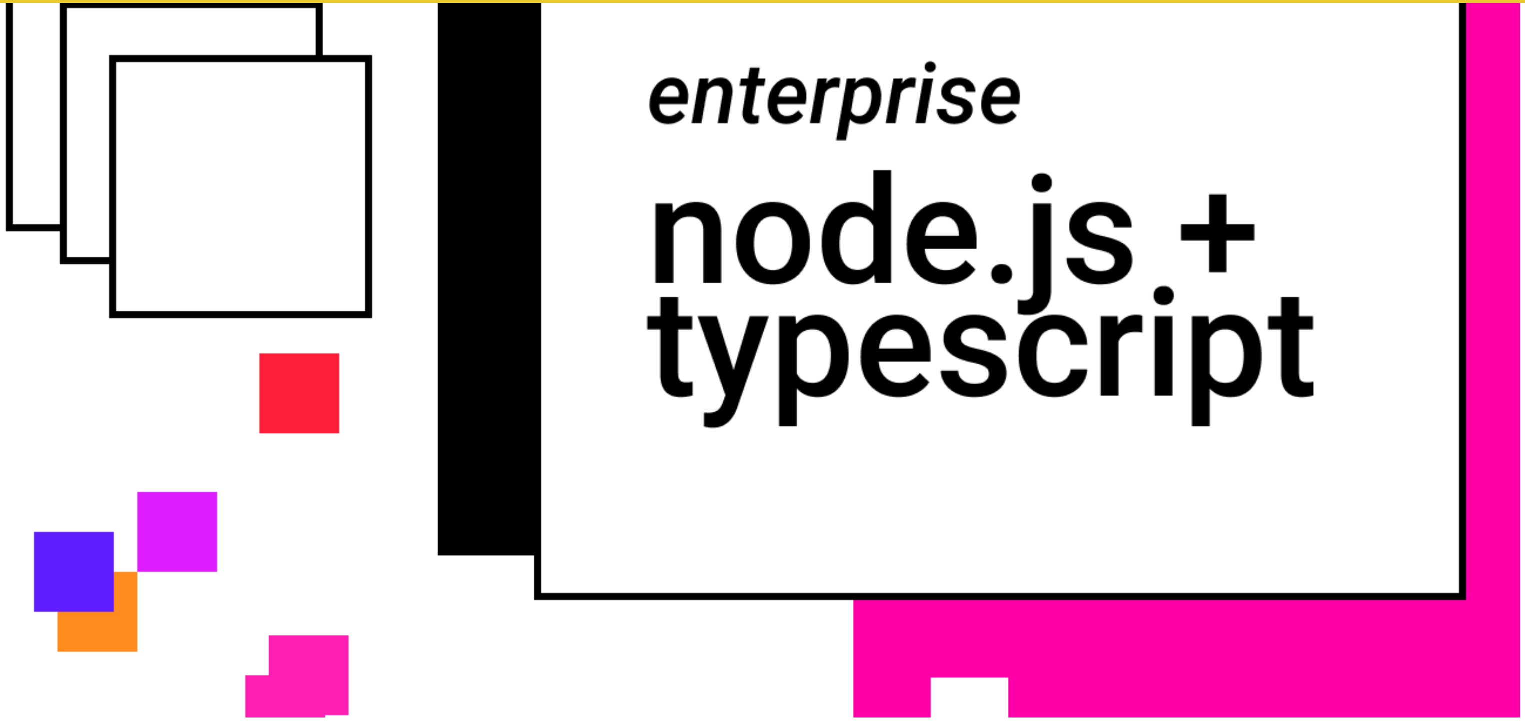
khalilstemmler.com

Better Software Design with Application Layer Use Cases | Enterprise Node.js + TypeScript

Enterprise Node + TypeScript

In this article, we explore how organizing application logic as Use Cases in the application layer helps to make large typescript projects easier to reason about.

node.js typescript use cases enterprise software



The term *Use Case* is used to describe one of the potential ways that our software can be used.

Another word that's sometimes used is *feature*, though the concept is nearly identical.

The whole purpose of building software is to address one or more Use Cases.

khalilstemmler.com

What if there was a particular *construct* that actually **appeared** in our code that would **describe** all of the different capabilities of our app?

Something that would help towards encapsulating, organizing, and keeping track of all of the things that our application is capable of.

Well, it *does* exist, and it's properly called: *you guessed it*, a **Use Case**.

In the Clean Architecture, Use Cases are an **application layer** concern that encapsulate the business logic involved in executing the features within our app(s).

In this article, we'll cover the following topics towards structuring Node.js/TypeScript applications using Use Cases in the application layer:

- How to discover Use Cases
- The role of the application layer

To see the code used in this article, check out:

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

Discovering Use Cases

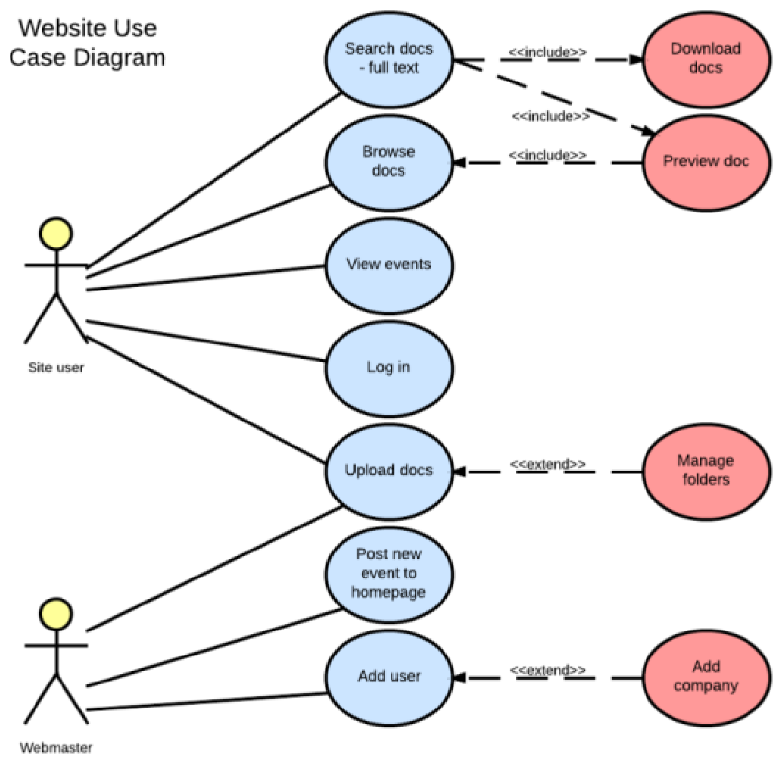
There are several different ways to plan out building an application. For a long time, I simply planned out how I would actually build something by designing the API first.

Later on, I ended up moving more towards wire-framing and starting from the interface first, because the front-end would often dictate what's really needed and what's YAGNI.

It wasn't until the projects I started working on got so complex that I realized I needed to take a more traditional approach to software planning: **Use Case design**.

Traditional approach

who uses it), and the circles represent all of the actual use cases that we'd like to enable them to execute with our software.



things out using a kind of free-form approach.

Use Case Basics

The most important things to understand about identifying use cases are:

- 1. Who the `actors` of the system are (who's executing the use cases)
- 2. That use cases are either **commands** or **queries**
- 3. That use cases belong to a particular **subdomain** which may be deployed on separate **bounded contexts**

1. Actors

It's easy to just call every `actor` a `User`. We could do that, but it doesn't say much about the domain itself.

about their *role* in the domain.

Here are some alternatives to `User` depending on the domain:

- A billing system: `Customer`, `Subscriber`, `Accountant`, `Treasurer`, `Employee`
- A blogging system: `Editor`, `Reviewer`, `Guest`, `Author`
- A recruitment platform: `JobSeeker`, `Employer`, `Interviewer`, `Recruiter`
- Our vinyl-trading application: `Trader`, `Admin`
- An email marketing company: `Contact`, `Recipient`, `Sender`, `ListOwner`

Get the point? **Role** matters.

2. Use Cases are Commands and Queries

A Use Case will be either a **command** or a **query**.

For example, in our Vinyl-Trading app, "White Label", an example of a particular **command** is to **add vinyl to our wishlist**. That might appear in our code as a class called `AddVinylToWishlist`.

An example of a **query** would be to **get our wishlist**, which might appear as `GetWishlist`.

In **Command-Query Segregation**, COMMANDS perform changes to the system **but return no value**, and QUERIES pull data out of the system, **but produce no side effects.**

3. Use cases belong to a particular subdomain

Generally speaking, most applications are built up of *several* subdomains.

If you don't remember what a **subdomain** from domain-driven design is, it's a logical separation of the **entire problem domain.**

For example, White Label, the Vinyl-Trading app that I'm building, is about trading vinyl.

But the problem domain isn't *only* about enabling traders to trade vinyl. There's much more that needs to be accounted for.

In addition to the trading aspect (`Trading`), the enterprise also has to account for several other subdomains: identity and access management (`Users`), cataloging items (`Catalog`), billing (`Billing`), notifications and more.

Billing
subdomain
Trading
subdomain
Catalog
subdomain
Users
subdomain

> designs that are copies of the communication structures of these organizations."

Conway's law actually helps answer a lot of questions like:

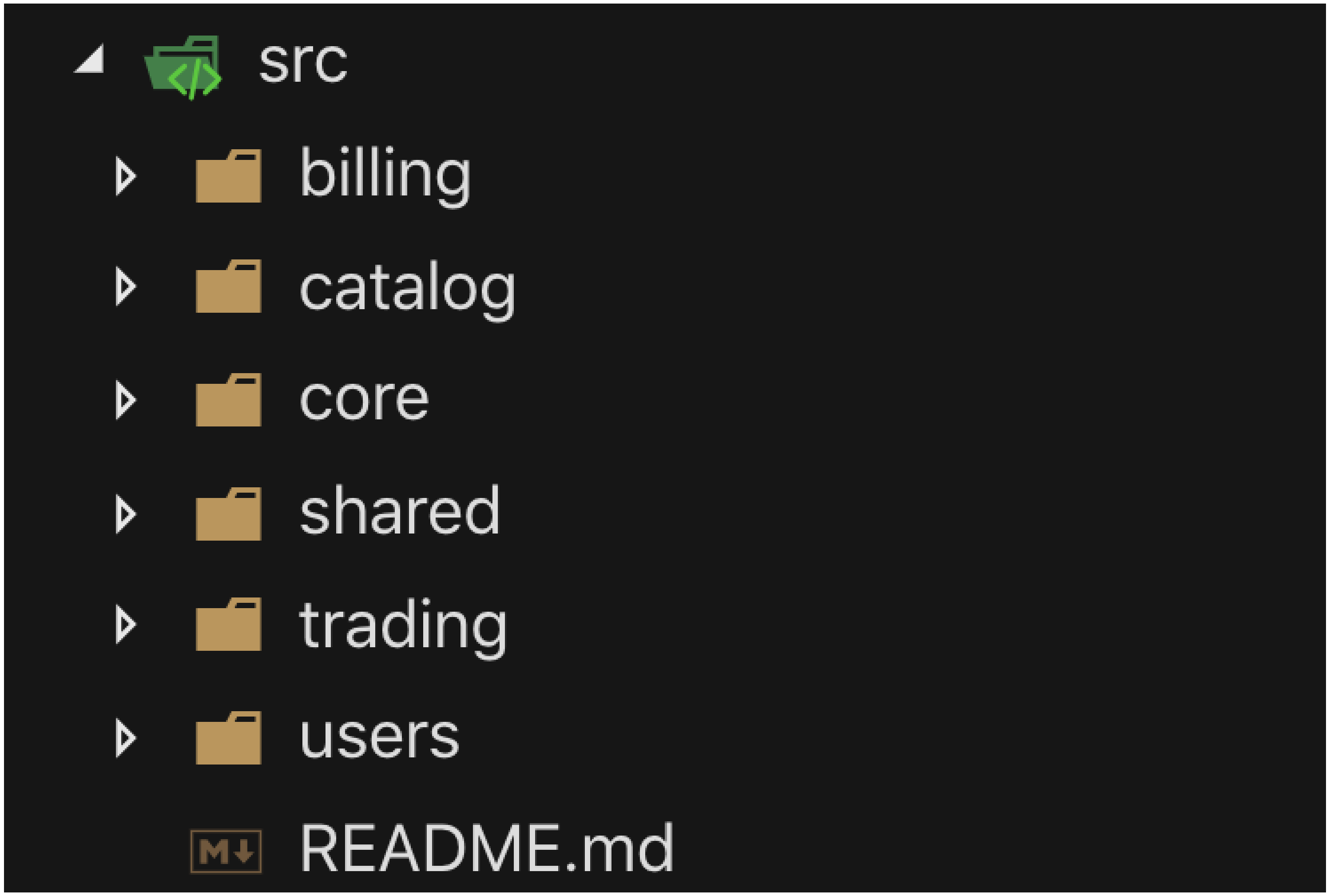
- how do we decide on our subdomains?
- how do we decide which subdomain a use case should belong to?
- how do we make it easier to change use cases in the future?

For more on Conway's Law and how it can help make these decisions, check out this article.

In any domain-driven project, we're usually able to decompose the entire problem domain into separate subdomains; some of which are essential that *we* develop ourselves (like the `Trading` and probably `Catalog` subdomain), and some of which

khalilstemmler.com

be logically separate.



```
src
  billing
  catalog
  core
  shared
  trading
  users
  README.md
```

would be **essential** in order for us to *physically* separate our problem domain into several independently deployable units.

In other words: microservices.

In DDD-lingo: separate **bounded-contexts**.

We discover use cases through conversation

A huge misconception about software development is that developers just code away in the corner and never have to talk to people.

That's *so* not true. So much of software development (especially in DDD) is consistently attempting to identify the correct language to effectively create a lasting software implementation of the model from real life.

Here's an example of the conversation to discover some of the Use Cases in White Label.

"OK, cool. So like, Discogs?"

"Yeah, pretty much. But it's only for vinyl. True hipsters."

"Awesome, I like it. What if I don't want to trade my vinyl? Can I just *buy* something?"

"Yeah, let's say: you can either trade your own vinyl for someone elses, or you can *trade* them money for it instead."

"Oh, what about this? Users can make complex offers, like hey, I'll give you these 2 Devo albums, a Sex Pistols album and 60 bucks for that limited edition Birthday Party vinyl".

"Ah, so there's `offers` and `trades` . And an `offer` can include several records and/or money in exchange for one or more records. And the person who receives the offer can either accept it, or decline the offer."

"Yeah, that sounds pretty accurate. And if they want to decline it, they can decline 'with comments' or something, saying why they decline it, and give them another

"So what are the use cases we've discovered so far?"

"

- `MakeOffer(offer: Offer)`
- `DeclineOffer(offerId: OfferId, comments?: string)`
- `AcceptOffer(offerId: OfferId)`

"

"Probably also the ability to *get* all the offers, and *get* an offer by id. We have to think about the UI as well. It's going to need some use cases."

"Ah, yes. So also `GetAllOffers(userId: User)` , `GetOfferById(offerId: OfferId)` ".

"Hmm, where did `User` come from?"

`Traders` , or `RecordCollectors` .

"Ahh, I see, the term `User` probably belongs more in a `Users & Identity` subdomain, not a... `Trading` subdomain, which is what we've been discussing so far, right?"

"Yeah, that's right."

"OK, let's go with `Traders` ."

"Awesome. So far, in the `Trading` subdomain, the Use Cases we have are:

- `MakeOffer(offer: Offer)`
- `DeclineOffer(offerId: OfferId, comments?: string)`
- `AcceptOffer(offerId: OfferId)`
- `GetAllOffers(traderId: TraderId)` and

"Me neither, let's go with that for now."

"And it looks like we've identified some of the Entities as well. `Offer` and `Trader` , right?"

"Yeah, `Offer` is probably going to be an Aggregate Root for all of the `OfferItems` as well (a collection of money + vinyl). We can figure that out later. Sounds good so far.

"Oh, and since we brought up the `Users & Identity` subdomain, should we address that as well?"

"Eh, yeah- we could. It's probably going to be the same as every other app."

"What do you mean?"

"Well, the use cases are pretty common. There's normally like something like:

- `verifyEmail(emailVerificationToken: EmailVerificationToken)`
- `changePassword(passwordResetToken: Token, password: UserPassword)`

You know what I mean? You've probably done this plenty of times."

"Ah, isn't that something we could probably outsource?"

"Yeah, we could probably try Auth0 for that."

"What do they call that in DDD-lingo, again? The type of subdomain..."

"A generic subdomain."

"Meaning?"

"Meaning while it *might* be a critical part of the business, yeah, it's not the *core* of the business. The core is probably going to be the `Trading` subdomain".

If you've been following the series on Enterprise Node.js + TypeScript, you'll recall that the Domain Layer holds all of the entities, value objects, has 0 dependencies to outer layers, and is the *first* place that we aim to place business logic, especially if it pertains to one particular entity.

For example, in White Label, if I were trying to figure out where to place invariant logic that would ensure that a `Vinyl` can only have up to at max 3 different `Genre` s, *that logic* would belong in the `Vinyl` class (which is an **aggregate root**).

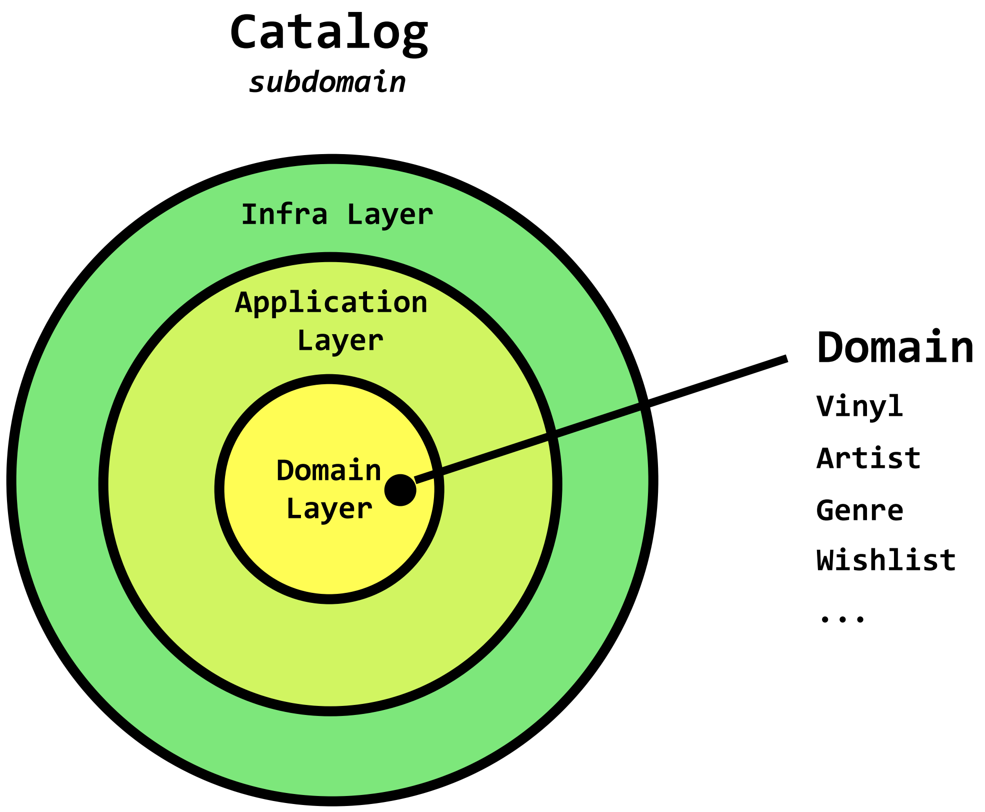
```
class Vinyl extends AggregateRoot<VinylProps> {
  ...

  addGenre (genre: Genre): Result<any> {
    if (this.props.genres.length >= MAX_NUMBER_OF_GENRES_PER_VINYL) {
      return Result.fail<any>('Max number of genres reached')
    }

    if (!this.genreAlreadyExists(genre)) {
      this.props.genres.push(genre)
    }
```

Ensuring domain model integrity by placing validation logic within the model itself.

`Vinyl` is one of many domain models from the **Domain Layer** from the `Catalog` subdomain.



So then, what's the role of the **application layer**?

> The application layer contains the **Use Cases** for a **particular subdomain** in our application.

The use cases describe the *features* of the **application**, which may be independently deployed or deployed as a monolith.

That means that when we put the use cases directly into a subdomain, we can understand the capabilities of that subdomain right away.

In DDD-lingo, Use Cases are the **application services**. They're responsible for retrieving the domain entities in addition to the information that they need in order to execute some domain logic.

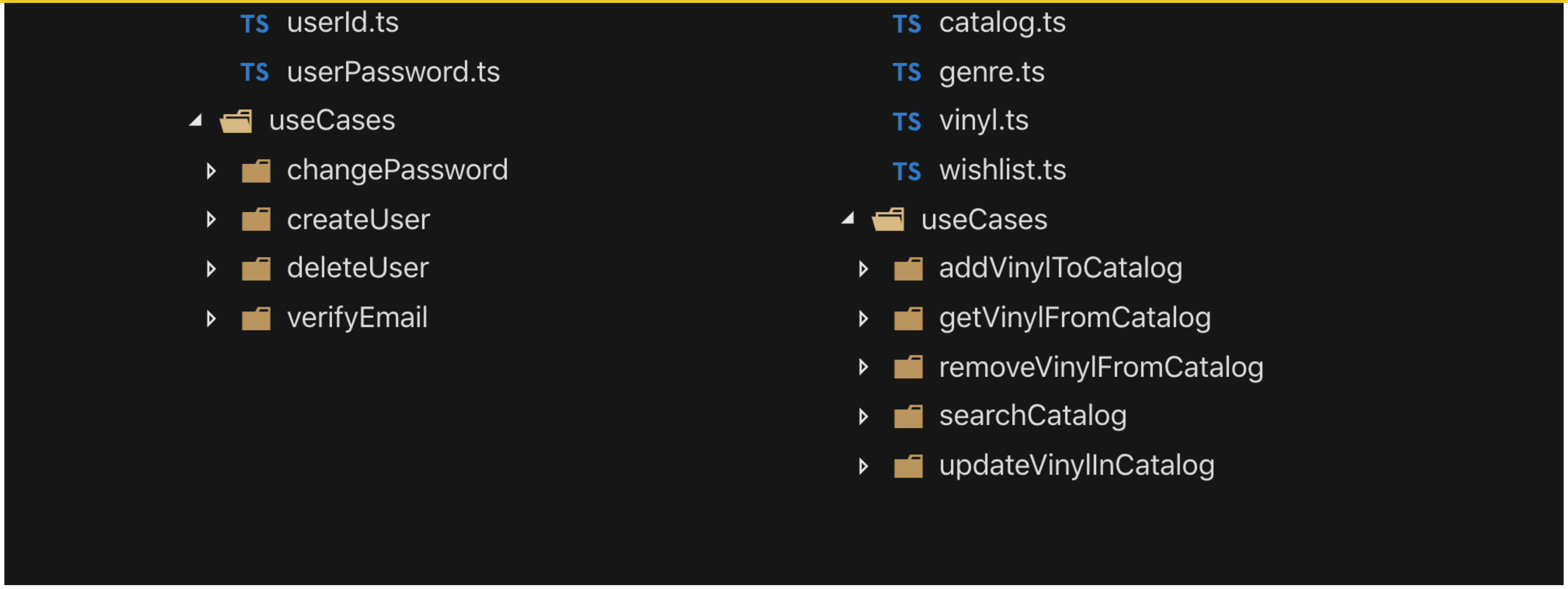
For example, in the `AcceptOffer(offerId: OfferId)` use case, all I have is the `OfferId` . That's not enough for me to do the *accept* action. I'm going to need the entire `offer` entity in order to save `offer.accept()` and dispatch a `OfferAcceptedEvent`

Let's look at how we might structure a project around use cases.

Structuring projects around use cases

Uncle Bob identified a pattern called "Screaming Architecture". It means that by just looking at the project structure itself, it should be figuratively *screaming* at us: the **type** of project we're working on, in addition to the capabilities of the system.

Here's a little bit of what it looks like in White Label when we split it into `Subdomain` => `Use Cases` + `entities` .



At a glance, this tells us a *ton* about what the `users` subdomain is and what it does, in addition to what the `catalog` subdomain is and what it does.

A Use Case interface

Use Cases are simple in principle. They have an optional request and response.

```
export interface UseCase<IRequest, IResponse> {
  execute (request?: IRequest) : Promise<IResponse> | IResponse;
```

implementation", we can create an interface to represent a Use Case like this.

Simple enough, right?

Implementing a Use Case

Let's take a look at how we might implement this. Let's do the `AddVinylToCatalogUseCase` from the `Catalog` subdomain.

First, we'll create the class and implement the interface, using `any` for the Generic DTOs (data transmission objects).

```
export class AddVinylToCatalogUseCase implements UseCase<any, any> {
  public async execute (request: any): Promise<any> {
    return null;
  }
}
```

Alright, so in order to update a `Vinyl`, we need to provide everything necessary in order to **create** it, in addition to the `Trader` 's id that we're adding it to.

```
  artistNameOrId: string;
  traderId: string;
  genresArray?: string | string[];
}

export class AddVinylToCatalogUseCase
  implements UseCase<AddVinylToCatalogUseCaseRequestDTO, any> {

  async execute (request: AddVinylToCatalogUseCaseRequestDTO) : Promise<any> {
    return null;
  }
}
```

We're ready to actually implement the use case algorithm now.

Since our `Vinyl` aggregate root class needs an actual instance of an `Artist`, we'll have to determine whether to retrieve the artist by `id` or by `artistName`.

If the request fails, we'll use our result class to safely return an error, otherwise we'll use a `VinylRepo` to save the `Vinyl` to persistence.

```
interface AddVinylToCatalogUseCaseRequestDTO {
  vinylName: string;
  artistNameOrId: string;
  traderId: string;
  genresArray?: string | string[];
}

export class AddVinylToCatalogUseCase
  implements UseCase<AddVinylToCatalogUseCaseRequestDTO, Result<Vinyl>> {

  private vinylRepo: IVinylRepo;
  private artistRepo: IArtistRepo;

  constructor (vinylRepo: IVinylRepo, artistRepo: IArtistRepo) {
    this.vinylRepo = vinylRepo;
    this.artistRepo = artistRepo;
  }

  public async execute (request: AddVinylToCatalogUseCaseRequestDTO): Promise<Result<Viny
    return null;
  }
}
```

```
  artistNameOrId: string;
  traderId: string;
  genresArray?: string | string[];
}

export class AddVinylToCatalogUseCase
  implements UseCase<AddVinylToCatalogUseCaseRequestDTO, Result<Vinyl>> {

  private vinylRepo: IVinylRepo;
  private artistRepo: IArtistRepo;

  constructor (vinylRepo: IVinylRepo, artistRepo: IArtistRepo) {
    this.vinylRepo = vinylRepo;
    this.artistRepo = artistRepo;
  }

  public async execute (request: AddVinylToCatalogUseCaseRequestDTO): Promise<Result<Viny
    const { vinylName, artistNameOrId, traderId, genresArray } = request;
    let artist: Artist;

    const isArtistId = TextUtil.isUUID(artistNameOrId);

    if (isArtistId) {
      artist = await this.artistRepo.findById(artistNameOrId);
    } else {
        name: ArtistName.create(artistNameOrId).getValue(), genres: []
      }).getValue();
    }

    const vinylOrError = Vinyl.create({
      title: vinylName,
      artist: artist,
      traderId: TraderId.create(new UniqueEntityID(traderId)),
      genres: []
    });

    if (vinylOrError.isFailure) {
      return Result.fail<Vinyl>(vinylOrError.error)
    }

    const vinyl = vinylOrError.getValue()

    await this.vinylRepo.save(vinyl);
    return Result.ok<Vinyl>(vinyl)
  }
}
```

That's it! Now how do we hook this up to our application?

As long as we can provide the inputs, they can execute **commands** and **queries** on our system.

That means that they can be hooked up by Express.js `controllers` or any other external services from the **infrastructure layer**.

```
import { BaseController } from "../../../../../infra/http/BaseController";
import { AddVinylToCatalogUseCase } from "./CreateJobUseCase";
import { DecodedExpressRequest } from "../../../../../domain/types";
import { AddVinylToCatalogUseCaseRequestDTO } from "./AddVinylToCatalogUseCaseRequestDTO"

export class AddVinylToCatalogUseCaseController extends BaseController {
  private useCase: AddVinylToCatalogUseCase;

  public constructor (useCase: AddVinylToCatalogUseCase) {
    super();
    this.useCase = useCase;
  }

  public async executeImpl (): Promise<any> {
    const req = this.req as DecodedExpressRequest;
    const { traderId } = req.decoded;
    const requestDetails = req.body as AddVinylToCatalogUseCaseRequestDTO;
```

```
        return this.ok(this.res, resultOrError.getValue());
      } else {
        return this.fail(resultOrError.error);
      }
    }
  }
```

Use cases can **also** be executed by *other Use Cases* from within the **application layer** as well (but not from the Domain-layer as per Uncle Bob's [Dependency Rule](https://khalilstemmler.com)). And that's *really* cool.

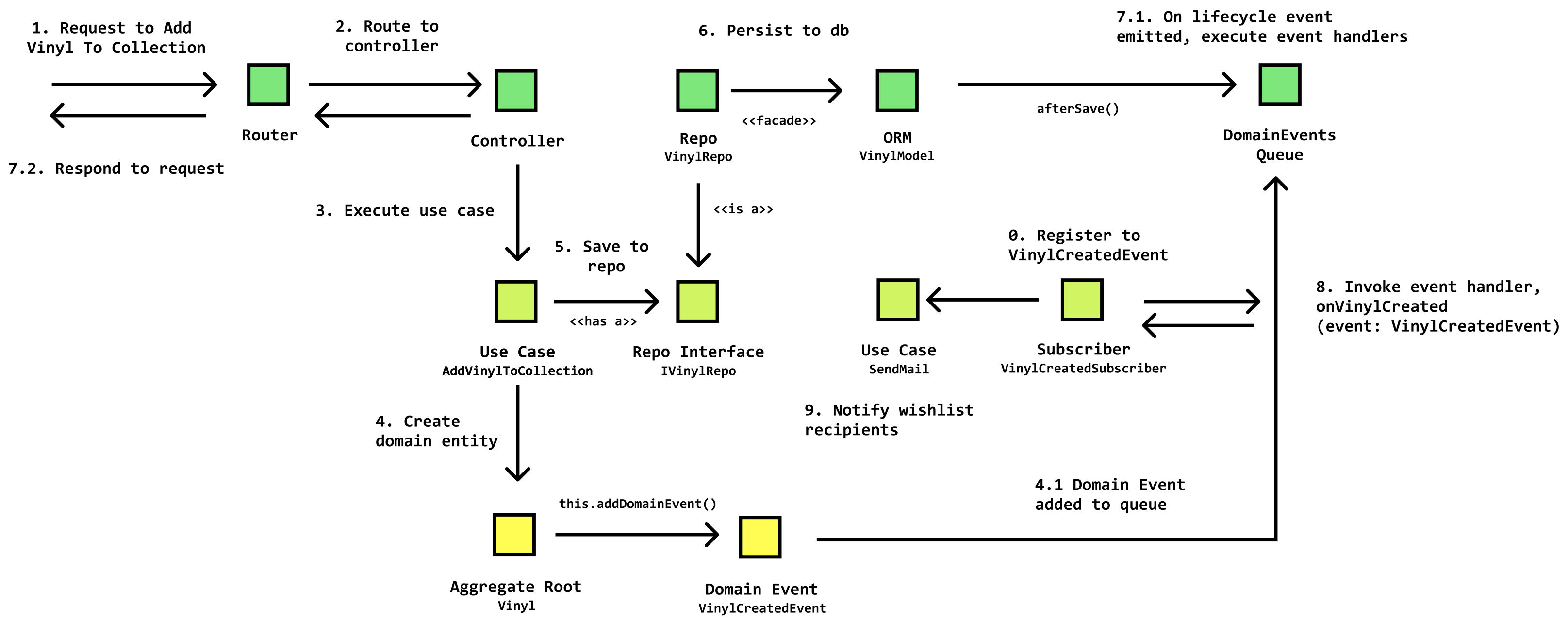
Elegant usage of Use Case with Domain Events

There's actually a really elegant way to **chain** Use Cases together.

You'd want to chain Use Cases together when one event might trigger another Use Case to be executed in certain scenarios. In Domain-Driven Design, we'd identify this behaviour through an Event storming exercise and use the **Observer pattern** to emit Domain Events.

are interested in that particular `vinyl` or `artist` will be notified that it was posted. That way, they can make an `offer` to the owner for the `vinyl` that they're interested in.

The following diagram is a simplification of the communication between layers and use cases.


1. Request to Add Vinyl To Collection
Router
2. Route to controller
Controller
7.2. Respond to request
3. Execute use case
6. Persist to db
Repo VinylRepo
<<facade>>
ORM VinylModel
<<is a>>
5. Save to repo
<<has a>>
Use Case AddVinylToCollection
Repo Interface IVinylRepo
4. Create domain entity
this.addDomainEvent()
Aggregate Root Vinyl
Domain Event VinylCreatedEvent
7.1. On lifecycle event emitted, execute event handlers
afterSave()
DomainEvents Queue
0. Register to VinylCreatedEvent
Use Case SendMail
Subscriber VinylCreatedSubscriber
8. Invoke event handler, onVinylCreated (event: VinylCreatedEvent)
9. Notify wishlist recipients
4.1 Domain Event added to queue

We'll follow up with the nitty-gritty on hooking up Domain Events to execute chained Use Cases in a de-coupled way using the observer pattern in a future article.

Codebase

All the code in this article is from White Label, a Vinyl-Trading enterprise app built with Node.js + TypeScript using Domain-Driven Design. You can check it out on GitHub:

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

Discussion

Liked this? Sing it loud and proud 👨‍🎤.

Share on Twitter

6 Comments

Comment

Submit

scns 7 months ago

```
in TypeScript you can shorten the constructor to this:

constructor (
  private vinylRepo: IVinylRepo,
  private artistRepo: IArtistRepo
) {}
```

ryan 7 months ago

u can, but this feature is not clear for reader, buddy, easy to write, hard to read

```
in TypeScript you can shorten the constructor to this:

constructor (
  private vinylRepo: IVinylRepo,
  private artistRepo: IArtistRepo
```

khalilstemmler.com

Stephen Horvath 5 months ago

In the source code you're storing both the repo implementation and interface in the "repo" folder. Would that be mixing infrastructure and domain objects? I would think that the implementation is infrastructure and the interface would actually be part of the domain. Then, the use case (application layer) could import the interface without violating the Clean Architecture Dependency Rule. And the implementation would be located safely outside of the domain and application circles.

Khalil Stemmler 5 months ago

You're 100% correct. I should update that.

A better source code example is DDDForum, which got a lot more love recently.

Pjotr 5 months ago

How do we know when to choose Domain Service and when to choose Use Case?
Couldn't we add AddVinylToCatalogUseCase to a method on VinylService e.g. VinylService.addVinylToCatalog

Khalil Stemmler 5 months ago

Hey Pjotr,

In Domain-Driven Design, every construct has a specific purpose.

For example, in DDDForum, we have a `PostService` domain service with an `upvoteComment` method. Check out the method signature- look at how many objects need to be involved in this in order to encapsulate the business rules.

```typescript
public upvoteComment (
post: Post,
member: Member,
comment: Comment, existingVotesOnCommentByMember: CommentVote[]
): UpvoteCommentResponse {
...
```

Use a domain service when you need to locate some business logic that doesn't belong to a particular domain entity.

The problem with domain services is that they often rely on domain objects that need to have been pulled from persistence already, but they're not allowed to reference infrastructure layer concerns like repositories...

That's where Application Services come in.

Use cases *are* application services.

Services and Entities.

Antonio 4 months ago

Hi Khalil, in this article you presented a pattern to create a separate class whenever a new command or query is identified.

Thinking about code reuse of similar/related commands I can imagine these use case classes being grouped in one class where each method is a use case/command. Do you see a bad side in this? Besides overdoing it, of course. :)

Another question, but this I think is for the announced Observer Pattern article, is where the domain logic of triggering commands whenever events occur would be placed? In other words, how do you implement the policies identified in an Event Storming model?

Thanks for the article and congrats for its quality!

Mike K. Shum a month ago

You're awesome.

Stay in touch!

khalilstemmler.com

Email

Get notified

About the author

Khalil Stemmler,

Developer Advocate @ Apollo GraphQL

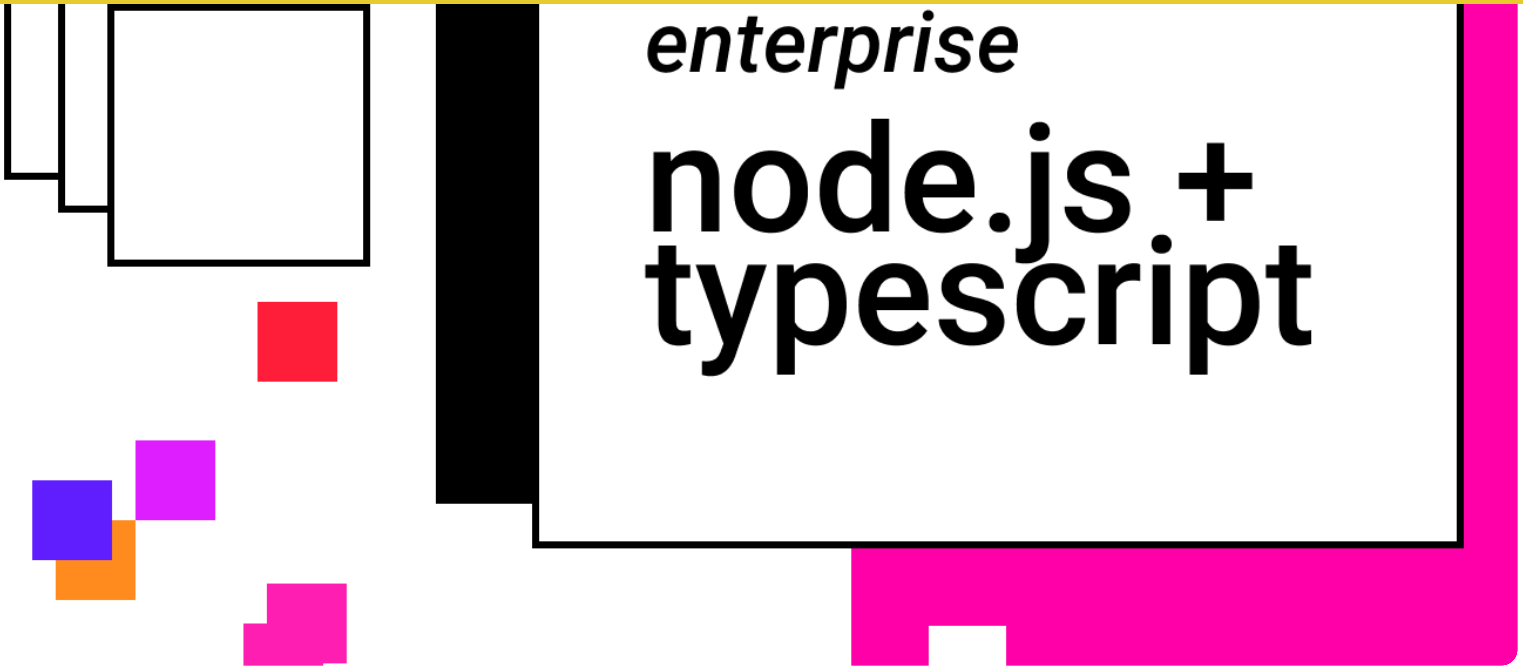
Khalil is a software developer, writer, and musician. He frequently publishes articles about Domain-Driven Design, software design and Advanced TypeScript & Node.js best practices for large-scale applications.

Follow @stemmlerjs 2,315 followers Follow 606

khalilstemmler.com

You may also enjoy...

A few more related articles



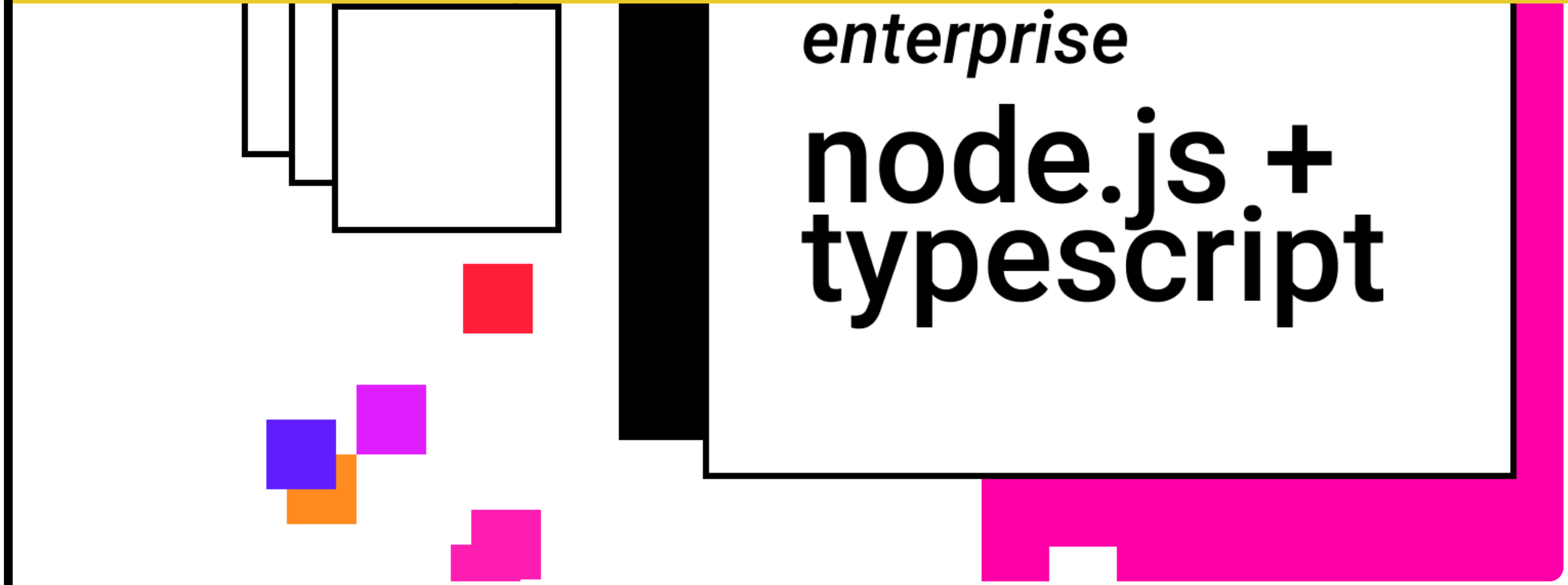
Functional Error Handling with Express.js and DDD | Enterprise Node.js + TypeScript

Enterprise Node + TypeScript

node.js typescript functional programming ddd express.js enterprise software

How to expressively represent (database, validation and unexpected) errors as domain concepts using functional programming concept...

khalilstemmler.com

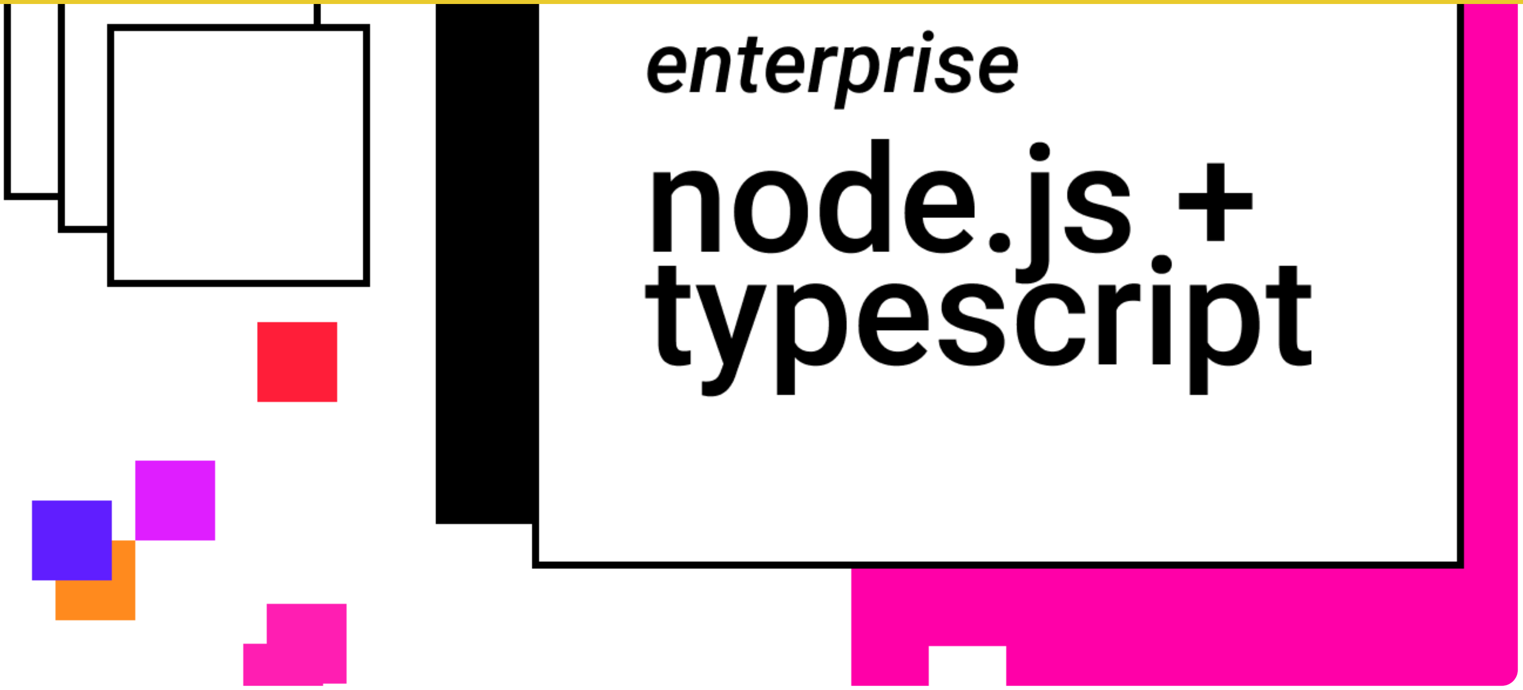


Knowing When CRUD & MVC Isn't Enough | Enterprise Node.js + TypeScript

Enterprise Node + TypeScript

node.js typescript architecture enterprise software

MVC the granddaddy of application architectures. In this article, we explore common MVC patterns, the responsibilities of the "M"-...



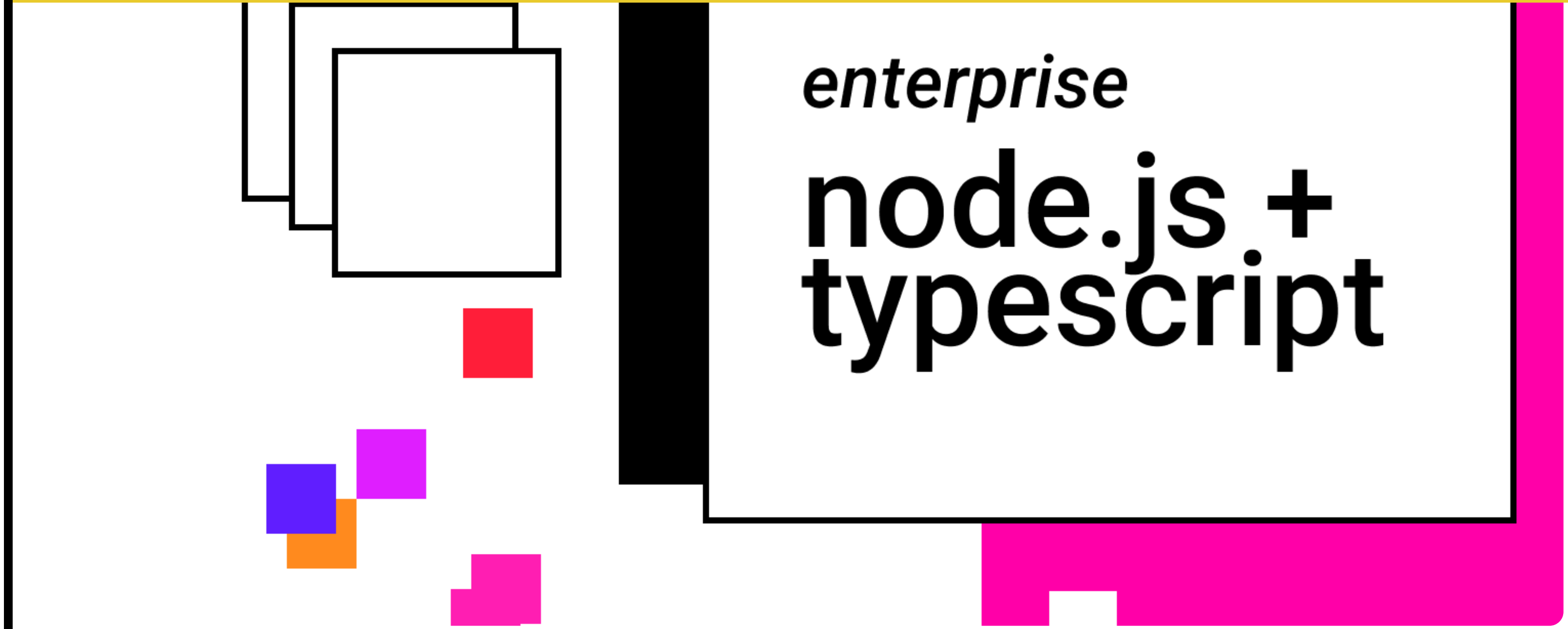
Flexible Error Handling w/ the Result Class | Enterprise Node.js + TypeScript

Enterprise Node + TypeScript

node.js typescript express.js enterprise software

Purposefully throwing errors can have several negative side effects to the readability and traceability of your code. In this arti...

khalilstemmler.com



Clean & Consistent Express.js Controllers | Enterprise Node.js + TypeScript

Enterprise Node + TypeScript

node.js typescript express.js enterprise software

In this article, we explore how to structure a clean and consistent Express.js controller by using abstraction and encapsulation w...

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

Email
@stemmlerjs

Social

khalilstemmler.com