Dependinte

Best C++ Code Formatter/Beautifier

When using config files (named .clang-format) styles can be per directory - the closest such file in parent directories shall be used for a particular file.[[1]](#footnote-1)

Styles can be inherited from a preset (say LLVM or Google) and can later override different options

It is used by Google and others and is production ready.

How to install clang-format on MinGW (Windows)?

You can get it as part of the LLVM compiler build for Windows. You can download it from <https://llvm.org/builds/>. Once installed, **clang-format.exe** can be found in **C:\Program Files\LLVM\bin**. [[2]](#footnote-2)

In this tutorial, you configure Visual Studio Code to use the GCC C++ compiler (g++) and GDB debugger from [mingw-w64](http://mingw-w64.org) to create programs that run on Windows.[[3]](#footnote-3)

Aplicatie ASP.net core cu react[[4]](#footnote-4)

In this article, you learn how to build an ASP.NET Core project to act as an API backend and a React project to act as the UI.

you can use the method described in this article to create ASP.NET Core Single Page Applications that:

* Put the client app in a separate project, outside from the ASP.NET Core project
* Create the client project based on the framework CLI installed on your computer
* Visual Studio 2022 or later with the **ASP.NET and web development** workload installed. Go to the [Visual Studio downloads](https://visualstudio.microsoft.com/downloads/) page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
* npm ([https://www.npmjs.com/](https://www.npmjs.com/package/npm)), which is included with Node.js
* npx (<https://www.npmjs.com/package/npx>)

Create the frontend app, creat un nou proiect react

Create the backend app

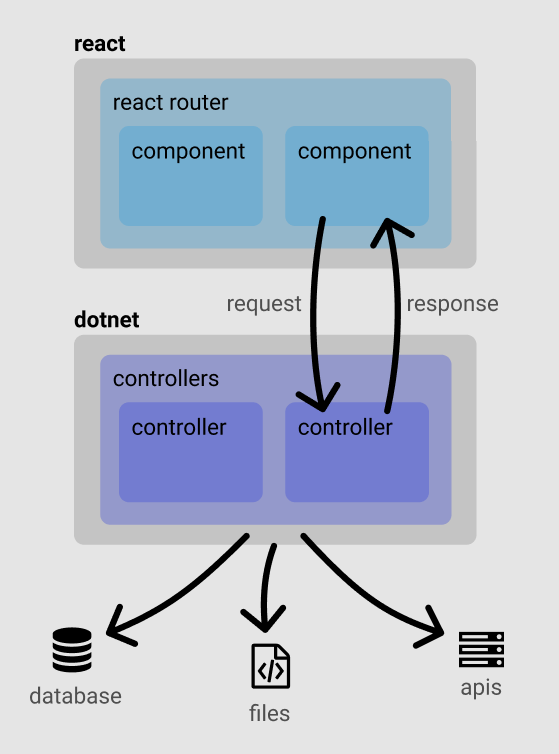
Search and select the ASP.NET Core Web API project.

In this article, we'll take a look how and why you might want to choose React to handle your front end considerations, and let ASP.NET manage the backend[[5]](#footnote-5)

**Architecture**

First, let's consider the range of responsibilities that each solution offers:

* **Templating** - scaffolding out HTML markup based on page data
* **Routing** - converting a request into a response
* **Middleware Pipeline** - building assets
* **Model Binding** - building usable model objects from HTML form data
* **API Services** - handling data requests on a server
* **Client Side Interaction** - updating a web page based on user interaction
* Because ASP.NET performs templating on the server, it's difficult to immediately respond to changes in state on the client without making a round trip, or writing the client side logic entirely separate from the rest of your application.
* Also, React doesn't have anything to say about server-side considerations or API calls, so part of building any React app will involve picking how you want to handle requests on the server.
* Here's roughly how we'll structure our architecture, with React taking care of all the client side concerns, and .NET handling the server side API.



Prerequisites - If you don't have these, you'll need em'

Install node & npm

Install vs code

## MVC Controller - That's the C in MVC!

The controller will orchestrate all of the backend services.

forward the request on the backend, but we could also hit a database, filesystem, or any other backend interfaces. We'll deserialize the API into a strongly typed class using the new [System.Text.Json.JsonSerializer](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializer)

(wrapped up in a Task because we made our method async).

If we run the project, we should now be able to fetch data by navigating to /api/breed in the browser.

If this JavaScript is looking unfamiliar, it's because it's using JSX as well as ES6 syntax that's not supported by all browsers, which is why React needs to get preprocessed, by Babel, into something more universally understood to browsers. But let's take a look at what's going on piece-by-piece.

In order for JSX to be processed, we need to import React into our file (until React 17 that is).

The component function's primary responsibility is to return the HTML markup that we want to use one the page. Everything else is there to help manage state, or inform how to template and transform the data into HTML.

In order to manage state inside of a component, we use the [React Hooks](https://reactjs.org/docs/hooks-intro.html) function useState, which returns two items in an array that we can deconstruct like this:

This allows us to update the state of this property and informs React to dynamically re-render the output when we do so. For example, while loading, we can display some boilerplate text until the data has returned, and then update the view appropriately.

We'll want to fetch data on load, but sending async fetch calls during the initial execution can have some weird side effects, so we can use another hook called useEffect. Every time the component renders, we want to introduce *an effect*, so we'll put the initial call for data inside of there and it will fire as soon as the component goes live:

This hits our backend API, gets the data, and updates our component state not only with the data, but also by toggling the loading indicator. Any updates to state that are used in the render function will cause it to kick off again and update the DOM automatically.

## Navigation - How to get around

Let's imagine we create a second component, roughly the same as the first, to go get a random cat image. How do we handle navigation between both components. Many of the advantages to choosing React occur from preventing full page reloads on navigation. After all, your header, footer, loaded JS, and CSS libraries haven't changed - why should they get shipped to and parsed on the client from scratch. So we'll handle all navigational concerns within React as well.

There is also an alternate way to build a client-side application architecture by leveraging the command and **npm** command of **create-react-app** command.

If you don’t know how to use this, please read about how to use **create-react-app** command which creates the boilerplate for react application and on top of that you can start developing the ReactJS application based on the structure which suits your project requirement better.  Such type of application can be easily used with MVC Web API where you have no MVC View but only APIs to deal with[[6]](#footnote-6).

Cra[[7]](#footnote-7)

[Create React App](https://github.com/facebookincubator/create-react-app) is a comfortable environment for **learning React**, and is the best way to start building **a new** [**single-page**](https://reactjs.org/docs/glossary.html#single-page-application) **application** in React.

It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production. You’ll need to have [Node >= 14.0.0 and npm >= 5.6](https://nodejs.org/en/) on your machine. To create a project, run:

npx create-react-app my-app

cd my-app

npm start

it just creates a frontend build pipeline, so you can use it with any backend you want. Under the hood, it uses [Babel](https://babeljs.io/) and [webpack](https://webpack.js.org/), but you don’t need to know anything about them.

The project is best considered as two entirely separate entities which happen to be co-located in the same place.[[8]](#footnote-8)

* A React application (using “Create React app”)
* An ASP.NET Core Web API backend

## React front end

ClientApp houses the frontend part of your new application.

So long as you’re using version 2.1 or higher of the .NET SDK you’ll find this part of the application is a standard React application created using something called “Create React App”.

When you run the application ClientApp/public/index.html will be served to your browser.

In there you’ll see a div which looks like this…

<div id="root"></div>

This element is all important as React will render your application into this div.

Take a look at ClientApp/src and you’ll see some .js files and a components folder.

In index.js you’ll see the very “top” of your application; this is where it all starts.

ReactDOM.render(

<BrowserRouter basename={baseUrl}>

<App />

</BrowserRouter>,

rootElement);

The first few lines bring in various modules from other js files and node modules you need to run the app.

Then a variable is declared which points to that div we noticed earlier in index.html (with id root).

ReactDOM is called to render your application which consists of a BrowserRouter and App components into this root div.

Now if you head into the <App /> component you’ll see yet more of that funny looking “almost HTML” which points to yet more components…

This React component (App) nests a few Route components inside a Layout component.

So when you run the app in a browser and head over to /fetch-data the FetchData component will be rendered but if you head to /Counter then FetchData will be omitted from the DOM and you’ll get the Counter component being shown instead.

In this tutorial, classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app. These model classes are used with [Entity Framework Core](https://docs.microsoft.com/en-us/ef/core) (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes created are known as ***POCO*** classes, from **P**lain **O**ld **C**LR **O**bjects. POCO classes don't have any dependency on EF Core. They only define the properties of the data to be stored in the database.[[9]](#footnote-9)

### Examine the generated database context class and registration

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. The database context is derived from [Microsoft.EntityFrameworkCore.DbContext](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext) and specifies the entities to include in the data model.

### Dependency injection

ASP.NET Core is built with [dependency injection (DI)](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0). Services, such as the database context, are registered with DI in Program.cs. These services are provided to components that require them via constructor parameters.

## Dependency injection in the controller

Open the Controllers/MoviesController.cs file and examine the constructor:

The constructor uses [Dependency Injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0) to inject the database context (MvcMovieContext) into the controller. The database context is used in each of the [CRUD](https://wikipedia.org/wiki/Create,_read,_update_and_delete) methods in the controller.

The id parameter is generally passed as route data. For example, https://localhost:5001/movies/details/1 sets:

* The controller to the movies controller, the first URL segment.
* The action to details, the second URL segment.
* The id to 1, the last URL segment.

The id can be passed in with a query string, as in the following example:

https://localhost:5001/movies/details?id=1

# Configuration in ASP.NET Core

Application configuration in ASP.NET Core is performed using one or more [configuration providers](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-6.0#cp). Configuration providers read configuration data from key-value pairs using a variety of configuration sources:[[10]](#footnote-10)

* Settings files, such as appsettings.json

# Safe storage of app secrets in development in ASP.NET Core

## Secret Manager

The Secret Manager tool stores sensitive data during the development of an ASP.NET Core project. In this context, a piece of sensitive data is an app secret. App secrets are stored in a separate location from the project tree. The app secrets are associated with a specific project or shared across several projects. The app secrets aren't checked into source control.

# Dependency injection into controllers in ASP.NET Core

* Article
* 06/04/2022
* 4 minutes to read

ASP.NET Core MVC controllers request dependencies explicitly via constructors. ASP.NET Core has built-in support for [dependency injection (DI)](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0). DI makes apps easier to test and maintain.[[11]](#footnote-11)

# ASP.NET Core fundamentals overview

This article provides an overview of the fundamentals for building ASP.NET Core apps, including dependency injection (DI), configuration, middleware, and more.[[12]](#footnote-12)

ASP.NET Core apps created with the web templates contain the application startup code in the Program.cs file. The Program.cs file is where:

* Services required by the app are configured.
* The app's request handling pipeline is defined as a series of [middleware components](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-6.0).

## Dependency injection (services)

ASP.NET Core includes [dependency injection (DI)](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0) that makes configured services available throughout an app. Services are added to the DI container

Services are typically resolved from DI using constructor injection. The DI framework provides an instance of this service at runtime.

The following code uses constructor injection to resolve the database context and logger from DI:

## Middleware

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an [HttpContext](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/http-context?view=aspnetcore-6.0) and either invokes the next middleware in the pipeline or terminates the request.

By convention, a middleware component is added to the pipeline

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseAuthorization();

## Servers

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of [request features](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/request-features?view=aspnetcore-6.0) composed into an HttpContext.

## Configuration

ASP.NET Core provides a [configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-6.0) framework that gets settings as name-value pairs from an ordered set of configuration providers. Built-in configuration providers are available for a variety of sources, such as .json files, .xml files,

By [default](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-6.0#default), ASP.NET Core apps are configured to read from appsettings.json

For managing confidential configuration data such as passwords, .NET Core provides the [Secret Manager](https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-6.0#secret-manager).

## Routing

A route is a URL pattern that is mapped to a handler. The handler is typically a Razor page, an action method in an MVC controller, or a middleware. ASP.NET Core routing gives you control over the URLs used by your app.

# ASP.NET MVC: Business Logic as a Separate Layer

ASP.NET MVC offers a great way of how to separate different application layers. View layer is responsible for data representation, the controller layer is responsible for receiving and replying to requests, and models are used as two-way information carriers between the previous two layers.[[13]](#footnote-13)

This separation of concerns is convenient for developers because there is no spaghetti mix of HTML layout and business logic. [ASP.NET MVC](https://www.diatomenterprises.com/technologies/asp-net/) welcomes developers to use dependency injection

MVC assumes that: Model stores, View represents, Controller handles requests and returns a response. A natural question arises – Where do I put my domain specific logic? There are three ways that developers tend to go for:

1. **Model.** In this case, the Model becomes rich and wise: it is able to request something from the data store, format it in an appropriate way and return to View. But there are disadvantages:

* View may be very large and complicated, and Model class becomes extended;

Model may be used as the result of GET response and data of POST request. In POST request case we don’t need all the methods which exemplar of a Model has; we need only fields posted to a server – the information itself.

1. **Controller.** If we put logic in Controller it becomes loaded with responsibilities it should not have – it is in charge of complex validation, business rules, helper calls, model generation, formatting and redirection. This approach also has disadvantages:

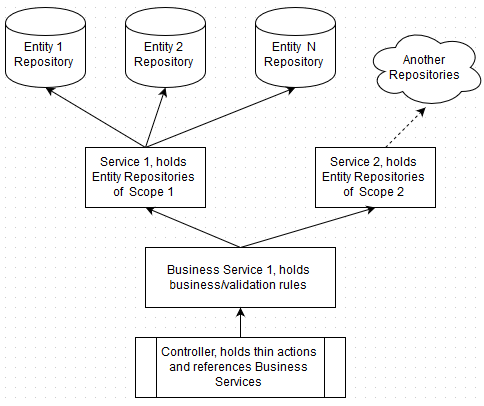
* Controller’s methods may become significantly extended.

action method, but still, business rules may change and then you are in trouble.

It is clear that three of these approaches violate basic OOP principles. Classes become bloated, objects are getting larger and larger to address these problems.

It is reasonable to let Controller be in charge of MVC-specific things like model binding and redirection. If we do not put business logic in Controller, then we need to do that in another layer of an application and Controller should be dependent on it.

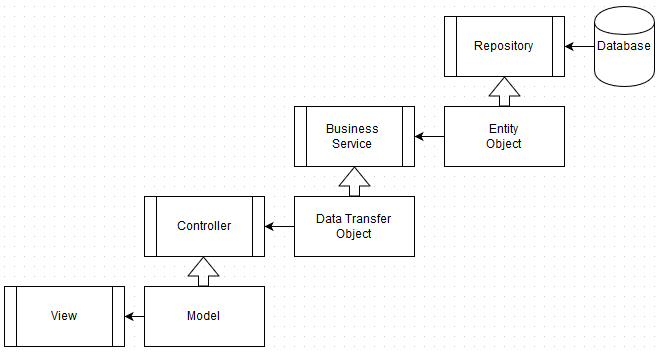
At the same time, business rules are usually dependent on data received from a store. Usual work-flow of GET-request processing: is request from store, filter using business logic, format, display.  
The layer which holds business logic can be called Business Service and in this case layer hierarchy may be the following:



Controller is dependent on Business Service. Having Business Service in place, there is no need to have long and fat Controller actions. Controller actions become thin and easily-readable:

## Data Transfer Objects

In this scheme database stores, records are mapped to Entity Objects. Entity Objects are requested by appropriate Repository and sent to Business Service, where they are processed (filtered, combined, etc.) and converted to DTOs. DTOs are sent to a Controller. In Controller DTOs may be either mixed or converted to a Model and sent to a View.



10

I know this has already been answered, but I categorize models into 3 groups[[14]](#footnote-14)

ViewModels - These are light weight (often poco) classes that model the data needed by a page on your site. These classes handle the mundane boilerplate of what gets shown to the user, and changes when the data that you want to display changes.

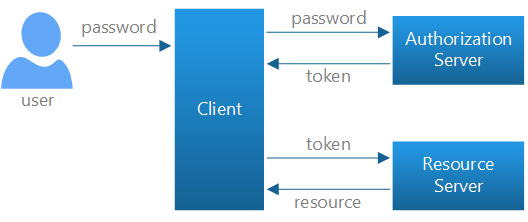
DomainModels - These are normally heavy weight business logic classes. They normally model the core business rules for what you're doing. These classes are often highly cohesive and are where the majority of the work that makes your site special happens. I said these models are normally heavyweight but in reality if all your project does is take that data from the user and stick it in the database, this class is going to be a little data mapping class. Many times you'll see these classes being composed of persistence models and returning view models.

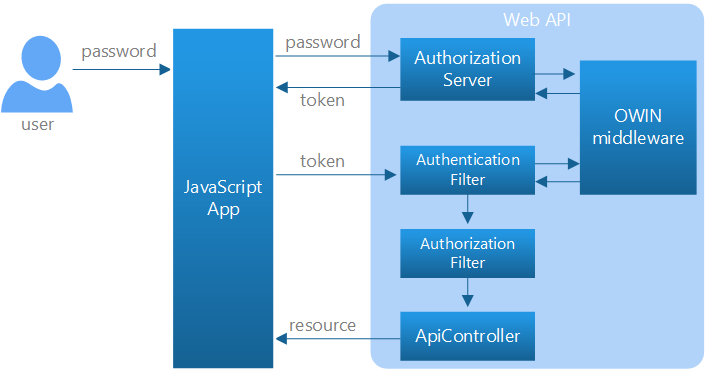
PersistenceModels - These are models of your persistence mechanism. For most of us this means modeling a database table, but could also be a complex nosql document or json (or whatever) data that's returned from an api request. Their responsibility is to handle the mundane boiler plate of what shape your external data takes.

Keep in mind also that you don't always need to have all three of these types of models present in your project. Sometimes your view model will be line for line what you're persistence model is. In that case you'd be wasting your clients money to write the whole thing twice and add a domain model to map one to the other. You're the developer and it's your job to know when to build a air-craft carrier to go to the store for groceries.

Individual accounts provide two ways for a user to log in:[[15]](#footnote-15)

* **Local login**. The user registers at the site, entering a username and password. The app stores the password hash in the membership database. When the user logs in, the ASP.NET Identity system verifies the password.
* The user enters a name and password into the client.
* The client sends these credentials to the authorization server.
* The authorization server authenticates the credentials and returns an access token.
* To access a protected resource, the client includes the access token in the Authorization header of the HTTP request.





## Sending an Unauthorized Request

To get started, run the app and click the **Call API** button. When the request completes, you should see an error message in the **Result** box. That's because the request does not contain an access token, so the request is unauthorized.

## Sending an Unauthorized Request

To get started, run the app and click the **Call API** button. When the request completes, you should see an error message in the **Result** box. That's because the request does not contain an access token, so the request is unauthorized.

The **Call API** button sends an AJAX request to ~/api/values, which invokes a Web API controller action. Here is the section of JavaScript code that sends the AJAX request.

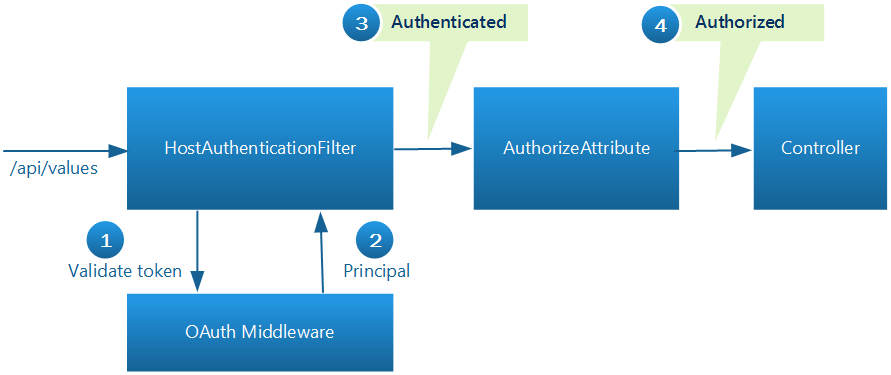
The **Log In** button sends a request to the token endpoint. The body of the request contains the following form-url-encoded data:

* grant\_type: "password"
* username: <the user's email>
* password: <password>

Here is the JavaScript code that sends the AJAX request:

If the request succeeds, the authorization server returns an access token in the response body. Notice that we store the token in session storage, to use later when sending requests to the API. Unlike some forms of authentication (such as cookie-based authentication), the browser will not automatically include the access token in subsequent requests. The application must do so explicitly.

Now that we have a bearer token, we can make an authenticated request to the API. This is done by setting the Authorization header in the request. Click the **Call API** button again to see this.



# Authentication and Authorization in ASP.NET Web API

You've created a web API, but now you want to control access to it. In this series of articles, we'll look at some options for securing a web API from unauthorized users. This series will cover both authentication and authorization.

* *Authentication* is knowing the identity of the user. For example, Alice logs in with her username and password, and the server uses the password to authenticate Alice.
* *Authorization* is deciding whether a user is allowed to perform an action. For example, Alice has permission to get a resource but not create a resource.

Let's start by defining a few key terms:[[16]](#footnote-16)

MVC

A software pattern that defines a separation of concerns between three distinct components: a model, a view, and a controller.

ASP.NET MVC

An application framework created by Microsoft that is inspired by and attempts to implement the MVC pattern.

Model

The component of the MVC pattern that is responsible for the behavior and data of an application.

Entity

A class involved in data persistence, typically as a representation of a table in a relational database.

So what exactly is the Model in ASP.NET MVC? Well, it's not one thing. Its function is split over three components:

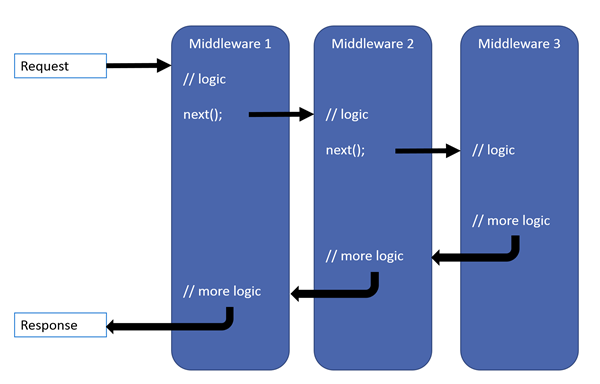
1. **Entity** - Holds the data that is persisted.
2. **View Model** - Holds the business logic related to presenting that data to the user or allowing the user to interact with that data (forms).
3. **Repository/Service** - Holds the business logic related to selecting, filtering, etc. the data from the persistence-store.

# ASP.Net Core Custom Middleware[[17]](#footnote-17)

ASP.Net core middleware is a component or piece of code executes in the request/response pipeline which decides whether to pass the request to the next component in the pipeline or not. Middleware does some kind of work before deciding to pass the request to next component in the pipeline.

Let me explain with an example, say I want to log all the incoming request first before it hits to any other component in the application. Also, I want to read a custom header value from the request and make it available throughout the application.

As you can see in the diagram above request pipeline consists of a sequence of middleware’s, called one after the other. Each middleware can perform operations before and after it delegates the request/response to the next middleware.



# [Storing files in SQL Server](https://stackoverflow.com/questions/13420305/storing-files-in-sql-server)[[18]](#footnote-18)

It's an old question I know, but with SQL Server 2012 is it finally ok to store files in the database, or should they really be kept in the filesystem with only references to them in the database?

There's a really good paper by Microsoft Research called [To Blob or Not To Blob](http://research.microsoft.com/apps/pubs/default.aspx?id=64525).[[19]](#footnote-19)

Their conclusion after a large number of performance tests and analysis is this:

* if your pictures or document are typically below 256K in size, storing them in a database VARBINARY column is more efficient
* if your pictures or document are typically over 1 MB in size, storing them in the filesystem is more efficient (and with SQL Server 2008's FILESTREAM attribute, they're still under transactional control and part of the database)
* in between those two, it's a bit of a toss-up depending on your use

# Create a REST API with Attribute Routing in ASP.NET Web API 2

Web API 2 supports a new type of routing, called *attribute routing*. For a general overview of attribute routing, see [Attribute Routing in Web API 2](https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2). In this tutorial, you will use attribute routing to create a REST API for a collection of books. The API will support the following actions:[[20]](#footnote-20)

| **Action** | **Example URI** |
| --- | --- |
| Get a list of all books. | /api/books |
| Get a book by ID. | /api/books/1 |
| Get the details of a book. | /api/books/1/details |
| Get a list of books by genre. | /api/books/fantasy |
| Get a list of books by publication date. | /api/books/date/2013-02-16 /api/books/date/2013/02/16 (alternate form) |
| Get a list of books by a particular author. | /api/authors/1/books |

All methods are read-only (HTTP GET requests).

For the data layer, we'll use Entity Framework.

## Add DTO Classes

If you run the application now and send a GET request to /api/books/1, the response looks similar to the following. (I added indentation for readability.) Instead, I want this request to return a subset of the fields. Also, I want it to return the author's name, rather than the author ID. To accomplish this, we'll modify the controller methods to return a data transfer object (DTO) instead of the EF model. A DTO is an object that is designed only to carry data.

## Add Route Attributes

Next, we'll convert the controller to use attribute routing. First, add a **RoutePrefix** attribute to the controller. This attribute defines the initial URI segments for all methods on this controller.

The route template for each controller method is the prefix plus the string specified in the **Route** attribute. For the GetBook method, the route template includes the parameterized string "{id:int}", which matches if the URI segment contains an integer value.

1. https://stackoverflow.com/questions/841075/best-c-code-formatter-beautifier [↑](#footnote-ref-1)
2. https://superuser.com/questions/1505283/how-to-install-clang-format-on-mingw-windows [↑](#footnote-ref-2)
3. https://code.visualstudio.com/docs/cpp/config-mingw [↑](#footnote-ref-3)
4. https://docs.microsoft.com/en-us/visualstudio/javascript/tutorial-asp-net-core-with-react?view=vs-2022 [↑](#footnote-ref-4)
5. https://www.thisdot.co/blog/adding-react-to-your-asp-net-mvc-web-app [↑](#footnote-ref-5)
6. https://blog.e-zest.com/setting-up-hybrid-structure-of-mvc-and-reactjs [↑](#footnote-ref-6)
7. https://reactjs.org/docs/create-a-new-react-app.html [↑](#footnote-ref-7)
8. https://jonhilton.net/understanding-the-asp-net-react-template/ [↑](#footnote-ref-8)
9. https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-6.0&tabs=visual-studio [↑](#footnote-ref-9)
10. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-6.0 [↑](#footnote-ref-10)
11. https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-6.0 [↑](#footnote-ref-11)
12. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-6.0&tabs=windows [↑](#footnote-ref-12)
13. https://diatomenterprises.com/asp-net-mvc-business-logic-as-a-separate-layer/ [↑](#footnote-ref-13)
14. https://stackoverflow.com/questions/14657656/asp-net-mvc-business-logic-in-domain-model-vs-service-layer [↑](#footnote-ref-14)
15. https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/individual-accounts-in-web-api [↑](#footnote-ref-15)
16. https://cpratt.co/entities-are-not-models/ [↑](#footnote-ref-16)
17. https://medium.com/swlh/asp-net-core-custom-middleware-89206c27f22b [↑](#footnote-ref-17)
18. https://stackoverflow.com/questions/13420305/storing-files-in-sql-server [↑](#footnote-ref-18)
19. https://www.microsoft.com/en-us/research/publication/to-blob-or-not-to-blob-large-object-storage-in-a-database-or-a-filesystem/?from=https%3A%2F%2Fresearch.microsoft.com%2Fapps%2Fpubs%2Fdefault.aspx%3Fid%3D64525 [↑](#footnote-ref-19)
20. https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/create-a-rest-api-with-attribute-routing [↑](#footnote-ref-20)