khalilstemmler.com

REST-first design is Imperative, DDD is Declarative [Comparison] - DDD w/ TypeScript

Domain-Driven Design

A comparison between designing Node.js applications using REST-first design and Domain-Driven Design.

ddd typescript software design crud restful apis

Also from the Domain-Driven Design with TypeScript series.

When you get a new Node.js project, what do you start coding first?

Do you start with the **database schema?**

Do you start with the **RESTful API**?

Do you start with the **Models**?

REST-first Design is a term I've been using it to describe the difference between what Domain-Driven Design projects and REST-first CRUD projects look like on a code level.

Just for reminders, REST stands for "Representational State Transfer", which is a architectural style towards designing APIs on the web with HTTP.

In this article, I'm going to explain what a **REST-first Designed** codebase looks like, how it's **imperative** and how it differs from a **Domain-Driven Designed** project.

Imperative vs. Declarative

Do we remember what **imperative** code is?

Imperative

need to be very explicit for how the program's state gets changed.

Find the max number in an array [Imperative]

Here's an example of how we would determine the max number in an array of numbers, imperative style.

```
const numbers = [1,2,3,4,5];
let max = -1;
for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] > max) {
    max = numbers[i]
  }
}
```

Because imperative programming requires you to specify the exact commands to specify "how" program state changes, in this example, we:

- have a list of numbers
- create a for loop starting at 0
- increment i up to numbers.length
- and if number at the current index is greater than the max, then we’ll set that as the max

closer at that statement in a few moments.

Declarative

Declarative programming is more concerned about the *“what”*.

Because of this, declarative code is a bit more *"wordy"* and abstracts away a lot of the details for expressability.

Let's look at the same example.

> ### Find the max number in an array [Declarative]

```
const numbers = [1,2,3,4,5]
const max = numbers.reduce((a,b) => Math.max(a,b))
```

Take particular notice of the **language** being used in this example.

Ask yourself,

> "which of the two examples would non-programmers be quicker to understand?"

Notice that in this example, the **language** better describes the *“what”* than the imperative equivalent?

khalilstemmler.com

Declarative style code is one of the primary benefits of designing software using **Domain-Driven Design**.

Now that we're refreshed on **Imperative** and **Declarative** style coding, we'll dive deeper into my statements.

REST-first Design

When we build RESTful applications, we tend to think more about designing our applications from either:

- the database up and
- the API calls up

Because of this, there's a tendency to place the majority of our business logic in either controllers or **services.**

You might remember from Uncle Bob's "Clean Architecture", for controllers this is definitely no-no.

And if you read his book of the same name, you might recall the *potential* service-oriented fallacy of putting all the domain logic into services (hint: Anemic Domain Models).

But this is the type of code that gets written when:

- we want to respond to prototype apps
- we're working on small apps
- we're working on problems that are either #1 or #2 from the Hard Software Problems

And it does suffice for a large number projects!

However, for complex domains with complicated business rules and policies, this has the potential to become incredibly difficult to change and extend as time goes on.

In REST-first CRUD applications, we almost solely **write imperative code to satisfy business use cases**. Let's take a look at what that looks like.

REST-first code

Let’s say we were working on an application where `Customers` could rent `Movies` .

Designing REST-first using Express.js and the Sequelize ORM, my code might look like this:

```
class MovieController {
  public async rentMovie (movieId: string, customerId: string) {
    // Sequelize ORM models
    const { Movie, Customer, RentedMovie, CustomerCharge } = this.models;
```

khalilstemmler.com

```
      if (!!movie === false) {
        return this.notFound('Movie not found')
      }

      // 401 error if not found
      if (!!customer === false) {
        return this.notFound('Customer not found')
      }

      // Create a record which signified a movie was rented
      await RentedMovie.create({
        customer_id: customerId,
        movie_id: movieId
      });

      // Create a charge for this customer.
        await CustomerCharge.create({
        amount: movie.rentPrice
      })

      return this.ok();
    }
  }
```

In this code example, we pass in a `movieId` and a `customerId`, then pull out the appropriate Sequelize models that we know we're going to need to use. We do a quick null check and then if both model instances are returned, we'll create a `RentedMovie` and a `CustomerCharge`.

This is quick and dirty and it shows you just how quickly we can get things up and running REST-first.

Let's add some constraints to this. Consider that `Customer` isn't allowed to rent a movie if they:

> A) have rented the maximum amount of movies at one time (3, but this is configurable)

> B) have unpaid balances.

How exactly can we enforce this business logic?

A primitive approach would be to enforce it directly in our `MovieController`'s `purchaseMovie` method like so.

```
class MovieController extends BaseController {
  constructor (models) {
    super();
    this.models = models;
  }
  public async rentMovie () {
    const { req } = this;
    const { movieId } = req.params['movie'];
    const { customerId } = req.params['customer'];

    // We need to pull out one more model,
    // CustomerPayment
    const {
      Movie,
      Customer,
```

khalilstemmler.com

```
const movie = await Movie.findOne({ where: { movie_id: movieId }});
const customer = await Customer.findOne({ where: { customer_id: customerId }});

if (!!movie === false) {
  return this.notFound('Movie not found')
}

if (!!customer === false) {
  return this.notFound('Customer not found')
}

// Get the number of movies that this user has rented
const rentedMovies = await RentedMovie.findAll({ customer_id: customerId });
const numberRentedMovies = rentedMovies.length;

// Enforce the rule
if (numberRentedMovies >= 3) {
  return this.fail('Customer already has the maxiumum number of rented movies');
}

// Get all the charges and payments so that we can
// determine if the user still owes money
const charges = await CustomerCharge.findAll({ customer_id: customerId });
const payments = await CustomerPayment.findAll({ customer_id: customerId });

const chargeDollars = charges.reduce((previousCharge, nextCharge) => {
  return previousCharge.amount + nextCharge.amount;
});

const paymentDollars = payments.reduce((previousPayment, nextPayment) => {
  return previousPayment.amount + nextPayment.amount;
})

// Enforce the second business rule
if (chargeDollars > paymentDollars) {
      customer_id: customerId,
      movie_id: movieId
    });

    await CustomerCharge.create({
      amount: movie.rentPrice
    })

    return this.ok();
  }
}
```

There. It works. But there are **several drawbacks**.

Lack of encapsulation

Another developer could inadvertently circumvent our **domain logic** and business rules when developing a new feature that intersects with these rules, because it lives in a place where it shouldn't be.

We *could* easily move this domain logic to a service. That would be a small improvement, but really, it's just re-locating where the problem happens since other developers will still be able to write the code we just did, in a separate module, and **circumvent the business rules**.

_There are more reasons. If you'd like to know more about how services can get out of hand, read this.

When you look at a class and it's methods for the first time, it should accurately describe to you the capabilities and limitations of that class. When we co-locate the capabilities and rules of the `Customer` in an infrastructure concern (controllers), we lose some of that discoverability for what a `Customer` can do and *when it's allowed to do it*.

Lack of flexibility

Most of the time, we're concerned about making our application deliverable through HTTP.

For CRUD applications, yes- this is *usually* how we will be delivering it since the world lives on RESTful APIs.

However, if you want your application to be multi-platform, integrate with an **older system** or deliver your application as a **desktop app** as well, we'll need to ensure that none of the business logic lives in controllers, and instead resides within the Domain Layer.

We'll want to do that so that different infrastructure technologies can execute the *use cases* of the application.

Going back to how this code is **imperative**, it's **imperative** because we have to specify exactly *"how"* everything happens.

CRUD-first design is a "Transaction Script" approach

In the enterprise software world, Martin Fowler would call this a **Transaction Script** (article coming soon).

I first came to knowledge of the term after skimming through Fowler's Patterns of Enterprise Application Architecture.

I also came to realize that the Transaction Script approach was the single approach I used to writing all of my backend code.

> REST-first Design (more often than not) is a Transaction Script

How do we improve upon that? We use a **domain model**.

DDD

In Domain Modeling, one of the primary benefits is that we eventually hit an inflection point where the **declarative** language for specifying business rules becomes so expressive, that it takes us no time to add new capabilities and rules.

It also makes our business logic that much more readable, abstracting away **how** it gets done, and presenting more of **what** can get done and

khalilstemmler.com

the controller code would probably look more like this:

```
class MovieController extends BaseController {
  private movieRepo: IMovieRepo;
  private customerRepo: ICustomerRepo;

  constructor (movieRepo: IMovieRepo, customerRepo: ICustomerRepo) {
    super();
    this.movieRepo = movieRepo;
    this.customerRepo = customerRepo;
  }

  public async rentMovie () {
    const { req, movieRepo, customerRepo } = this;
    const { movieId } = req.params['movie'];
    const { customerId } = req.params['customer'];

    const movie: Movie = await movieRepo.findById(movieId);
    const customer: Customer = await customerRepo.findById(customerId);

    if (!!movie === false) {
      return this.fail('Movie not found')
    }

    if (!!customer === false) {
      return this.fail('Customer not found')
    }

    // The declarative magic happens here.
    const rentMovieResult: Result<Customer> = customer.rentMovie(movie);

    if (rentMovieResult.isFailure) {
      return this.fail(rentMovieResult.error)
    } else {
```

khalilstemmler.com

```
    }
  }
```

See that? Notice how much is abstracted away?

From our controller, we no longer have to worry about:

- if the `Customer` has more than the max number of rented movies
- if the `Customer` has paid their bills
- billing the `Customer` after they rent the movie.

In following articles on Domain **Entities** and **Aggregate Roots**, we'll go into more depth on how this works.

This is the **Declarative** essence of DDD. **How** it is done is abstracted, but the **ubiquitous language** being used effectively describes what the domain objects are allowed to do and when.

Additional reading

3 Categories of Hard Software Problems

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

khalilstemmler.com

Liked this? Sing it loud and proud 👨‍🎤.

Share on Twitter

2 Comments

Name

B I U

Comment

Submit

Rahul 8 months ago
Great piece of content again! I've implemented a project in Nest.js that is going to grow in complexity a lot. I would love to take some learnings from the DDD approach and use them to refactor my backend. Can you be more specific on what exactly about Nest forces you into an anemic model, and what to focus on to get out of it?

Khalil Stemmler 8 months ago
Thanks!

Yeah, for sure :) great question btw. Love that.

OK, so an Anemic Domain Model is one where **all of the domain logic ends up in the services** and **none of it ends up in the entities and value objects.**

validation rules?

To my knowledge, Nest.js requires you to define each of the attributes of an entity as a primitive.

So if I have a `User` class, the `email` field has to be `string`. This isn't great for encapsulating validation logic. I briefly answered your comment on this very problem in the value object article. We want to be able to have:

1) Good encapsulation of validation logic (so that means that instead of `userEmail: string`, we create a *wrapper type* to deal with `userEmail: UserEmail` instead, making illegal states unrepresentable.

2) Nominal typing of business objects (so even if we create a `UserEmail` to wrap a string, and we create another class called `UserPhoneNumber` which also wraps a string, we can only pass in `UserEmail` to `User.create(userEmail: UserEmail): User` and not `UserPhoneNumber` because the *names of the types don't match, even though they wrap the same primitive*.

3) Exposing only valid operations to the domain (if `User` has has a public *setter* for `userId`, does it make sense to the domain for me to be able to alter it from outside the class? For example, should you **ever be able to change a `User` model's `userId` field**? Probably not, right? Doing so would be invalid to the domain, and we'd corrupt that user. To my knowledge, Nest.js needs us to have all fields public so that it can create the columns at the persistence layer. Also, this tight coupling of the domain layer to the persistence layer breaks the dependency rule).

Ayyappa 5 months ago
Thanks for detailed explanations! It looks great!

I have a question which I can't resist to ask.

instead of a service and had all checks inside the controller.

Ideally both seem same. Can you please confirm?

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.

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 3,732 followers

khalilstemmler.com

View more in [Domain-Driven Design]

Learn to write testable, flexible and maintainable code

Khalil Stemmler

SOLID

SOLID

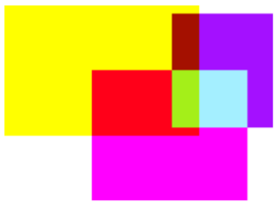
The Software Design
& Architecture Handbook

Get the book

You may also enjoy...

A few more related articles

khalilstemmler.com



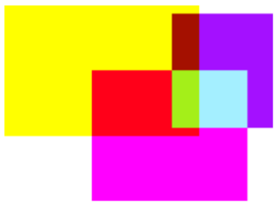
How to Handle Updates on Aggregates - Domain-Driven Design w/ TypeScript

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

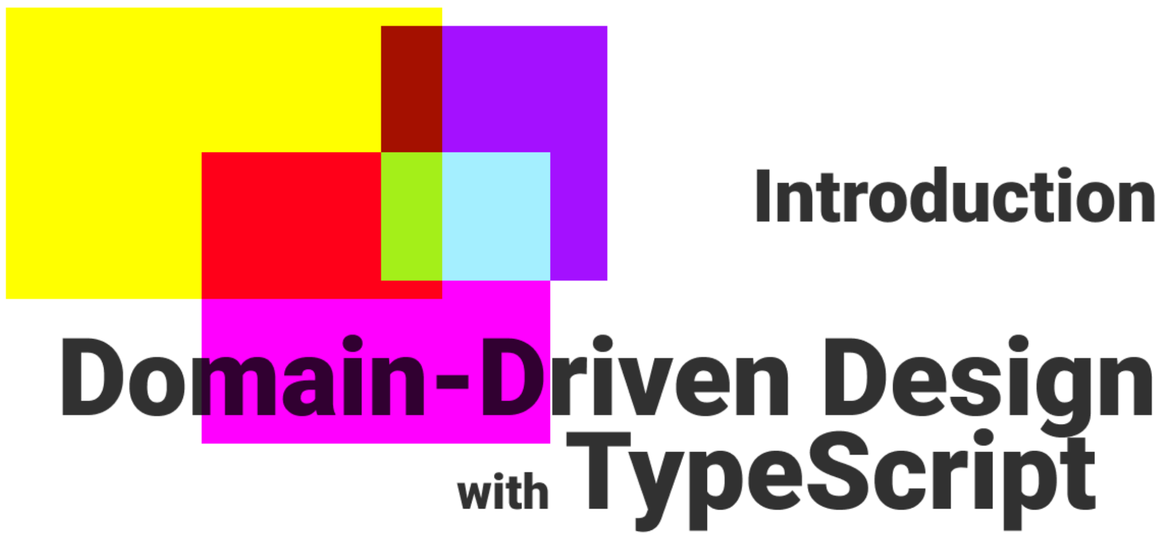
ddd typescript software design domain events sequelize typeorm

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

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

khalilstemmler.com

applications? How far does domain modeling reach from ...



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

Domain-Driven 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
- Courses
- Books
- Newsletter
- Portfolio
- Wiki

khalilstemmler.com

Social

GitHub
Twitter
Instagram
LinkedIn

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