(Demo prep)

- Ensure no VS Code extensions installed
- Cleanup the browser
- Ensure no docker images, volumes, networks available

Adding the REST API operations

1. In order to add our API operations we will need to introduce a controller, specifically a web api controller. The controller groups the set of actions that can handle the API requests, including the routes, authorization and a series of other rules usually needed in REST APIs.

2. Create ItemsController.cs

3. Remove the navigation pane

4. **ControllerBase** provides many properties and methods useful when handling HTTP requests, like the BadRequest, NotFound and CreatedAtAction methods, which we will use in this and in future lessons.

   The **ApiController** attribute enables a series of features that improve your REST api developer experience like having model validation errors automatically return a 400 Bad Request error or how to bind incoming requests into our method parameters.

   The **Route** attribute specifies the URL pattern that this controller will map to. For instance, if we use “items” here, it means that this controller will handle routes that start with /items, like https://localhost:5001/items

   Now, since in this lesson we would like focus on the REST API part of or microservice we will not be interacting with a real database just yet. We will use an in-memory list of items that our API operations will interact with, and in a future lesson we will introduce a proper repository class that will take care of interacting with our database.

5. Implement ItemsController:

```
[ApiController]
[Route("items")]
public class ItemsController : ControllerBase
{
   Private static readonly List<ItemDto> items = new()
   {
      new ItemDto(Guid.NewGuid(), "Potion", "Restores a small amount of HP", 5,
DateTimeOffset.UtcNow),
      new ItemDto(Guid.NewGuid(), "Antidote", "Cures poison", 7, DateTimeOffset.UtcNow),
      new ItemDto(Guid.NewGuid(), "Bronze sword", "Deals a small amount of damage", 20,
DateTimeOffset.UtcNow)
```

```
        };

        [HttpGet]
        public IEnumerable<ItemDto> Get()
        {
            return items;
        }

        [HttpGet("{id}")]
        public ItemDto GetById(Guid id)
        {
            var item = items.Where(item => item.Id == id).SingleOrDefault();
            return item;
        }

    }
```

6. Try GET in Swagger UI

7. Add POST:
   (ActionResult allows us to return a type that represents one of several HTTP status codes, like 200 OK, or 400 BadRequest. It also allows us to return a more specific type, like a DTO type, if we need to)

```
[HttpPost]
public ActionResult<ItemDto> Post(CreateItemDto createItemDto)
{
    var item = new ItemDto(Guid.NewGuid(), createItemDto.Name, createItemDto.Description,
createItemDto.Price, DateTimeOffset.UtcNow);
    items.Add(item);

    return CreatedAtAction(nameof(GetById), new { id = item.Id }, item);
}
```

8. Try POST in Swagger UI

9. Add PUT:

```
[HttpPut("{id}")]
public IActionResult Put(Guid id, UpdateItemDto updateItemDto)
{
    var existingItem = items.Where(item => item.Id == id).SingleOrDefault();

    var updatedItem = existingItem with
    {
        Name = updateItemDto.Name,
        Description = updateItemDto.Description,
```

```
        Price = updateItemDto.Price
    };

    var index = items.FindIndex(existingItem => existingItem.Id == id);
    items[index] = updatedItem;

    return NoContent();
}
```

10. Update an item in Swagger UI

11. Add DELETE:

```
[HttpDelete("{id}")]
public IActionResult Delete(Guid id)
{
    var index = items.FindIndex(existingItem => existingItem.Id == id);
    items.RemoveAt(index);

    return NoContent();
}
```

12. Delete an item in Swagger UI

Our REST API seems to be working fine, however try querying for an un-existing item or perhaps creating an item without a name. We need to do something about those. In the next lesson we will see how to properly handle these kinds of invalid inputs.