

khalilstemmler.com

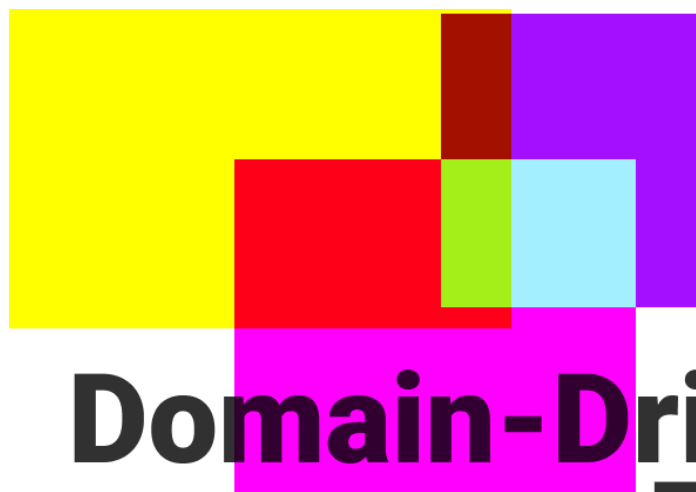
# Implementing DTOs, Mappers & the Repository Pattern using the Sequelize ORM [with Examples] - DDD w/ TypeScript

Domain-Driven Design

There are several patterns that we can utilize in order to handle data access concerns in Domain-Driven Design. In this article, we talk about the role of DTOs, repositories & data mappers in DDD.

ddd   typescript   software design

khalilstemmler.com



**DTOs, Mappers  
& Repositories**

# **Domain-Driven Design with TypeScript**

This is part of the [Domain-Driven Design w/ TypeScript & Node.js](#) course. Check it out if you liked this post.

*Also from the [Domain-Driven Design with TypeScript](#) series.*

What's responsible for handling validation logic? *Value Objects*.

Where do you handle domain logic? As close to the *Entity* as possible, otherwise *domain services*.

Perhaps one of the hardest aspects towards learning DDD is being able to determine *just what that tool is* needed for the particular task.

In DDD, the Repositories, Data Mappers and DTOs are a critical part of the entity lifecycle that enable us to store, reconstitute and delete domain entities.

This type of logic is called "*Data Access Logic*".

For developers coming from building REST-ful CRUD APIs using MVC without much attention to encapsulating ORM data access logic, you'll learn:

- problems that occur when we don't encapsulate ORM data access logic
- how DTOs can be used to stabilize an API
- how Repositories act as a facade over complex ORM queries

khalilstemmler.com

and ORM models

## How do we usually use ORM models in MVC apps?

In a [previous article about MVC](#), we looked at some of the most common approaches to utilizing ORMs like [Sequelize](#).

Take this simple `controller` where we create a `User` .

```
class UserController extends BaseController {
  async exec (req, res) => {
    try {
      const { User } = models;
      const { username, email, password } = req.body;
      const user = await User.create({ username, email, password });
      return res.status(201);
    } catch (err) {
      return res.status(500);
    }
  }
}
```

khalilstemmler.com

- this code is increasingly easy to read
- on small projects, this approach makes it easy to quickly become productive

However, as our applications grow and get more complex, this approach leads to several drawbacks *which may introduce bugs*.

Summarizing [what we spoke about last time](#), the main reason why it's problematic is because there's a lack of a separation of concerns. This block of code is responsible for too many things:

- handling the API request ([controller](#) responsibility)
- performing validation on the domain object (not present here, but a [domain entity](#) or [value object](#) responsibility)
- persisting a domain entity to the database (repository responsibility)

When we add more and more code to our project, it becomes really important that we pay attention to assigning [singular responsibility](#) to our classes.

khalilstemmler.com

Here's an example where lack of encapsulation towards how we retrieve data from ORMs may lead to introducing bugs.

Let's say we were working on our [Vinyl Trading app](#) and we're tasked with creating 3 different API calls.

```
GET /vinyl?recent=6      - GET the 6 newest listed vinyl
GET /vinyl/:vinylId/     - GET a particular vinyl by it's id
GET /vinyl/owner/:userId/ - GET all vinyl owned by a particular user
```

In each of these API calls, we need to return `Vinyl` view models [1](#).

So let's do the first controller: returning recent vinyl.

```
export class GetRecentVinylController extends BaseController {
  private models: any;

  public constructor (models: any) {
    super();
    this.models = models;
  }
}
```

khalilstemmler.com

```
const count: number = this.req.query.count;

const result = await Vinyl.findAll({
  where: {},
  include: [
    { owner: User, as: 'Owner', attributes: ['user_id', 'display_name'] },
    { model: Genre, as: 'Genres' },
    { model: Track, as: 'TrackList' },
  ],
  limit: count ? count : 12,
  order: [
    ['created_at', 'DESC']
  ],
})

return this.ok(this.res, result);
} catch (err) {
  return this.fail(err);
}
}
```

OK. Not bad. If you're familiar with Sequelize, this is probably pretty standard for you.

khalilstemmler.com

```
private models: any;

public constructor (models: any) {
  super();
  this.models = models;
}

public async executeImpl(): Promise<any> {
  try {
    const { Vinyl, Track, Genre } = this.models;
    const vinylId: string = this.req.params.vinylId;

    const result = await Vinyl.findOne({
      where: {
        vinyl_id: vinylId
      },
      include: [
        { model: User, as: 'Owner', attributes: ['user_id', 'display_name'] },
        { model: Genre, as: 'Genres' },
        { model: Track, as: 'TrackList' },
      ]
    })

    return this.ok(this.res, result);
  } catch (err) {
    return this.fail(err);
  }
}
```



khalilstemmler.com

Not too much is different between those two classes, eh? 😊

So this is definitely not following the DRY principle, because we've repeated a lot here.

And you can expect that the 3rd API call is going to be somewhat similar to this.

So far, the main problem that we're noticing is:

code duplication

While that's bad, because if we were to need to add a new *relationship* to this model (like `Label` ), we'd have to *remember to* scan through our code, locate each `findOne()` and `findAll()` for the `Vinyl` model, and add the new `{ model: Label, as: 'Label' }` to our `include` array, it's not the worst aspect to this.

There's another problem on the brewing on the horizon...

khalilstemmler.com

Notice how we're passing back the ORM query results directly?

```
return this.ok(this.res, result);
```

That's what's going back out to the client in response to the API calls.

Well, what happens when we perform migrations on the database and add new columns? Worse- what happens when we remove a column or *change* the name of a column?

We've just broken the API for each client that depended on it.

Hmm... we need a tool for this.

Let's reach into our enterprise toolbox and see what we find...

Ah, the DTO (Data Transfer Object).

khalilstemmler.com

## separate systems.

When we're concerned with web development, we think of DTOs as **View Models** because they're faux models. They're not really the REAL domain models, but they contain as much data that the view needs to know about.

For example, the `Vinyl` view model / DTO could be built up to look like this:

```
type Genre = 'Post-punk' | 'Trip-hop' | 'Rock' | 'Rap' | 'Electronic' | 'Pop';
```

```
interface TrackDTO {  
  number: number;  
  name: string;  
  length: string;  
}
```

```
type TrackCollectionDTO = TrackDTO[];
```

```
// Vinyl view model / DTO, this is the format of the response
```

```
interface VinylDTO {  
  albumName: string;  
  label: string;  
  country: string;  
  yearReleased: number;
```

The reason why this is so powerful is because we've just standardized our API response structure.

Our DTO is a **data contract**. We're telling anyone who uses this API, "hey, this is going to be the format that you can always expect to see from this API call".

Here's what I mean. Let's look at how we could use this in the example of retrieving Vinyl by id.

```
export class GetVinylById extends BaseController {  
  private models: any;  
  
  public constructor (models: any) {  
    super();  
    this.models = models;  
  }  
  
  public async executeImpl(): Promise<any> {  
    try {  
      const { Vinyl, Track, Genre, Label } = this.models;  
      const vinylId: string = this.req.params.vinylId;
```

khalilstemmler.com

```
    { model: User, as: 'Owner', attributes: ['user_id', 'display_name'] },
    { model: Label, as: 'Label' },
    { model: Genre, as: 'Genres' },
    { model: Track, as: 'TrackList' },
  ]
});

// Map the ORM object to our DTO
const dto: VinylDTO = {
  albumName: result.album_name,
  label: result.Label.name,
  country: result.Label.country,
  yearReleased: new Date(result.release_date).getFullYear(),
  genres: result.Genres.map((g) => g.name),
  artistName: result.artist_name,
  trackList: result.TrackList.map((t) => ({
    number: t.album_track_number,
    name: t.track_name,
    length: t.track_length,
  })))
}

// Using our baseController, we can specify the return type
// for readability.
return this.ok<VinylDTO>(this.res, dto)
} catch (err) {
```

khalilstemmler.com

That's great, but let's think about the responsibility of this class now.

This is a `controller` , but it's responsible for:

- defining the how to map persisted ORM models to `VinylDTO` , `TrackDTO` , and `Genres` .
- defining just how much data needs to get retrieved from the Sequelize ORM call in order to successfully create the DTOs.

That's quite a bit more than `controllers` should be doing.

Let's look into **Repositories** and **Data Mappers**.

We'll start with **Repositories**.

## Repositories

Repositories are Facades to persistence technologies (such as ORMs)

A **Facade** is some design pattern lingo that refers to an object that provide a *simplified* interface to a larger body of code. In *our* case, that larger body of code is domain entity persistence and domain entity retrieval logic.

## The role of repositories in DDD & clean architecture

In DDD & clean architecture, repositories are [infrastructure-layer](#) concerns.

Generally speaking, we said that repos **persist** and **retrieve** domain entities.

### | Persistence objectives

- Scaffold complex persistence logic across [junction](#) and relationship tables.
- Rollback transactions that fail

With respect to doing the "create if not exists, else update", that's the type of complex data access logic that we don't want any other constructs in our domain to have to know about: only the repos should care about that.

## | Retrieval objectives

We've seen a little bit of this but the goals are ultimately to:

- Retrieve the entirety of data needed to create domain entities
  - ie: we've seen this with choosing what to `include: []` with Sequelize in order to create DTOs and Domain Objects.
- Delegate the responsibility of *reconstituting* entities to a `Mapper` .

## Approaches to writing repositories

There are several different approaches to creating repositories in your application.



khalilstemmler.com

that you'd have to do to a model like `getById(id: string)` , `save(t: T)` or `delete(t: T)` .

```
interface Repo<T> {  
  exists(t: T): Promise<boolean>;  
  delete(t: T): Promise<any>;  
  getById(id: string): Promise<T>;  
  save(t: T): Promise<any>;  
}
```

It's a good approach in the sense that we've defined a common way for repositories to be created, but we might end up seeing the details of the data access layer leaking into calling code.

The reason for that is because saying `getById` is feels like of *cold*. If I was dealing with a `VinylRepo` , I'd prefer to say `getVinylById` because it's a lot more descriptive to the Ubiquitous Language of the domain. And if I wanted all the vinyl owned by a particular user, I'd say `getVinylOwnedByUserId` .

Having methods like `getById` is pretty YAGNI.

khalilstemmler.com

I like being able to quickly add convenience methods that make sense to the domain that I'm working in, so I'll usually start with a slim base repository:

```
interface Repo<T> {  
  exists(t: T): Promise<boolean>;  
  delete(t: T): Promise<any>;  
  save(t: T): Promise<any>;  
}
```

And then extend that with additional methods that say more about the domain.

```
export interface IVinylRepo extends Repo<Vinyl> {  
  getVinylById(vinylId: string): Promise<Vinyl>;  
  findAllVinylByArtistName(artistName: string): Promise<VinylCollection>;  
  getVinylOwnedByUserId(userId: string): Promise<VinylCollection>;  
}
```

*The reason why it's beneficial to always define repositories as interfaces first is because it adheres to the [Liskov Substitution Principle](#) (which enables concretions to*

khalilstemmler.com

Let's go ahead and create a concrete class of our `IVinylRepo` .

```
import { Op } from 'sequelize'
import { IVinylRepo } from './IVinylRepo';
import { VinylMap } from './VinyMap';

class VinylRepo implements IVinylRepo {
  private models: any;

  constructor (models: any) {
    this.models = models;
  }

  private createQueryObject (): any {
    const { Vinyl, Track, Genre, Label } = this.models;
    return {
      where: {},
      include: [
        { model: User, as: 'Owner', attributes: ['user_id', 'display_name'], where: {} },
        { model: Label, as: 'Label' },
        { model: Genre, as: 'Genres' },
        { model: Track, as: 'TrackList' },
      ]
    }
  }
}
```

khalilstemmler.com

```
        where: { vinyl_id: vinyl.id.toString() }
    });
    return !!result === true;
}

public delete (vinyl: Vinyl): Promise<any> {
    const VinylModel = this.models.Vinyl;
    return VinylModel.destroy({
        where: { vinyl_id: vinyl.id.toString() }
    })
}

public async save(vinyl: Vinyl): Promise<any> {
    const VinylModel = this.models.Vinyl;
    const exists = await this.exists(vinyl.id.toString());
    const rawVinylData = VinylMap.toPersistence(vinyl);

    if (exists) {
        const sequelizeVinyl = await VinylModel.findOne({
            where: { vinyl_id: vinyl.id.toString() }
        });

        try {
            await sequelizeVinyl.update(rawVinylData);
            // scaffold all of the other related tables (VinylGenres, Tracks, etc)
            // ...
        }
    }
}
```

khalilstemmler.com

```
        await VinylModel.create(rawVinylData);
    }

    return vinyl;
}

public getVinylById(vinylId: string): Promise<Vinyl> {
    const VinylModel = this.models.Vinyl;
    const queryObject = this.createQueryObject();
    queryObject.where = { vinyl_id: vinylId.toString() };
    const vinyl = await VinylModel.findOne(queryObject);
    if (!!vinyl === false) return null;
    return VinylMap.toDomain(vinyl);
}

public findAllVinylByArtistName (artistName: string): Promise<VinylCollection> {
    const VinylModel = this.models.Vinyl;
    const queryObject = this.createQueryObject();
    queryObjectp.where = { [Op.like]: `%${artistName}%` };
    const vinylCollection = await VinylModel.findAll(queryObject);
    return vinylCollection.map((vinyl) => VinylMap.toDomain(vinyl));
}

public getVinylOwnedByUserId(userId: string): Promise<VinylCollection> {
    const VinylModel = this.models.Vinyl;
    const queryObject = this.createQueryObject();
```

khalilstemmler.com

```
}
```

See that we've encapsulated our sequelize data access logic? We've removed the need for repeatedly writing the `include` s because all of the required include statements are here now.

We've also referred to a `VinylMap` . Let's take a quick look at the responsibility of a **Mapper**.

## Data Mappers

The responsibility of a **Mapper** is to make all the transformations:

- From Domain to DTO
- From Domain to Persistence
- From Persistence to Domain

khalilstemmler.com

```
public static toDomain (raw: any): Vinyl {
  const vinylOrError = Vinyl.create({
    albumName: AlbumName.create(raw.album_name).getValue(),
    artistName: ArtistName.create(raw.artist_name).getValue(),
    tracks: raw.TrackList.map((t) => TrackMap.toDomain(t))
  }, new UniqueEntityID(raw.vinyl_id));
  return vinylOrError.isSuccess ? vinylOrError.getValue() : null;
}

public static toPersistence (vinyl: Vinyl): any {
  return {
    album_name: vinyl.albumName.value,
    artist_name: vinyl.artistName.value
  }
}

public static toDTO (vinyl: Vinyl): VinylDTO {
  return {
    albumName: vinyl.albumName.value,
    label: vinyl.Label.name.value,
    country: vinyl.Label.country.value,
    yearReleased: vinyl.yearReleased.value,
    genres: vinyl.Genres.map((g) => g.name),
    artistName: vinyl.artistName.value,
    trackList: vinyl.tracks.map((t) => TrackMap.toDTO(t))
  }
}
```

khalilstemmler.com

OK, now let's go back and refactor our controller from earlier using our `VinylRepo` and `VinylMap` .

```
export class GetVinylById extends BaseController {
  private vinylRepo: IVinylRepo;

  public constructor (vinylRepo: IVinylRepo) {
    super();
    this.vinylRepo = vinylRepo;
  }

  public async executeImpl(): Promise<any> {
    try {
      const { VinylRepo } = this;
      const vinylId: string = this.req.params.vinylId;
      const vinyl: Vinyl = await VinylRepo.getVinylById(vinylId);
      const dto: VinylDTO = VinylMap.toDTO(vinyl);
      return this.ok<VinylDTO>(this.res, dto)
    } catch (err) {
      return this.fail(err);
    }
  }
}
```



khalilstemmler.com

## CONCLUSION

In this article, we took an in-depth look at **DTOs**, **Repositories** and **Data Mappers** which all have their own single responsibility in the infrastructure layer.

## The *actual* repo

If you'd like to see some real life code, all of the code that was used in this article can be found in the my Vinyl Trading app (Node.js + TypeScript + Sequelize + Express) that I'm working on for the upcoming DDD course. Check it out here:

- <https://github.com/stemmlerjs/white-label>

This is part of the [Domain-Driven Design w/ TypeScript & Node.js](#) course. Check it out if you liked this post.

<sup>1</sup> View models are *essentially* the same thing as DTOs (Data Transfer Objects).

khalilstemmmler.com

Liked this? Only if you read and proud 🙌

Share on Twitter

## 7 Comments

**B** *I* U 🔗 </>

Submit

**Sebas** 5 months ago

Hi Khalil! I'm learning a lot with all your articles! Continue like that! just AWESOME!

**Pjotr** 5 months ago

Great articles! Your blog became one of my favourites. I have one question:

khalilstemmler.com

**Khalil Stemmler** 5 months ago

Hey Pjotr, thanks for reading!

The question here is "when to use static methods/functions vs. dependency injection".

When your functions (static methods are a form of functions) are pure and have no dependencies, using dependency injection is overkill.

Another statement I could make is that it doesn't make sense to dependency inject functions because the prime benefit of dependency injection is the ability to invert the dependency graph ([dependency inversion](#)). When functions are pure, there's usually no need to abstract away functions with an abstraction.

Think about it this way: can you think of a good reason to dependency inject the `JSON.stringify` function? What about the `parseInt()` function? Not really, right?

**Vasu Bhardwaj** 5 months ago

I've been going through this series of articles all day. Really informative. Thanks

**Jake** 4 months ago

khalilstemmler.com

Could you go into a little more detail about where this method would be used elsewhere in an application and how to decide what data should be included in it's return value?

**Khalil Stemmler** 3 months ago

Ah yes, ``toPersistence()`` would probably be better named something like ``toSequelize()`` because it's the format that Sequelize needs it to be in for it to be saved.

The ``toPersistence()`` method is simply to satisfy Sequelize's ``update(props: any)`` and ``create(props: any)`` API.

If you were using MongoDB, or Postgres, it might be the same, it might be a different format. The Mapper class is helping to serialize the data so that it can be saved by whichever persistence mechanism you want to use.

The ``toDTO()`` method is used to convert a domain object to a DTO (view model in the format that a RESTful API or GraphQL type expects it to be in).

**Demisx** 4 months ago

Great and very helpful article. Please run it through a spellcheck, though. :)

**Lucho** 2 months ago

Dude! I'm loving these series!

khalilstemmler.com

The main takeaway is that whoever wants a vinyl repo doesn't care if it is an interface or a concrete class, it cares that it respects the expected contract, and then we could potentially have different implementations, e.g. MockVinylRepo, or a more extreme case: MongoVinylRepo.

**test** 2 months ago

testtestsetsetes

## Stay in touch!

We're just getting started 🙌 Interested in how to write professional JavaScript and TypeScript? Join 5000+ other developers learning about Domain-Driven Design and Enterprise Node.js. I won't spam ya. 🙌 Unsubscribe anytime.

Get notified



khalilstemmler.com

Khalil is a software developer, writer, and musician. He frequently publishes articles about Domain-Driven Design, software design and Advanced TypeScript & Node.js best practices for large-scale applications.

Follow @stemmlerjs

1,976 followers


 Follow

605

View more in [Domain-Driven Design](#)

khalilstemmler.com

**Learn to write  
testable, flexible  
and maintainable  
code**



**Khalil Stemmler**

**S O L I D**  
**S O L I D**

The Software Design  
& Architecture Handbook

**Get the book**

## You may also enjoy...

A few more related articles

khalilstemmler.com



# How to Handle Updates on Aggregates - Domain-Driven Design w/ TypeScript

Domain-Driven Design

ddd   typescript   software design   aggregate root   aggregate   sequelize

In this article, you'll learn approaches for handling aggregates on Aggregates in Domain-Driven Design.



khalilstemmler.com



# Decoupling Logic with Domain Events [Guide] - Domain-Driven Design w/ TypeScript

## Domain-Driven Design

ddd   typescript   software design   domain events   sequelize   typeorm

In this article, we'll walk through the process of using Domain Events to clean up how we decouple complex domain logic across the...

khalilstemmler.com



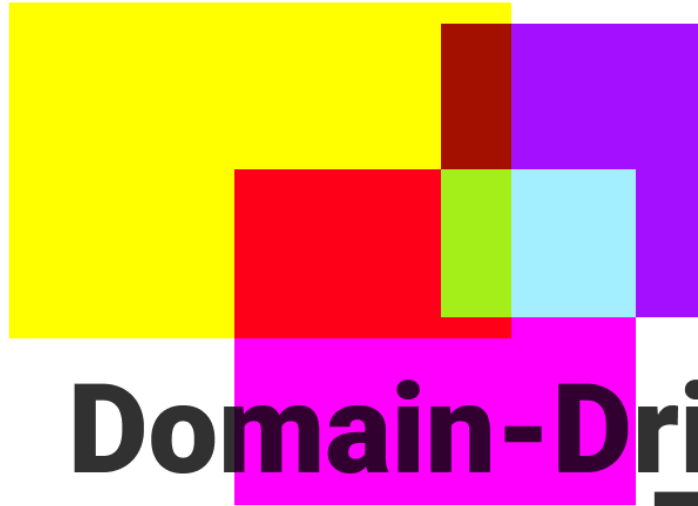
# Does DDD Belong on the Frontend? - Domain-Driven Design w/ TypeScript

Domain-Driven Design

ddd   typescript   software design   frontend development

Should we utilize Domain-Driven Design principles and patterns in front-end applications? How far does domain modeling reach from ...

khalilstemmler.com



# Introduction

# Domain-Driven Design

## with TypeScript

## An Introduction to Domain-Driven Design - DDD w/ TypeScript

Domain-Driven Design

ddd   typescript   software design

khalilstemmler.com

I'm Khalil. I teach advanced TypeScript & Node.js best practices for large-scale applications. Learn to write flexible , maintainable software.

## Menu

About

Articles

Blog

Portfolio

Wiki

## Contact

khalilstemmler.com

## Social

GitHub

Twitter

Instagram

LinkedIn

© khalilstemmler • 2020 • Built with  • Open sourced on  • Deployed on 