



Advanced Software Engineering

DDD Project

Angewandte Informatik

at the Cooperative State University Baden-Württemberg Karlsruhe

by

Abdelrahman Saleh (6614515)

May 2021

1 Domain Driven Design

1.1 Event Storming

Event Storming is a set of meetings where developers come together with domain experts to deeply understand the domain and to make sure we are all on the same page.

Event Storming Goals

- Provide visibility during the discussion. This should remove assumptions when many people are discussing the same thing with different terms. It also eliminates some of the ambiguity and brings it to the surface for further exploration.
- Have a modeling language that people understand. UML is not an option, and the usual boxes and arrows have no real notation, so people can get confused and start spending time trying to clarify the meaning of things.
- Involve many people simultaneously. In traditional meetings, only one person can effectively deliver the message, while everyone else needs to shut up and listen. As soon as many people start talking at the same time there is no conversation anymore. But, assuming people with different interests and backgrounds are present in one session, they might show a lack of interest and get bored.
- Find a way to express terms, behavior, model processes, and decisions, not features and data.

1.2 Ubiquitous Language Analysis

To ensure a ubiquitous language we will describe what happens on our side when customers use our services.

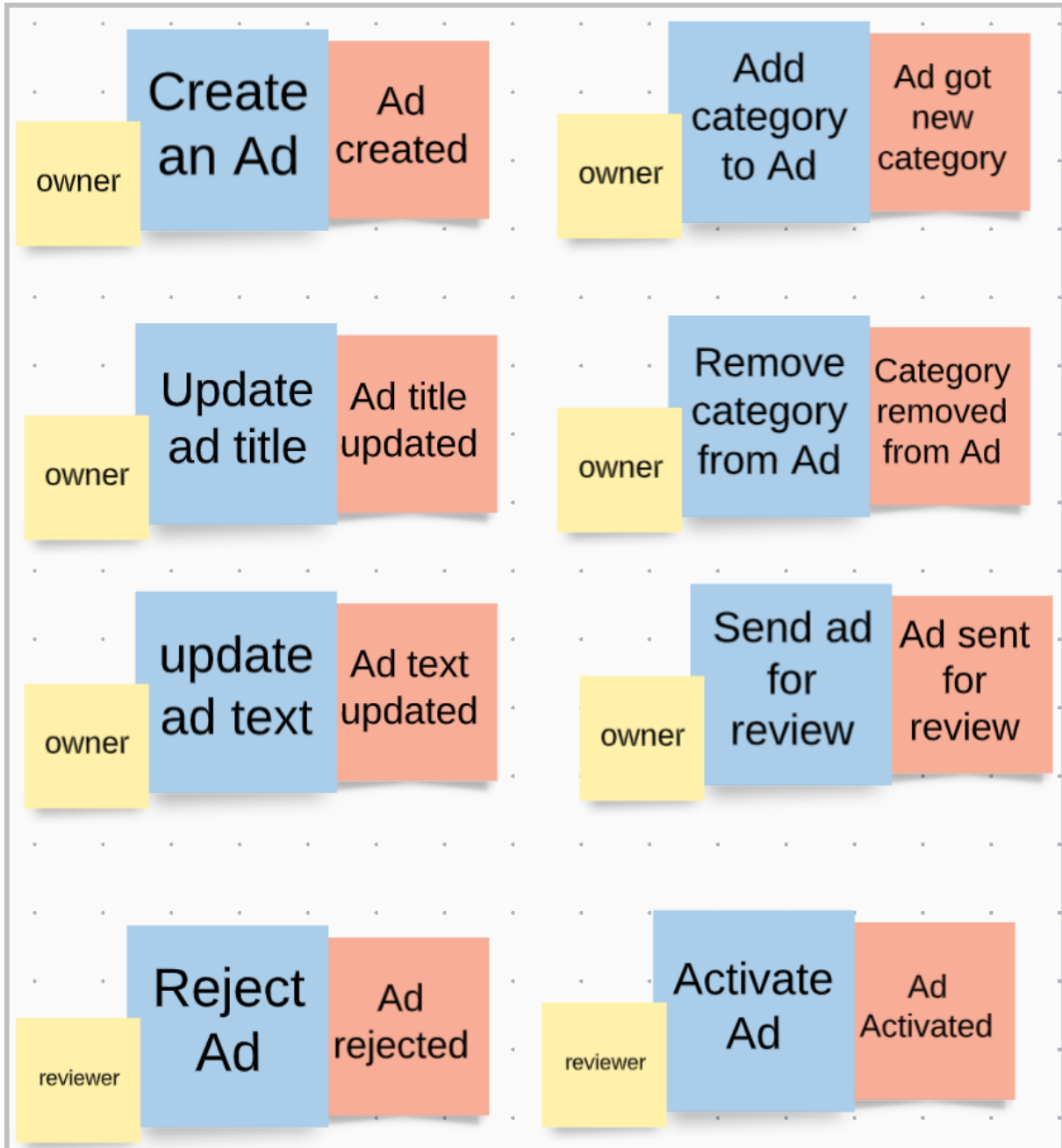


Figure 1.1:

Recruiter and Potential Employee

In the application everyone can create a Job Ad, so there is no difference between a recruiter and a potential employee when it comes to creating and managing an Ad. So we will use the term Ad Owner.

Reviewer

A Reviewer is an app admin, who is responsible for approving ads and rejecting them.

Update an Ad

There is no such thing as update an entire ad, rather there is `update ad title`, `update ad text`, `update ad category`, ...etc.

Activate vs Approve

When the owner creates an ad, the ad then will be sent for review, and when the ad is approved it will be activated, but basically being approved and activated are the same thing, so we are sticking with ad activated, rather than having two methods which are doing the same thing.

1.3 Repositories

Repository have responsibility to add an object, get objects by identifier or complex criteria and eventually to remove an object. There are also use cases that require aggregations like How many objects are in the system.

```
namespace JobMarket.Domain
{
    2 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
    public interface IJobAdRepository
    {
        1 reference | Rahman Saleh, 15 hours ago | 1 author, 1 change
        Task<bool> Exists(JobAdId id);

        1 reference | Rahman Saleh, 15 hours ago | 1 author, 1 change
        Task<JobAd> Load(JobAdId id);

        2 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
        Task Save(JobAd entity);
    }
}
```

Figure 1.2:

1.4 Entities

Of course the first entity we have in the application is JobAd Entity. It has an ID and includes all the methods which the Ad owner could use.

```
8 references | Rahman Saleh, 12 hours ago | 1 author, 1 change
public class JobAd : Entity<JobAdId>
{
    6 references | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public JobAdId Id { get; private set; }
    2 references | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public UserId OwnerId { get; private set; }
    3 references | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public JobAdTitle Title { get; private set; }
    3 references | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public JobAdText Text { get; private set; }
    3 references | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public Salary Salary { get; private set; }
    5 references | 1/1 passing | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public JobAdState State { get; private set; }
    1 reference | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public UserId ApprovedBy { get; private set; }

    2 references | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public JobAd(JobAdId id, UserId ownerId) =>
        Apply(event: new Events.JobAdCreated
        {
            Id = id,
            OwnerId = ownerId
        });

    5 references | 2/4 passing | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public void SetTitle(JobAdTitle title) =>
        Apply(event: new Events.JobAdTitleChanged
        {
            Id = Id,
            Title = title
        });

    5 references | 2/4 passing | Rahman Saleh, 12 hours ago | 1 author, 1 change
    public void UpdateText(JobAdText text) =>
        Apply(event: new Events.JobAdTextUpdated
        {
            Id = Id,
            AdText = text
        });
}
```

Figure 1.3:

1.5 Value Objects

Value objects allow declaring entity properties with explicit types that use Ubiquitous Language. Besides, such objects can explicitly define how they can be created and what operations can be performed within and between them. It is a perfect example of making implicit, explicit.

We used the value object in our `UserId` class. In the `userId` constructor we make sure that the `UserId` value isn't empty:



```
namespace JobMarket.Domain
{
    8 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
    public class UserId
    {
        2 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
        private Guid Value { get; set; }

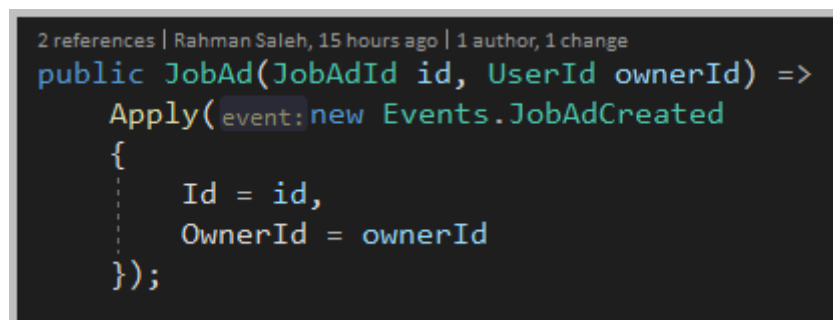
        3 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
        public UserId(Guid value)
        {
            if (value == default)
                throw new ArgumentNullException(nameof(value), message: "User id cannot be empty");

            Value = value;
        }

        public static implicit operator Guid(UserId self) => self.Value;
    }
}
```

Figure 1.4:

Then inside our Entity we pass the `UserId` into the constructor, with that our Entity has no check for the `ownerId`, since by receiving the argument of type `UserId` we guarantee that the value is valid.



```
2 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
public JobAd(JobAdId id, UserId ownerId) =>
    Apply(event: new Events.JobAdCreated
    {
        Id = id,
        OwnerId = ownerId
    });
```

Figure 1.5:

And we did the same with `JobAdId`

```
15 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
public class JobAdId : IEquatable<JobAdId>
{
    6 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
    private Guid Value { get; }

    4 references | Rahman Saleh, 15 hours ago | 1 author, 1 change
    public JobAdId(Guid value)
    {
        if (value == default)
            throw new ArgumentNullException(nameof(value), message: "Job Ad id cannot be empty");

        Value = value;
    }
}
```

Figure 1.6:

2 Entwurfsmuster (Design Pattern)

One of the design patterns that are used in the project is **Factories**.

Factories are functions that are used to create instances of domain objects, which are, by definition, valid. Factory functions can execute some logic to construct valid instances, and such logic could be different per factory. Factories also help to make implicit things more explicit by using proper naming.

We have got a requirement in `JobAdTitle` to partially support markdown. For now we made it only support italic and bold. We do need to validate the existing factory argument since any string is a valid Markdown string anyway. But, if we can get input from some online editor that can only produce pure HTML, we can do a conversion in a new factory function:

```
public static JobAdTitle FromHtml(string htmlTitle)
{
    var supportedTagsReplaced:string = htmlTitle
        .Replace(oldValue: "<i>", newValue: "*")
        .Replace(oldValue: "</i>", newValue: "*")
        .Replace(oldValue: "<b>", newValue: "***")
        .Replace(oldValue: "</b>", newValue: "***");

    var value:string = Regex.Replace(input:supportedTagsReplaced,
        pattern:"<.*?>", replacement:string.Empty);
    CheckValidity(value);

    return new JobAdTitle(value);
}
```

Figure 2.1:

3 Clean Architecture

3.1 Onion Architecture

We are using an Onion Architecture, where the center of the application is the domain. Application services and Infrastructure are kept outside and form layers around this core of the system. Unlike a layered architecture, which has dependencies going down from the UI layer to the data layer, we can see that the Domain is the center of everything, and everything depends on it.

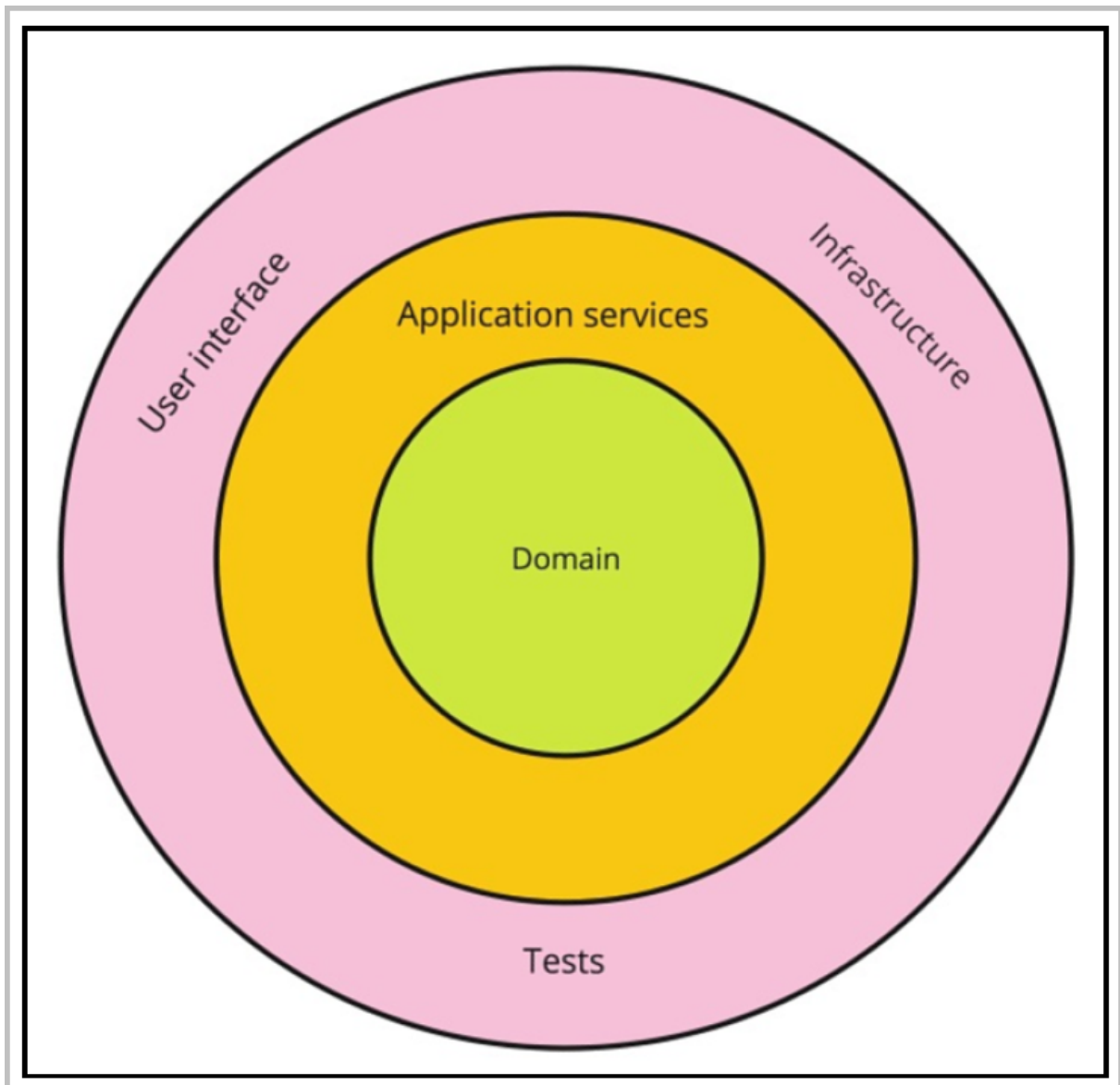


Figure 3.1:

3.2 CQRS

Separating commands and queries on the system level means that any state transition for the system can be expressed by a command, and such a command should be handled efficiently, optimized to perform the state transition. Queries, on the other hand, return data derived from the system state, which means that queries can be executed differently and can be optimized for reading the state or any derivative of the state if such a derivative exists.

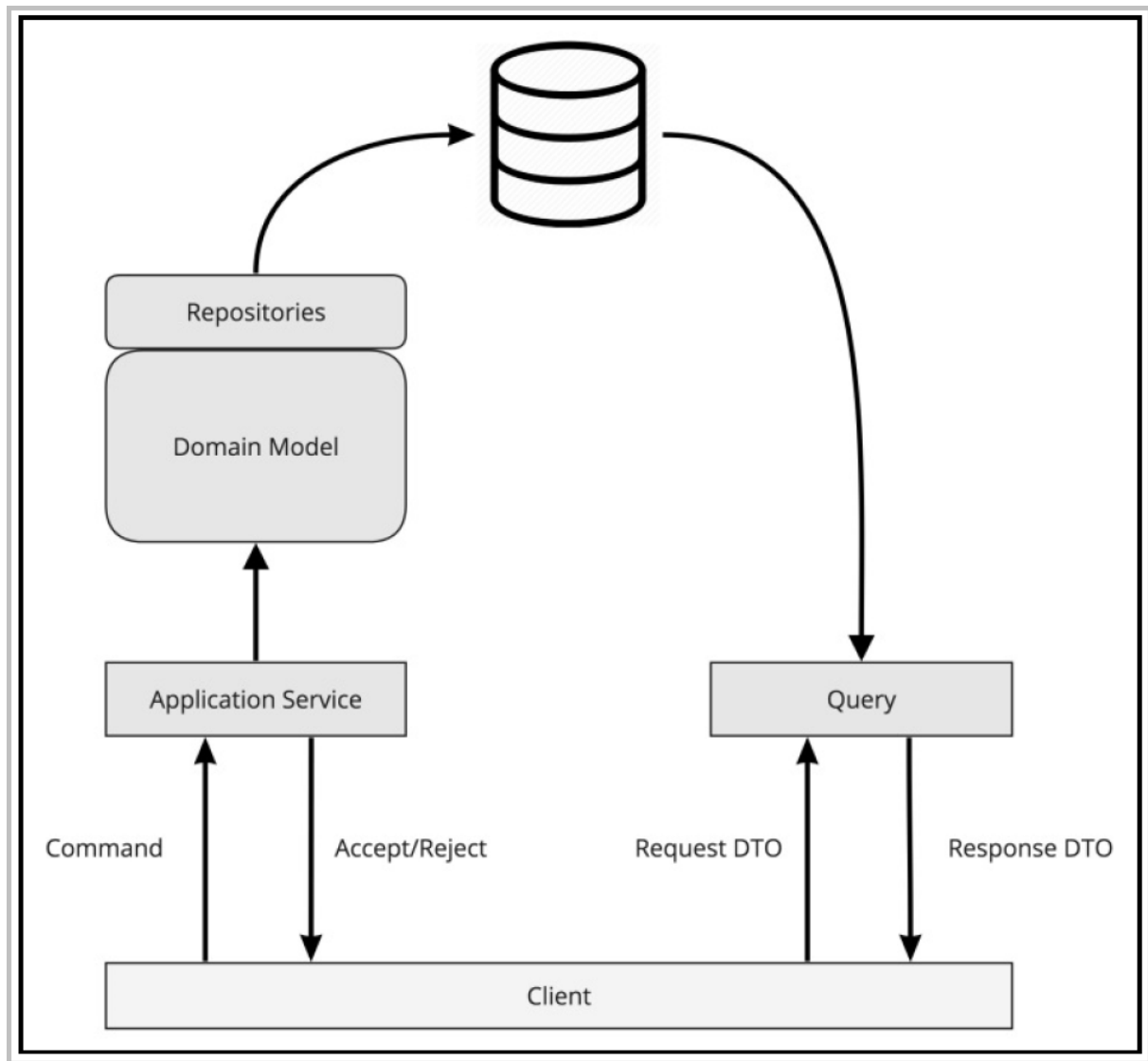


Figure 3.2:

In our system, the blue sticky notes represents commands and the red sticky notes represent state transition.

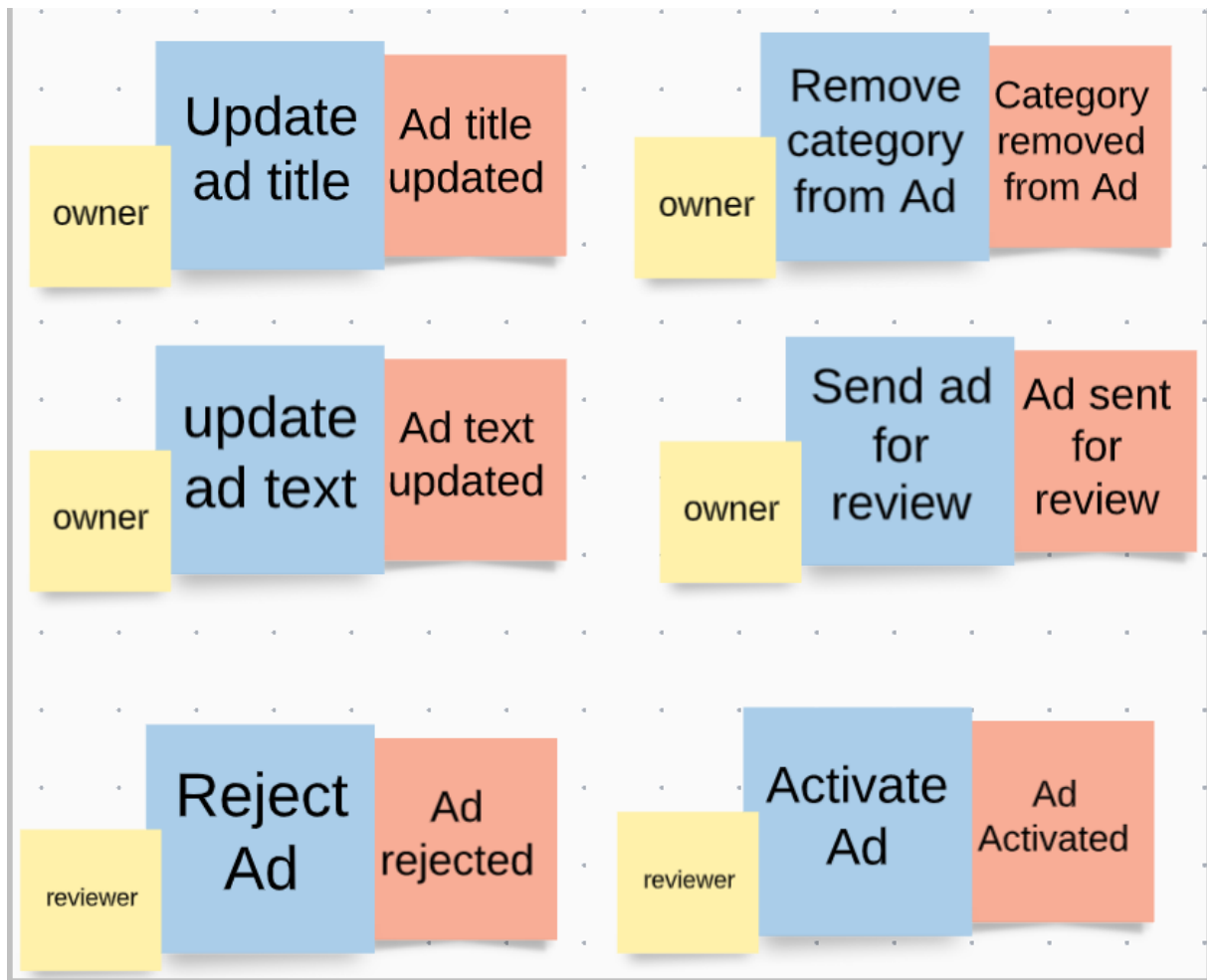


Figure 3.3:

We then took the same methods from the Event Storming sticky notes and converted them into commands.

```
5 references | 2/4 passing | Rahman Saleh, 16 hours ago | 1 author, 1 change
public void SetTitle(JobAdTitle title) =>
    Apply(event: new Events.JobAdTitleChanged
    {
        Id = Id,
        Title = title
    });

5 references | 2/4 passing | Rahman Saleh, 16 hours ago | 1 author, 1 change
public void UpdateText(JobAdText text) =>
    Apply(event: new Events.JobAdTextUpdated
    {
        Id = Id,
        AdText = text
    });

5 references | 3/4 passing | Rahman Saleh, 16 hours ago | 1 author, 1 change
public void UpdateSalary(Salary salary) =>
    Apply(event: new Events.JobAdSalaryUpdated
    {
        Id = Id,
        Salary = salary.Amount,
        CurrencyCode = salary.Currency.CurrencyCode
    });

6 references | 3/5 passing | Rahman Saleh, 16 hours ago | 1 author, 1 change
public void RequestToPublish() =>
    Apply(event: new Events.JobAdSentForReview { Id = Id });
```

Figure 3.4:

4 Refactoring

In the beginning we implemented everything inside our Entity JobAd, including raising events. But since we are going to have multiple Entities, we refactored the code to have a class Entity, which then the JobAd Entity and all future entities are going to inherit from.

```
namespace JobMarket.Library
{
    2 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public abstract class Entity<TId> where TId : IEquatable<TId>
    {
        private readonly List<object> _events;

        0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        protected Entity() => _events = new List<object>();

        5 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        protected void Apply(object @event)
        {
            When(@event);
            EnsureValidState();
            _events.Add(@event);
        }

        2 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        protected abstract void When(object @event);

        0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        public IEnumerable<object> GetChanges() => _events.AsEnumerable();

        0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        public void ClearChanges() => _events.Clear();

        2 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        protected abstract void EnsureValidState();
    }
}
```

Figure 4.1:

```
8 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
public class JobAd : Entity<JobAdId>
{
    6 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public JobAdId Id { get; private set; }
    2 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public UserId OwnerId { get; private set; }
    3 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public JobAdTitle Title { get; private set; }
    3 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public JobAdText Text { get; private set; }
    3 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public Salary Salary { get; private set; }
    5 references | 1/1 passing | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public JobAdState State { get; private set; }
    1 reference | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public UserId ApprovedBy { get; private set; }

    2 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public JobAd(JobAdId id, UserId ownerId) =>
        Apply(event: new Events.JobAdCreated
        {
            Id = id,
            OwnerId = ownerId
        }));
```

Figure 4.2:

5 Unit Tests

We created a fake mockup to check if the currency is real, and then we implemented it in the Money_spec class which is used in the Salary Class.

```
5 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
public class FakeCurrencyLookup : ICurrencyLookup
{
    private static readonly IEnumerable<Currency> currencies =
        new[]
        {
            new Currency
            {
                CurrencyCode = "EUR",
                DecimalPlaces = 2,
                InUse = true
            },
            new Currency
            {
                CurrencyCode = "USD",
                DecimalPlaces = 2,
                InUse = true
            },
            new Currency
            {
                CurrencyCode = "JPY",
                DecimalPlaces = 0,
                InUse = true
            },
            new Currency
            {
                CurrencyCode = "DEM",
                DecimalPlaces = 2,
                InUse = false
            }
        };
};
```

Figure 5.1:

```
0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
public class Money_Spec
{
    private static readonly ICurrencyLookup CurrencyLookup =
new FakeCurrencyLookup();

    [Fact]
    0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public void Two_of_same_amount_should_be_equal()
    {
        var firstAmount:Money = Money.FromDecimal(amount:5, currency:"EUR", CurrencyLookup);
        var secondAmount:Money = Money.FromDecimal(amount:5, currency:"EUR", CurrencyLookup);

        Assert.Equal(expected:firstAmount, actual:secondAmount);
    }

    [Fact]
    0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public void Two_of_same_amount_but_different_Currencies_should_not_be_equal()
    {
        var firstAmount:Money = Money.FromDecimal(amount:5, currency:"EUR", CurrencyLookup);
        var secondAmount:Money = Money.FromDecimal(amount:5, currency:"USD", CurrencyLookup);

        Assert.NotEqual(expected:firstAmount, actual:secondAmount);
    }
}
```

Figure 5.2:

And here we wrote some tests for our JobAd Entity, which tests some Entity methods.

```

1 reference | Rahman Saleh, 2 hours ago | 1 author, 2 changes
public class JobAd_Publish_Spec
{
    private readonly JobAd _jobAd;

    0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public JobAd_Publish_Spec()
    {
        _jobAd = new JobAd(
            id: new JobAdId(Guid.NewGuid()),
            ownerId: new UserId(Guid.NewGuid()));
    }

    [Fact]
    0 references | Rahman Saleh, 2 hours ago | 1 author, 2 changes
    public void Can_publish_a_valid_ad()
    {
        _jobAd.SetTitle(JobAdTitle.FromString("Test ad"));
        _jobAd.UpdateText(JobAdText.FromString("Job Description"));
        _jobAd.UpdateSalary(
            Salary.FromDecimal(amount: 100.10m, currency: "EUR", new FakeCurrencyLookup()));

        _jobAd.RequestToPublish();

        Assert.Equal(expected: JobAd.JobAdState.PendingReview,
            actual: _jobAd.State);
    }

    [Fact]
    0 references | Rahman Saleh, 2 hours ago | 1 author, 2 changes
    public void Cannot_publish_without_title()
    {
        _jobAd.UpdateText(JobAdText.FromString("Job Description"));
        _jobAd.UpdateSalary(
            Salary.FromDecimal(amount: 100.10m, currency: "EUR", new FakeCurrencyLookup()));

        Assert.Throws<InvalidEntityStateException>(testCode: () => _jobAd.RequestToPublish());
    }
}

```

Figure 5.3:

6 API

Inside the application we created a folder to hold all our API files. The first API is for our JobAd Entity, where we will have all our JobAd endpoints.

```
namespace JobMarket.API
{
    [Route(template: "/ad")]
    1 reference | Rahman Saleh, 16 hours ago | 1 author, 1 change
    public class JobAdsCommandsApi : Controller
    {
        private readonly JobAdsApplicationService
            _applicationService;

        0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        public JobAdsCommandsApi(
            JobAdsApplicationService applicationService
        )
        => _applicationService = applicationService;

        [HttpPost]
        0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        public async Task<IActionResult> Post(V1.Create request)
        {
            await _applicationService.Handle(request);
            return Ok();
        }

        [Route(template: "title")]
        [HttpPut]
        0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        public async Task<IActionResult> Put(V1.SetTitle request)
        {
            await _applicationService.Handle(request);
            return Ok();
        }

        [Route(template: "text")]
        [HttpPut]
        0 references | Rahman Saleh, 16 hours ago | 1 author, 1 change
        public async Task<IActionResult> Put(V1.UpdateText request)
        {
            await _applicationService.Handle(request);
            return Ok();
        }
    }
}
```

Figure 6.1:

We are using Swagger UI to show and test our endpoints.



Figure 6.2: