otherwise you'll need to write a bit of extra in the command when it comes to running your containers. But if you use this convention then we can just use a very simple command to get things up and running.

A black background with white text

Description automatically generated

Now we've got a prompt here from VS code asking me if I want to install the recommended Docker extension. I am going to say yes because when we're composing these yml files they're very fussy and if we use this **Docker extension** then it will also give us **syntax support** **for the yml files** as well. So that's now installed.

A black background with blue and orange text

Description automatically generatedAnd the way that we write these is that we **specify the services we want to add** and notice that we do get auto completion support inside this file. So I'm going to specify **services:** And then very specifically we need to tab and then we can specify the **name** of the service, what we're going to call it. In this case, I'm just going to call it **postgres**. And then we add another **tab** and then we specify the **image** that we want.

Now if you try and do this at the **wrong level** and I start typing image here, notice that I don't get any auto completion and that's because we have to provide the level that we want. This particular configuration section. If I'm at the **right level**, then I'm going to get the image **auto completion** popping up to say that I can use this at this level. I would recommend you have that **Docker** **extension** installed just to keep you on the right track when you're typing these files because it will identify when you've done something wrong.

Now **below the image** - and the image, by the way, of course is going to be Postgres - we're just using the **standard postgres image** for this,

and then we can supply this with some **environment** **variables**. And the one that we need to provide is the **password** we're going to use. So I'm going to specify an environment variable called PSOTGRES\_PASSWORD, and the password is going to match what we used inside the **appsettings.development.json**, so postgrespw,

and then we specify the **ports** that we want this to run on. Now **Docker networking** provides an **external port** that we can use from our operating system to **connect to the service** inside Docker, and it also has an **internal port** which the **service** is going to be running on.

Now in this case we're going to go for **5432:5432** which means that we access Postgres from the **outside** of Docker on that **port** and from the **inside of Docker on the same port**.

And because this is a **database server**, we need to provide this with some **storage** because it's going to persist our data. And we'll specify that the **volumes** we're going to have **pgdata:/var/lib/postgresql/data**. And this allows Docker to **use external storage to Docker**, our operating system storage **to store the data for our database**.

A screen shot of a computer

Description automatically generated

And then below this we're going to specify **volumes** and **pgdata**. And importantly, we need a **colon** after pgdata as well.

So with this Docker compose file in place now we can run a Postgres server. I'm just going to open up the **terminal** once again. I'm going to go to a new tab and we want to be at the **solution level** where our **Docker compose file** is located.

A screenshot of a computer

Description automatically generated

If I list the contents, then we can see we have this **particular file**.

Now in order *to run the services inside Docker from this Docker compose file*, we just need to **run** **docker compose up -d**, so it runs in **detached mode** so we don't see the logs inside the terminal and press return.

A screen shot of a computer screen

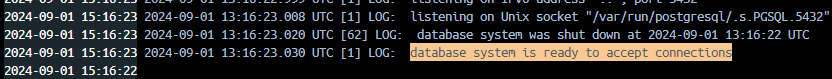
Description automatically generated

And the first time we run this, it's going to go out to Docker repository and pull down the image from Postgres and start it. It might take a bit longer on your machine if you've not done this with this particular image before. But once you've done that, we should have the fact that it is **started**.

A screenshot of a computer

Description automatically generated

And if we go take a look at the **Docker desktop**, then we can see that we've got our **Postgres image**. It's running on ports 5432.



And if we **click on the image**, we can go and **see its logs** and we want to see that the **database** **system is ready to accept connections**. And that means Docker is running and it started successfully.

****

Now if we go back to VS code, we can execute **dotnet ef database update** and press return.

And what we should find is that our **table gets created inside the Postgres database server** and indeed we've got the create table action taking place and the create table items taking place.

|  |
| --- |
| check environment variables in container:  docker exec -it <container\_id> env  verify database connectivity from Inside the Container:  docker exec -it <container\_id> psql -U postgres  when getting this command without any auth errors, you’re connected    review the logs from the PostgreSQL container  docker logs <container\_id>  when connection errors remain while doing the first ef update, this can be due to another postgres service running on the same pc, so kill the process and try again, this solved the issue for me |

Now, if we want to see what's inside our database, we can do so. But that does mean adding another **extension**.

A blue background with white text

Description automatically generated

I'll just go to the extensions and I'm going to search for Postgres. And the one that is quite good is this one called PostgreSQL by Chris Coleman. And I'm going to go ahead and **install** this one.

A black screen with white text

Description automatically generatedAnd this puts a **little tab** on the left hand side with the **Postgres SQL Explorer**. And if I click on this, then we've got the option to add a **new connection**.

So I'm just going to create a **new connection** and we specify the **host name** of the database that's running on **localhost**. The Postgres SQL **user** is going to be **postgres**. The password is what we specified. And the **port number** is going to be 5432. Do we want to use an **SSL** connection? **No**, we want the **standard connection** and we want to specify auctions as the name of the database and then the **display name** of the database. I'm just going to keep as **localhost**.

A screenshot of a black screen

Description automatically generated

And now we've got a connection to our database server. And if I go in I can see the **Auctions** **database** has been created inside **public**. We can see the **tables** and we've got a table for **Auctions** with all of its columns and same for the **Items**.

A screenshot of a computer

Description automatically generated

And we've got an extra **table for** **migrations** **history**, which Entity Framework uses to check which migrations have been applied already to list database. And if we select the top 1000, then we'll see in the results here that we have an entry for the migration that we created and the version number of the dotnet **runtime** that it was created on.

So that looks good. But we've got an **empty database** for our auctions and our items and we'll take a look at addressing that next.

### 13. Creating seed data

Now that we have the database up and running or the database server up and running, let's go and **create some seed data** so that we have something to work with and we'll populate these tables.

So let's go ahead and I don't need to save any of those queries, but let's go ahead to Solution Explorer inside the auction service, inside the **data** folder. Let's create a new file. If you're using that C# dev kit and then inside the data folder, we'll get the option to add a new file and we can pick a **class** and this one is going to be called **DbInitializer**.

And this is what we'll use to seed our data. Inside this class we're going to create a static method. I'm going to say public static void. We're not returning anything from this and I'm going to say **InitDb** and we're going to pass it as an argument the **web application** and specify that name as **app**. We're using a **static method** here because this allows us to use this method inside another class without initializing a new class of DB initializer.

A screen shot of a computer

Description automatically generated

We need to get access to our **DbContext** in order to add data to it. And where we're going to use this DbInitializer class is inside the **startup of our application**. In fact, **before** the application even **runs**, we're going to be using it here (before app.Run()).

We need to get hold of our service, the **DbContextService**. But before we've got the ability to actually **inject this service** **into a class**, once the framework has got the application up and running and the way that we'll need to do that is we'll need to **create a scope for the service** inside here itself.

We're going to say **using var scope = app.Services.CreateScope()** and we add the **using keyword** here because this will allow the framework once we're finished with this method to **dispose of any** **of the services** **we've used inside this scope** that we're creating, and then we'll call another method that we don't have yet called **SeedData**.

And we're going to pass this as an argument **scope.ServiceProvider**, and we're going to say **.GetService** and we want to get hold of our **AuctionDbContext** and add the closing parentheses there and then we can put our cursor inside the SeedData, use the quick fix and say **generate** **method** seed data and this goes ahead and scaffolds the method that we're going to use to actually seed the data into our database.





I'll just change auctionDbContext to **context** in this case.



So now we have our context. We can do things in code to do something with the database. So I'm going to say first of all that we're going to use context dot database and we're going to use the **Migrate method.** And if we take a look at the **description** of the Migrate method, this is going to **apply any pending migrations for the context to the database** and we'll **create the database if it does not already exist**.

It's going to take a look at our migrations, apply any that have not already been applied. And if the database has not even been created at this point, it's actually going to create a database as well. All inside this single command.

A black screen with white text

Description automatically generated

Following this, we're going to attempt to seed some data in, but we're going to **check**, first of all, to see if we already have some **auctions** in our database. We're going to say if context auctions dot any. Then we're simply going to return from this method. But we'll add a console dot write line here as well, just for debugging purposes and say that we already have data, no need to seed. And then we return after that.

A screen shot of a computer program

Description automatically generated

But if we do not have any auctions in our database, then we're going to **seed some auctions**. And the way that we can do that is specify var auctions = new list of type Auction. And then inside here we can go ahead and create a **new auction** and start filling out the properties for each auction. But that would be incredibly tedious. So I'm just going to say auctions inside a comment there for now and close that off. And then I'm going to say context.AddRange(auctions) will **populate in memory**. Entity framework is going to keep the auctions that we add in memory when we use the add range.

And then if we actually want to update our database, we can use the **context.SaveChanges** and we just need to fill in the gap here.

So we've got some auctions to work with. Now it's quite tedious to write out this kind of seed data. So inside the course assets folder, inside **snippets**, you should have an auctionsSeed.txt and inside here is just a bunch of auctions that we're going to add for cars of course, because we're a car auction site, so I'm just going to control a control C, go back to the DbInitializer class and overwrite this comment with the list of auctions from the clipboard. And there's ten auctions in here for various different cars of various different prices with various different auction end dates and times, and there should be ten of them. So if we scroll back down, just make sure there's no errors in this class and that looks fine.

A screen shot of a computer program

Description automatically generatedAnd then we can head over to our **program class** to actually make use of this. So inside here, just **below the MapControllers statement**, we'll add a try catch block. And we'll specify **DbInitializer** and we'll use the **InitDb** method and we'll pass it the **app** as a parameter. And if we do get an error. And that's just take out system exception and change that to exception e. Then all we'll do with the error is just simply **log it out to the console**. With that in place, we should now be able to **populate** **our database** with some data.

A black screen with white text

Description automatically generated

Let's open up the console. We'll need to stop our application from running because first of all, I want to **drop the database** that we already have just so that we can confirm that when we do restart our application, it's going to create a database and seed the data. So to drop the database, we can use **dotnet ef database drop**. And we should get a confirmation asking if we're sure and I'm going to say yes. And that's our database gone.

What we should be able to do now is just type in **dotnet watch** and we should see activity to populate our database with the values that were provided in the DB initializer class. And we can see lots of **insert into statements** there, which suggests that has worked successfully.

A screenshot of a computer

Description automatically generated

And we can further confirm this if we go to the Postgres Explorer, let's just go up to the auctions and **refresh** items. And if I say select and select top 1000, then what we should see is a **list of auctions** which we can inside there.

A black screen with white text

Description automatically generated

And if we take a look at the **Items** and select the top 1000, then we can see that this table has also been **populated** with the details of the **cars** as well.

Now that we have data in our database, we can start to think about how we're going to return this data to the client. And in the next lesson, we're going to take a look at what we're using to **shape our** **data**, which is **DTOs** **and** **AutoMapper**.

bij opnieuw runnen (om postgres container te starten):

docker compose up -d

### 14. Shaping the data to return

Now that we've got the data in the database, we don't want to return our data like this. We'd rather **return a single class**. That has all the **properties flattened for the auction** **and the item** as well.

Let's just clear up the stuff at the top of this screen. And if we head over to the [spec](04%20CourseAssets/CourseAssets/specs/auctionSvcSpec.pdf) sheets, let's take a look at what we are supposed to return from our endpoints in terms of an auction, DTO and effectively this is just the auction and the item mostly **combined**, but returning things like the **status** **as a string** instead of the enum value, which is going to be an integer. And what else do we have? Okay, that looks fine, but it's really just an object that's been **flattened into a single object** **from two objects**. And that's how we're going to **shape our data to return**.

And we're going to use **DTOs** to do that. So **data transfer object** so that we can go from our auction and item entity into an auction DTO. And inside the auction service, we'll create a **new folder** and call it **DTOs**. And inside the DTOs we're going to add a **new file** that's going to be a class called **AuctionDTO**.

A screen shot of a computer program

Description automatically generatedAnd inside here we're effectively going to have a **combination** of the auction and item entities.

So I'm just going to go to the **auction** class. First of all, I'm just going to copy all of the properties apart from the item into my clipboard and paste them into the auction DTO. And the status here. We're going to swap from the **status** enum to a **string** as we want to return the value of the enum, not the database value, which was just an integer. And let's see, we can **remove the optional** properties here as well as this is just a thing that we're returning and we'll **remove the default values** from this as well. We don't want default values in our DTOs. A DTO really just should be a collection of properties effectively.

And we'll just go to our **item** and we don't need the ID of the item, but we'll copy the **properties of the car** and we'll also go to our auction DTO and paste those in as well.

And that looks good also so that we can **create an auction**. We'll need the properties that we require from the user when they attempt to create an auction, and we'll do that via a DTO. So we'll create another new **class** and we'll call this one **CreateAuctionDto.** And inside here we're going to need several properties, car related properties.

A screen shot of a computer program

Description automatically generatedSo I'm just going to go to the auction DTO and copy the **properties that we're going to take from the user**, which is going to be the make model, the year, the color, the mileage, the image URL. And we'll put that inside here. And we also need to get from them what they want the reserve price to be, the minimum amounts that their car could be sold for. So we'll specify **reserve price** and we'll also have the **date time** **for the auction**.

And now we want to make **all of these properties required**. We don't want auctions in our database that do not have any of these properties and we'll make them explicitly states if they don't want to reserve price, we want them to specify that's going to be zero.

So if somebody puts in $1 for their bid, then that's what the price would have to be sold at. So it's down to the user what they want for this, whether or not they want the reserve price. But we are going to make all of these fields **required** in here and we can use **data** **annotations** to do so simply by specifying required in square brackets against each of these individual properties. And I'll just do this for all of these.

A screen shot of a computer

Description automatically generatedWe'll create one more DTO inside here if they want to **update an auction**. So we'll create another new file inside the DTOs folder, another class, and we'll call this one **UpdateAuctionDto**, and we'll just take the properties that we need for the car, just the make model year, color and mileage. We won't let them adjust the image URL, just five properties in this case. And we'll get to our update auction data and paste those in there.

Now we've got the dto's in place. We need some **mapping profiles for AutoMapper** so we can automatically map us from the auction to the auction DTO. Now AutoMapper is a great tool. It takes a look at the property names. So for example, if we have a property in here called ReservePrice in the entity and we want AutoMapper to automatically map something into the auctionDto so long as it's got the **same name** as the property, then AutoMapper will take care of this mapping for us.

So it's a really useful tool and we'll set this up and configure it so it does exactly that. Inside the auction service, I'm going to create another **new folder** called **RequestHelpers**. And inside that folder, I'm going to create another new class and add new file that's going to be a class and call it **MappingProfiles**. And inside this class we need to **derive from AutoMapper**.

And we're going to use the profile from AutoMapper that we're going to derive from. So we've got AutoMapper there. And the namespace, I'm just going to update this as well to request helpers. And inside this class that we derived from profile, we need to create an empty constructor.

A screen shot of a computer program

Description automatically generated

I'll do that and remove the parameters. We don't need parameters for this, but then we can go ahead and create our mapping profiles and tell AutoMapper what we want it to map from and what we want it to map to.

We're going to have a **create map going from the auction to the auctionDto**. But we also want it to **map the item** as well. So after this, we'll add parentheses and we're going to tell it to **include** **members**. And then we're going to say X goes to X dot item.

We also need a mapping **from the item to the auctionDto** as well. So we'll create another map below this one, but this time going from item to auction DTO and we don't need to include members for that one.

We'll also create a mapping going **from the CreateAuctionDto to the auction**. And because we've got **car properties** in here, we'll need to go from item properties.

And because we're also going **from CreateAuctionDto to the item** as well, we just need a mapping to take care of that. So we'll add another mapping here.

And one more bit of configuration we need to do for AutoMapper before we move forward is to **provide this as a service inside our program class**. So let's open up the **Program.cs** and just underneath our **services** that we already have, we'll just say builder dot services and we'll say add AutoMapper.



And then we specify the location of where our mapping profiles are. And typically nowadays I just use app domain current domain and say get assemblies which provides the location of effectively the assembly that this application is running in. It's going to take a look for any classes that derive from the profile class and register the mappings in memory so that when it comes to using AutoMapper, it's already set up and good to go. So now we're all set up. What we'll take a look at next is creating a web API controller so we can start returning data that we retrieve from our database.

### 15. Adding an API controller

We spend a bit of time setting up to get to this point, but now we're ready to **create a web API** **controller** so that we can **start returning some data to a client**.

Inside Solution Explorer, let's go to our **controllers folder** and we'll add new file and we'll create a class and we'll call this **AuctionsController**. This particular class is going to **derive from** a framework class called **ControllerBase**.



And if we take a look at the **description** of this class, this provides or this is a base class for an MVC controller without view support. And this is a controller that's **just going to contain API endpoints**. It's not going to return a view to a client. We've got a client app we'll build later on that's going to take care of the View. We're only concerned with **returning the data in Json format to the client** from these API endpoints.

A screen shot of a computer screen

Description automatically generated

A couple more attributes that we need inside an API controller. One of them is the **ApiController** **attribute**. This gives our controller some extra abilities that makes our lives as developers a little bit easier, especially when it comes to **data validation**. When we create an auction, we've got those required properties. This is going to **check those required** properties and return a **bad request** if it fails validation. And it also allows us to **bind to properties** that are sent up as arguments to our API endpoints as well.

We also need a **route** so that the framework knows **where to direct the Http request** when it comes into our service. So we're going to have a route and we'll open parentheses and we're going to say the route is api/auctions and then we're going to **add a constructor**.

And this is where we get to use **dependency injection**. I mentioned earlier on that inside our program class, the services that we're creating, such as the DbContext and AutoMapper in this case. When we want to make use of these services, we can **inject** **them into classes** in our application and we need to use our **DbContext** so that we can **access the data** and we need to access to **AutoMapper** so that we can **shape the data** and **automatically map it** from the auction entity that we get from the database to the auction DTO, which is what we want to return to the client.

So inside the auctions controller, let's take a look. We're going to add the auction dbcontext and call it context, and we're going to add the IMapper interface that we get from Automapper and we'll call it mapper.

A screenshot of a computer

Description automatically generated

And inside here, if we just put our cursor inside the context and use the quick fix and we can use the **create and design field context** and this populates a field at the top, a **read only field** that we can then use in the rest of the class after this has been injected.

And the way that **dependency injection** works is that when our framework creates a new instance of the auctions controller, which it will do when it receives a request into this particular route, then it's going to take a look at the arguments inside the controller and it's going to say, I see you want a dbcontext and a mappern and it's going to **instantiate these classes and make them available** inside here.



Now, the only thing is I don't particularly like this **syntax** with how that VS code has decided to add this particular field. Typically most developers nowadays, and it's not a law, it's just a convention, will typically use an **underscore when defining a private read only field** and instead of using this.context, they'll specify, this.\_context.

But I don't want to do that manually every time. I inject something into a class because I'm going to be doing that quite a lot. I'd like that to happen **automatically**. Now there was an extension, I used to add that provided this functionality, but I don't want to add that extension just for this very simple task of defining an underscore. So we'll approach it a slightly different way. And one of the things we can do with VS code is provide it with some **editor configuration** that allows us to **define tasks** **like this** so that it takes care of doing that **adjustment** of swapping this from an underscore or to an underscore automatically.

A screenshot of a computer program

Description automatically generated

So we'll do that and inside not solution explorer but the files inside the root of the file of our solution. I'm just going to create a new file and it has to be called **.editorconfig**. And inside here, if we just go back across to the **snippets**, there's an editorconfig.txt which contains the configuration to swap or change that behavior. So it does use an **underscore for private fields**. So just copy all of this code and paste it inside this editor config file.



And what should happen now is if I just undo what I did before and map it back in. But now if I use the quick fix to do what I did, which is to create an assign field, we've got an **option** to use underscore context now. We do the same for the mapper parameter.

Now that we have that, let's **create a couple of endpoints** so that we can test our progress.

A screen shot of a computer code

Description automatically generatedWe'll create an **HttpGet** because this is going to be a **get request** so that we can go and get our data from the database and we'll say public async task. And what we're returning from this is an **action result** and an action result lets us send back **Http responses** such as a **200** or a **404** not found.

And then inside the action result, we're going to specify that we want to return a type of **list** which is going to be a type of **auction DTO**. And we'll just call this method get all auctions that takes no arguments and we'll specify var auctions. Equals await and context auctions. And we want to **include or eagerly load our related properties** here. And the related property is the item. So we're going to specify **include**. Let's include it along with the auction. I'm going to specify some **ordering** just to give them some kind of order and I'm going to order by. I'll specify to list async. And then to return. I'm just going to say we want to return and we'll use Automapper and we're going to say we want to map to a list of auction DTO. And get that from the auctions.

A screen shot of a computer code

Description automatically generatedSo that takes care of the list of items and we'll just **create another endpoint** that we can use to **get** **an** **individual** **auction**. So we'll specify a route parameter here of ID inside curly brackets and we'll say public async Task of action results. That's going to return an auction DTO. And we'll call this one get auction by ID. And this is going to take a guid of the ID and this property name needs to match the property name that we specified inside square brackets in our route parameter here.

And for this one we'll just specify var auction equals await and context auctions. And we'll say include item. And we'll say first or default async and we'll just go and get the first item that matches the ID.

So we'll say if auction = null, then we're going to return Not found. But if we do have the auction, then we're going to return and say mapper dot map and say auction DTO. And pass in the auction. And now we've got these two end points.

We can go **check to see if this is working**. But first of all, because we've added a new controller and made some changes, it's highly unlikely that hot reloading is going to be able to work with that. So if you check your terminal where you're running your app or just simply **restart your app**, then you'll probably see something like this. And basically it says, Hey, you've made a change that I can't cope with. Do you want to restart the app? And I'm going to say yes, of course. So the app restarts and now we can go test that what we've done is working.

And to do that, the tool that we're going to use for this training course is going to be **Postman**. So please load up Postman And what I'd suggest is just to keep things tidy as you create a **new** **workspace** and I'm going to call this workspace **Carsties** and make it **personal** and create workspace.

And then we can go ahead and create a **new collection**. And I'm going to supply a collection soon. So I'm just going to call this one scratch just for playing around and inside here. I'm just going to **create a new request**.

And to say this request name is get auctions. And inside here we're going to specify **http://localhost:7001/api/auctions**, which is the address of our API endpoint. And if we click **send**, what we should find is that we do **get the list of auctions back** from our API and this all looks good and formatted based on what we've done inside AutoMapper. We've got the **item details** along with the auction details as well and everything looks good there.

What we should also be able to do is **take one of these items** and take the ID. And we'll just add a forward slash and press return and we get the single auction based on its ID as well, which looks good. But typing out all of these requests in Postman can become tedious and annoying and to make life easier to check our progress as we go along, we'll take a look at a **postman collection** that I've used and provided to make it easier in this training course. And we'll take a look at that next.

### 16. Adding a postman collection

To make life a lot easier as we go about the process of building this application and the services, then **Postman** is the tool that we'll be using to **test our progress** as we go along. It's an excellent tool and what I want to do in this lesson is to show you the **postman collection** that I've created to make that process of testing our app as we go along easier, and just show you a couple of **features** in Postman that I've used to help us out here as well.

So we want to **import the collection** that I've created. So inside Postman, if we go to **file** and **import**, then we can drop anywhere to import and I'll just select the file. Inside the course assets inside postman that I'm going to import this Carsties postman collection json. Once that has imported we should see the collection inside here with folders for the different requests that will be using in different sections throughout this course.

A screenshot of a computer

Description automatically generatedNow for section two, for example, let's just take a look at a couple of requests that we can execute right now. What we have here is a request to get all auctions, but Postman lets us **use variables** in place of hard typing the URL that we're going to request from. So in this case we can see the variable is http localhost 7001, which matches the URL of our API service or our auction service.

A screenshot of a computer

Description automatically generatedAnd where's this information coming from? Well, if we click on Carsties at the **collection** level, we can see we have a number of **variables** inside here for the different services that we will be creating as we build this application. And one of them, the one I'm using here is auction API and that's where it's getting those particular values from. So if we go ahead and execute this request and we click send, then we can see that we get the list of auctions back. Of course we do, because that's the same request that we made previously. But what we also have inside here is **for certain requests**, I've made **tests** so that we can see we're **getting the expected results** as we go along.

A screenshot of a test results

Description automatically generatedAnd if we take a look at the **test results**, we can see that there was three tests inside here. Status code is 200, there should be ten items and the first item should be Audi R8, because we ordered our list of items inside our API.







And if I was to make a change to the ordering, for example, and instead of item dot make, I said let's order it by item dot model, then I'd expect that test to fail. And if I give that another go and I click send this time we get a failure because the first item should be Audi R8, but instead we got a Ferrari as the first item in the list.And if we take a look at the body, then the first item returned in this case is a Ferrari because this model happens to be first in the list.

A screen shot of a computer program

Description automatically generatedBut if we take a look at the test itself. What we're doing here is **postman tests**. First of all, we get a look at the **data** that's being returned from Postman via this line. We can get the postman response and get the **Json** **value** of that response and that's stored as Json data. We can check the **status code** that's being returned in this case is 200 and that does match and that test did pass. We've got a test to **check the number of items** equals ten. Obviously, if we do add another auction to our items, then that test will fail. So they're quite fragile. Tests we've got in here that are going to be used at various times. But as we **add more auctions**, then of course **this test will fail** because it's kind of fragile this particular test, and then we've got the test of the **first item** should be Audi, R8.

This one is what we're failing on currently because it's not equal in Audi or R8 as the first car that's being returned. So we can fix that test, of course, by rearranging what we're sending back to be the item make instead of model in terms of the ordering. And if I click send, then the test now passes.

So a very small taster of what's going on inside that particular request, just to show you that as e're checking things as we go along, you've got some reassurance that you're doing things in the same way I am. Of course, if you use different seed data, you've added more items, then you can expect these tests to fail and you would need to adjust them if you wanted to so that they do pass.

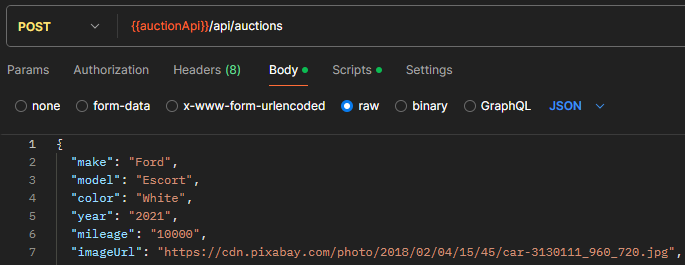
If we take a look at the next request, this is just to get a request of a specific car. The seed data did provide IDs for all of these. So using this, we can just send this request and we get a passing test saying, yep, this is a Ferrari spider. And that's because the test itself, if we take a look, is checking for those properties that we get back from the Json response in Postman.

A black background with white text

Description automatically generated

Another thing that's really useful about Postman, if you're wondering what's going on inside these variables, if you **take a look at the console**, you can get the full **details of the request** that's going to the API. So if you're **troubleshooting or debugging** and you're not sure what this is or where the request has actually gone or you want more details about the response, **expand** and you can see the **request** **headers** inside here, the **request** **response** **headers**, and also the **response** **body** as well, what's coming back from the API.

So this tool will use extensively throughout this course because this is an incredibly useful tool to **test our API as we build it**. When we move on to things like creating a new auction, then there's other tools inside Postman that I've used to help us with this.



Things like inside the **body** of the thing that we're going to send up to create the auction, then the date has to be specified in a certain format. So inside here is a variable called datestring.

A black background with white text

Description automatically generated

And if we take a look at the Pre-request script, what's going to happen before the request actually goes to our API? Then we've got the opportunity to do something. And in this case I just want to **set** **a date that's based on today's date plus 14 days** and then use that as the date for the end of the auction.

Also in here we've got some tests as well which is doing pretty similar things to what the previous ones did. But also what we've got inside here is a **collection variable set method** so that from the **created ID**, which we do not know at the time of creating the auction, then we're going to set that as a variable so that we can use that in future requests where we make use of the created item ID there. So after we've created an auction, we'll be able to update that created auction, etcetera.

So hopefully you'll find this postman collection useful. Please do import it into your version of Postman because we'll make heavy use of this as we go and build our application.



Also, what we can do inside Postman that's pretty cool is if I click on the **folder level**, then I've got the ability to **run all of these queries at the same time** and **run the tests at the same time**. So we're kind of use these as checkpoints sometimes to make sure that we've done everything we're supposed to do for this particular service.

### 17. Adding the create auction endpoint

Let's work on the endpoint for this particular request, which is going to make a **post request** to our auction API url. And it's going to accept the properties that we send up inside what effectively is the create auction DTO that we added earlier? So let's head back to VS code and implement this functionality. So we need an **Http post type of request** for this. And we **don't need to specify any** **parameters** for this because our framework is smart enough to realize that if a post request comes in to list routes, then - as long as we've only got one http post request at this route - then this is the method that's going to be executed.

A screen shot of a computer program

Description automatically generated

So we create a new post method CreateAuction and we do need some parameters for this. We need a CreateAuctionDto and I'll just call it auctionDto and the first thing we need to do here is **map** the auction or create auction **DTO** into an **auction entity** so we can use **Automapper** for that.

Now, at this point, what we would want to do is **check the username of the user that's creating** this auction, because we wouldn't want anonymous users to be able to create an auction. But we haven't added identity yet, so we're not working with usernames and authentication yet. So I'm just going to **add a todo** and say add current user as seller. But in the meantime, just so that we have something I'm going to say auction seller = “test” as a placeholder for the time being.

Following this, we can then add the auction using entity framework. What's happening here is entity framework is effectively **tracking this in memory**. So nothing's been saved to the database at this point. This is simply being added to **memory** and entity framework is **tracking** this because it is an **entity**.

Following this, we can actually **save** this to the database, with context and save changes async and we'll say greater than zero because this save changes async method **returns an integer for each change it was able to save in the database**. If it returns zero, that means nothing was saved into our database and we know our result is going to be false. But if the changes were more than zero, then we can presume that was successful and this will evaluate to true.

So we'll check the results and if we have no results we'll simply return a bad request. And we'll say could not save changes to the DB. And then one result there.

And what we want to **return** from a post request is what we should return is an **Http** with a **status** code of **201 created** to say that we've **created a resource** and we also need to tell the client **where** the resource was created at.

So we have a method inside here that we can use to do such a thing. And what we'll return is a **CreatedAtAction**. And we can specify the name of the action where this resource can be found. So in this case, we've got a method here called Get Auction by ID, and this is the **location** we would want to send back in the header to tell the client, we've created your resource and this is the **location** you can **get** your **resource**.

So this particular endpoint is where we'd want them to know about if they wanted to get the resource that's been created. So we can specify a name of and **GetAuctionById**. And this particular method, this takes an **argument** of the guid of the auction so we can specify as a **second** **parameter** in the created at action, we can specify **new** and then we can simply specify the auction ID as the **parameter** that's needed for this particular action or endpoint. And then as a **third parameter**, we can return the auction **data**.

So in order to return an auction data from this, we need to go **from** our auction **entity** **into** an auction **DTO**, so we'll utilize **mapper** **functionality** for this.

So that's our endpoint for creating an auction.

And now we can go **test in Postman**. So let's go take a look. In fact, let's just open up the terminal and see what's going on inside here. **Hot reload** makes a bit of a mess inside your console because when I forgot to add a semicolon, it prints out this nasty warning. But it does say at the end there hot reload of changes did succeed.

So let's give that a go and we'll head over to Postman and we'll see if we can actually create an auction from this particular request. So we've got our make the model, the color, the year, the mileage. And if I just to test the validation is working correctly, if I remove one of these properties, the color for instance, that is a required property.

A black background with white text

Description automatically generated

And if we do not provide a color, what we should expect to find is that based on what we have inside our create auction **DTO** if I miss the color which is a **required** property, what we should see is a **bad request**. And the reason that we should see it's a bad request is because of what we've done in the auctions controller. When we've got the API controller. This is responsible for **checking** **validation** and if it doesn't meet the validation requirements, then it's not even going to let the request get to this endpoint. It's going to the framework is going to **reject the request before it gets to the endpoint** that we're asking for. So let's see if that functionality works.

A screenshot of a computer

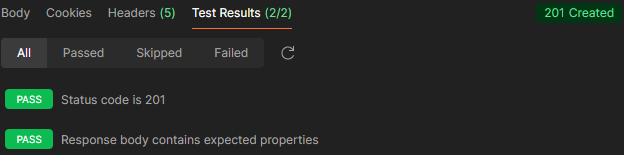
Description automatically generated

First of all, as a simple test. So I've removed the color and if I click send, then what we get is a **400** **bad request** and it tells us that the **color field is required**.

So that functionality is working, which is good. And if I put the color back in again, then this time we should meet the validation requirements and the auction should be created and everything should be okay. So let's click Send, we get a successful response, we get the auction back from the endpoint or the auction DTO, we can see that we've got the status of **201 created**.



And if we take a look at the headers because it is a 201 Created, what we also get is a **location** **header** in the **response** that tells us the **location** of what has been created. And we can see that there in this particular **response**, which is good.



And we can also see the **test results** are **passing** with the status code of 201. And the response body contains the expected properties.

pm.test("Response body contains expected properties", function () {

    pm.expect(jsonData).to.haveOwnProperty('id');

    pm.expect(jsonData).to.haveOwnProperty('make');

    pm.expect(jsonData).to.haveOwnProperty('model');

    pm.expect(jsonData).to.haveOwnProperty('year');

    pm.expect(jsonData).to.haveOwnProperty('imageUrl');

    pm.expect(jsonData).to.haveOwnProperty('seller');

    pm.expect(jsonData.status).to.eq('Live');

});

So if you use **different property names** for the auction, then please do take a look in here and take a look at what's **expected**. And if you did want to use different property names, that's fine, of course, but you would need to **update the test to reflect the properties** **that you're expected** to get back or simply remove the test that contains these properties.

Either way is fine, but it's only a postman test and it's not going to affect your application anyway. But now we have the ability to create a new auction.

Our next task is to take a look at updating the created auction or any auction, and we'll take a look at that next.

### 18. Adding the update auction endpoint

Our next task is to deal with the **updating of an auction**. So let's head back to our API controller and we're going to **add an endpoint** to deal with this, which is going to be an **Http put endpoint**. To **update** a **resource**.

And this one does require us or we need to **supply the ID as a route parameter** here. So we'll specify ID inside the route parameters for this particular one. So once again, we need a public async task action result. We're **not going to** **return** anything from this in terms of what we're updating. And the reason is that the **client should already know** what they're updating. So we don't need the API to return what has been updated.

So we can just **return an action result without any type parameter** here and we'll specify the name of this endpoint as update auction and as a route parameter. We're going to take the guid as the ID and we're also going to take in the update auction DTO. And just call this update auction DTO.

A screen shot of a computer program

Description automatically generated

So in an update request, then we need to go and **get the auction**, first of all, **from the database** that **matches** this particular **ID**. We need to include the items because we're updating the car properties in this. So we're going to include the item. And then we'll say first or default async that matches that particular ID.

So then we're going to check effectively to make sure that we have the auction.

Now, if we do have the auction at this point, we would want to **check that the seller name** matches the username that's updating the auction. But of course, we do not have identity available to us yet. So we're just going to add a **to do** here and say check seller equals username, but we can't do anything about that yet, so we won't do anything about that yet.

From here, all we want to do is **update the properties** to the updated property in the update auction DTO or if that's not provided, we want to keep the original property of the entity.

So we're going to say the auction item dot make is equal to the update auction DTO make and if that's null or undefined, then we're going to use the **null conditional operator double question mark** and we're going to set the auction item dot, make to what it was inside the entity.

And we're going to just copy this down. We're not going to use mapping for lists because there's only five properties we're allowing them to update. We need to make the Year and the Mileage **int** **optional** for the null **conditional** operator.

Whether or not you would actually want somebody to be able to update these things in a real application is debatable on an auction site because once somebody started bidding on something and then the seller starts changing properties of that item, that's kind of dodgy ground that you're on then because somebody put a bid in for something that's then changed after the bid has been accepted. So you probably wouldn't do this, but I'm really just providing an example of a normal Crud application at this stage of the training course. Whether or not we use this functionality later is debatable, but for now we're just going to put it in as it is.

So now we've updated our auction which is being tracked in memory by entity framework at this point and we've updated its properties. We don't actually need to do anything with our context to say that this item has been updated. We can simply use or simply **save the changes** at this point because this is being tracked in memory.

A screenshot of a computer

Description automatically generated

Let's just check that the restart has taken place and that looks okay. And if we go to **Postman** and we've got the created item ID which is based on because of the test inside this post request, this represents the auction that we just created. And if we go back to that request and take a look at its body of the response, then we can see this is the ID that was created.



And if we take a look at the **update auction**, then that should match the ID property that's inside there. And then in the body of this, all we've got is a make to say Ford updated and we've got a test that reflects well just to say that the response was okay in this case.

A black background with white text

Description automatically generated



So if we click send, what we should get is a 200. And we do we've got another request here to get the **updated auction** again with the same ID,



and this one just has a **test** to make sure that the make includes updated in the text of the make of that car. And if we click send on this one, we should get the test result saying that is what it is. And if we take a look we can see the maker's Ford updated there. So almost there. We're just going to create another endpoint for deleting an auction. And we'll take a look at that next.

### 19. Adding the delete auction endpoint

Okay, one more endpoint to go, which is the delete endpoint. Once again, I'd say that this probably wouldn't make it into the final product because if we allow users to create auctions and they're immediately set to live, which is how we're approaching the design of this app, and then they forget to set a reserve price and somebody comes along, bids $1 for their car, then allowing them to **delete it once those bids are already placed on it would not be ethically** okay, but just for the sake of creating a simple Crud operation for our purposes in this training course, I'm going to include adding this delete endpoint even if it's not something we're typically want to include.

A screen shot of a computer program

Description automatically generated

So I want an **Http delete**. A public async Task that's going to return an action result. Once again, we don't **return anything from a delete method** in a typical rest API, and I'm just going to call it delete auction. That's going to take the guid as the ID.

And first of all, because we're deleting an auction, we need to actually **get the auction from the** **database**. And this time we'll use the **find async method**. And this can just take in the ID as a parameter.

We can use context auctions dot **remove** and we can specify the auction. Once again, we'll just use var results equals await context or save changes async that's **greater than zero**.

And if the result is not true, we can return a bad request and just say could not update DB. And we'll just return Ok.

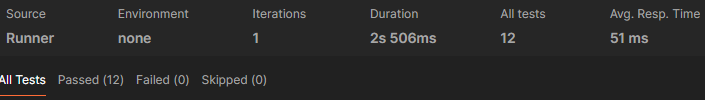
Let's go check to make sure the Http delete works and we've got a delete auction request once again using that created item ID. And just a simple test here to make sure we get a 200 response.

I click send test passes 200 Ok. And that auction should be gone. If I try and repeat this request, then this should fail because the auction has already been deleted. And if we try and get the deleted auction, of course we're going to get a 404 not found perfect.

And that means we've now successfully created our Crud operation.

A screenshot of a black screen

Description automatically generatedJust as a kind of checkpoint, what you should be able to do and we'll see if this works, but you can click on a folder inside Postman and you can run all of the requests inside that folder and it will give you a kind of report of how many tests passed, how many failed, how many were skipped, etcetera.



We've got the test passes is 12. The number of tests that failed is zero and the number of tests skipped is zero. So this is where we should be at this stage of the course. If you have managed to create loads of extra new auctions and you would like to see a clean bill of health on your tests, then you can just drop the database and restart the app and that will give you your ten auctions that we seeded back into the database and then you'll be in a clean place to move forward if you wanted to. So in the next lesson, we'll take a look at what we need to do to save our changes into source control.

### 20. Saving changes into source control

Okay, we've made a start and we've got our first service kind of complete. We've still got more work to do on this particular service. And certainly if we go back and take a look at the spec that we're working towards, then we've got things that we cannot do just yet, as in dealing with micro service based things like emitting events and consuming events. And we've also got identity that we do not have yet that we need for certain endpoints that will require authentication. So we've made some progress. We've got our first very basic service, just a simple Crud application with these endpoints and we're returning the data that's specified in that spec.

Now's a good time to **save our changes into source control**. Now you might have source control public repository preferences, but what I'd really recommend is you do **use GitHub** for this particular training course.

We are going to **use some GitHub functionality** a bit later when it comes to **continuous integration**. And one great tool for doing that is **GitHub actions**, but it does rely on you having a GitHub account. if you do not have a GitHub account at this time, please do create one. It is free of charge. You also need to have **Git installed on your computer** because this is not a beginner course. I'm going to presume you already have git on your machine and you are reasonably comfortable with using it.

And what we'll do first of all is we've got a bunch of code inside our application that we **don't need** **to save into source control**. If we take a look at the File Explorer, then we have these things like **obj folders** that contains a bunch of code, and we have a **bin folder** that contains a bunch of dlls and stuff that's recreated every time that we restart our application. We **don't need to save that stuff into source control** because that's not code that we have written. We only really care about the stuff that we've created ourselves. what we'll do to address that concern is that we'll create a **git ignore file**.



But first of all, let's **initialize our git repository** and we can do that by using the git command and say **git init**. And this initializes an empty git repository in this location.



What we can also do is **add a gitignore file** and the **Dotnet SDK provides this**. Now that's suitable for a .Net application. What we can do for this is say **dotnet new gitignore**. And this is going to create a gitignore file.

And if we take a look at this, then inside the file system what we should have is this gitignore file that **ignores a whole bunch of things that we don't need to save into source control** like the bin folder, the obj folder, that kind of thing. We don't need to adjust this. This is fine.



What we can do then is we can **commit changes** and we're going to use **git add** followed by a **period**. And this is going to change all of the not ignored stuff and add it or **track the changes** that we've made inside Git.



Then we can **commit** **our changes into our local source repository** with git **commit -m** (m for the message) and we'll say end of section two as the **commit message**.

Now we'd also like to **store these changes in a public source repository like GitHub**, just in case we have a catastrophic hardware or hard disk failure, then that would lose our **local** git repository and all our hard work would be for nothing.

But if we use a public repository like GitHub, then our changes are **saved in the** **cloud** somewhere and if we replace our computer then we can get back all of our code from a public repository. And this is the one that I'm using on this training course and I really do heartily recommend that you do the same. So please do create an account if you do not already have one. Once you've got past all their security requirements, such as **two factor authentication** which is required for GitHub nowadays,

we can then go ahead and **create a new repository**, which I'm going to do now, and we need to give it a **name** and I'm going to say **Carsties2024** as the name of this repository. I'm going to make the repository **public**, which means anyone on the internet can see this repository, but they are not allowed to update the repository as read only when we use public. But as long as we're **authenticated**, then we can **make changes to our own repository**. I'm not going to do anything else with this for now, and I'm just going to say **create repository**.

A screenshot of a computer

Description automatically generated

And once this has gone ahead and done its thing, then it **gives us some things that we can do**. The Git branch or **we've done the first three things on here**. We've got the git init.

We're not going to worry about a Readme for now. I'll add something later for that. And we've given our commit a name and committed it locally.

A computer screen shot of text

Description automatically generated

This git branch allows us to **rename the branch**. If your branch was created using master, that's probably going to be removed from the terminology at some point in the future and they recommend you change it to something like **Main**. And if I go double check, then this is the name of my repository main, which is fine, but if yours is something else like Master, then you'd want to use this command to **change it to main**.

**Then we can add the remote repository to our git** which uses this command. **git remote add** followed by the **repository location**.



so I'm just going to copy that into my clipboard and I'm going to add that

A screen shot of a computer

Description automatically generated

and then we can say **git** **push origin main** and it uses the address here to push our code up into the GitHub repository.

And there we go. It's now **located on our public git repository**.

And if I refresh this page then we can see the code that we've added and we've got our source code along with all of the code that we have written.