Overview

Reference: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api

API Spec:

| API | Description | Request body | Response body |
| --- | --- | --- | --- |
| GET /api/todo | Get all to-do items | None | Array of to-do items |
| GET /api/todo/{id} | Get an item by ID | None | To-do item |
| POST /api/todo | Add a new item | To-do item | To-do item |
| PUT /api/todo/{id} | Update an existing item | To-do item |  |
| DELETE /api/todo/{id} | Delete an item | None. No request body- | None |

API Design:

+

* The client is whatever consumes the web API (browser, mobile app, and so forth). We aren’t writing a client in this tutorial. We'll use [Postman](https://www.getpostman.com/) to test the app.
* A *model* is an object that represents the data in your application. In this case, the only model is a to-do item. Models are represented as simple C# classes (POCOs).
* A *controller* is an object that handles HTTP requests and creates the HTTP response. This app will have a single controller.
* To keep the tutorial simple, the app doesn’t use a database. Instead, it just keeps to-do items in memory. But we’ll still include a (trivial) data access layer, to illustrate the separation between the web API and the data layer. For a tutorial that uses a database, see [Building your first ASP.NET Core MVC app with Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/index).

## Repository class

A repository is an object that encapsulates the data layer. The repository contains logic for retrieving and mapping data to an entity model.

## Register the repository

By defining a repository interface, we can decouple the repository class from the MVC controller that uses it. Instead of instantiating a TodoRepository inside the controller we will inject an ITodoRepository using the built-in support in ASP.NET Core for [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection).

This approach makes it easier to unit test your controllers. Unit tests should inject a mock or stub version of ITodoRepository. That way, the test narrowly targets the controller logic and not the data access layer.

In order to inject the repository into the controller, we need to register it with the DI container.

services.AddSingleton<ITodoRepository, TodoRepository>();

# **Getting started with ASP.NET Core MVC and Entity Framework Core using Visual Studio**

<https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro>

# **Asynchronous Repositories**

<https://blogs.msdn.microsoft.com/mrtechnocal/2014/03/16/asynchronous-repositories/>

**Implementing Repository Pattern using Async/Await**

Changes:

* Convert every return value T to async Task<T>
* Use data base async operations inside
* Add 'Async' suffix to method names