Project Write-Up

So, the first thing I began with was the model for the medicines which will be stored at the API side. The Medicine model has the following properties shown in the figure below.

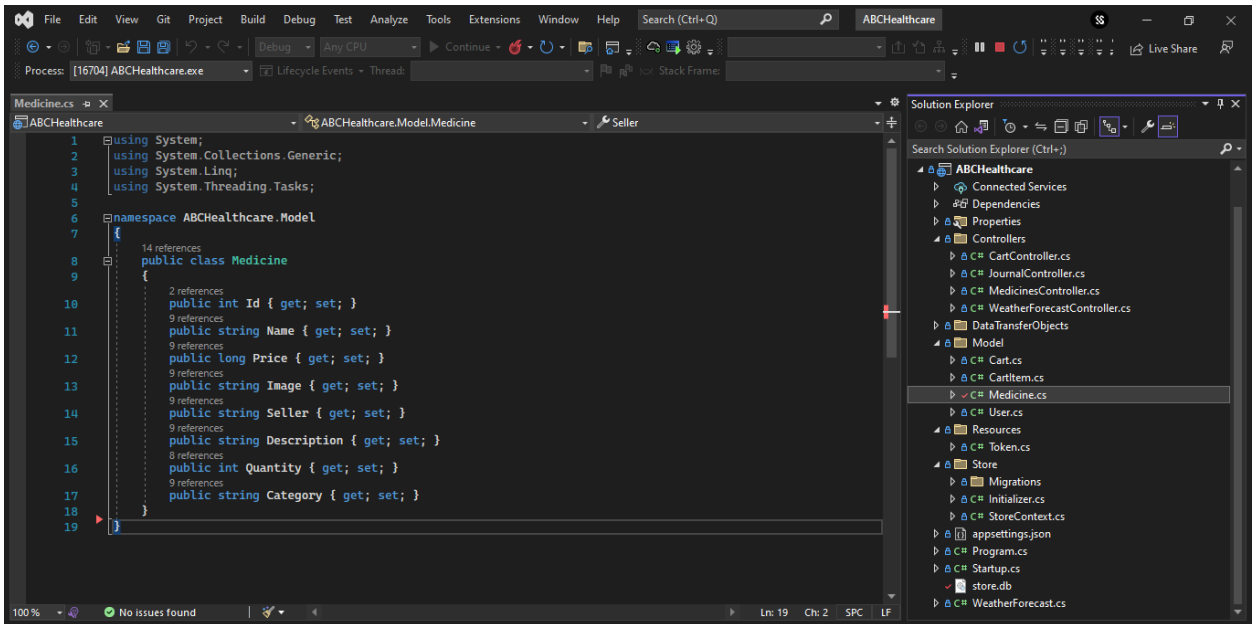


Fig. 1: Model/Medicine.cs

Now, the next model is for the 'Cart' that I will be using in the frontend for buying the medicines. Different users will have different cart IDs and BuyerIDs, so that the medicines that they add to cart are stored in the database. Now, each Cart has a list of CartItems which consists of some navigation properties to map the items inside the cart. Two void methods for adding items to the cart, and removing items from the cart are also included.

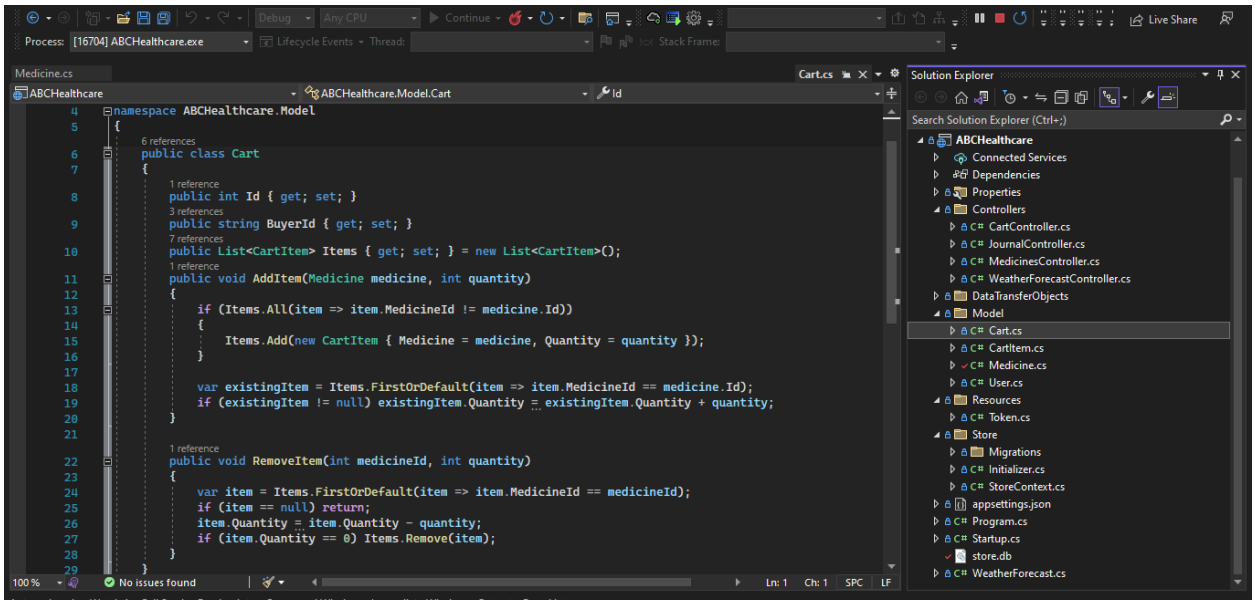


Fig. 2: Model/Cart.cs

CartItem model is used to store the ID of cart, ID of medicines inside each particular cart, and the Quantity of medicines added in the cart.

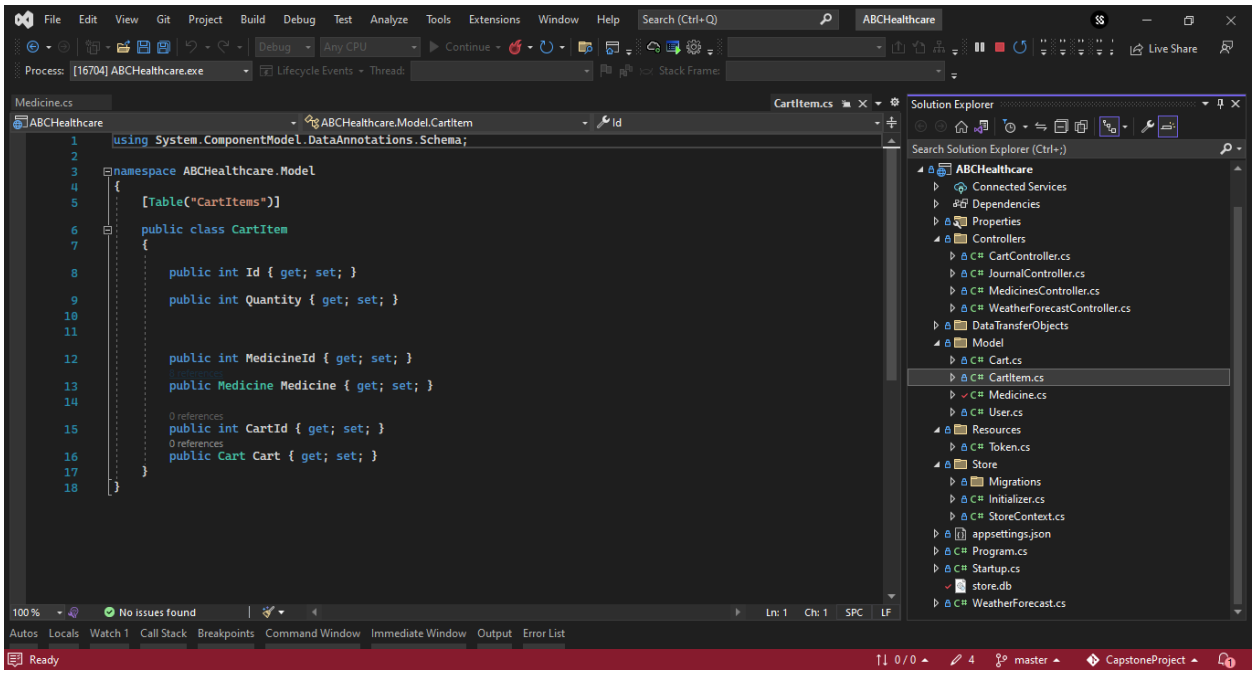


Fig. 3: Model/CartItem.cs

Next, is the DbContext that is used for creating the tables in the database as a object relational mapper using the entity framework.

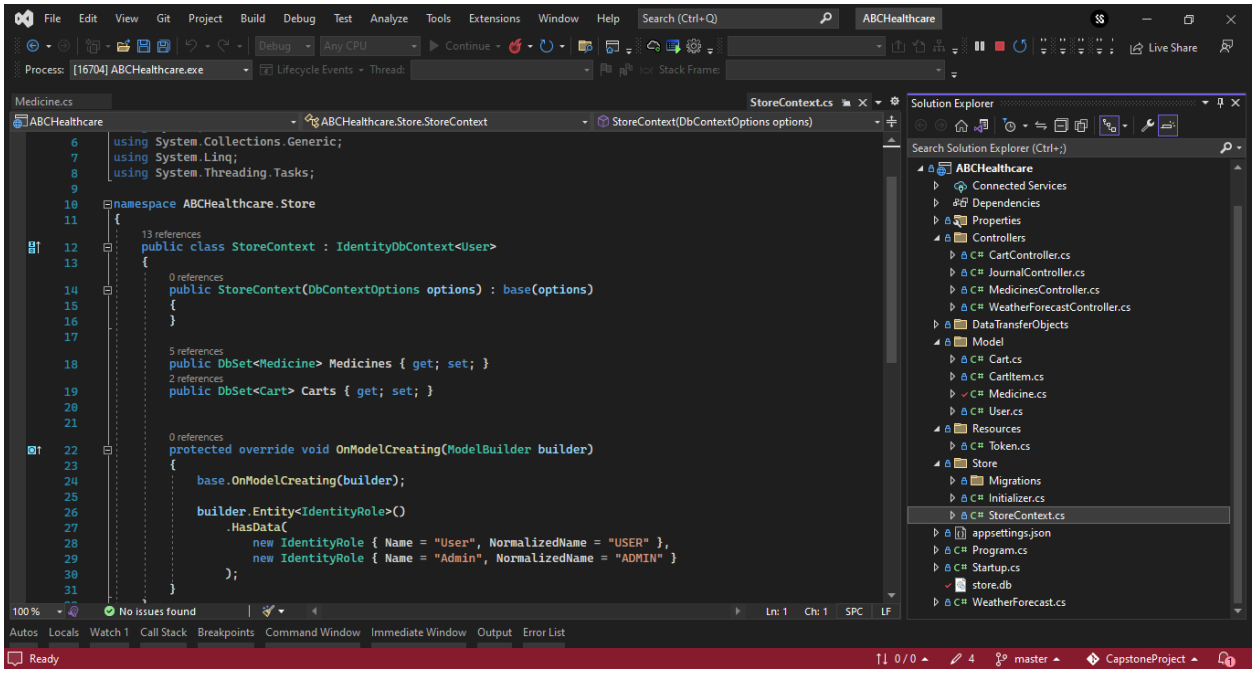


Fig. 4: StoreContext.cs

Finally, Initializer.cs is used to seed the initial data. I am adding some sample data input to begin with in the User and Admin tables here.

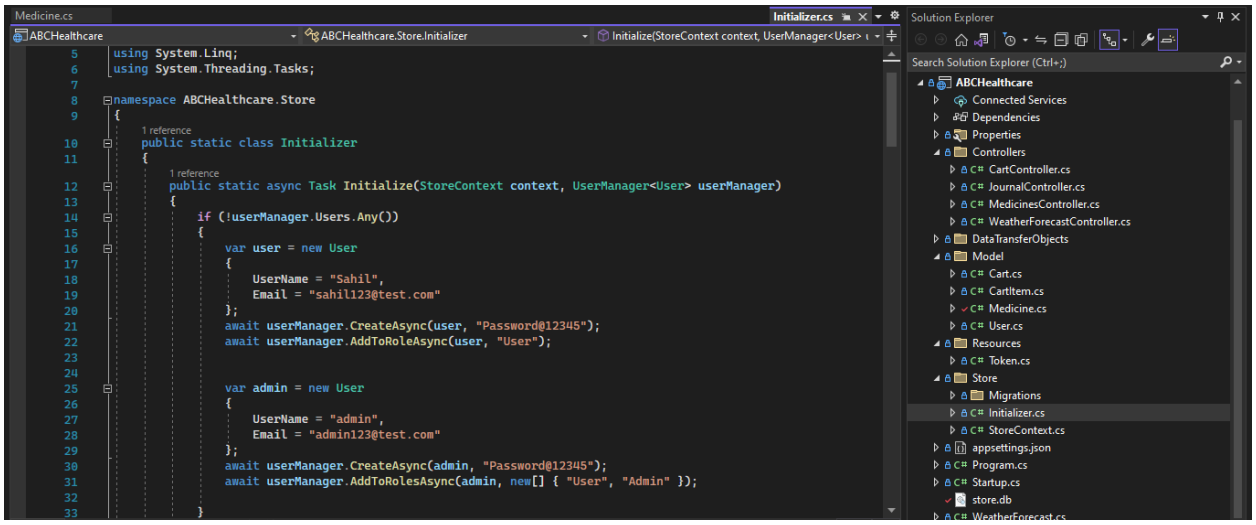


Fig. 5: Initializer.cs

The Initializer file also contains the seed data for some medicines.

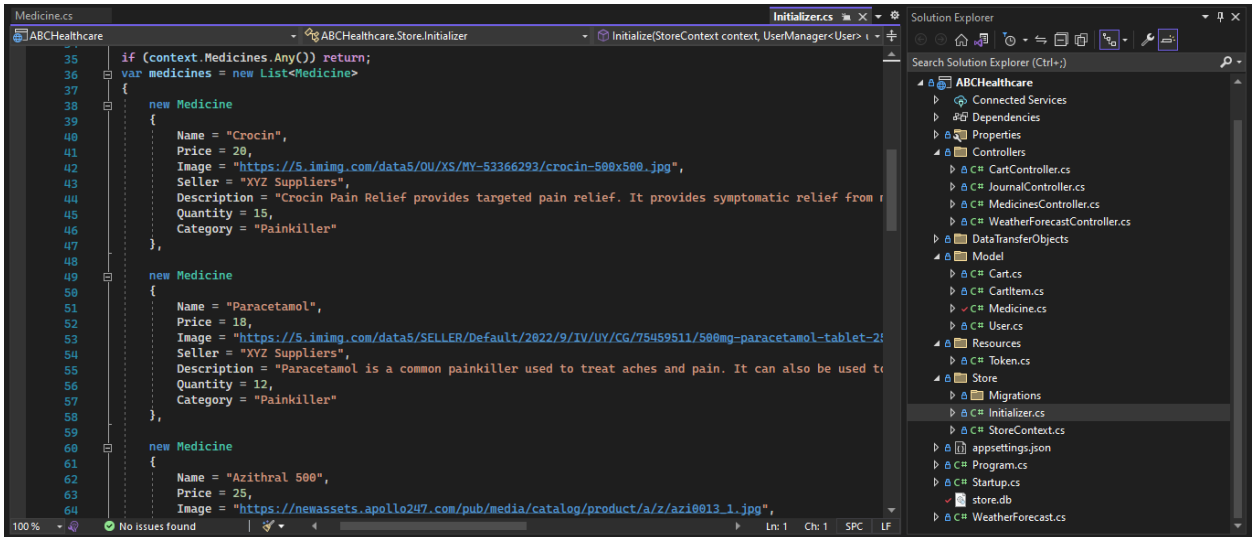
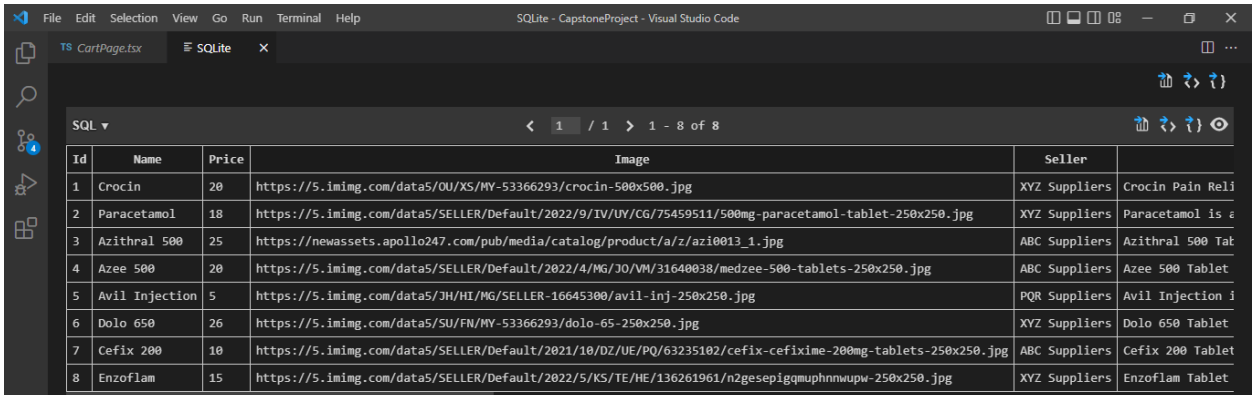


Fig. 6: Initializer.cs (contd…)



Id	Name	Price	Image	Seller	
1	Crocin	20	https://5.imimg.com/data5/OU/XS/MY-53366293/crocin-500x500.jpg	XYZ Suppliers	Crocin Pain Reli
2	Paracetamol	18	https://5.imimg.com/data5/SELLER/Default/2022/9/IV/UY/CG/75459511/500mg-paracetamol-tablet-250x250.jpg	XYZ Suppliers	Paracetamol is a
3	Azithral 500	25	https://newassets.apollo247.com/pub/media/catalog/product/a/z/azi0013_1.jpg	ABC Suppliers	Azithral 500 Tab
4	Azee 500	20	https://5.imimg.com/data5/SELLER/Default/2022/4/MG/JO/VM/31640038/medzee-500-tablets-250x250.jpg	ABC Suppliers	Azee 500 Tablet
5	Avil Injection	5	https://5.imimg.com/data5/JH/HI/MG/SELLER-16645300/avil-inj-250x250.jpg	PQR Suppliers	Avil Injection i
6	Dolo 650	26	https://5.imimg.com/data5/SU/FN/MY-53366293/dolo-65-250x250.jpg	XYZ Suppliers	Dolo 650 Tablet
7	Cefix 200	10	https://5.imimg.com/data5/SELLER/Default/2021/10/DZ/UE/PQ/63235102/cefix-cefixime-200mg-tablets-250x250.jpg	ABC Suppliers	Cefix 200 Tablet
8	Enzoflam	15	https://5.imimg.com/data5/SELLER/Default/2022/5/KS/TE/HE/136261961/n2gesepigqmuphnnwupw-250x250.jpg	XYZ Suppliers	Enzoflam Tablet

Fig. 7:.Medicine seed data reflected in database

Next, is the Migrations that I am using to create the schema for the tables. This is the Medicines table schema here. The command that I used here was: dotnet ef migrations add InitialCreate

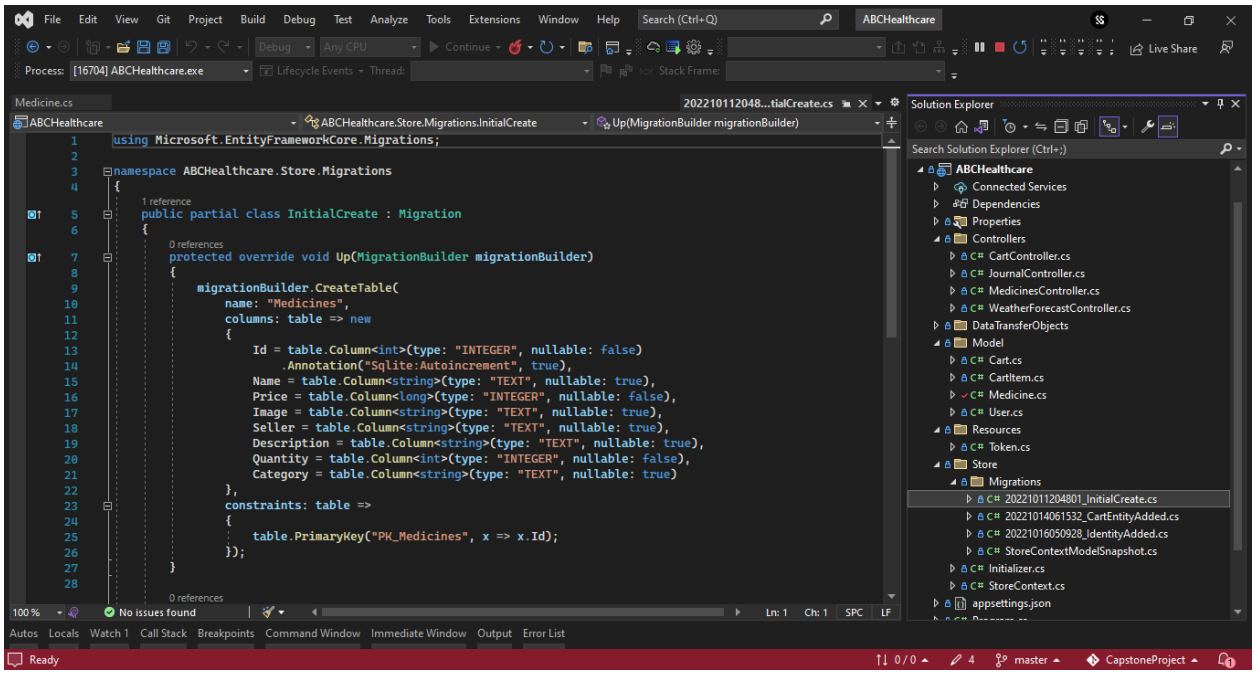


Fig. 8: InitialCreate.cs

The following file contains the Migrations for the Carts table and the CartItem table

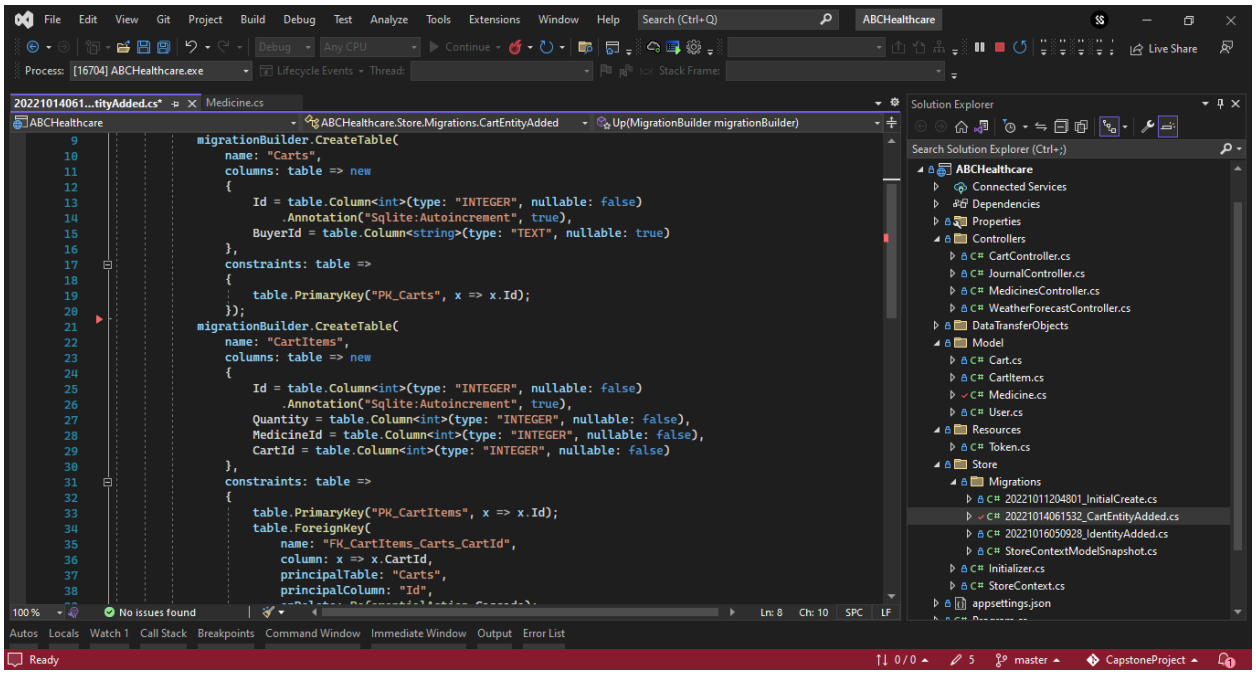
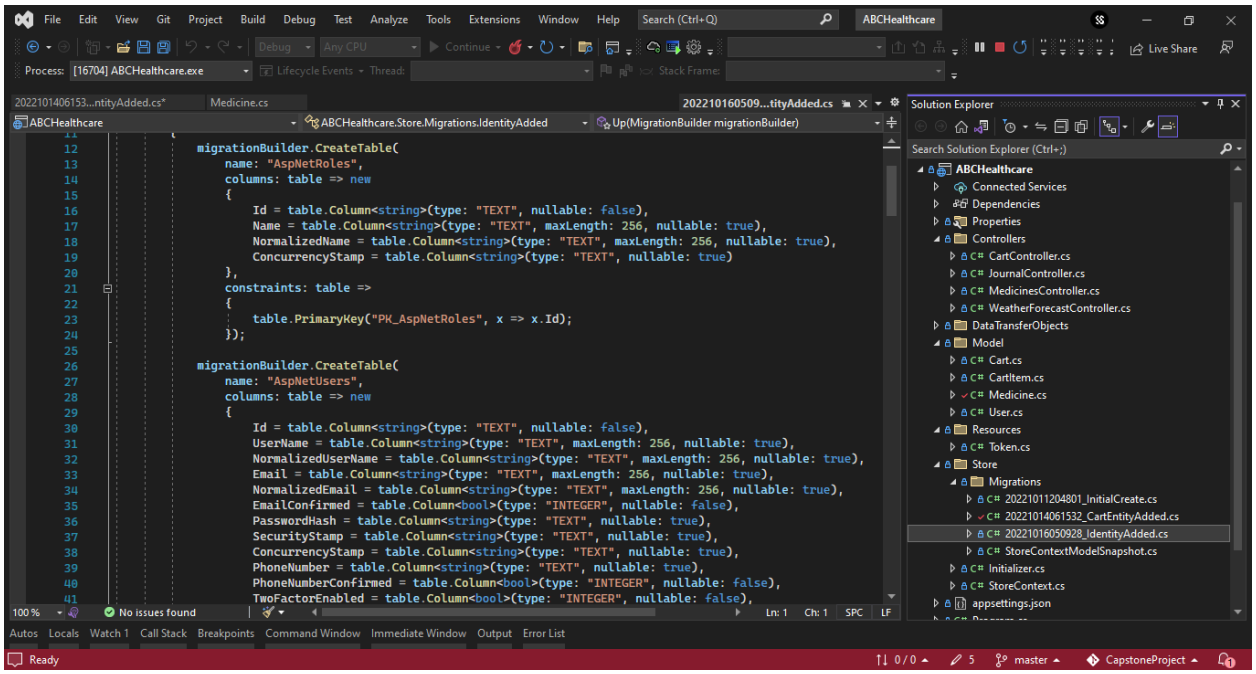


Fig. 9: InitialCreate.cs

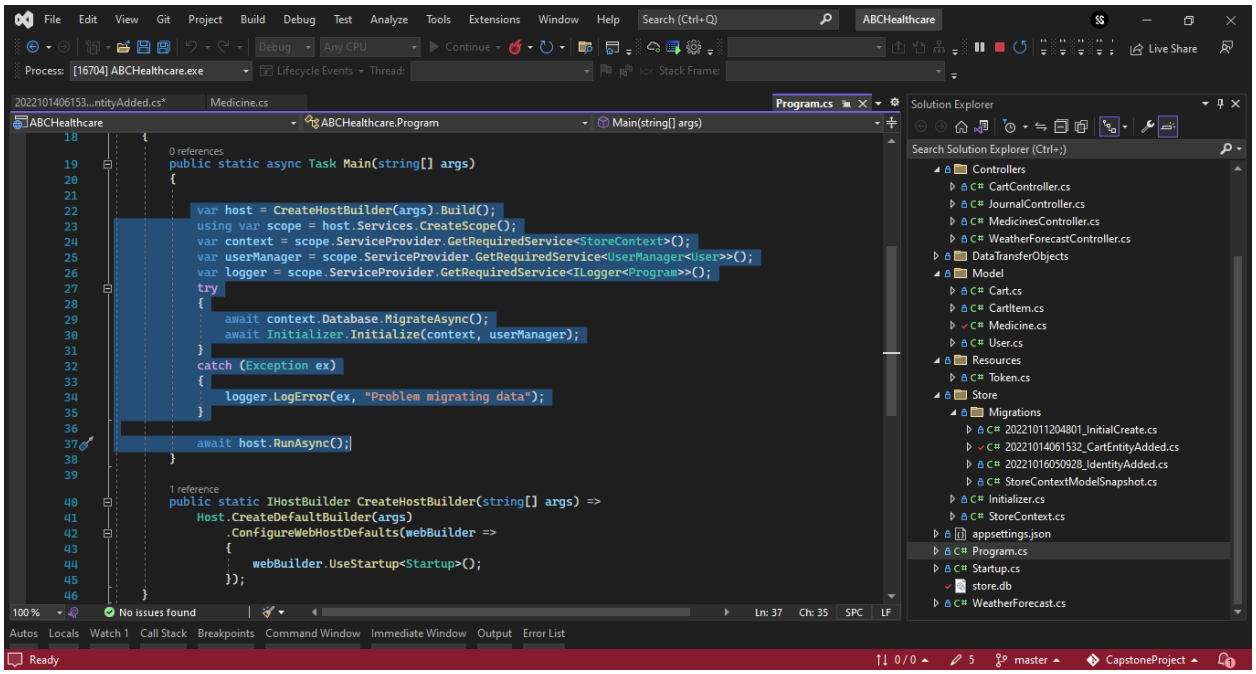
The following file contains the Migrations for Roles and Users tables



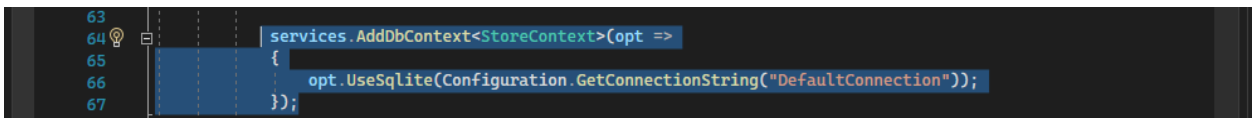
```
migrationBuilder.CreateTable(
    name: "AspNetRoles",
    columns: table => new
    {
        Id = table.Column<string>(type: "TEXT", nullable: false),
        Name = table.Column<string>(type: "TEXT", maxLength: 256, nullable: true),
        NormalizedName = table.Column<string>(type: "TEXT", maxLength: 256, nullable: true),
        ConcurrencyStamp = table.Column<string>(type: "TEXT", nullable: true)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_AspNetRoles", x => x.Id);
    });

migrationBuilder.CreateTable(
    name: "AspNetUsers",
    columns: table => new
    {
        Id = table.Column<string>(type: "TEXT", nullable: false),
        UserName = table.Column<string>(type: "TEXT", maxLength: 256, nullable: true),
        NormalizedUserName = table.Column<string>(type: "TEXT", maxLength: 256, nullable: true),
        Email = table.Column<string>(type: "TEXT", maxLength: 256, nullable: true),
        NormalizedEmail = table.Column<string>(type: "TEXT", maxLength: 256, nullable: true),
        EmailConfirmed = table.Column<bool>(type: "INTEGER", nullable: false),
        PasswordHash = table.Column<string>(type: "TEXT", nullable: true),
        SecurityStamp = table.Column<string>(type: "TEXT", nullable: true),
        ConcurrencyStamp = table.Column<string>(type: "TEXT", nullable: true),
        PhoneNumber = table.Column<string>(type: "TEXT", nullable: true),
        PhoneNumberConfirmed = table.Column<bool>(type: "INTEGER", nullable: false),
        TwoFactorEnabled = table.Column<bool>(type: "INTEGER", nullable: false),
```

Fig. 10: IdentityAdded.cs

Now, next is the Program.cs file where I have added the logic so that, on startup the migrations will be updated into the store.db database file using SQLite and passing the ConnectionString..



```
public static async Task Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();
    using var scope = host.Services.CreateScope();
    var context = scope.ServiceProvider.GetRequiredService<StoreContext>();
    var userManager = scope.ServiceProvider.GetRequiredService<UserManager<User>>();
    var logger = scope.ServiceProvider.GetRequiredService<ILogger<Program>>();
    try
    {
        await context.Database.MigrateAsync();
        await Initializer.Initialize(context, userManager);
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "Problem migrating data");
    }

    await host.RunAsync();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}
```



```
services.AddDbContext<StoreContext>(opt =>
{
    opt.UseSqlite(Configuration.GetConnectionString("DefaultConnection"));
});
```

Fig. 11: Program.cs

Also, I have added the ConnectionString for the connection with a data source for store.db

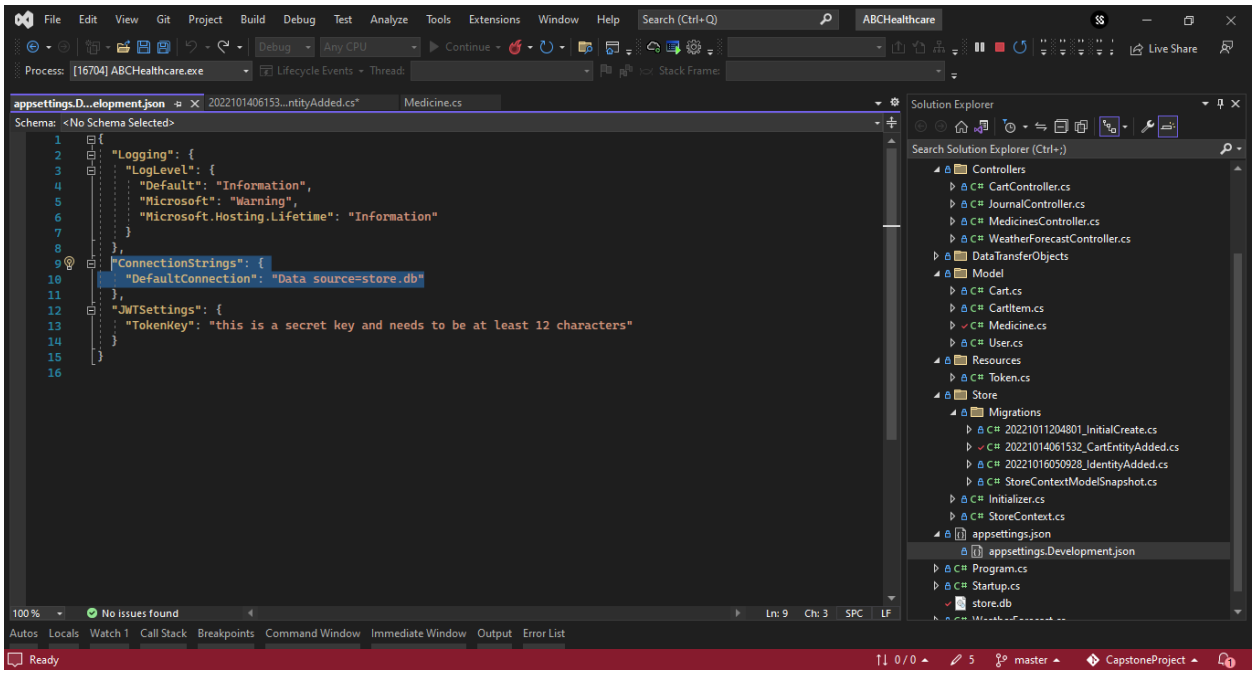


Fig. 12: appsettings.Development.json

Now, coming to the controllers, MedicineController has two ActionResut Tasks to basically fetch all the medicines, and fetching particular medicine using an ID.

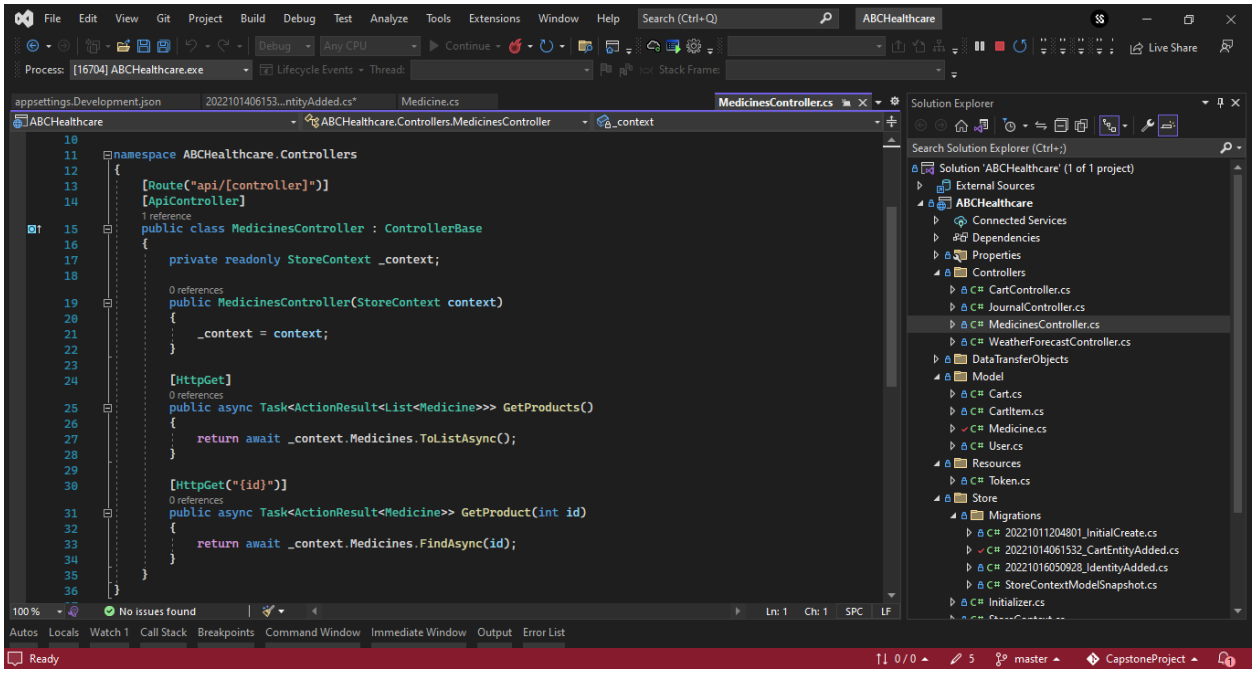
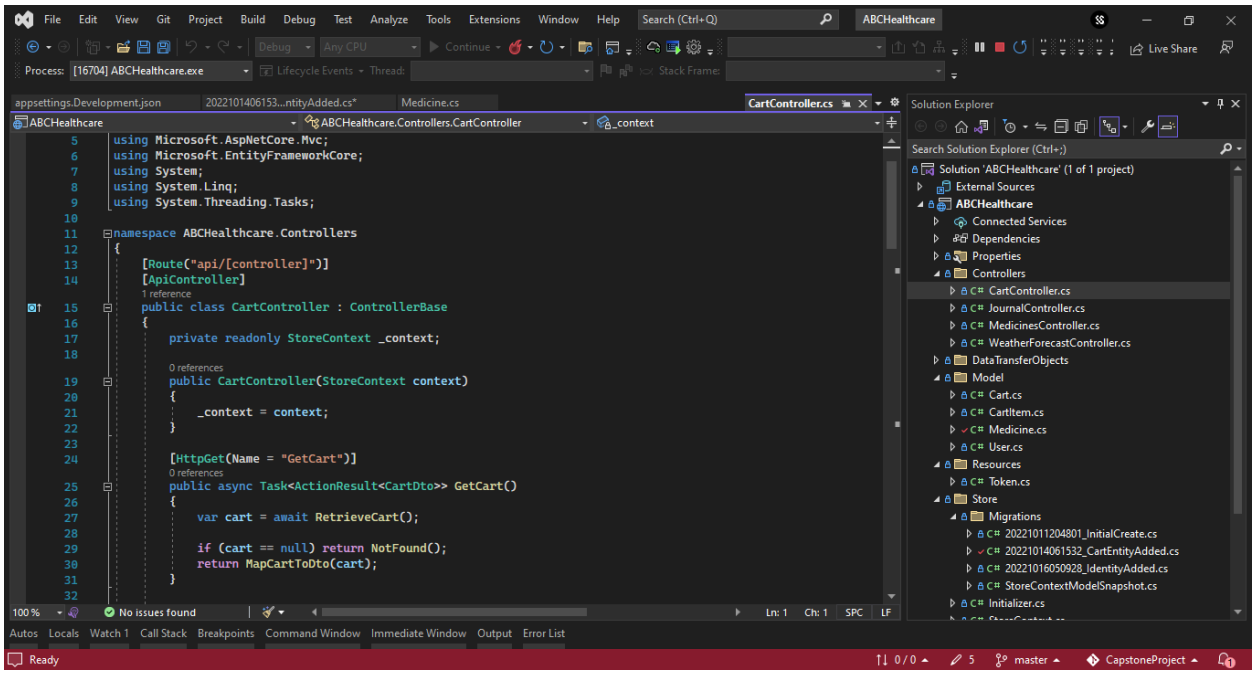


Fig. 13: Controllers/MedicineController.cs

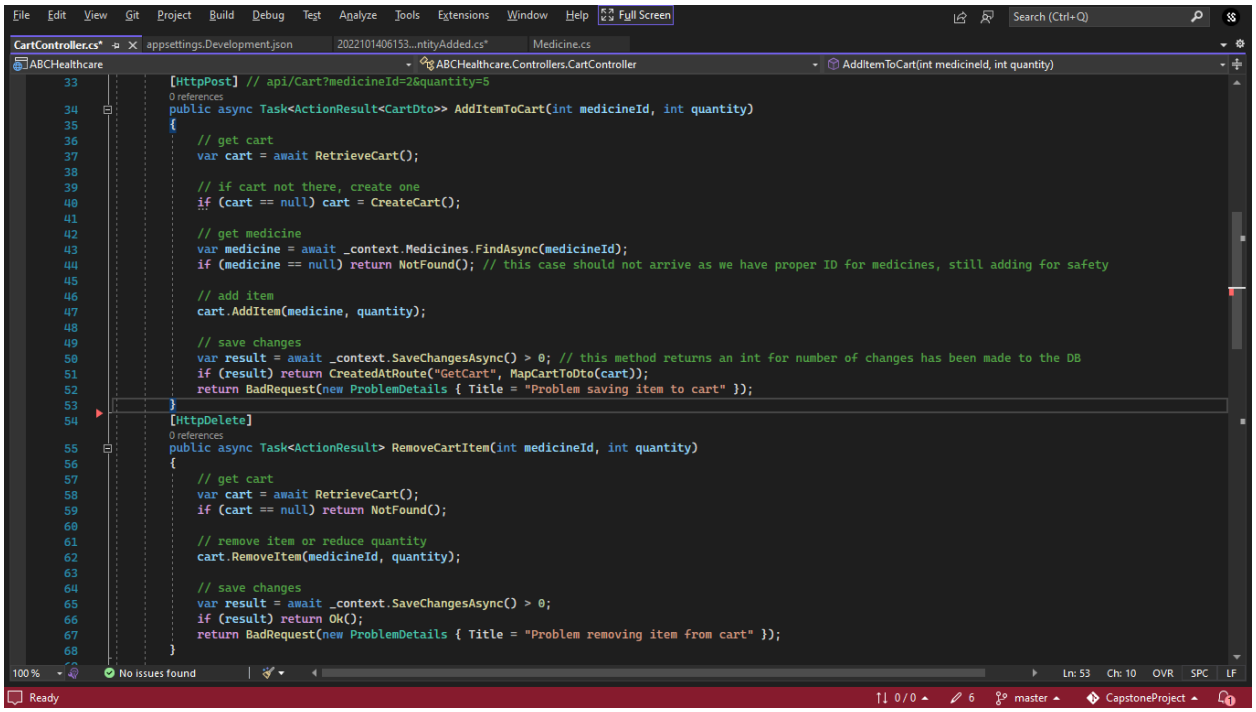
CartController gets the cart from API. If the cart doesn't exist it will just give a NotFound error



```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace ABCHealthcare.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class CartController : ControllerBase
    {
        private readonly StoreContext _context;

        0 references
        public CartController(StoreContext context)
        {
            _context = context;
        }

        [HttpGet(Name = "GetCart")]
        0 references
        public async Task<ActionResult<CartDto>> GetCart()
        {
            var cart = await RetrieveCart();

            if (cart == null) return NotFound();
            return MapCartToDto(cart);
        }
```

Fig. 14: Controllers/CartController.cs

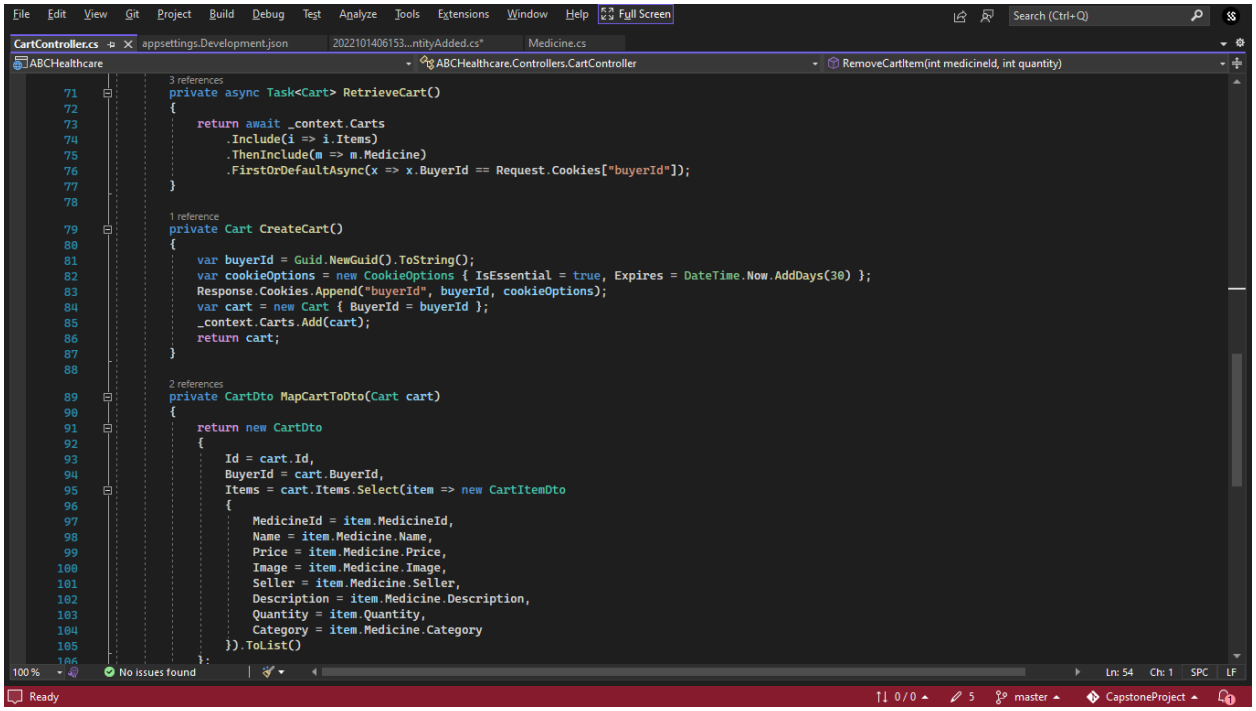
The CartController then also includes methods for adding and removing the medicines from cart.



```
        [HttpPost] // api/Cart?medicineId=2&quantity=5
        0 references
        public async Task<ActionResult<CartDto>> AddItemToCart(int medicineId, int quantity)
        {
            // get cart
            var cart = await RetrieveCart();

            // if cart not there, create one
            if (cart == null) cart = CreateCart();

            // get medicine
            var medicine = await _context.Medicines.FindAsync(medicineId);
            if (medicine == null) return NotFound(); // this case should not arrive as we have proper ID for medicines, still adding for safety

            // add item
            cart.AddItem(medicine, quantity);

            // save changes
            var result = await _context.SaveChangesAsync() > 0; // this method returns an int for number of changes has been made to the DB
            if (result) return CreatedAtRoute("GetCart", MapCartToDto(cart));
            return BadRequest(new ProblemDetails { Title = "Problem saving item to cart" });
        }
        [HttpDelete]
        0 references
        public async Task<ActionResult> RemoveCartItem(int medicineId, int quantity)
        {
            // get cart
            var cart = await RetrieveCart();
            if (cart == null) return NotFound();

            // remove item or reduce quantity
            cart.RemoveItem(medicineId, quantity);

            // save changes
            var result = await _context.SaveChangesAsync() > 0;
            if (result) return Ok();
            return BadRequest(new ProblemDetails { Title = "Problem removing item from cart" });
        }
```

Fig. 15: Controllers/CartController.cs (contd…)

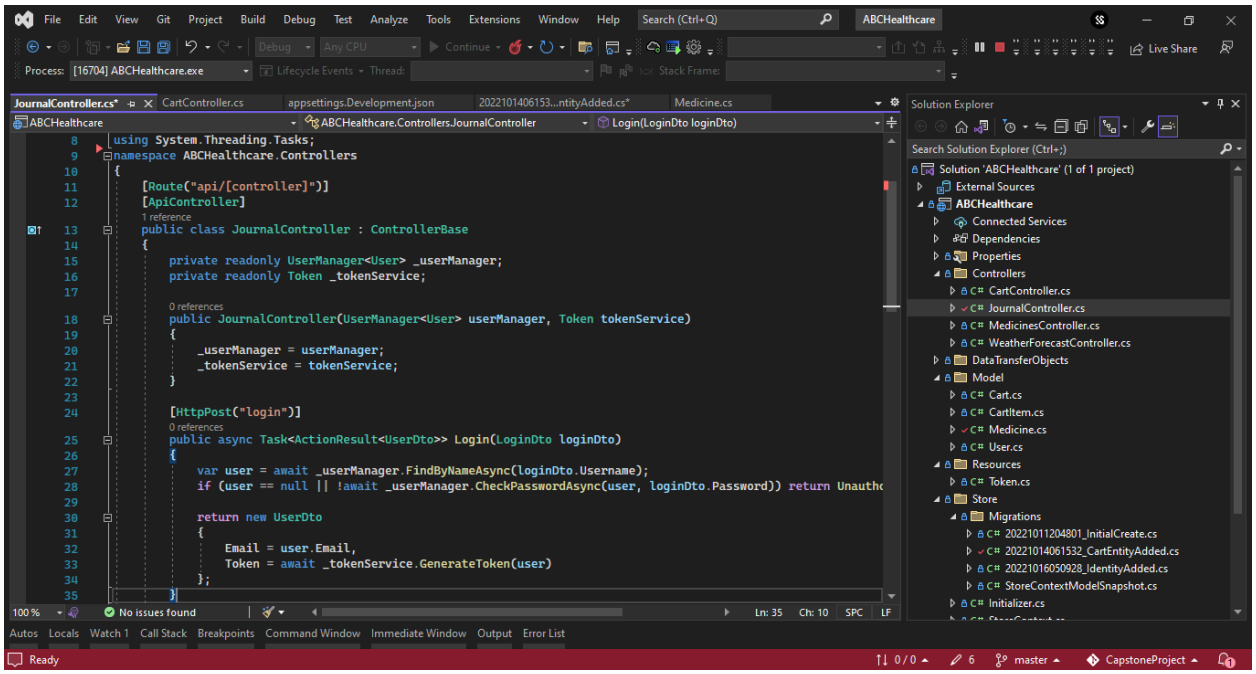
Finally, there are methods to retrieve cart if it already exists, otherwise a new cart will be created.



```
3 references
private async Task<Cart> RetrieveCart()
{
    return await _context.Carts
        .Include(i => i.Items)
        .ThenInclude(m => m.Medicine)
        .FirstOrDefaultAsync(x => x.BuyerId == Request.Cookies["buyerId"]);
}

1 reference
private Cart CreateCart()
{
    var buyerId = Guid.NewGuid().ToString();
    var cookieOptions = new CookieOptions { IsEssential = true, Expires = DateTime.Now.AddDays(30) };
    Response.Cookies.Append("buyerId", buyerId, cookieOptions);
    var cart = new Cart { BuyerId = buyerId };
    _context.Carts.Add(cart);
    return cart;
}

2 references
private CartDto MapCartToDto(Cart cart)
{
    return new CartDto
    {
        Id = cart.Id,
        BuyerId = cart.BuyerId,
        Items = cart.Items.Select(item => new CartItemDto
        {
            MedicineId = item.MedicineId,
            Name = item.Medicine.Name,
            Price = item.Medicine.Price,
            Image = item.Medicine.Image,
            Seller = item.Medicine.Seller,
            Description = item.Medicine.Description,
            Quantity = item.Quantity,
            Category = item.Medicine.Category
        }).ToList()
    };
```

Fig. 16: Controllers/CartController.cs (contd…)

The JournalController basically is a like an account manager which looks after the Login, Register and the informtion about the CurrentUser logged in the system.



```
using System.Threading.Tasks;
namespace ABCHealthcare.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class JournalController : ControllerBase
    {
        private readonly UserManager<User> _userManager;
        private readonly Token _tokenService;

        0 references
        public JournalController(UserManager<User> userManager, Token tokenService)
        {
            _userManager = userManager;
            _tokenService = tokenService;
        }

        [HttpPost("login")]
        0 references
        public async Task<ActionResult<UserDto>> Login(LoginDto loginDto)
        {
            var user = await _userManager.FindByNameAsync(loginDto.Username);
            if (user == null || !await _userManager.CheckPasswordAsync(user, loginDto.Password)) return Unauthc

            return new UserDto
            {
                Email = user.Email,
                Token = await _tokenService.GenerateToken(user)
            };
        }
```

Fig. 17: Controllers/JournalController.cs

Finally, the last part at the API end, I am using a Token to hash the password in the API.

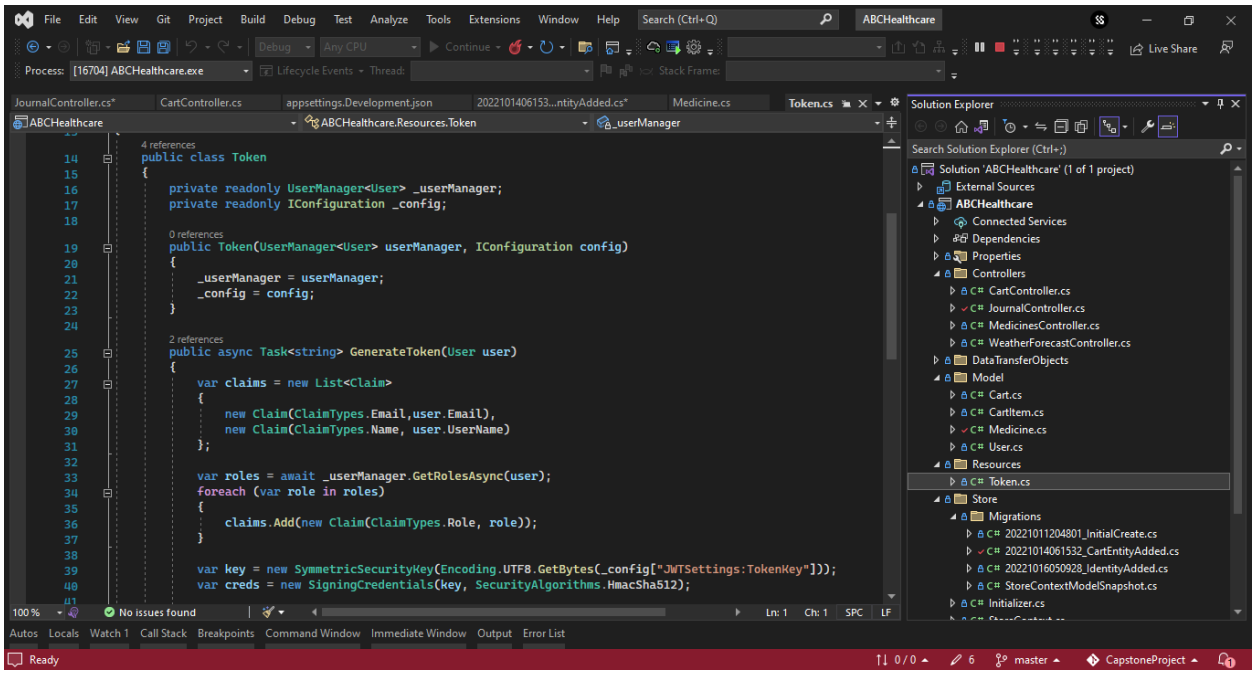


Fig. 18: Controllers/JournalController.cs

Here are the different APIs that can be seen through Swagger

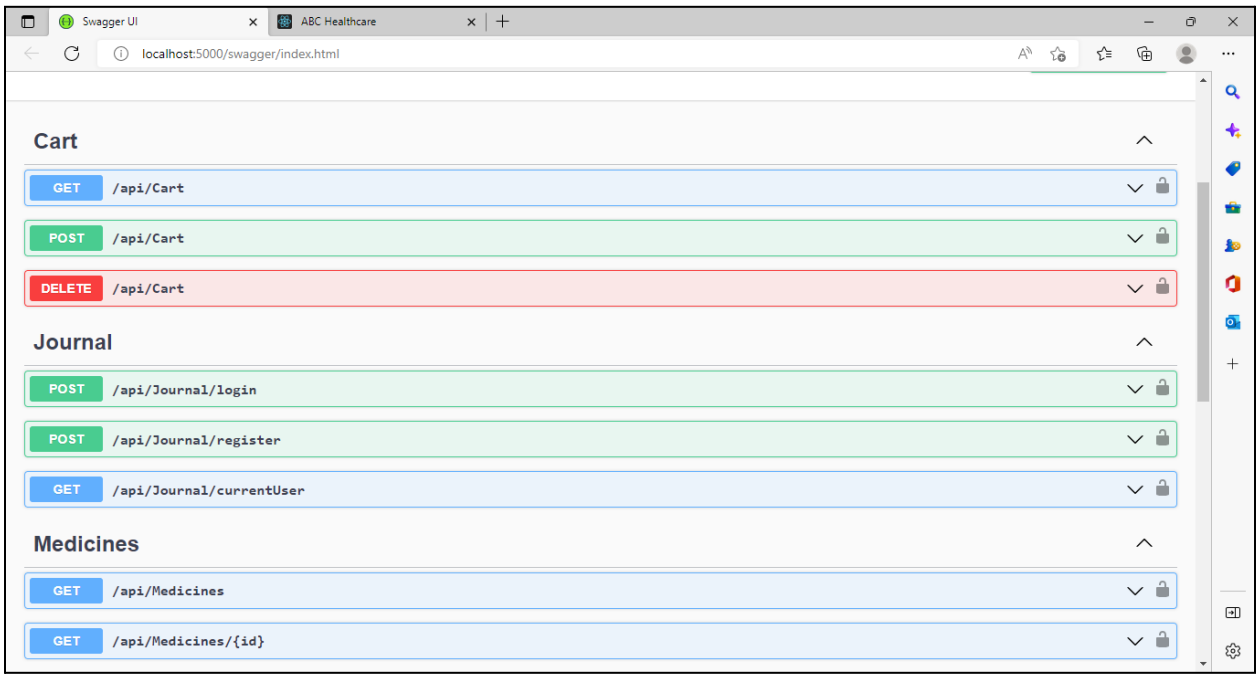


Fig. 19: Swagger/index.html

Using the GET method on Medicines API

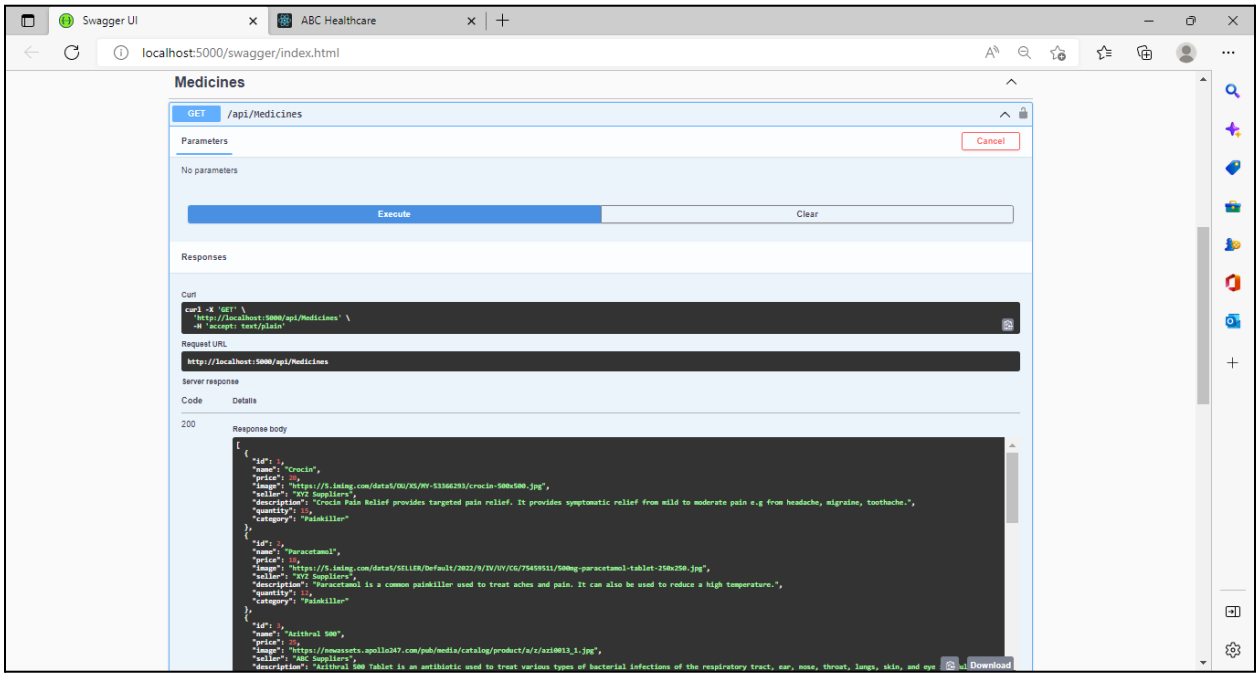


Fig. 20: Medicines GET method

Using the GET method on Medicines API for a particular ID

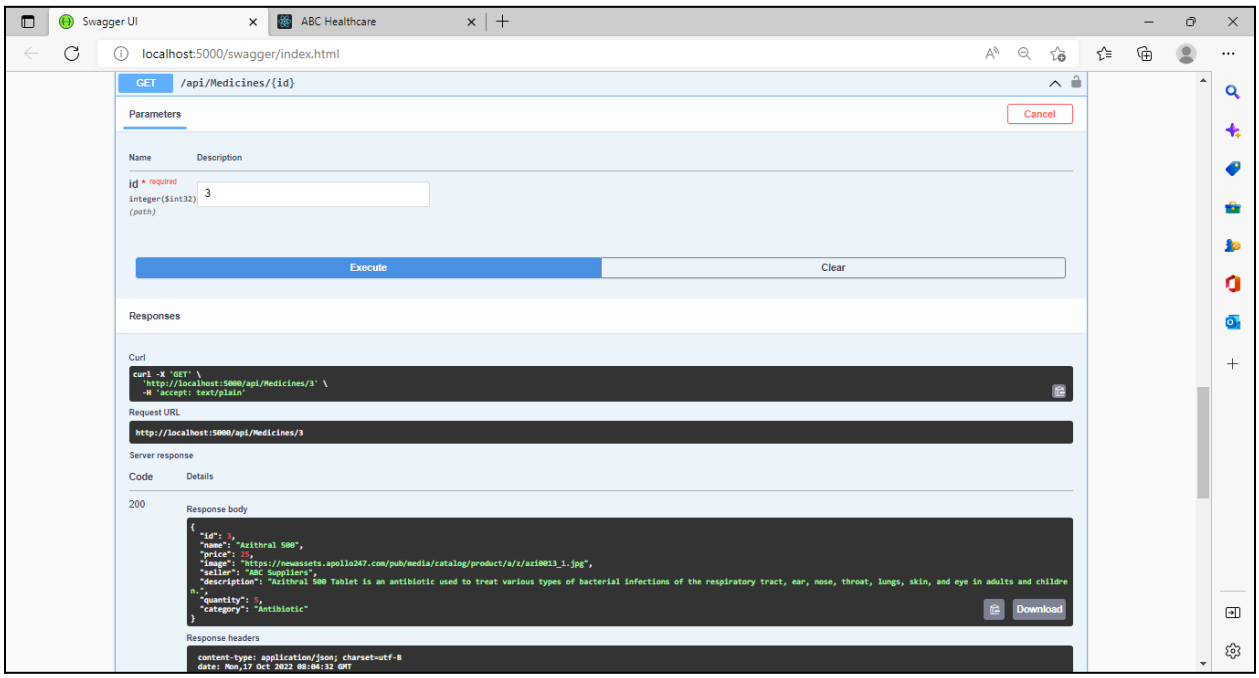


Fig. 21: Medicines GET method for a particular ID

Registering a new user into the system using a POST method

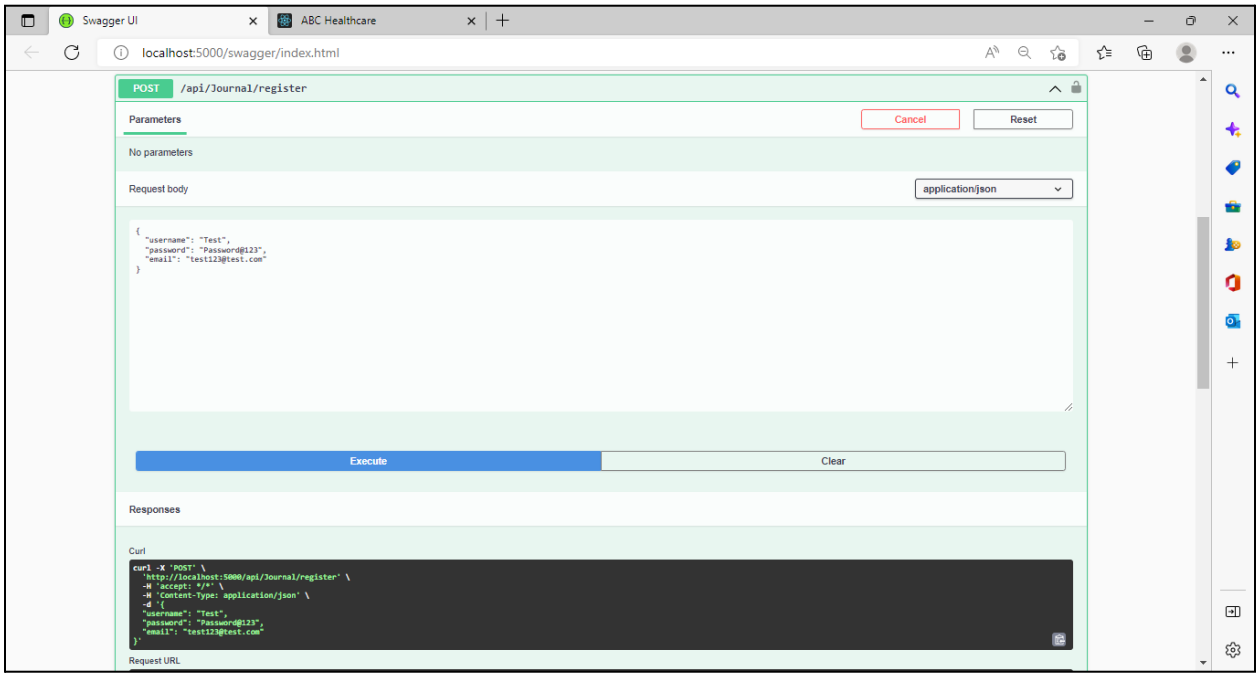
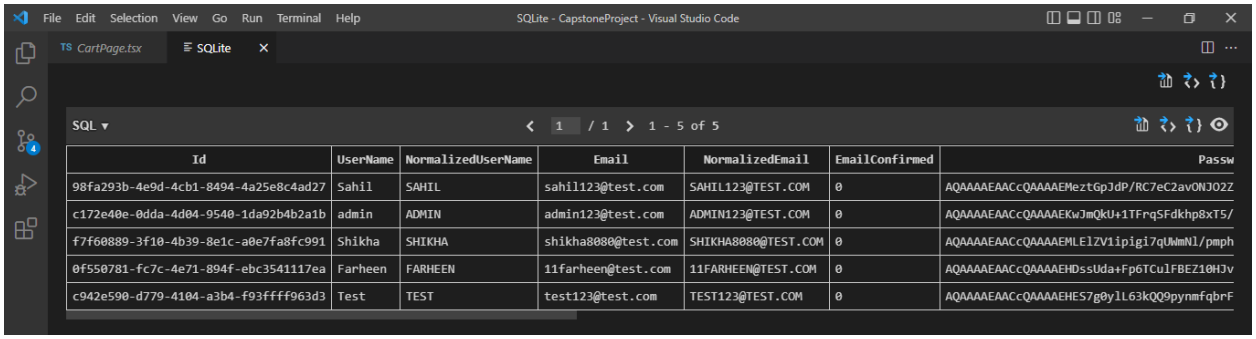


Fig. 22: Register POST method

User is successfully registered into the database



Id	UserName	NormalizedUserName	Email	NormalizedEmail	EmailConfirmed	Passw
98fa293b-4e9d-4cb1-8494-4a25e8c4ad27	Sahil	SAHIL	sahil123@test.com	SAHIL123@TEST.COM	0	AQAAAAEAACcQAAAAEMeztGpJdP/RC7eC2avONJO2Z
c172e40e-0dda-4d04-9540-1da92b4b2a1b	admin	ADMIN	admin123@test.com	ADMIN123@TEST.COM	0	AQAAAAEAACcQAAAAEKwJmQkU+1TFrqSFdkhp8xT5/
f7f60889-3f10-4b39-8e1c-a0e7fa8fc991	Shikha	SHIKHA	shikha8080@test.com	SHIKHA8080@TEST.COM	0	AQAAAAEAACcQAAAAEMLElZV1ipigi7qUWmNl/pmph
0f550781-fc7c-4e71-894f-ebc3541117ea	Farheen	FARHEEN	11farheen@test.com	11FARHEEN@TEST.COM	0	AQAAAAEAACcQAAAAEHDssUda+Fp6TCulFBEZ10HJv
c942e590-d779-4104-a3b4-f93ffff963d3	Test	TEST	test123@test.com	TEST123@TEST.COM	0	AQAAAAEAACcQAAAAEHES7g0yll63kQQ9pynmfqbrF

Fig. 23: Users table

Checking if the registered user is able to login

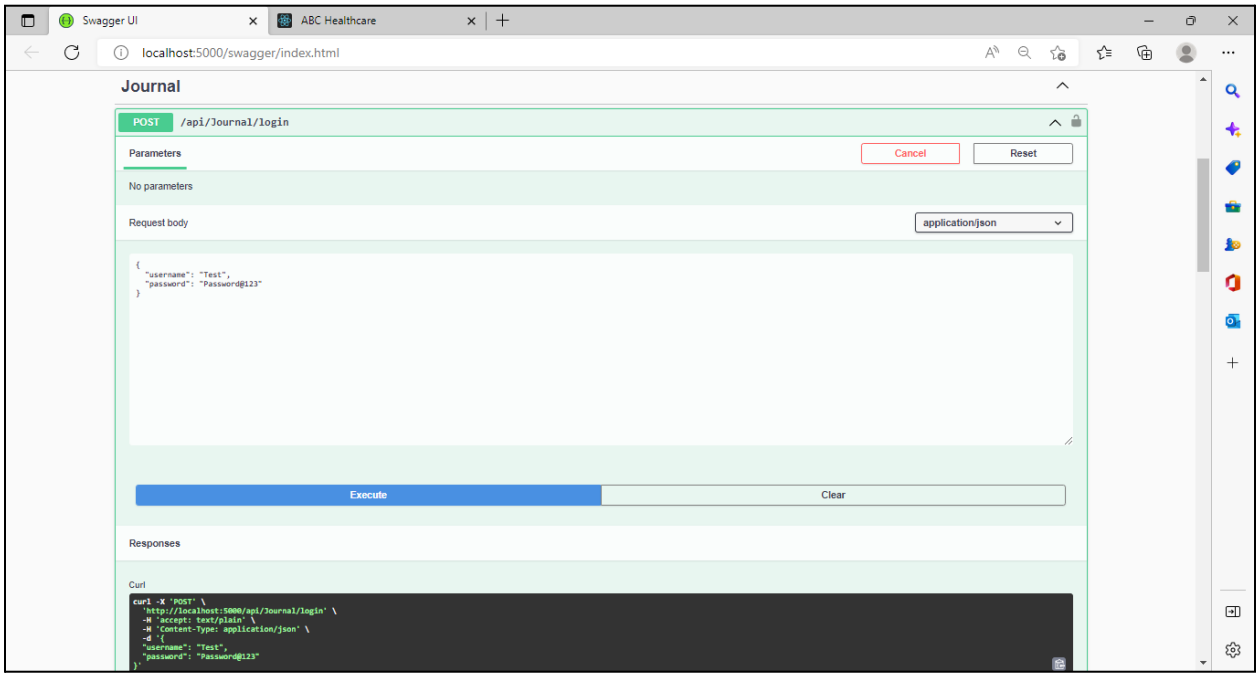


Fig. 24: Login POST method

Login successfully

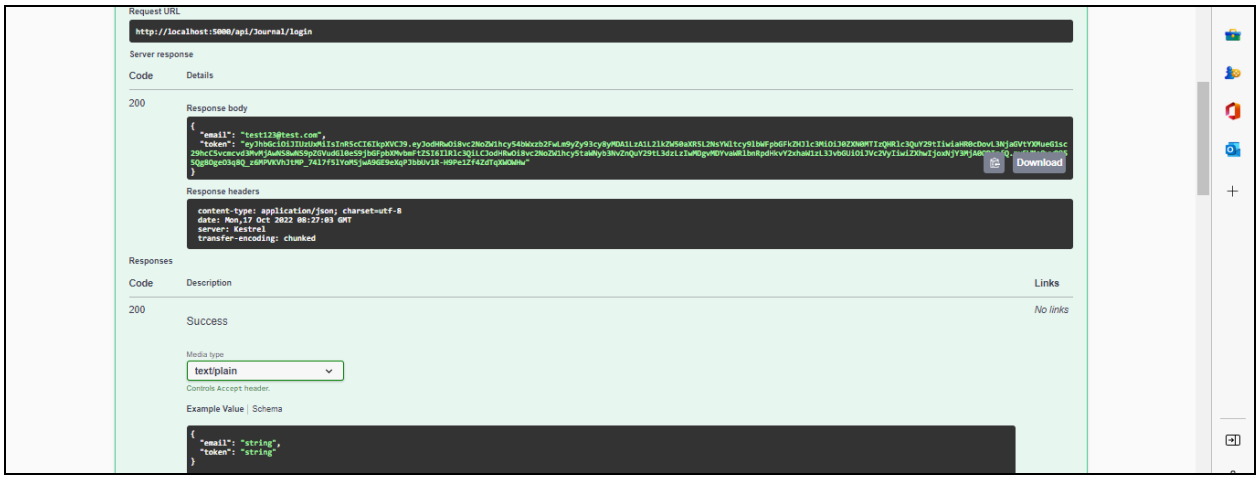


Fig. 25: Login POST method (contd…)

Existing cart with only one item inside it

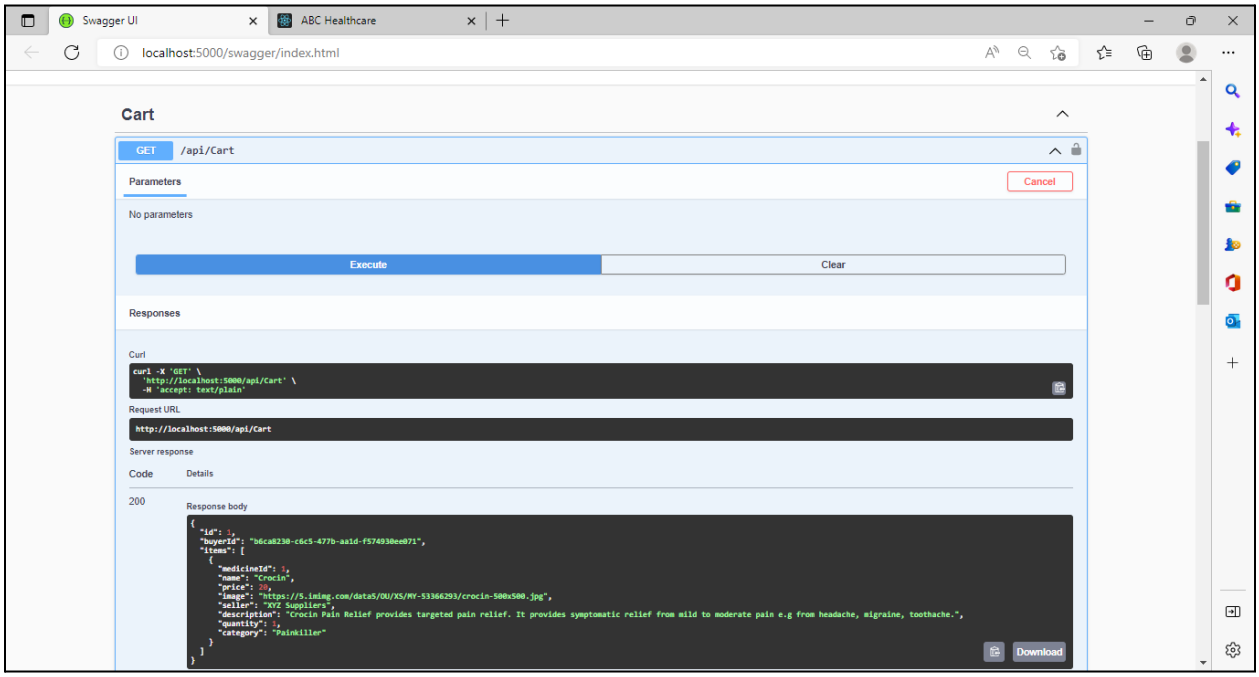


Fig. 26: Cart GET method

Adding items to the cart by specifying medicineId and quantity

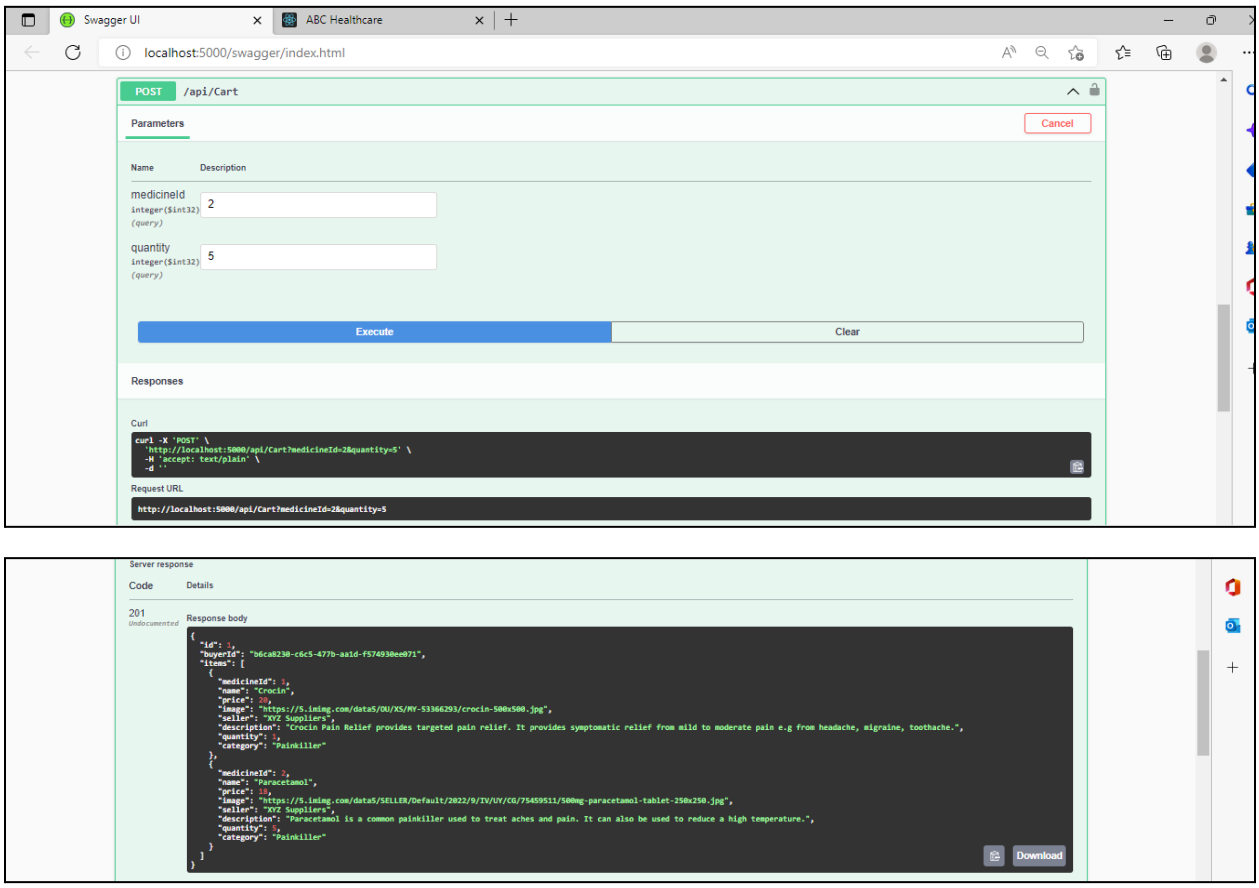


Fig. 27: Cart POST method

Removing item from the cart by providing medicineId and quantity

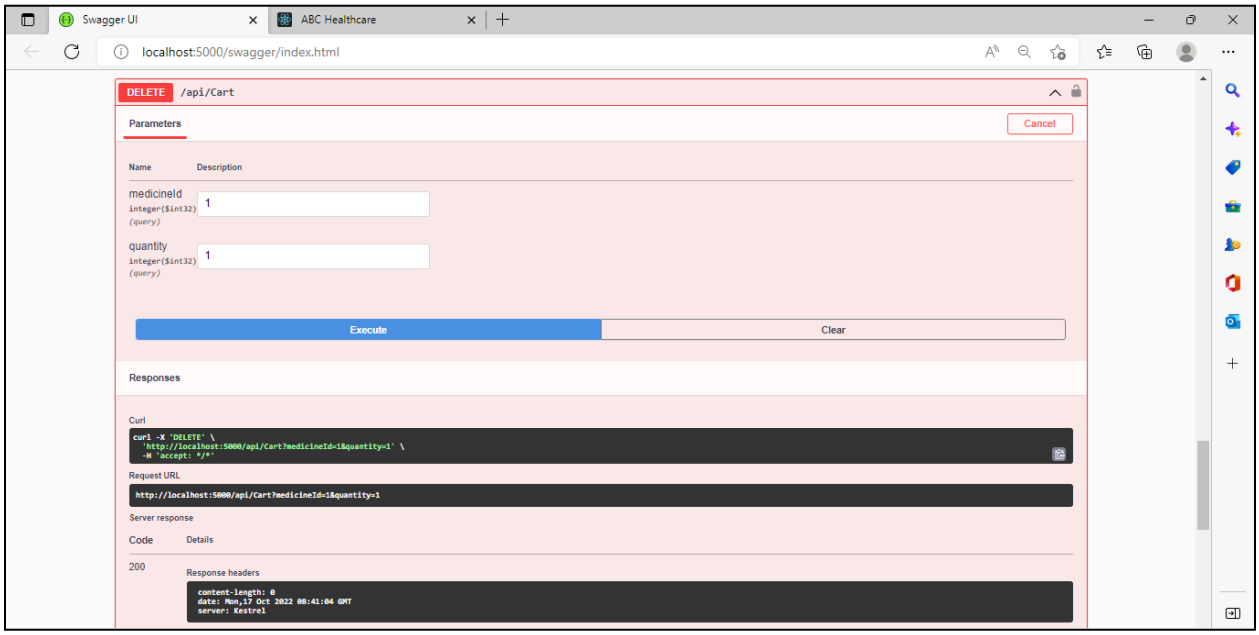


Fig. 28: Cart DELETE method

Checking the cart items once again

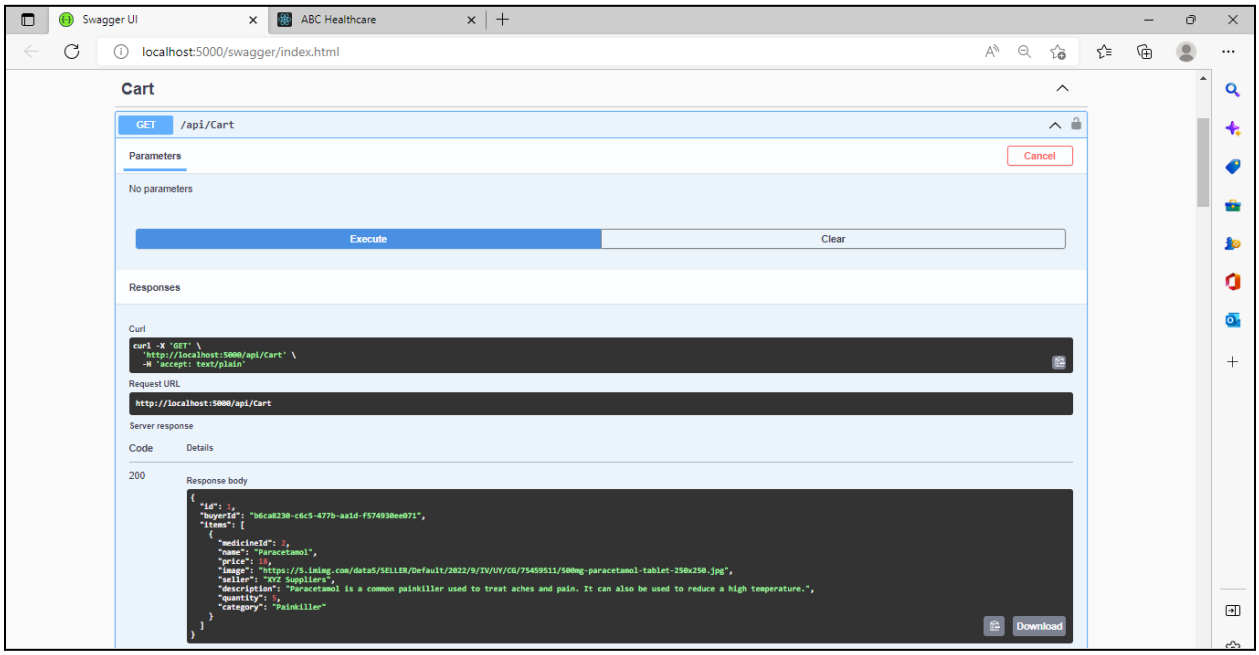


Fig. 29: Cart GET method

So, starting with the frontend part using react. This is the App component containing routes to different component pages. I have used react router here for the routing. Also, I have used a store-context basically to save the state of the cart. The useEffect() hook is used to generate a random cookie for a localhost session, and then save it as buyerId. The cart is then generated (or say 'set') as setCart for this unique buyerId.

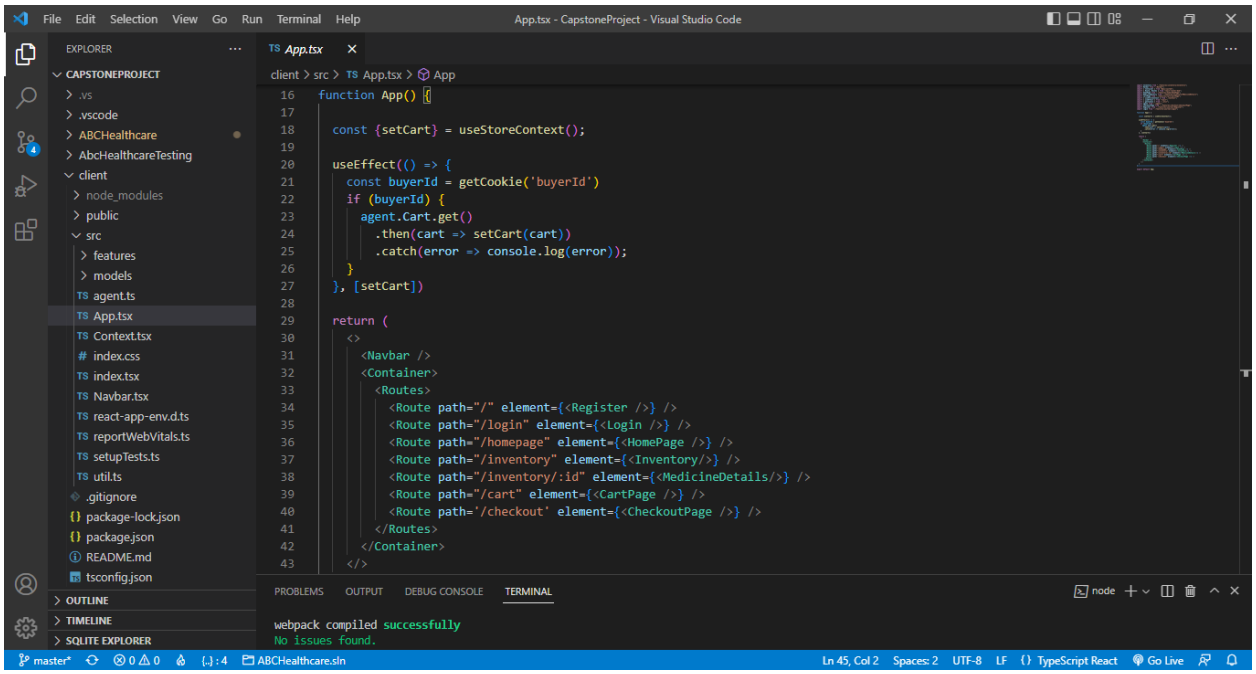


Fig. 30: App component

The StoreContext as mentioned above is basically used to save the cart state. Also, removeItem method is defined here which has the logic to reduce the quantity of medicines when items are removed from the cart.

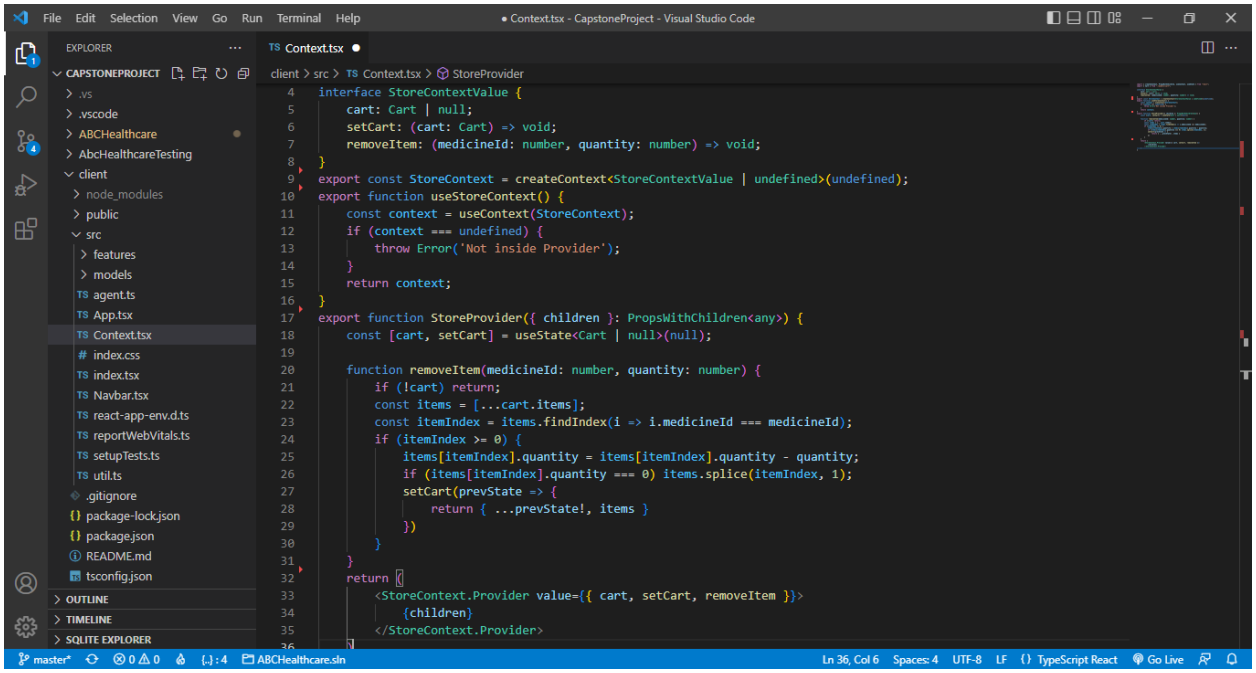


Fig. 31: Context.tsx

The agent.ts basically contains the base url for the localhost API which is used with ‘Axios’. A common ‘requests’ object is then used with the response data from axios. Different methods are then created using this request. The Inventory object contains a list, using the ‘get’ method to access the medicines API, also the details property is created to access particular medicines using the IDs. Cart object similarly has a get property to access the cart API; It also consists methods for adding and removing items from the cart. The Journal object contains ‘post’ method links to the login and register APIs.

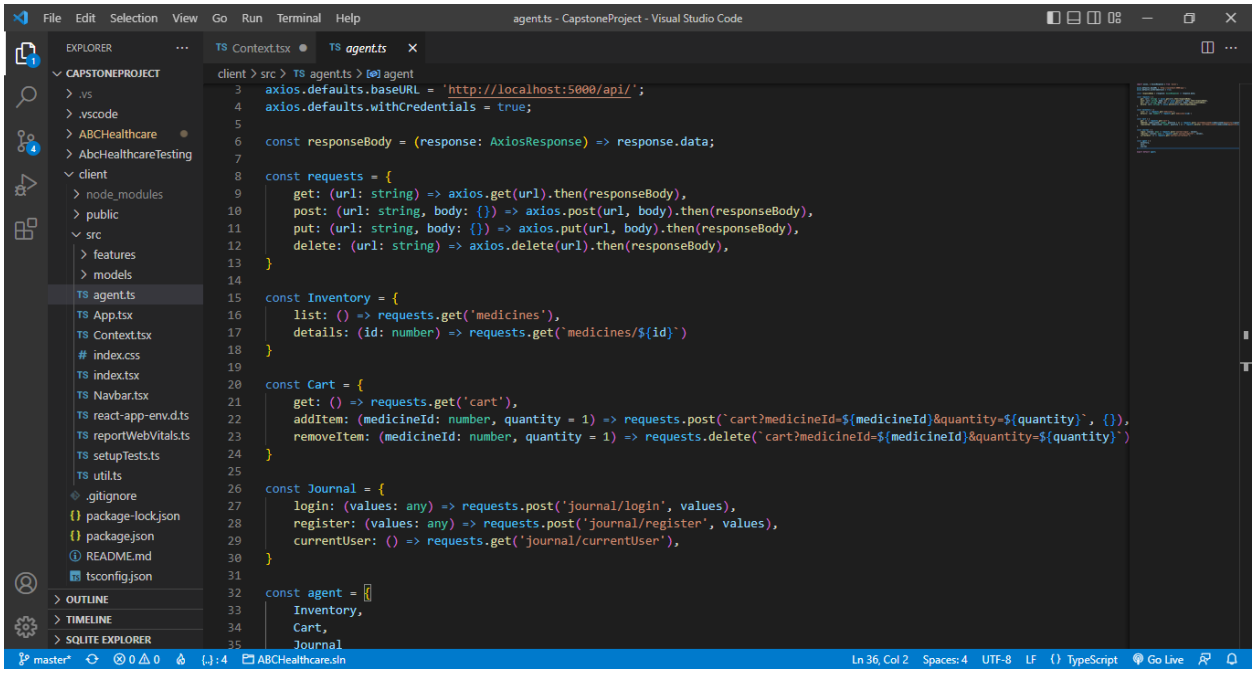


Fig. 32: agent.ts

Navbar component contains links to the Homepage, Inventory and the Cart

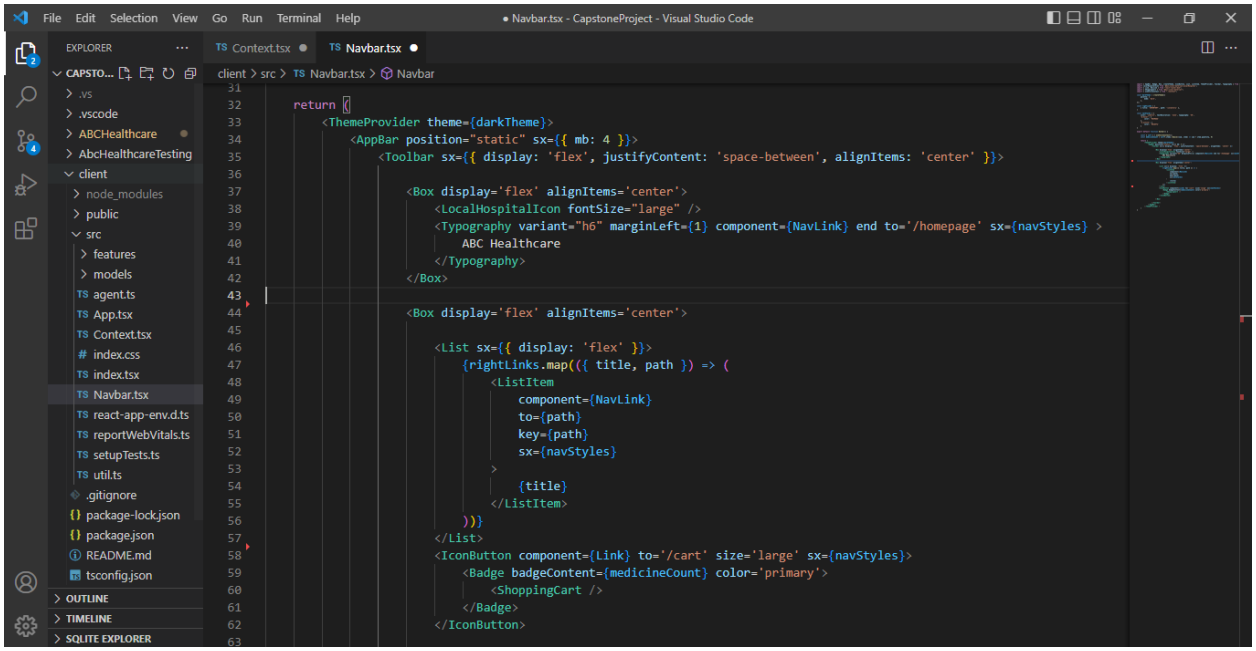


Fig. 33: Navbar component

For rendoring the App component, here we have to use BrowserRouter component from the react-router-dom so as to enable react routing. Also, the StoreProvider component is used to provide the store and the states (cart, setCart, removeItem).

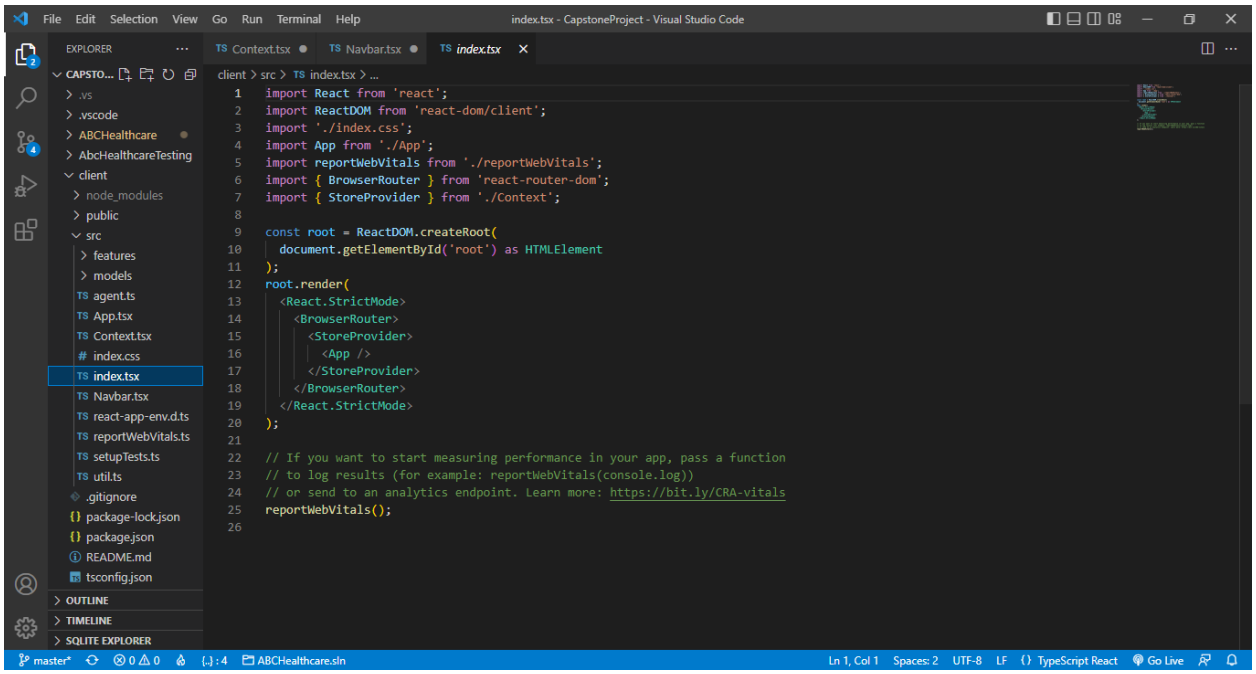


Fig. 34: index.tsx

As you can see, these are the interfaces I have used with the same properties as the API models.

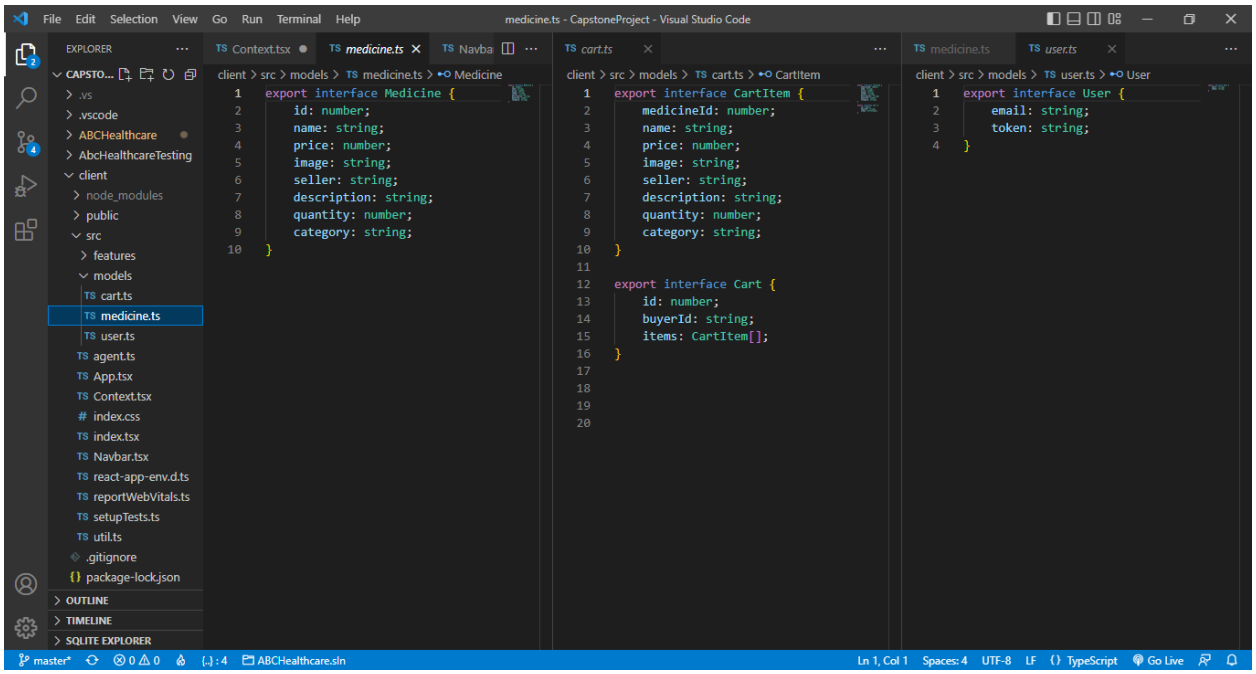


Fig. 35: medicine.ts, cart.ts, user.ts

Moving on to the Register feature, which is actually where the website begins. Here, I have used the useForm() hook which allows error handling for the input fields and eventually handles the submission of the form data into the API.

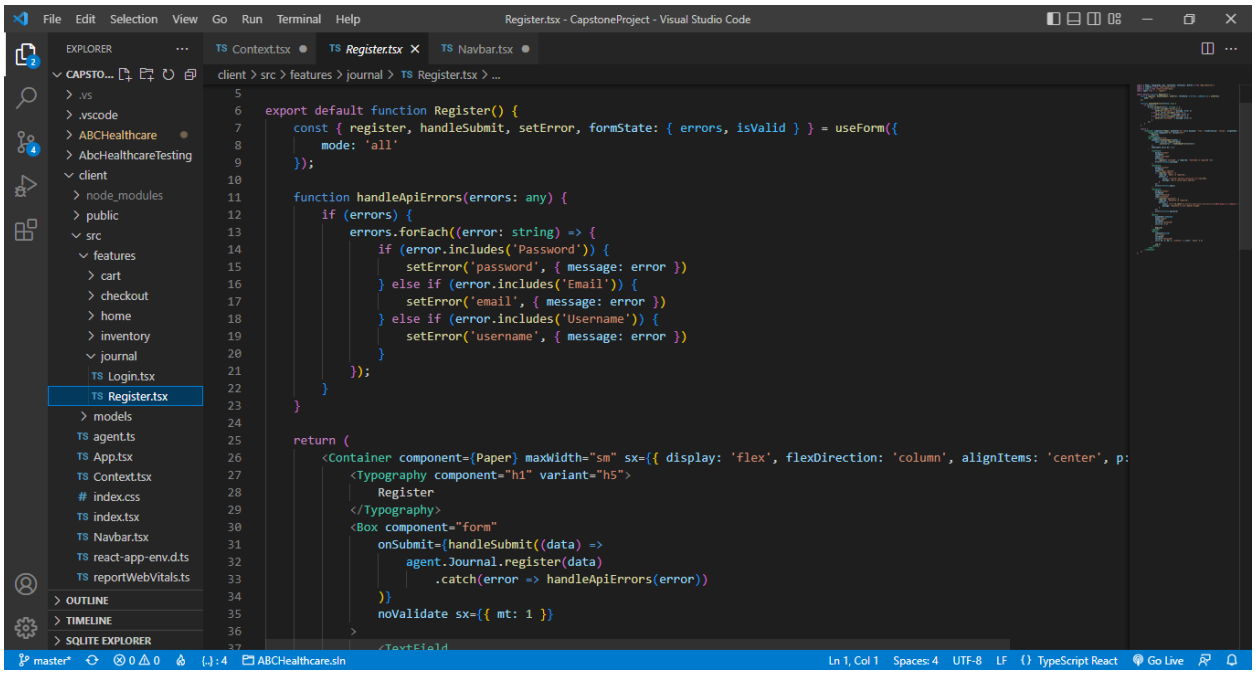


Fig. 36: Register.tsx

I have used RegEx for validating the inout fields for email and password

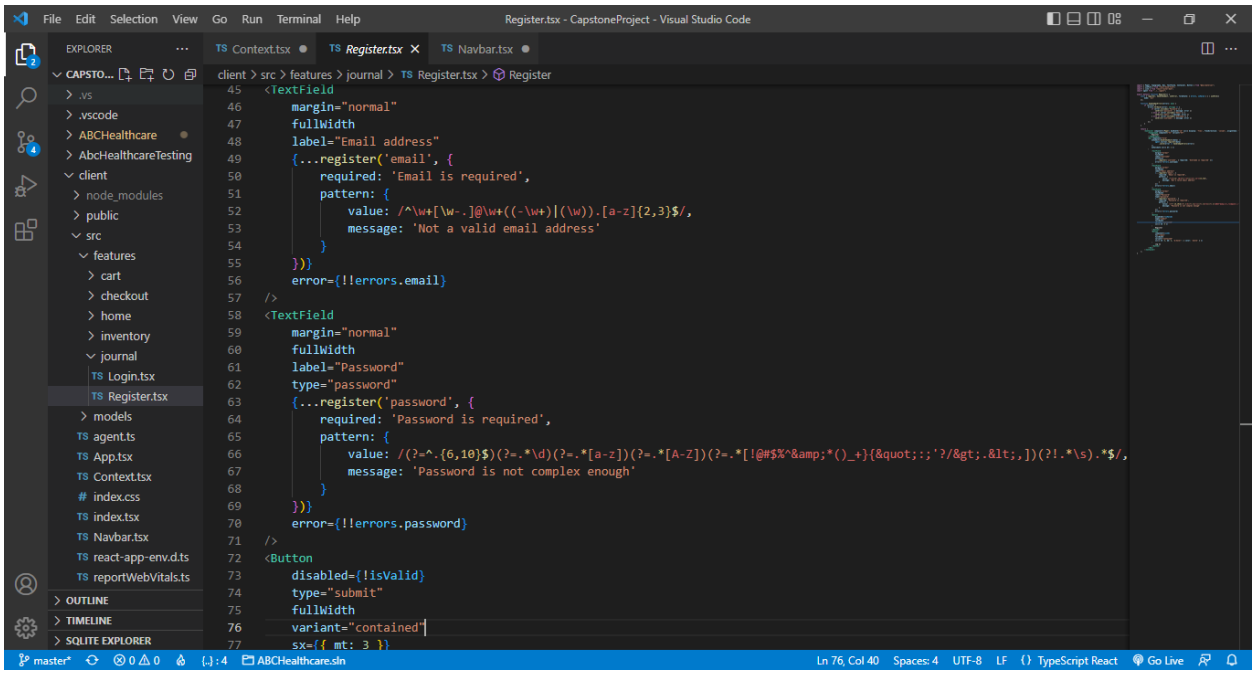


Fig. 37: Register.tsx (contd…)

The Login form also uses the useForm() hook for handling. This helps in disabling the submit button if the entries for the input fields are left blank.

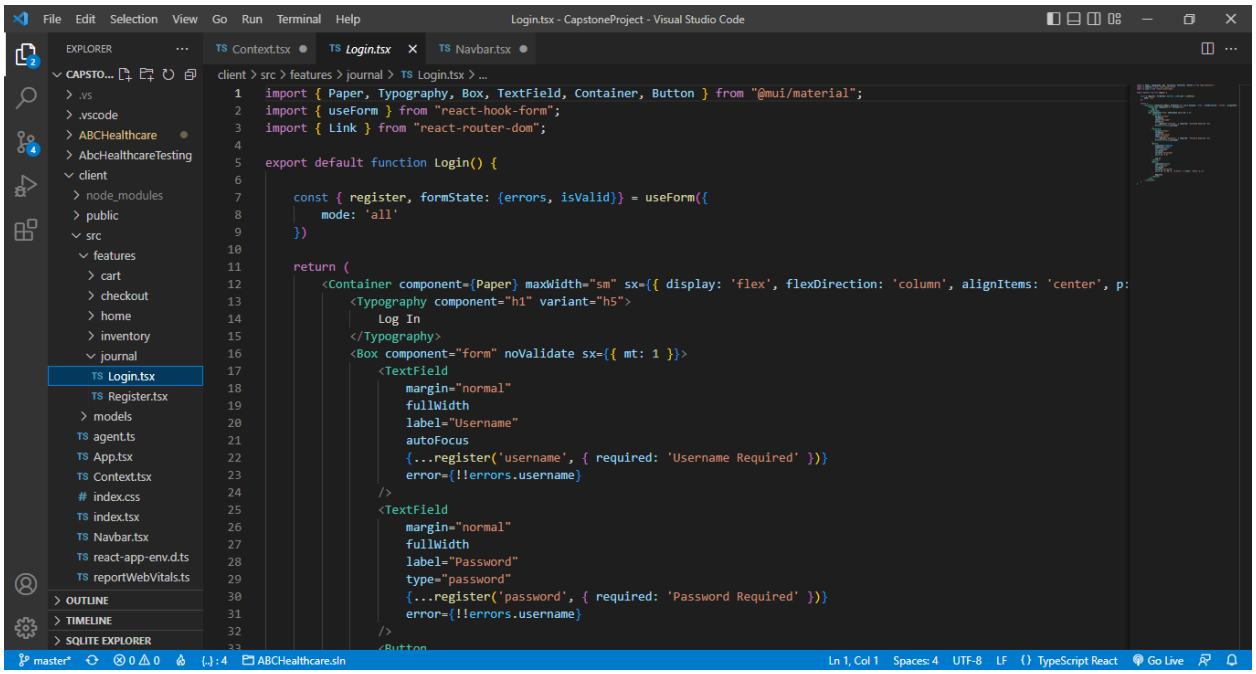


Fig. 38: Login.tsx

This is the entry point for the website. Here, we can see that validation is done. If the Username is kept empty, then there will be an error. And, if again the Email address does not match the RegEx pattern of a valid email, then there will be an error, also if the password is kept simple (without capital letters, small letters, numbers, special characters, and min length of 6) like shown below, again there will be an error.

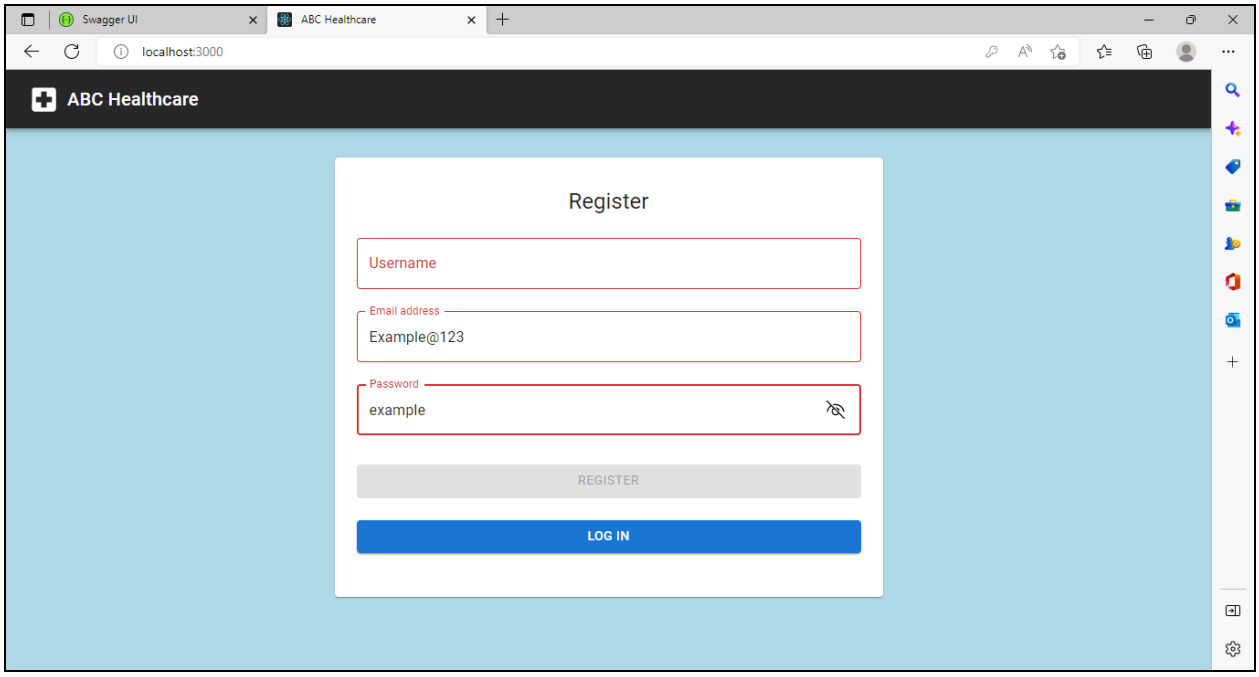


Fig. 39: Register Page

When we enter the values properly in the input fields, the register submit button enables, and now we can click on it. Also, this Register page contains a link to the Login page if the user is already registered.

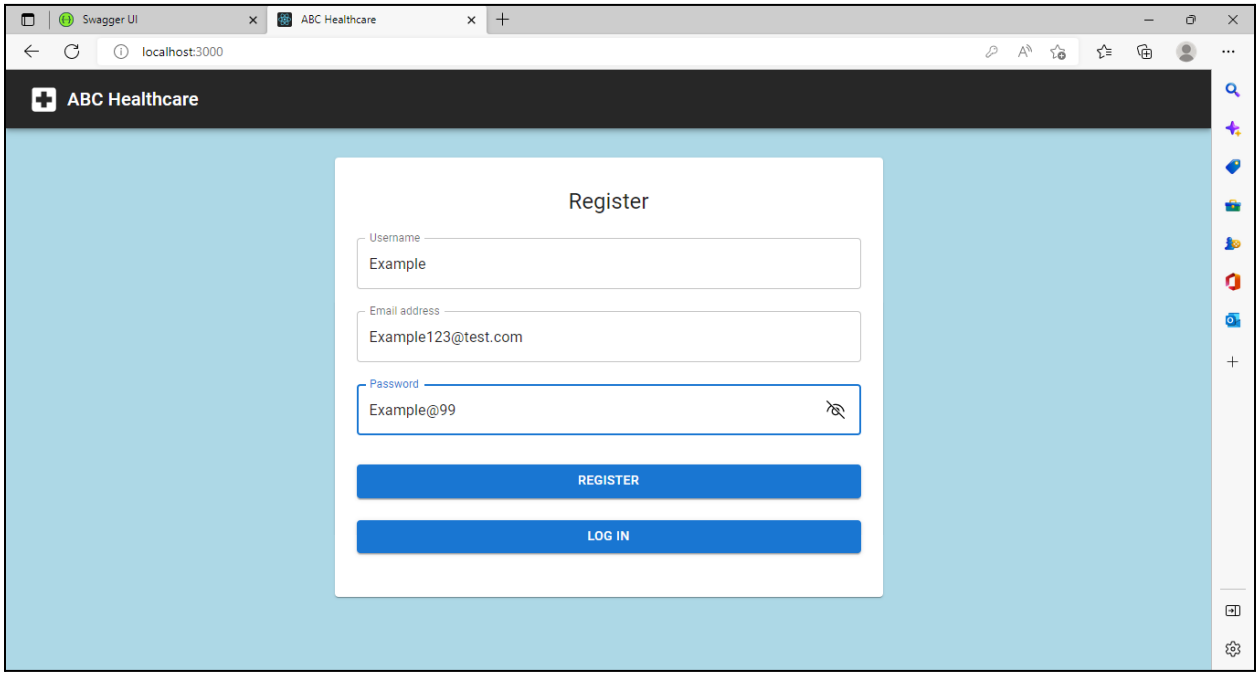


Fig. 40: Register Page (contd…)

The newly registered values are successfully stored into the database!

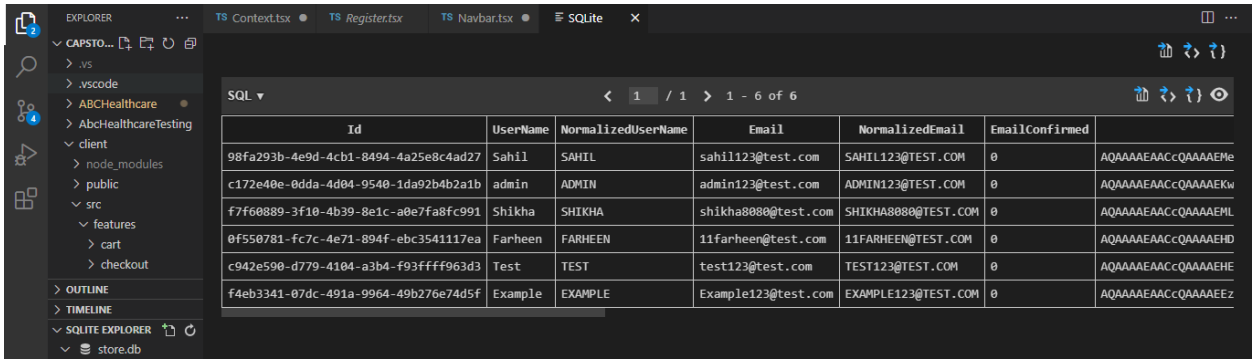


Fig. 41: Users table

Similar kind of validation is done for the Log-In form, where empty values cannot be submitted. A register button is used, which redirects to the Register page, if the user has not registered yet.

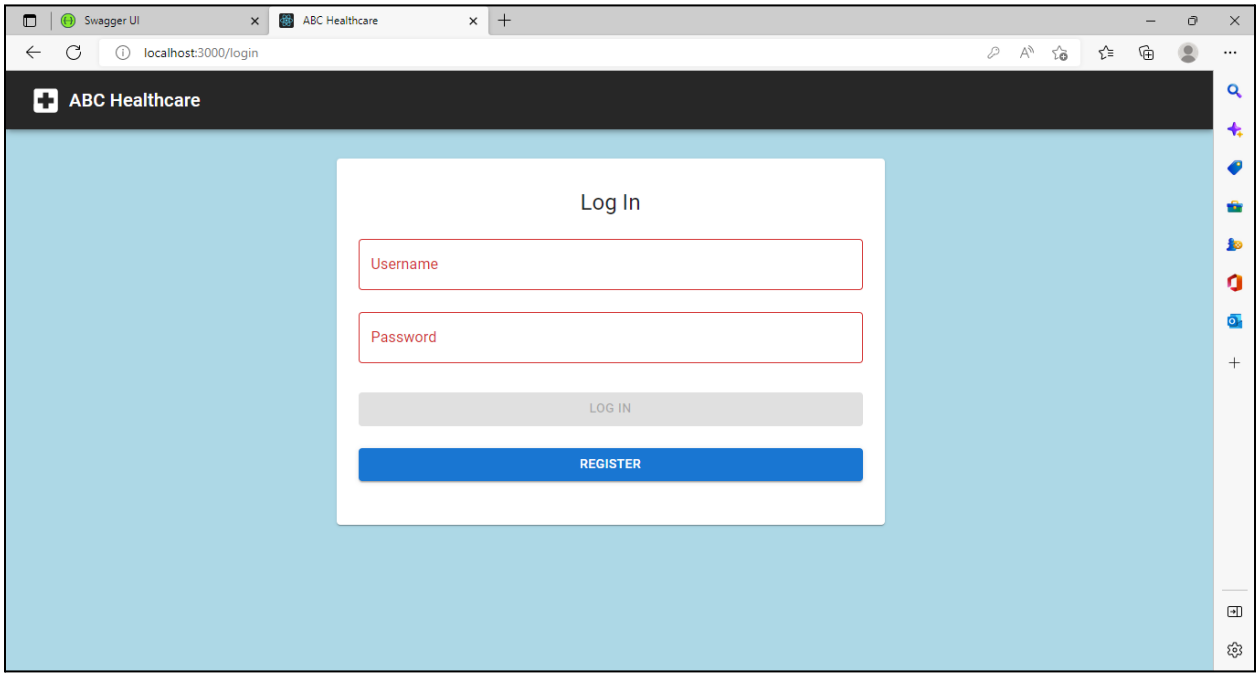


Fig. 42: Login Page

Now, logging in with correct values. Once the values match the database, the user will be redirected to the Homepage of the website.

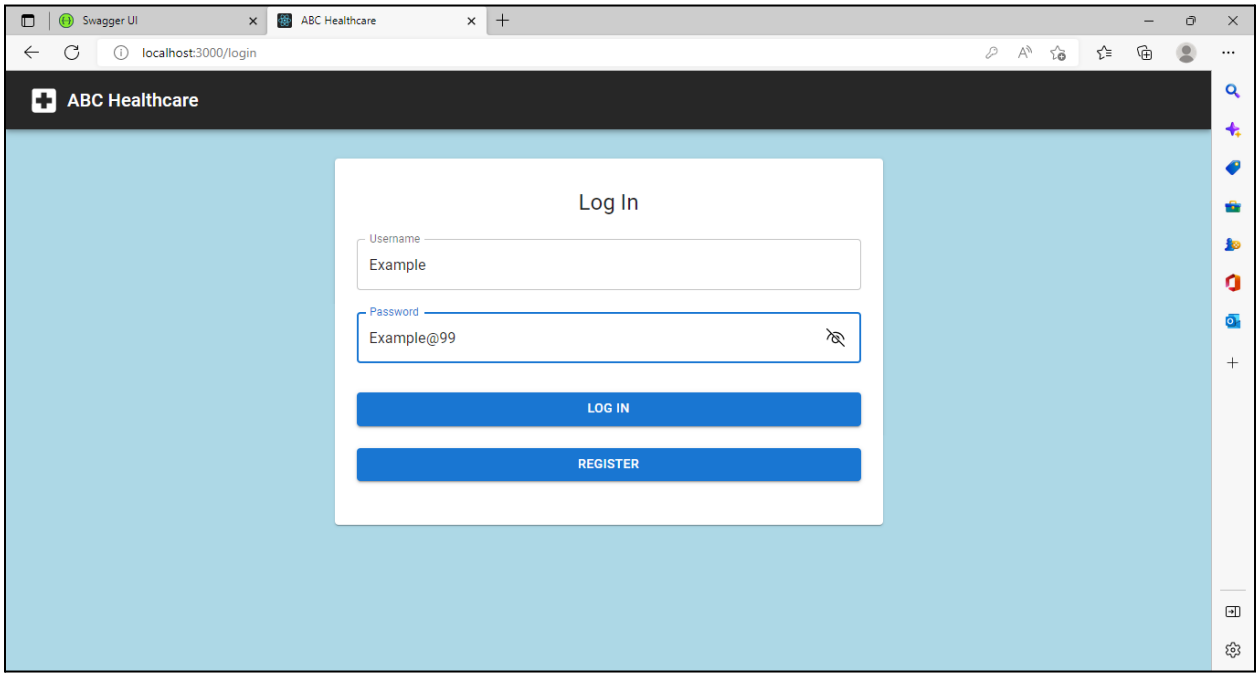


Fig. 43: Login Page (contd…)

After successful login, the user is navigated to the Homepage of the website.

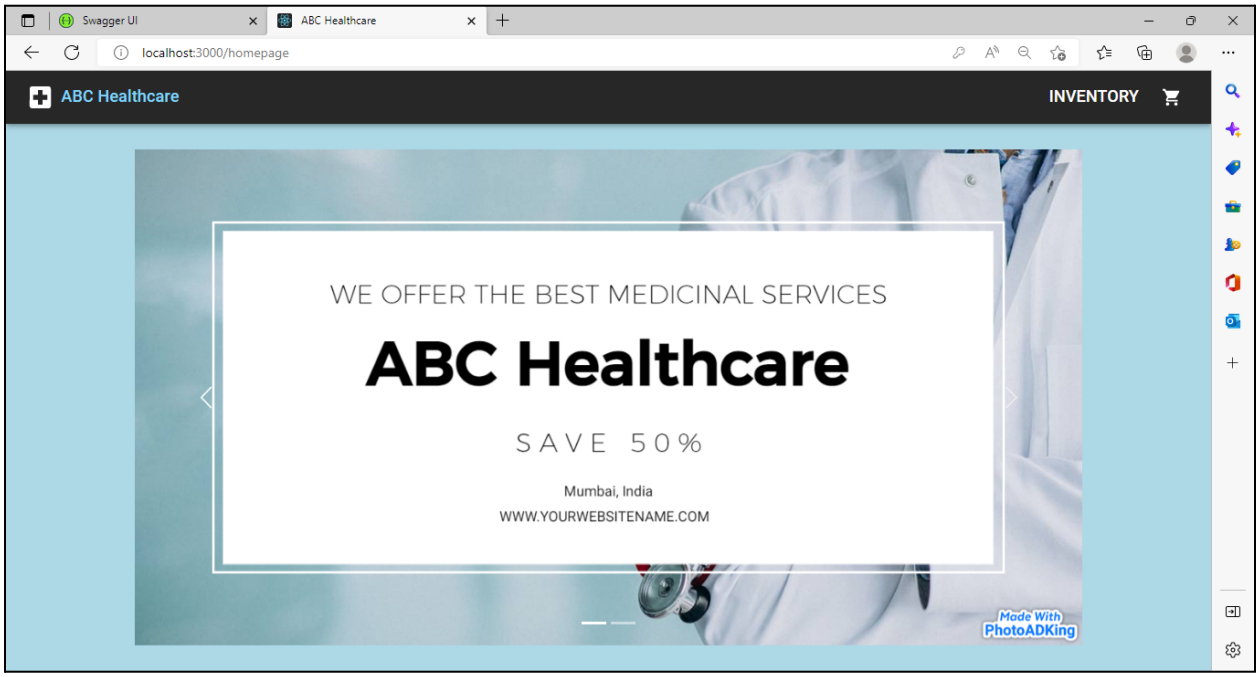


Fig. 44: HomePage

A react carousel is added on the homepage.

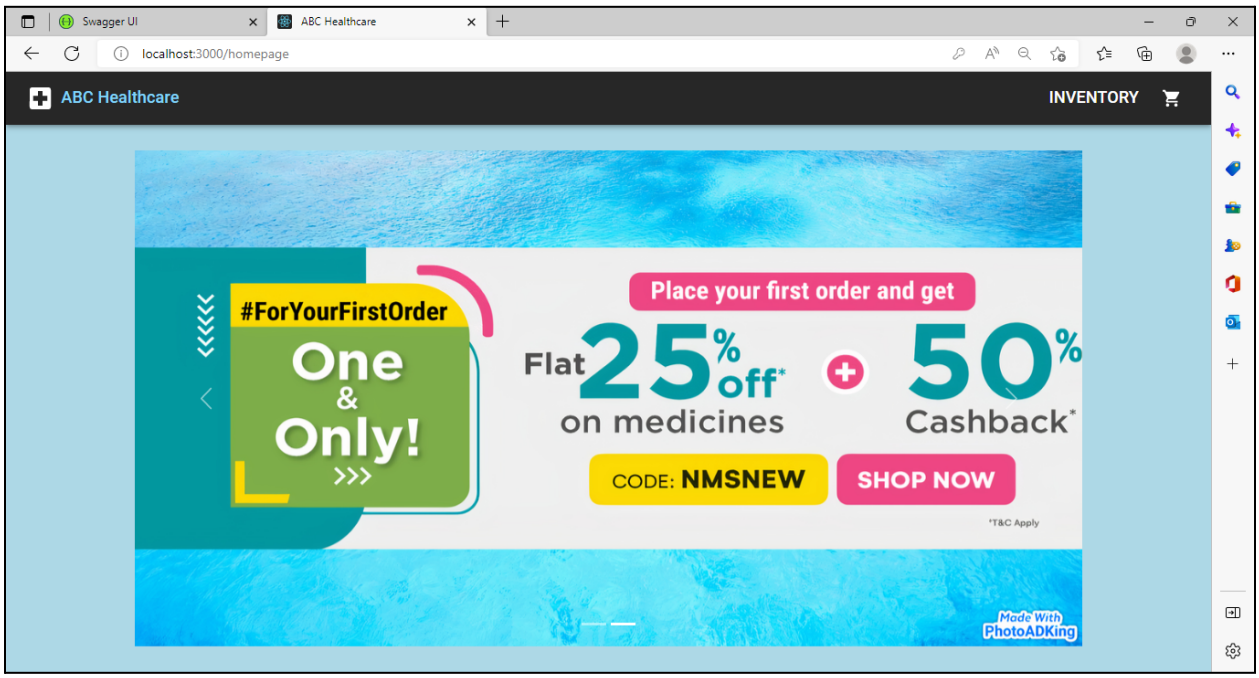


Fig. 45: HomePage (contd…)

Following is the code snippet for the above mentioned carousel in the Homepage

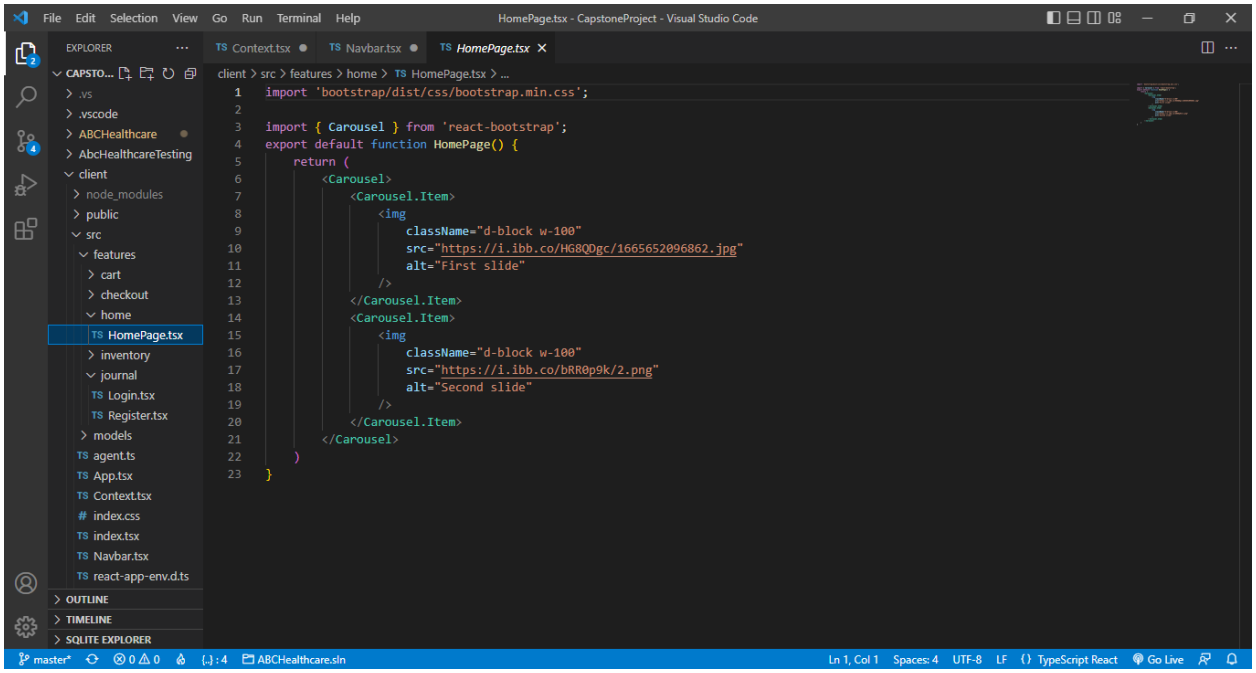


Fig. 46: HomePage.tsx

The user can navigate to the Inventory from the navigation bar. Once on the Inventory page, the user is able to see the different medicines available in the database. The user can either add them to the cart, or view the product details.

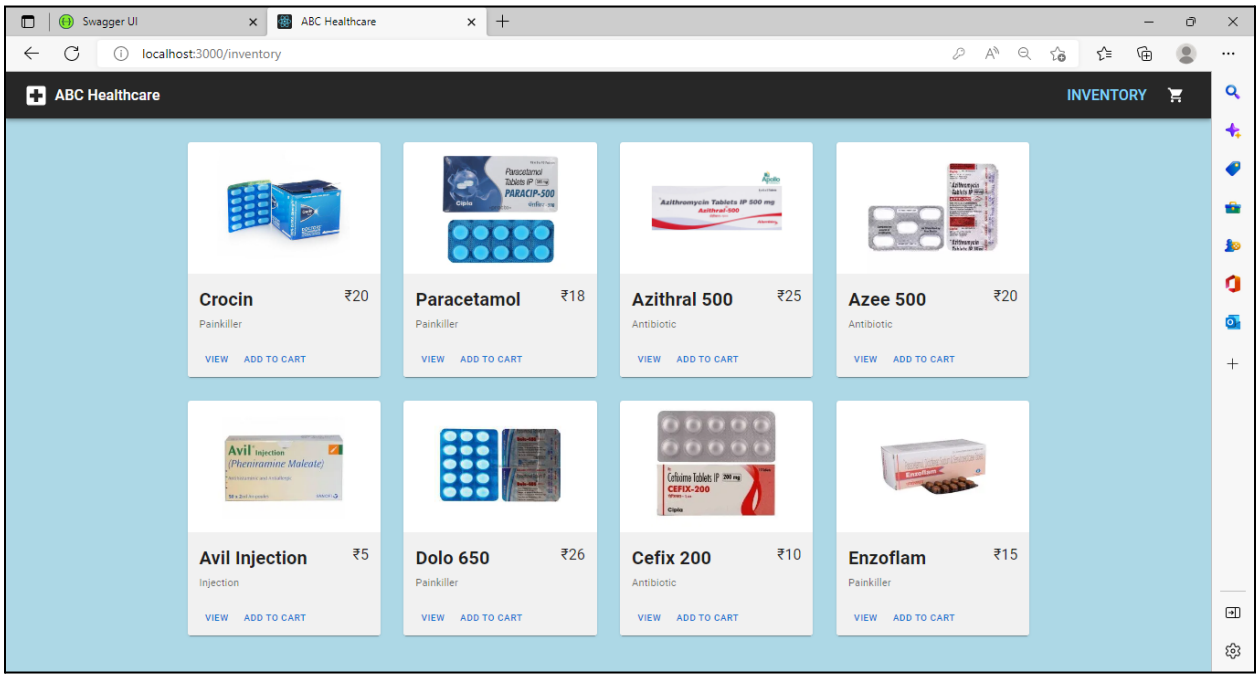


Fig. 47: Inventory Page

MedCard component contains the cards for the medicines shown on the Inventory page. Here, I have created a handleAddItem() method, which basically adds the item to the Cart using the agent.ts axios connection to the API as mentioned earlier.

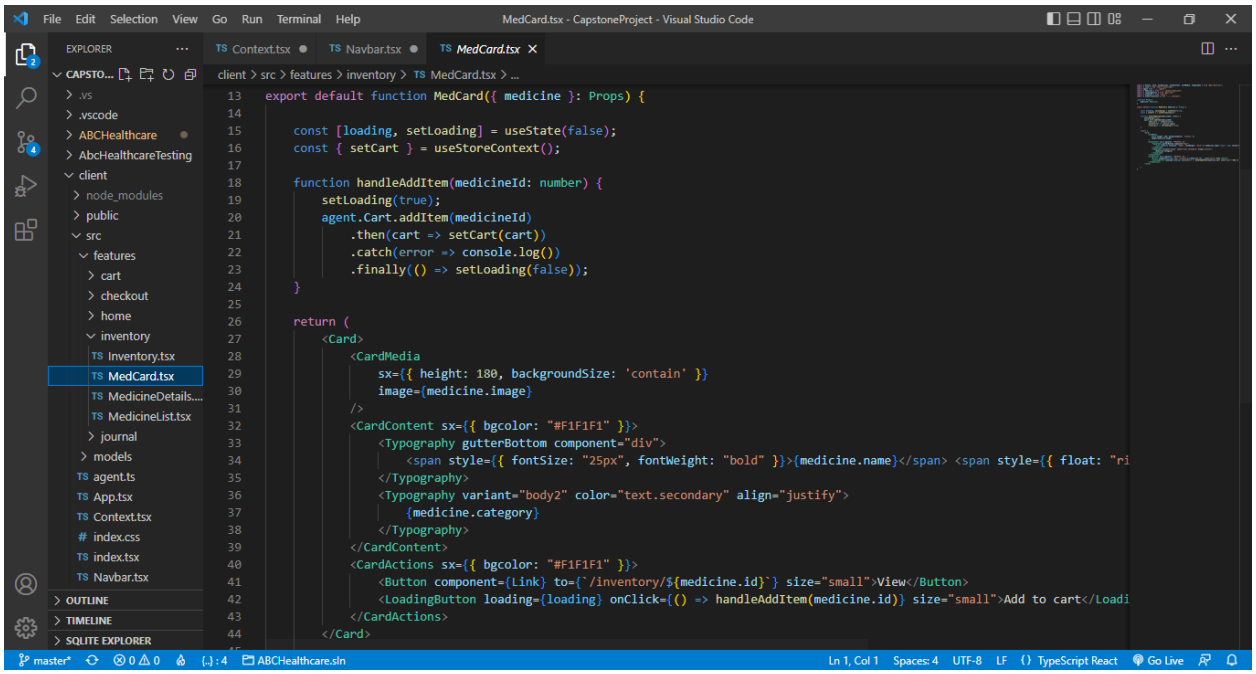


Fig. 47: inventory/MedCard.tsx

MedicineDetail component contains a table for displaying the details of each medicine when the user clicks on 'View' for any medicine on the Inventory page.

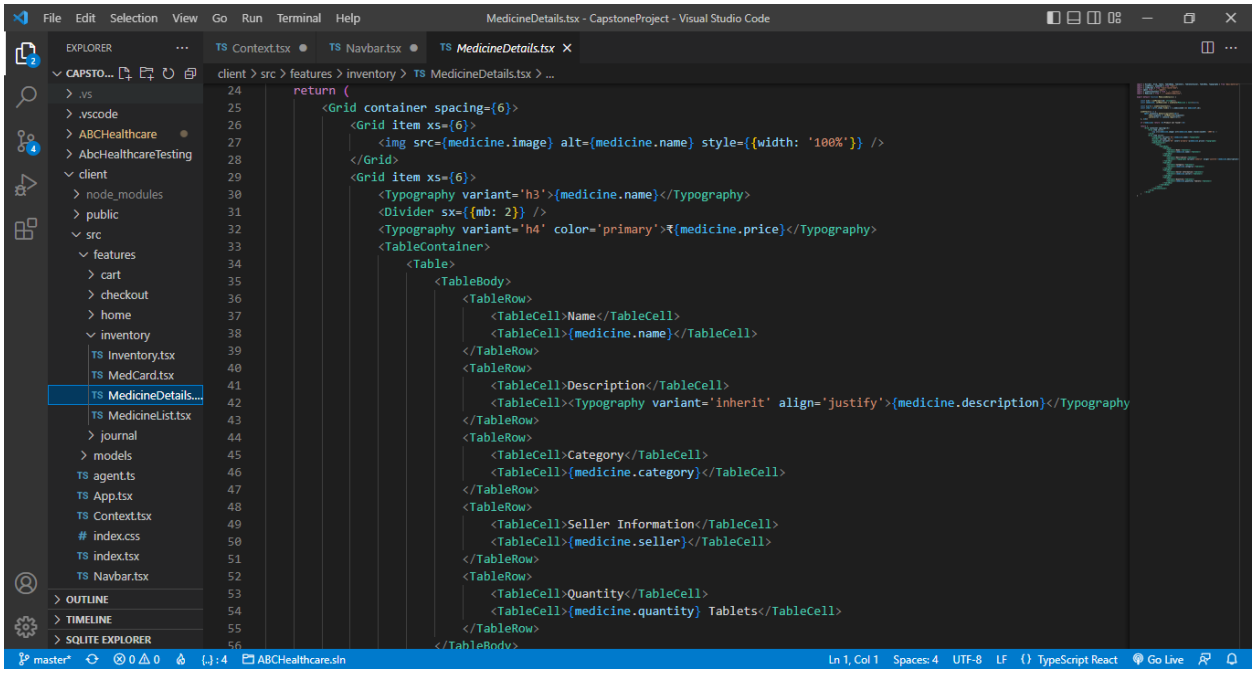


Fig. 48: inventory/MedicineDetails.tsx

The URL of the page shows that /inventory/1 page is currently being displayed, where '1' is actually the ID of this medicine.

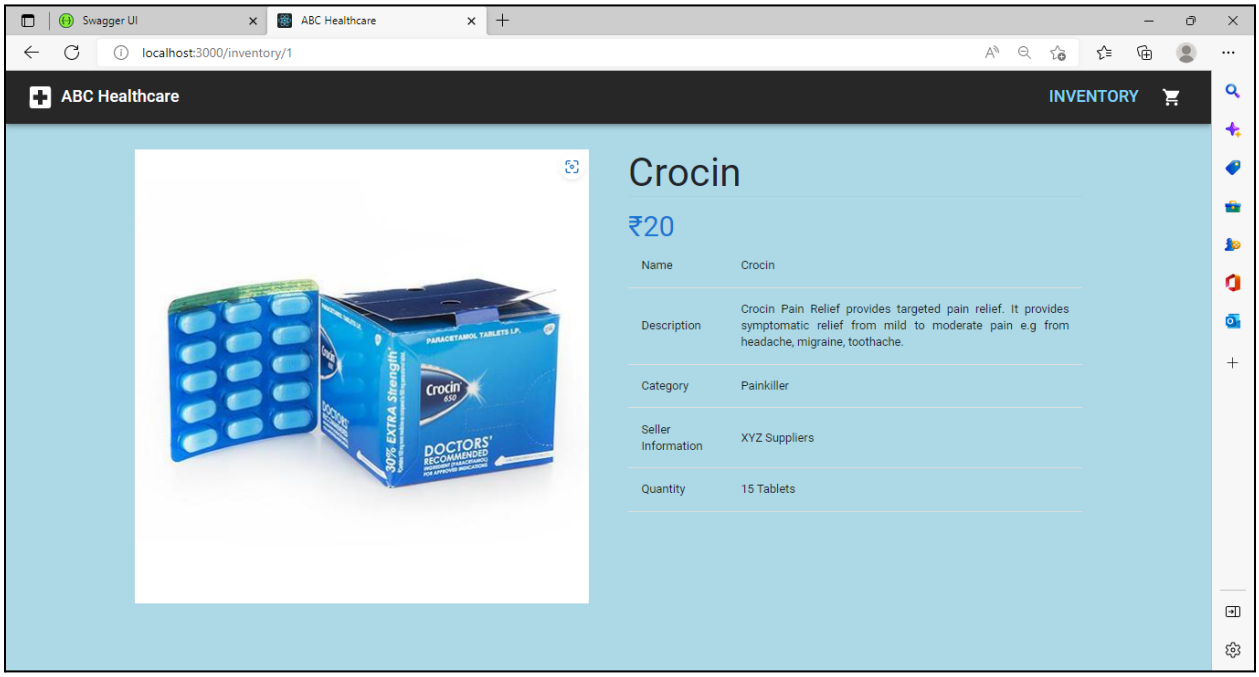


Fig. 49: Medicine-Details Page

The URL of the page shows that /inventory/3 page is currently being displayed, where '3' is actually the ID of this medicine.

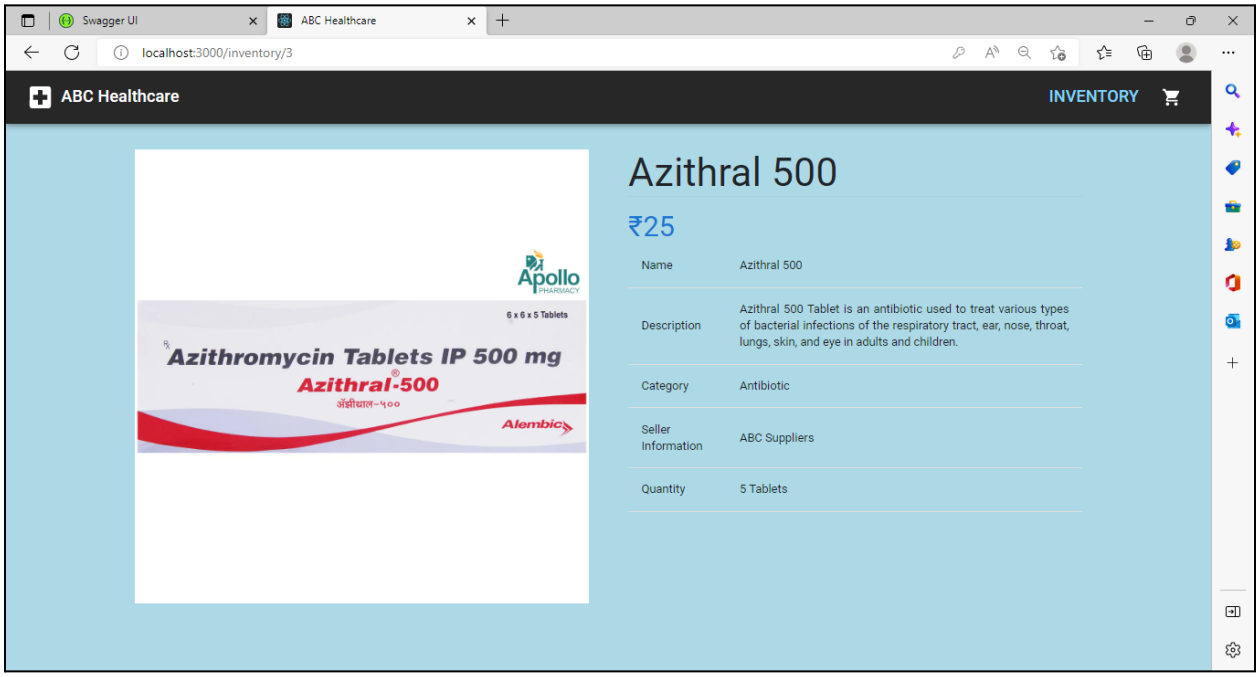


Fig. 50: Medicine-Details Page

Now, if bychance we provide some wrong ID in the URL of the pagewhere where there is no product, an error message will be displayed.

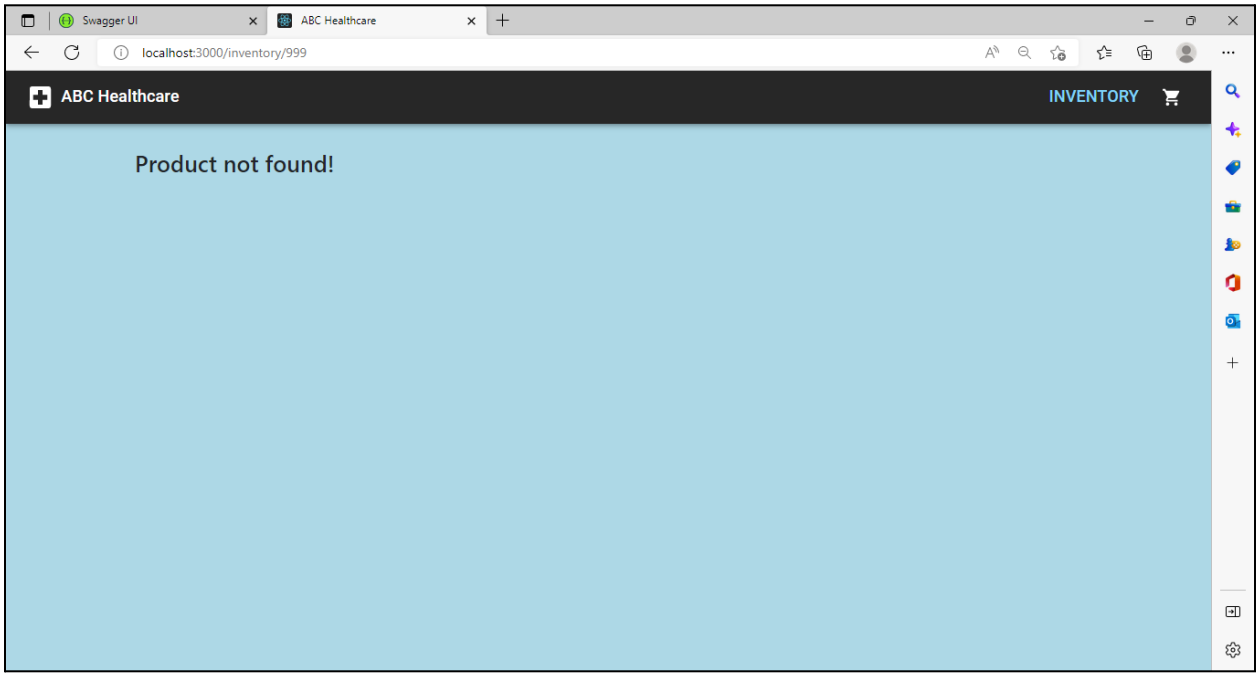


Fig. 51: Wrong Medicine-Details Page

Now, I have added some medicines to the cart. This is reflected in badge of ShoppingCart icon.

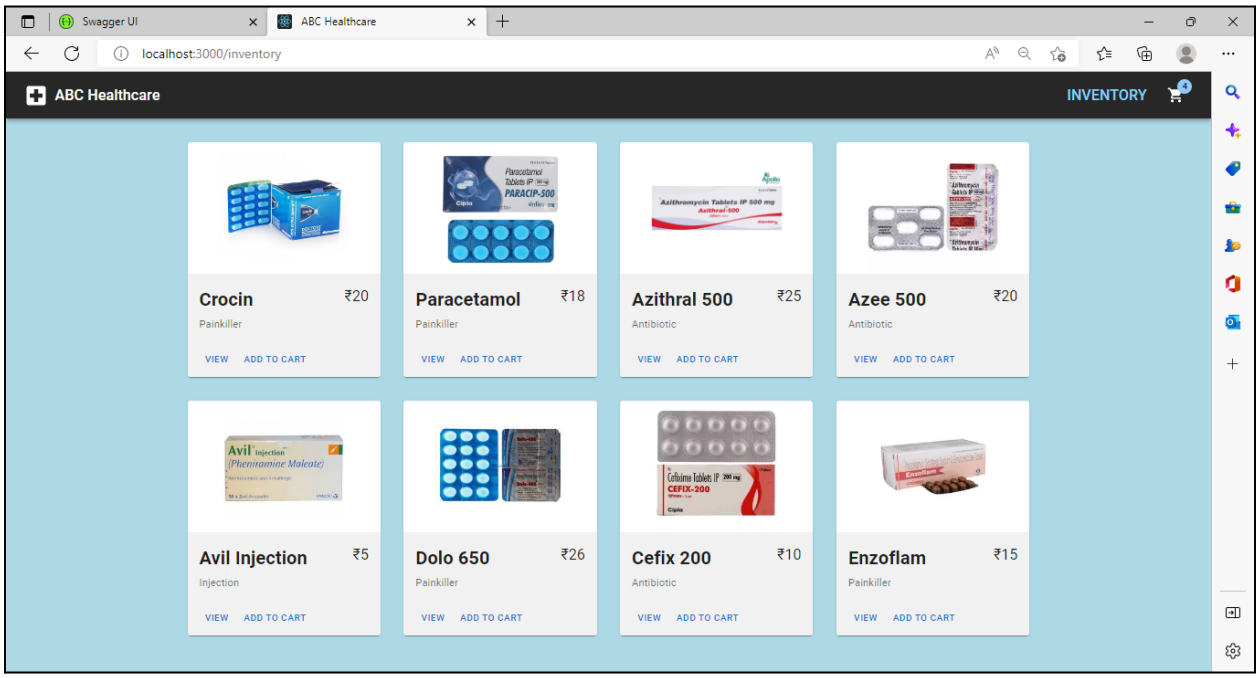


Fig. 52: Adding medicines to Cart

The added the products are reflected in the cart. The user can increase or decrease the quantity of medicines from here, or even remove the item from the cart by clicking on the Delete icon.

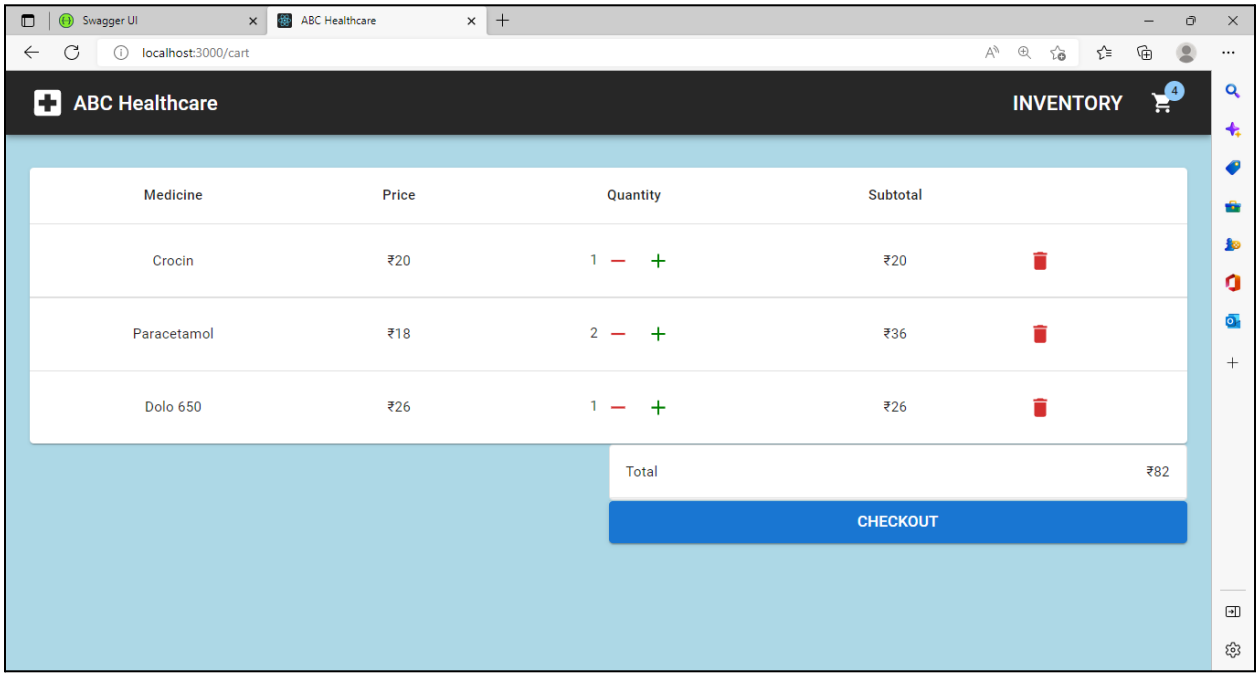


Fig. 53: Cart Page

As soon as we add the items to the cart at the frontend, the API creates a cart with an ID and unique buyerID. This create a cart that contains all the items which are added.

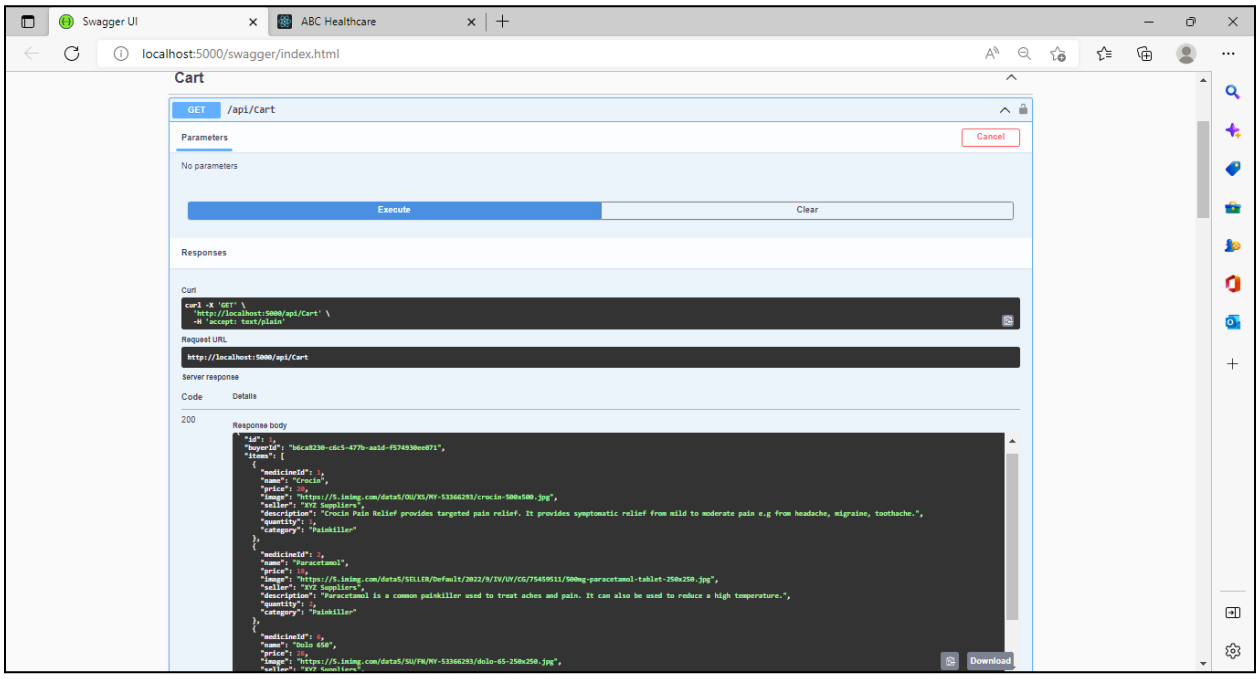


Fig. 54: Cart GET method

CartPage component firstly contains two methods to add or reduce the quantity of items; These methods use the properties that are defined in the agent.ts file whcih I created as shown already. The items are then updated using the context file hook I created, via setCart.

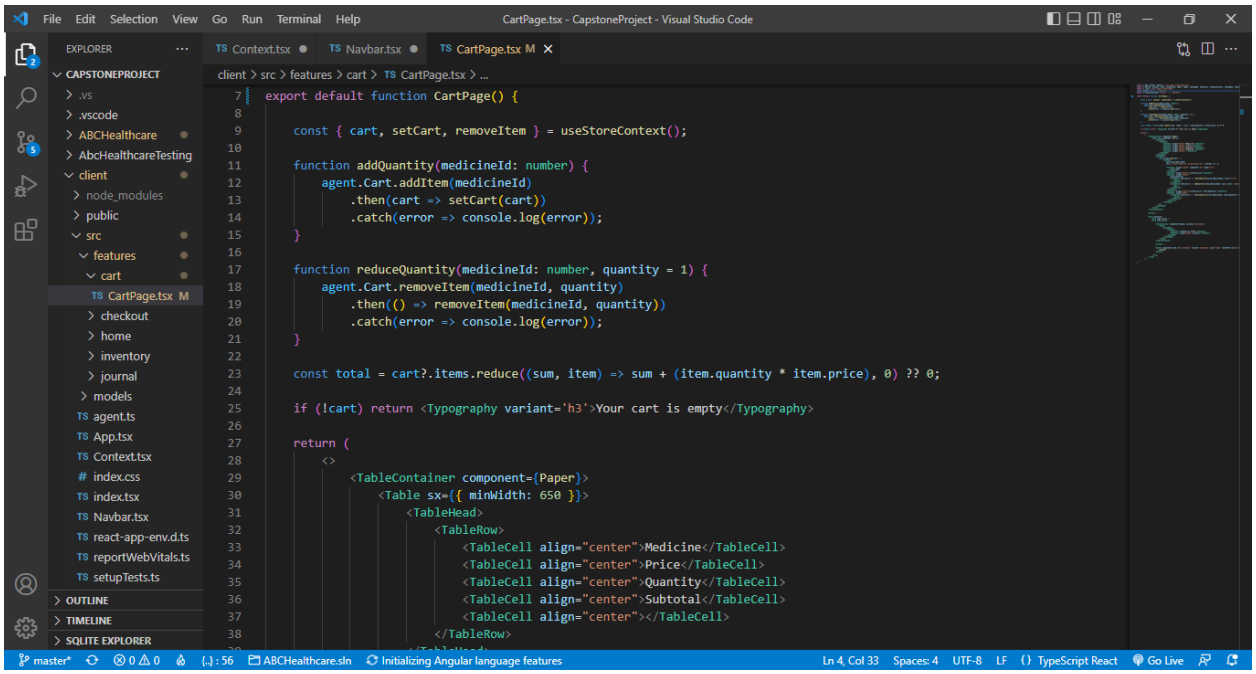


Fig. 55: CartPage.tsx

From the Cart page once we click on the checkout button, the order will be confirmed.

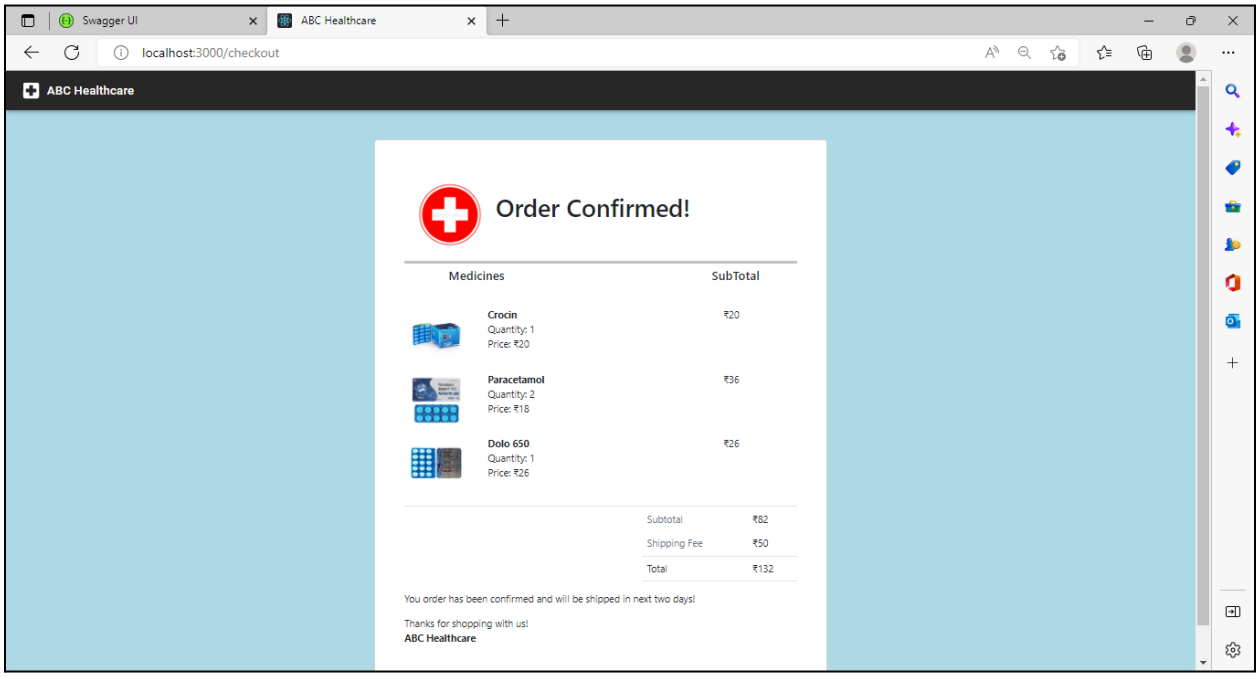


Fig. 56: Checkout Page

Again the context file is used to get the cart items for this page.

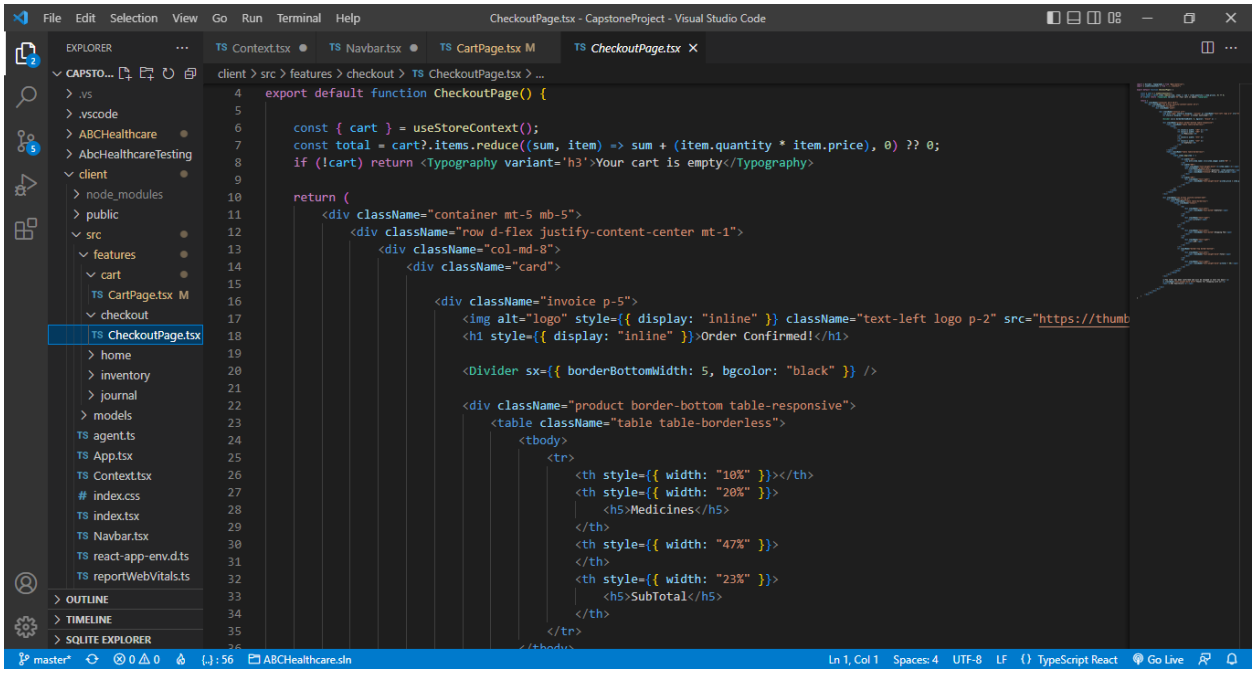


Fig. 57: CheckoutPage.tsx

Testing

RegisterUser feature testing is done to test whether the user is able to register into the system and also the data is stored in the database properly.

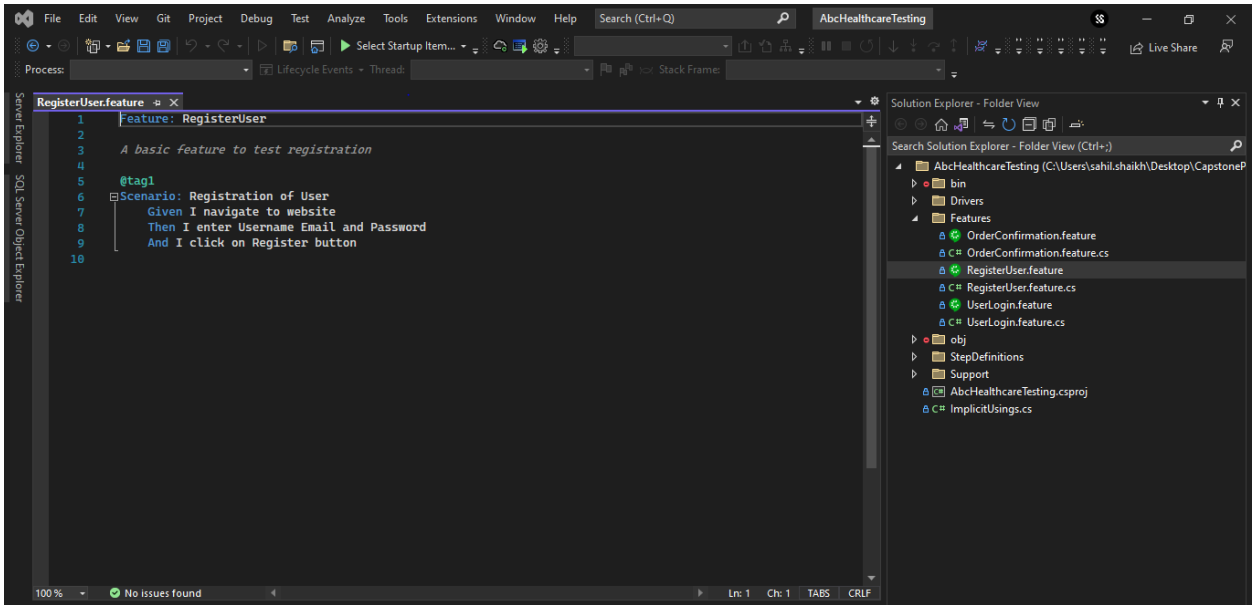


Fig. 58: RegisterUser.feature

In the StepDefinitions file for RegisterUser, firstly I initialized a driver as a chromeDriver. In the first step I navigated to the website URL, which will bring the Register page. Then, I looked for the elements using XPath as shown and used the SendKeys() method to input some test values.

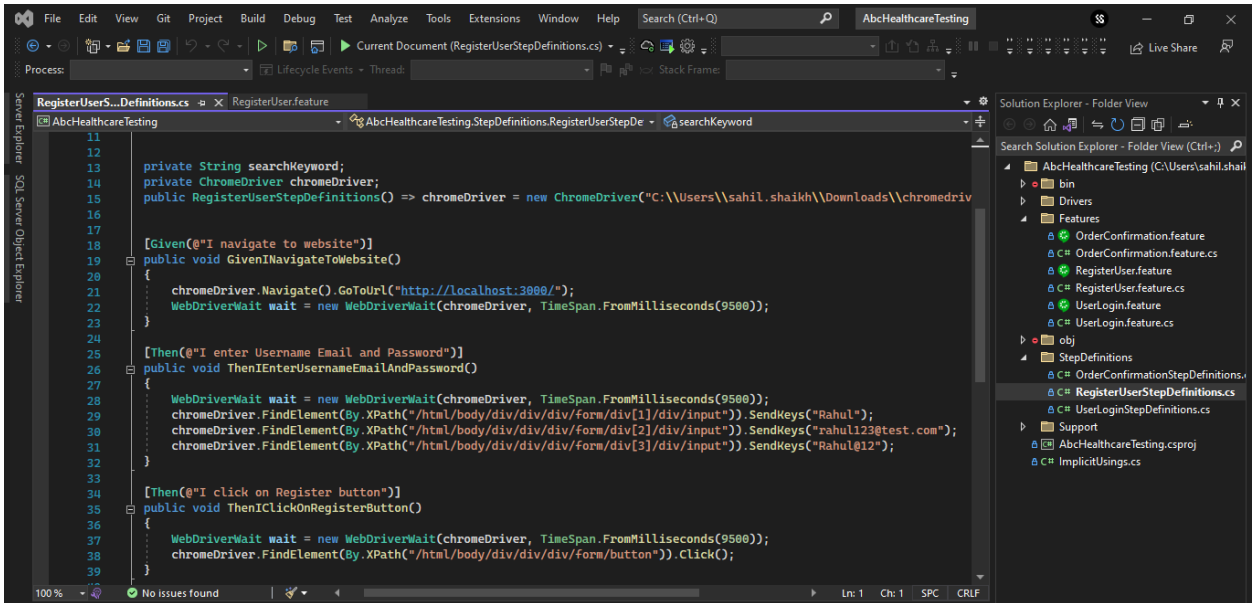


Fig. 59: RegisterUserStepDefinitions.cs

UserLogin feature testing is done to test whether the user is able to login into the system and then navigated to the homepage

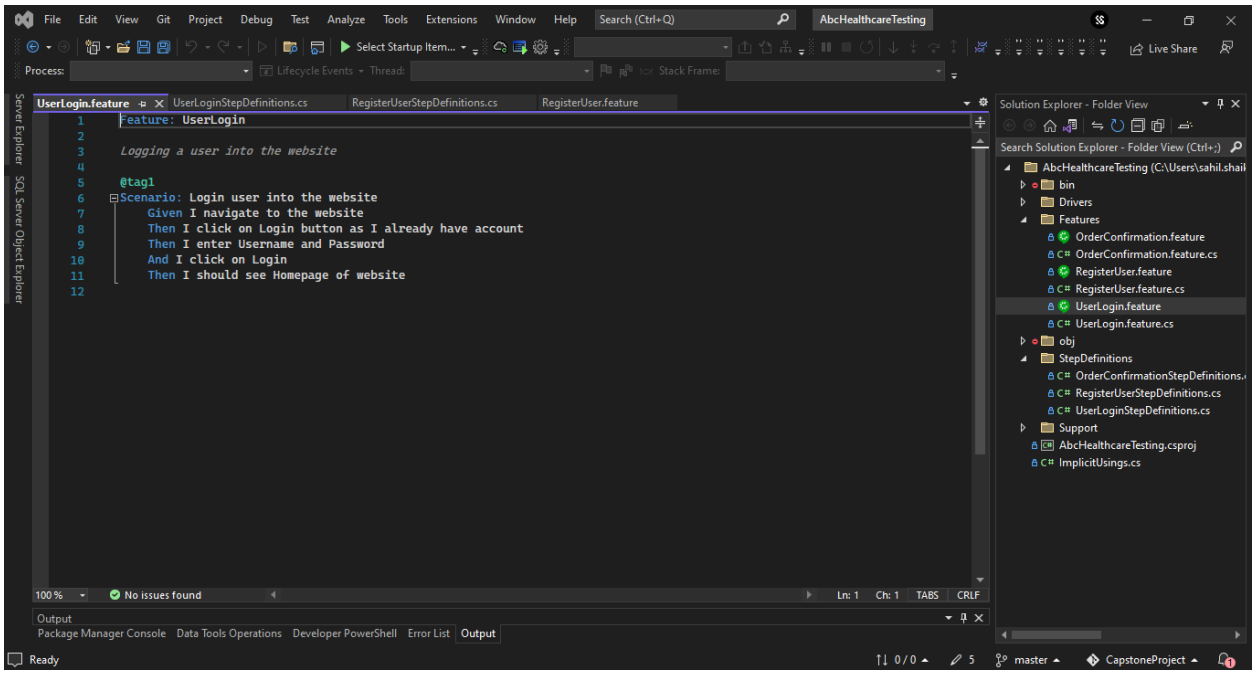


Fig. 60: UserLogin.feature

In the StepDefinitions file for UserLogin, firstly I navigated to the website URL, and then click on the Login button as the data is already registered. After arriving at the Login page then enter the credentials using the SendKeys() method to input some test values. After that click on Login button, and user should be navigated to the Homepage of the website.

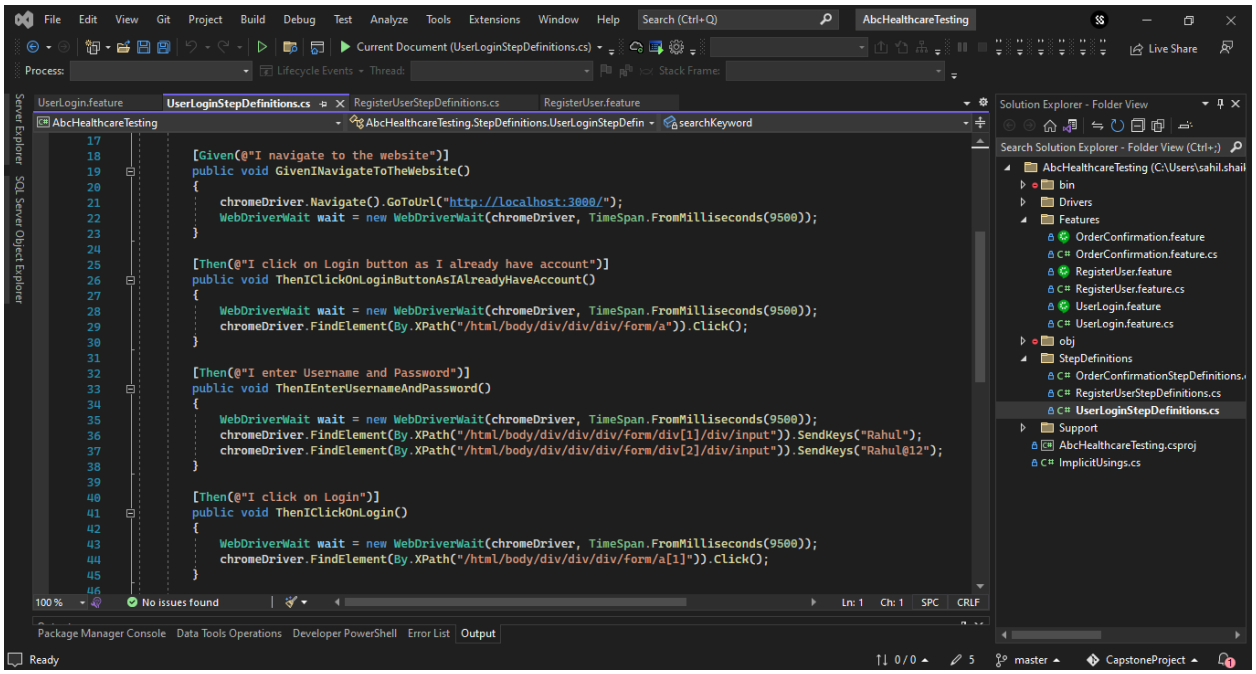


Fig. 61: UserLoginStepDefinitions.cs

OrderConfirmation feature testing is done to test whether the user is able to login into the system, add medicines to the cart, and then checkout.

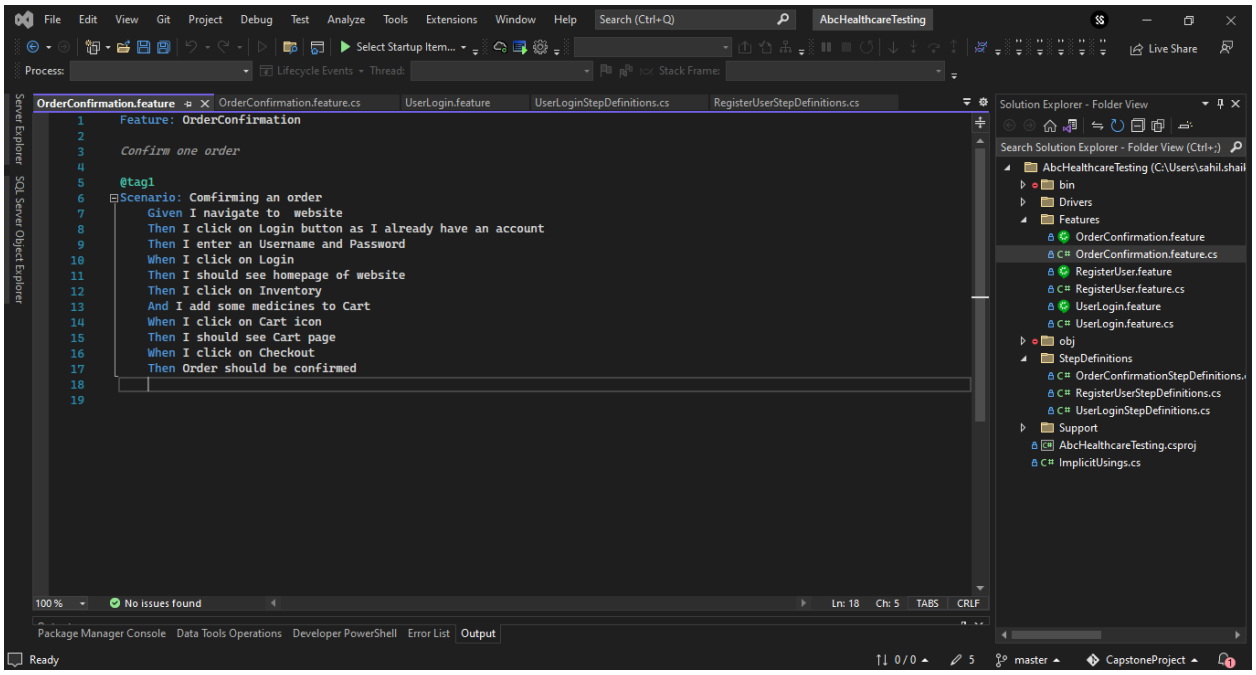


Fig. 62: OrderConfirmation.feature

In the StepDefinitions file, firstly I am logging in and arriving at the Homepage. Then I am navigating to the Inventory.

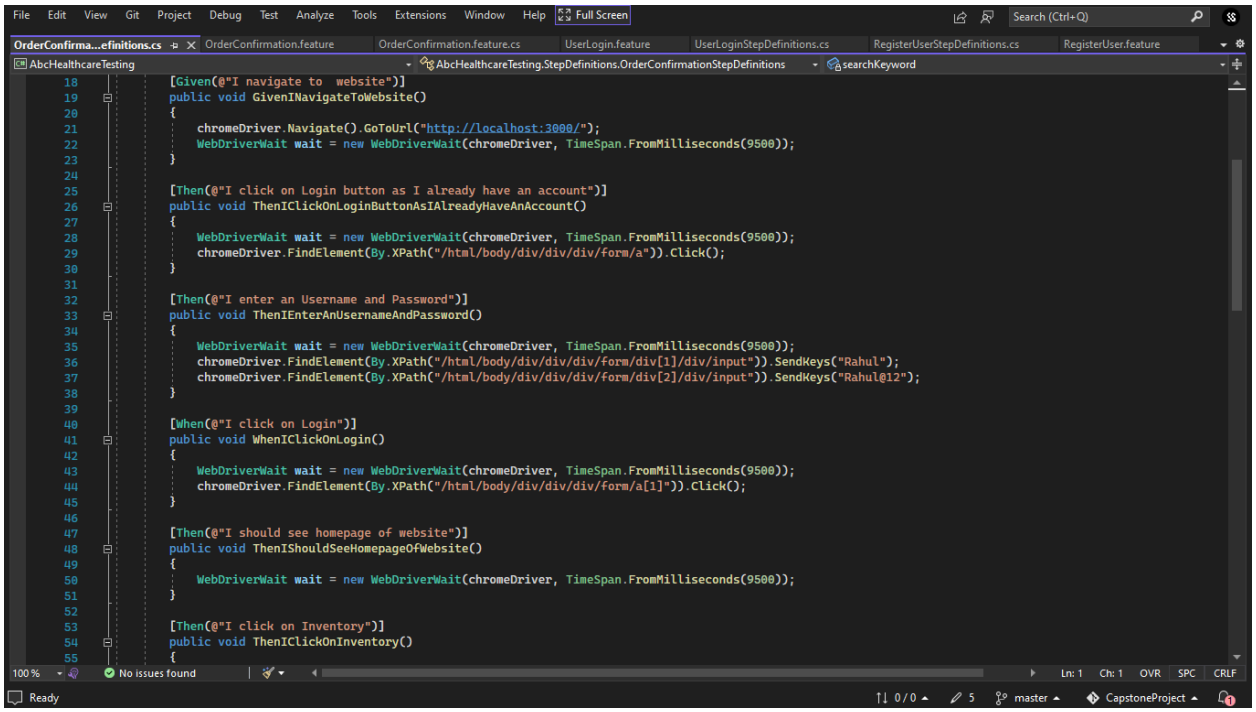
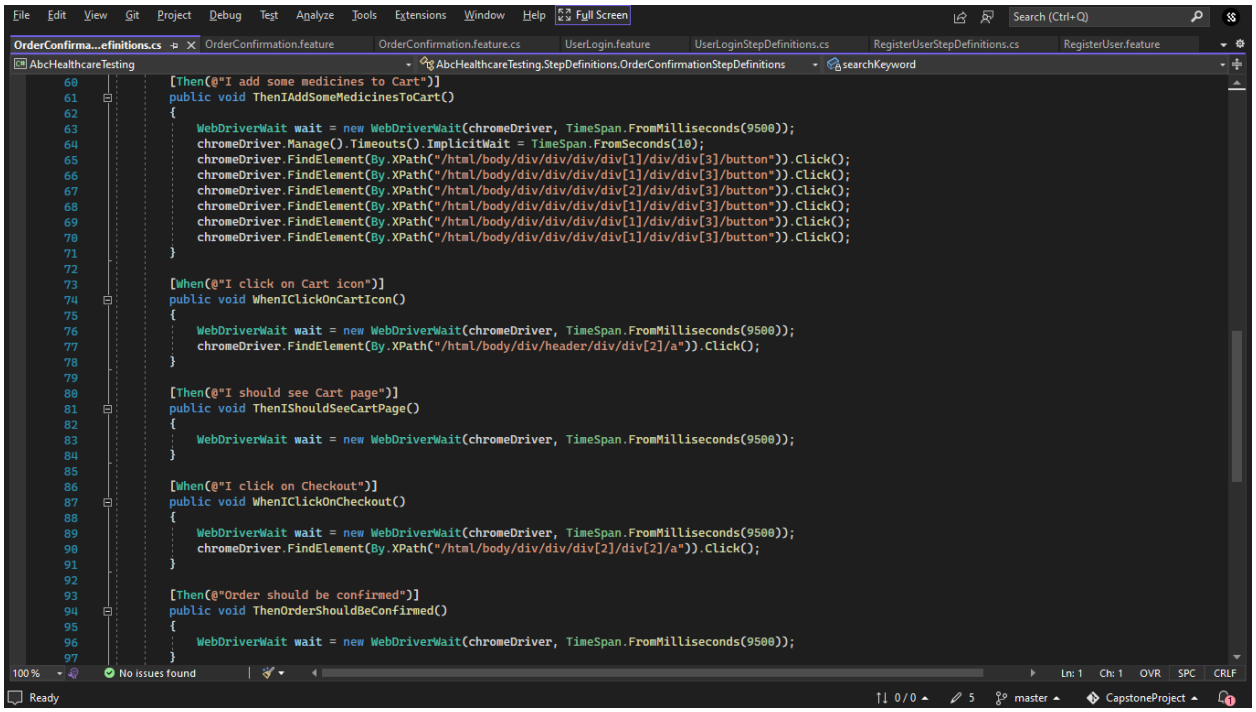


Fig. 63: OrderConfirmationStepDefinitions.cs

After coming to th Inventory page, I am adding some medicines to the cart. Finally, I am checking out.



```
[Then(@"I add some medicines to Cart")]
public void ThenIAddSomeMedicinesToCart()
{
    WebDriverWait wait = new WebDriverWait(chromeDriver, TimeSpan.FromMilliseconds(9500));
    chromeDriver.Manage().Timeouts().ImplicitWait = TimeSpan.FromSeconds(10);
    chromeDriver.FindElement(By.XPath("/html/body/div/div/div/div[1]/div/div[3]/button")).Click();
    chromeDriver.FindElement(By.XPath("/html/body/div/div/div/div[1]/div/div[3]/button")).Click();
    chromeDriver.FindElement(By.XPath("/html/body/div/div/div/div[2]/div/div[3]/button")).Click();
    chromeDriver.FindElement(By.XPath("/html/body/div/div/div/div[1]/div/div[3]/button")).Click();
    chromeDriver.FindElement(By.XPath("/html/body/div/div/div/div[1]/div/div[3]/button")).Click();
    chromeDriver.FindElement(By.XPath("/html/body/div/div/div/div[1]/div/div[3]/button")).Click();
}

[When(@"I click on Cart icon")]
public void WhenIClickOnCartIcon()
{
    WebDriverWait wait = new WebDriverWait(chromeDriver, TimeSpan.FromMilliseconds(9500));
    chromeDriver.FindElement(By.XPath("/html/body/div/header/div/div[2]/a")).Click();
}

[Then(@"I should see Cart page")]
public void ThenIShouldSeeCartPage()
{
    WebDriverWait wait = new WebDriverWait(chromeDriver, TimeSpan.FromMilliseconds(9500));
}

[When(@"I click on Checkout")]
public void WhenIClickOnCheckout()
{
    WebDriverWait wait = new WebDriverWait(chromeDriver, TimeSpan.FromMilliseconds(9500));
    chromeDriver.FindElement(By.XPath("/html/body/div/div/div[2]/div[2]/a")).Click();
}

[Then(@"Order should be confirmed")]
public void ThenOrderShouldBeConfirmed()
{
    WebDriverWait wait = new WebDriverWait(chromeDriver, TimeSpan.FromMilliseconds(9500));
}
```

Fig. 64: OrderConfirmationStepDefinitions.cs (contd…)

All 3/3 testcases have passed, and to the right we can see the last OrderConfirmation feature has completed successfully

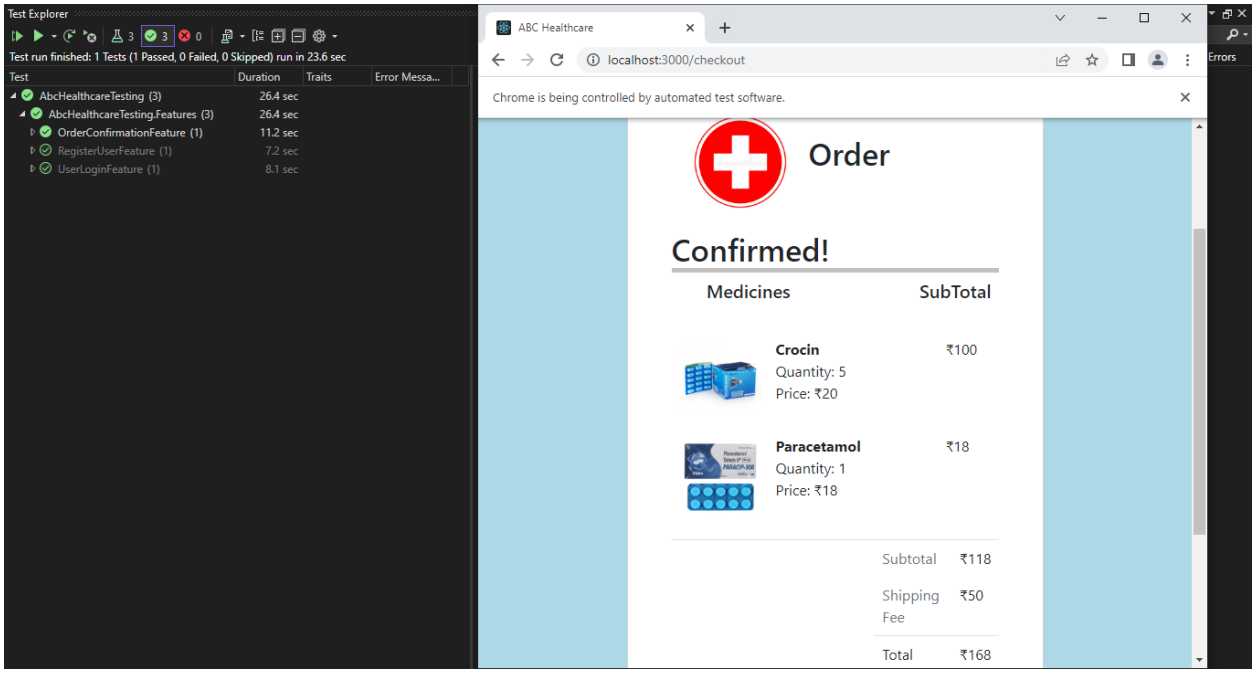


Fig. 65: All testcases passed