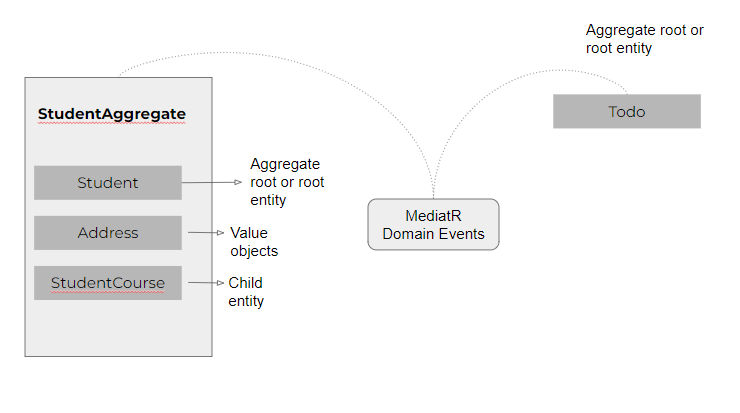
**What are domain events?**

Domain events are used to communicate with other entities or aggregates. Use domain events when something happens in the domain that different parts of the domain should be aware of. In DDD, side effects occur within the same domain.

For example, if you have a student entity and you want a student to add a todo, you would end up adding the code to the student entity that calls the Todo entity class, which is not good practice in DDD. Todo is an entity, and Student is also an entity; they should only manage its state and communicate by adding domain events and using a mediator object.

Domain-event communication happens in-process. A mediator object is used in-between to communicate with other entities. We use the **MediatR**library to dispatch domain events.



DDD communication with domain events

**What is MediatR?**

MediatR is a simple, unambitious mediator implementation in .Net It supports in-process messaging with no dependencies.

It also supports request/response, commands, queries, notifications, and events, synchronous and async, with intelligent dispatching via C# generic variance.

The MediatR is built to support CQRS, which stands for Command Query Responsibility Segregation, which defines a way to structure your read and write database operations, and it also supports the mediator pattern.

We can use the MediatR library to create the mediator pattern in your code. It is a class that communicates with another class or some classes by using the MediatR mediator object.

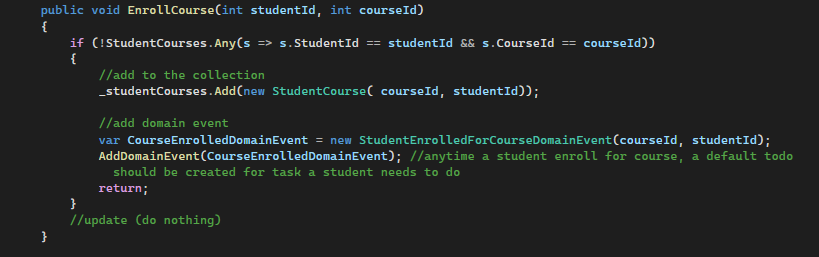
In MediatR, we can send an in-process message (messages are handled in-process) in a request/response style that is handled by a single handler.

You can also publish messages to multiple handlers by using the INotification and INotificationHandler interfaces.

Install [MediatR](https://www.nuget.org/packages/MediatR" \t "_blank) from the NuGet package manager

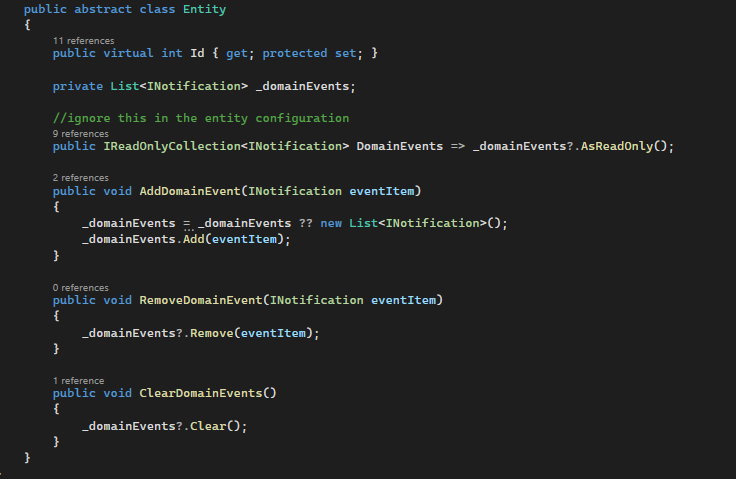
**Adding Domain Events**

In the student entity, we define a behavior that enables a student to enroll in a course. In the image below, you will notice the **AdddomainEvent()**method that adds the courseEnrolledDomainEvent object. Anytime a student enrolls in a course, a default to-do is created for a task the student needs to do.



Student Entity Course Enrollment Behavior

The **AddDomainEvent**code is defined in the Entity base class that all aggregate root or entities inherit. The entity class is where the **ID**primary key gets defined in the domain layer. The **AddDomainEvent**method just adds a list of **INotification**to a list collection.

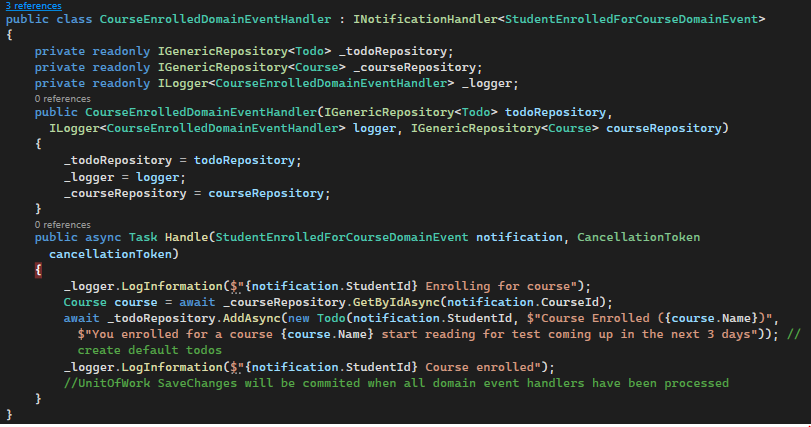


The Base Entity Class

**The Domain Event Handler**

The domain event handler defines a handler for a notification. Every domain event should have its own domain event handler. Whenever a domain event is added, the MediatR is responsible for dispatching the domain event to its handler.

The domain event handler can be defined in the **application layer**.

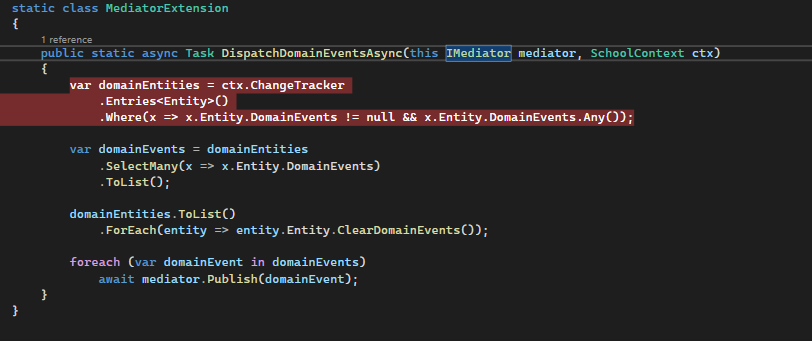


The Domain Event Handler For Course Enrollment

Now the question is, where do we add the dispatching code using the MediatR library?

**Dispatching Domain Events Using MediatR**

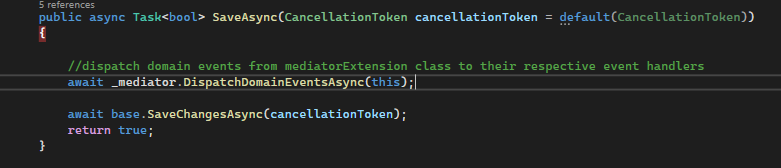
In the **infrastructure layer,**we define code that dispatches the domain events to their own handlers. This should be done right before **SaveChanges()**is called in the DbContext class.



The MediatorExtension

The DispatchDomainEventsAsync method defines code that dispatches the domain event to its handler. The code gets all domain events added to the collection, iterates through them, and publishes the domain event before changes are saved in the database.

*await mediator.Publish(domainEvent)*



Dispatch before SaveChanges()