Domain Driven Design

THEORY AND HANDS ON ,

MAHMOUD FADDO

Course Agenda

- Why Domain Driven design?
- Language and context
- Event storming
- Designing the model
- Implementing the model
- Acting with commands
- Consistency Boundary
- Aggregate persistence
- And much More ..!! (keep you excited)

second session agenda

- Why Domain Driven design? (Done)
- Language and context
- Event storming
- Designing the model
- Implementing the model
- Acting with commands
- Consistency Boundary
- Aggregate persistence
- CQRS
- And much More ..!! (keep you excited)

Outside the domain model

- We strive to keep our domain model intact from anything that is related to infrastructure, persistence, execution, and communication.
- This makes the domain model pure and keeps it focused on the business.
- So we need to create the whole system around it, keeping the domain model as the system core.

Exposing the Web API

- Lets code

Public API contracts

- To serve HTTP requests, we need more than just a controller. The controller needs to accept strongly typed requests, and the collection of those requests will be our public API and the models for those requests will be our contracts.
- Contracts are **data-transfer objects** (**DTOs**) and **plain-old C# objects** (**POCOs**). It means that they have no logic, they only contain primitive types and do not require any tricks to be serialized and deserialized.
- Practically, you can add complex types to your DTOs just because some complex types are used in many contracts. An example of such a type could be the Address type

Address Dto Example

```
public class UpdateCustomerAddressDetails
{
    public string BillingStreet { get; set; }
    public string BillingCity { get; set; }
    public string BillingPostalCode { get; set; }
    public string BillingCountry { get; set; }
    public string DeliveryStreet { get; set; }
    public string DeliveryCity { get; set; }
    public string DeliveryPostalCode { get; set; }
    public string DeliveryCountry { get; set; }
}
```

Address Dto Example Cont.

```
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
}
public class UpdateCustomerAddressDetails
{
    public Address BillingAddress { get; set; }
    public Address DeliveryAddress { get; set; }
}
```

Contracts with complex types drawbacks.

- complex types add compatibility issues.
  - since when you change the type, all contracts that use it will also change and this change will be implicit.
  - Since you have no internal consumers of this contract, you will not be able to see whether the clients that use this contract will be affected. Such information can only be obtained by tests.

Recognize breaking and non breaking changing

- Everything you publish outside of your development machine is considered public.
- The public API is something that anyone who has appropriate permissions can use.
- that you are not in control of who is using your API anymore.
- Therefore, any change in the published API can potentially break other parts of the system or other systems.
- Changes in public contracts need to be crafted with care since there are both non-breaking changes and breaking changes.

Non breaking changing

- Changing property type so any value of the type that was used before can be serialized to the new type. For example, we can change a property from being an integer to string, and it will be compatible.
- Adding a new property is also considered a non-breaking change. This is because when we try to deserialize an XML or JSON object that does not have this new property because the sender hasn't updated their contracts yet, most popular serializers will accept it and will use a default value if the value is not supplied.

Breaking changing

- of course, it is not always possible to make non-breaking changes.
- Therefore, we need to ensure that when we make a breaking change, everyone who is using the old API will not get exceptions and will be able to work as before, at least for some time.
- This is done by **API versioning.**

Code the contract.

- Go to code.
- Here, we use nested static classes, ClassifiedAds and V1, as a replacement for namespaces, so we can have more versions in one file when necessary. This method allows us to use static members import to keep code more concise.

Our first command has been created.

- What we have here is a *command*.
- Commands allow users and other systems to execute actions in our domain model.
- When a command is successfully processed, the domain model state changes and new domain events are emitted.
- we need to accept this command coming from the outside world, and we'll use an HTTP endpoint for this.

API as edge layer.

- This is our adapter for the HTTP infrastructure, which finds its place in the outermost layer of the onion architecture.
- That's we call this layer **the edge** because there is nothing outside of it, which we can consider a part of our app.
- The application can have multiple ways to communicate with the outside world, so if we added some other edge, such as messaging, we would expect for this new communication adapter to process the same commands.

Use Swagger

- Go to code.

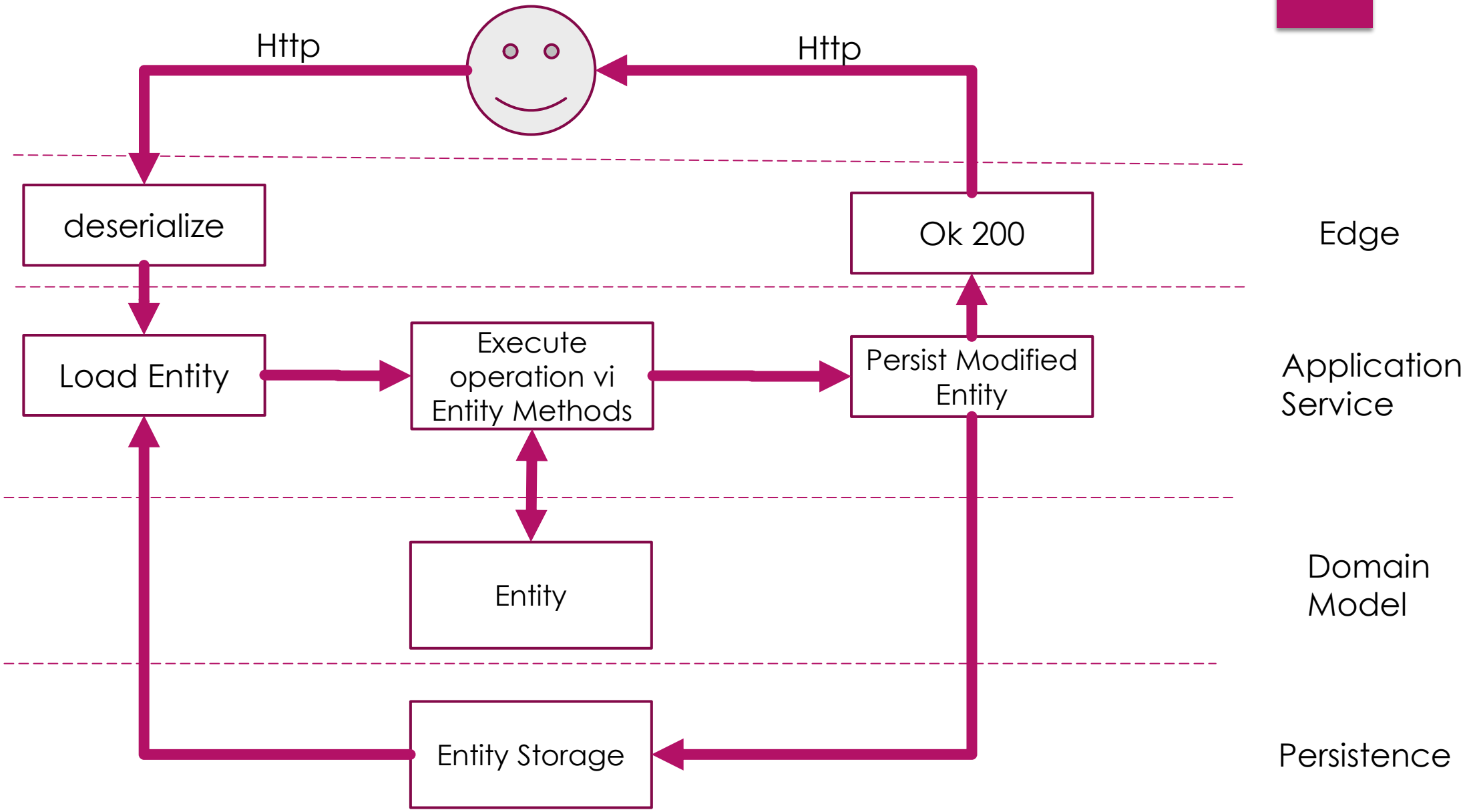
Application Layer

- The edge—in our case, it is a simple web API—accepts requests from the outside world.
- The main job for our edge component is to accept some request, sent as a JSON document, XML document, message, or any other communication channel and serialization type; convert it to a command; and then ensure this command gets handled.
- The edge can, of course, work directly with the domain model, but this would mean we accept the fact that we will always be working with one edge type only, with one communication protocol.
- edge components are usually heavily dependent on the communication infrastructure— creating unit tests for such components might be challenging

Application Layer (Cont)

- To isolate the communication infrastructure from the actual request handling, we can introduce the application layer. In this layer, we need a component that will accept commands from the edge and use our domain model to handle these commands. Such a component is called **application service**.
- the services need to have a way to load and store entities, because a typical operation of an application service

The whole Flow



Application Layer Implement

- ***Lets code***

Command Handling

- To execute requests, we need an *application service* that works as an intermediary between edge components and the domain model. The
- application service is also responsible for persisting entities.

The command handler pattern.

- The command handler is a class that has one method to handle a single command type.
  - IEntityStore
  - IHandleCommand
- ***Lets coed***

IEntity Store Vs Repos

- In fact, the entity store is not the exact math of the repository pattern. When the purpose of repositories is to mimic a collection of objects and hide the persistence, the entity store is a complete opposite.
- It does exactly what it tells you about -persist one single object and retrieves it back.

Controller and injection

- Lets code.

The command handler pattern Summary

- The command handler pattern is compelling, and it adheres to the **Single Responsibility Principle** (**SRP**).
- At the same time, each HTTP method in our API would require a separate command handler as a dependency.
- We might predict that we will have more than 10 methods in our classified ad web API, and an adequate number of command handlers
- Command handlers need the entity store as a dependency, and since all web API controllers are instantiated per scope, all command handlers will be instantiated and injected as well, with all their dependencies.

The command handler pattern Summary Cont.

- to resolve dependency tree:
  - using factory delegates instead of dependencies per request.
  - Lazy<IHandleCommand<T>> as a dependency.
  - using another dependency-injection container, such as Autofac, which supports automatic factory delegates and Lazy<T> out of the box
- The result is when using command handler pattern the dependency tree became very complicated and it forces us to use a very complicated methods to register our dependency. Unfortunately I wont talk about those complicated methods as this beyond this course.

Back to application service

- In fact, how our application service will look and behave is very similar to a bunch of command handlers.

Classic application service

- *classic* application service exposes some methods with multiple parameters. (code)
- Using this kind of declaration is perfectly fine, except it doesn't play that well with the composition.
- It is not easy to add such an application service to a *pipeline*, where we have logging, retry policies, and so on.
- To make a pipeline, we need all our handlers to have compatible parameters.
- these methods of IPaymentApplicationService just don’t allow us to go that way.
- Every other call in the pipeline must have the same set of parameters, and as soon as we want to add one more parameter to any method, we are doomed to make numerous changes in multiple classes that form our pipeline.

Classic application service cont.

- Alternatively, we can have one application service class that implements multiple IHandle<T> interfaces. This would work, but each command will then require a separate bootstrapping code, although we are adding the same elements to our pipeline.

Current Application service.

- we can generalize our application service to handle any type of command. Code
- By implementing our application service like this, we will have a single dependency to handle for all of our API calls, and we keep the door open to compose a more complex command processing pipeline, just as we were able to do with individual command handlers.
- Of course, the tradeoff here is that we have one class that handles several commands and some might see it as an SRP violation. At the same time, the level of cohesion for this class is high.

Let's expand our application service based on Domain model

- Extend contracts.
- Extend Edge.
- Extend handle Method.

Let's expand our application service based on Domain model

- As you can see, handling the Create command looks different from handling all other commands. This is natural since when we create a new entity, we need to ensure it does not exist yet. When we handle operations on the existing entity, it works the other way around. In this case, we need to ensure that the entity exists, otherwise, we cannot perform the operation and must throw an exception.
- the application service is responsible for translating primitive types, such as string or decimal, to value objects. The edge always uses serializable types that have no dependencies on the domain model. The application service, however, operates with domain concerns; it needs to tell our domain model what to do, and since our domain model prefers to receive data as value objects, the application service is then responsible for the conversion.

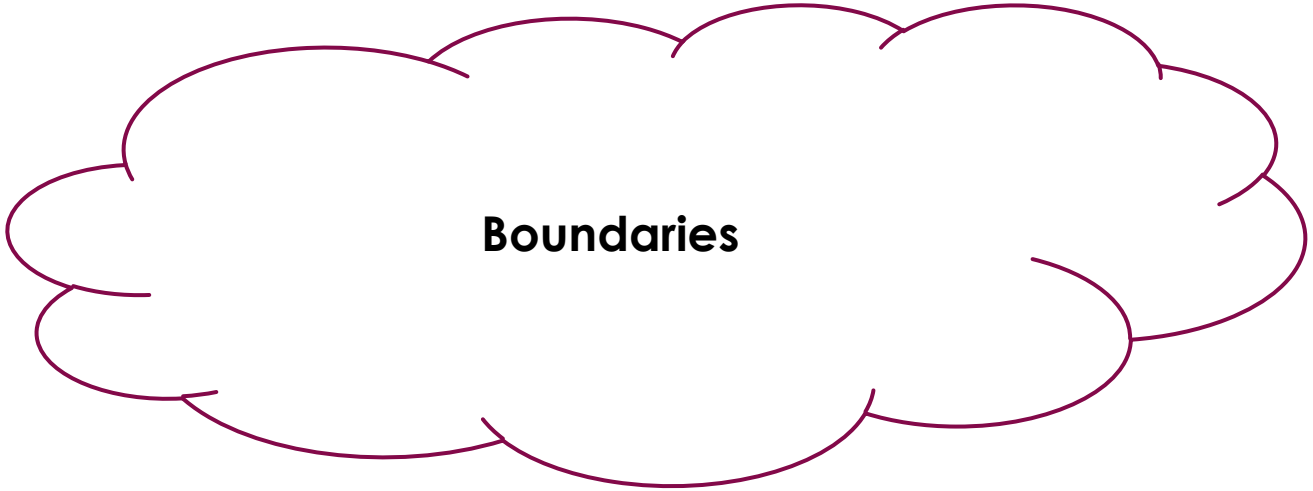
Simplify handle method

- As you can see, handling the Create command looks different from handling all other commands. This is natural since when we create a new entity, we need to ensure it does not exist yet. When we handle operations on the existing entity, it works the other way around. In this case, we need to ensure that the entity exists, otherwise, we cannot perform the operation and must throw an exception.
- the application service is responsible for translating primitive types, such as string or decimal, to value objects. The edge always uses serializable types that have no dependencies on the domain model. The application service, however, operates with domain concerns; it needs to tell our domain model what to do, and since our domain model prefers to receive data as value objects, the application service is then responsible for the conversion.

Summary

- From the application service code, it becomes clear that the application service itself plays the vital role of an intermediary between the application edge and the domain model. An edge could be anything that communicates with the outside world
- When we handle a command, no matter whether we use multiple command handlers or single application service, the sequence of operations is usually very similar. A command handler needs to fetch a persisted entity from the entity store, call the domain model to do the work, and then persist changes

What is next !! Keep excited



Hands on

- Your turn:
  - ***Pull the repo and try to understand the code base.***

Thank you

SESSION 6