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

Domain-Driven Design

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

ddd typescript software design aggregate root aggregate sequelize

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.*

# Introduction

> where a single field or only a few fields are present for an update against an aggregate".

Ah yes. Updates.

It's inevitable you'll want to perform update commands in a CRUD + MVC or DDD-based application. Everyone has their own way to handle updates in basic MVC applications, but it's not abundantly clear how to design updates against aggregates using DDD.

In the code sample, they had a `User` aggregate, which looked more or less like this:

> **Note:** I've left comments for improvements that I recommend based on patterns we've explored on this site in previous articles.

domain/user.ts

```
// I would recommend extracting these to props like in the
// "Understanding Domain Entities" guide.

// The way it is currently, getters and setters are exposed
// for everything, and that's not very "Intention Revealing".
// Also makes it hard to restrict invalid operations (like
// manually changing the id) and makes it hard to enforce
// model invariants like ensuring the Phone is a valid
// value object instance (can currently assign to null).
// Let's take full advantage of the language capabilities.

id: number;
phone: Phone;
email: Email;
address: Address;

private constructor (id, phone, email, address){
  // setting values here, I think
}

// Very good, there's a public factory method. Should type these
// props as their own interface UserProps {} though.
public static create (props) {

  // There should be a Guard class here to ensure all props are valid.
```

```
    }
}
```

There's definitely ways we could improve `User` aggregate, but the real pain points are felt from within the `UsersService`'s `updatePhone` method (which is something that I would normally advocate for representing as an application layer use case) instead.

usersService.ts

```
export class UserService {
  ...
  public updatePhone (updatePhoneDto) {

    const userEntity = userRepository.getUser(updatePhNoDto.userId);

    const userModel = User.update({
      id: updatePhNoDto.userId,
      phone: Phone.create(userEntity.phone),
      email: Email.create(userEntity.email),
      address: Address.create(userEntity.address)
    });
```

The primary drawback to address in this code is the fact that the `updatePhoneDto` probably has a shape like:

```
interface UpdatePhoneDto {
  userId: number;
  phone?: string;
  email?: string;
  address?: string;
}
```

And with all of those optional fields, we need to be able to "deal with situations where a single field or only a few fields are present for an update".

If any of either `phone`, `email`, or `address` aren't present, we'll break each value object's `create()` factory method.

That's not good.

## What about non 1-to-1 relationships?

For example, in White Label, an `Album` can have many different `Genre` s assigned to it.

We need some way that we can keep track or *mark* which `Genres` were updated or removed in an update so that we can perform the correct persistence commands (insert? delete? update?).

It can get pretty complex. And that's a necessary complexity when we use domain models to encapsulate business rules and language within code.

The flow is straightforward though.

## Plan of action for performing updates

Since in DDD, we usually implement the Data Mapper pattern, the object we retrieve from persistence before we update it will be a *plain 'ol TypeScript object*.

Our plan for performing an update against and aggregate will look like this:

- 3. Pass it off to a repo to `save()` (or perhaps `delete()` ).

The ***challenges* at step 2** are:

- Protecting model integrity (class invariants)
- Performing validation logic
- Representing errors as domain concepts
- Keeping "update code" DRY
- Choosing the best ways to represent updates

The **challenges at step 3** are:

- Performing an atomic update as a single transaction

Lets start with the basics.

In this article, we'll cover:

- How to handle updates (update, insert, delete) within aggregates with 1-to-1 relationships
- How mutations to your domain model can contain class invariants that need to be represented as domain concepts

We'll go into more advanced stuff like 1-to-many relationships in another article.

## Prerequisites

In order to get the most out of this article, there are a few things that you might want to read first.

- How to use DTOs, Repositories, and Mappers to persist and transform domain objects to other representations.
- The Command-Query Segregation Principle.
- How every feature of an application is a use case, which is either a command or a query.

## A basic example (1-to-1)

Let's take the example we looked at before: the `User` aggregate and it's relationship to `phone` , `email` and `address` .

### Creating the domain model

I'm going to model the `User` aggregate a little differently than the example provided based on things we've covered in the Domain-Driven Design w/ TypeScript series already.

domain/user.ts

```
import { UserId } from "./userId";
import { Email } from "./email";
import { Phone } from "./phone";
import { Address } from "./address";
import { Guard } from "../../../core/logic/Guard";

interface UserProps {
  phone: Phone;
  email: Email;
  address: Address;
}

export class User extends AggregateRoot<UserProps> {

  // Only getters so far, no way to perform changes to
  // the model after it's been created.

  get userId (): UserId {
    return UserId.create(this._id)
  }

  get phone (): Phone {
    return this.props.phone;
  }

  get email (): Email {
  }

  private constructor (props: UserProps, id?: UniqueEntityID) {
    super(props, id);
  }

  // The only way to create or reconstitute a User is to use the static factory
  // method here.

  public static create (props: UserProps, id?: UniqueEntityID): Result<User> {

    const guardResult = Guard.againstNullOrUndefinedBulk([
      { argument: props.phone, argumentName: 'phone' },
      { argument: props.email, argumentName: 'email' },
      { argument: props.address, argumentName: 'address' }
    ]);

    if (!guardResult.success) {
      return Result.fail<User>(guardResult.message);
    }

    const isNewUser = !!id === false;

    const user = new User(props, id);
```

```
    return Result.ok<User>(user);
  }

}
```

Cool, now let's talk about the **use case**.

## Use Case setup

We want to provide a way to update the `User` aggregate, so let's start by creating a new `UpdateUser` use case in our use cases folder for the `User` subdomain.

```
modules/
  users/                # `users` subdomain
    domain/
      ...
      user.ts
    ...
    useCases/
      updateUser/
        UpdateUser.ts   # Use case!
```

```
export class UpdateUser implements UseCase<any, Promise<any>> {

  constructor () {
    // import dependencies
  }

  public async execute (): Promise<any> {
    // execute application layer logic
  }

}
```

If you've read the Clean Architecture vs. Domain-Driven Design concepts article, you'll remember that the responsibility of use cases at this layer are to simply *fetch* the domain objects we'll need to complete this operation, allow them to interact with each other (at the domain layer), and then save the transaction (by passing the affected aggregate root to it's repository).

That's just what we'll do.

```
  UpdateUser.ts     # Use Case
  UpdateUserDTO.ts  # DTO
```

useCases/updateUser/UpdateUserDTO.ts

```typescript
export interface UpdateUserDto {
  userId: number;
  phone?: string;
  email?: string;
  address?: string;
}
```

And we'll update our use case to use that as the input.

useCases/updateUser/UpdateUser.ts

```typescript
import { UpdateUserDTO } from './UpdateUserDTO'

export class UpdateUser implements UseCase<UpdateUserDTO, Promise<any>> {
  constructor () {
    // import dependencies
}
```

Cool. Now let's dependency inject a `UserRepo` so that we can get access to the `user` aggregate that we want to change in this transaction, then let's get it.

useCases/updateUser/UpdateUser.ts

```
import { IUserRepo } from '../repos/interfaces/userRepo'
import { UpdateUserDTO } from './UpdateUserDTO'

export class UpdateUser implements UseCase<UpdateUserDTO, Promise<any>> {
  private userRepo: IUserRepo;

  constructor (userRepo: IUserRepo) {
    this.userRepo = userRepo;
  }

  public async execute (request: UpdateUserDTO): Promise<any> {
    let user: User;

    try {
```

```
    ...
    // Continue

  }
}
```

I'm going to hook up some expressive error handling because I don't want a "*Not Found*" error to go unswallowed by the consumer of this use case.

> **Note:** This is how we can strictly type and express any other application-level (use case) or domain layer errors that might get returned. The primary benefit of not throwing errors but instead representing them as first-class citizens of our app is that we *force the client* using our use case to handle the error states (that we *know for sure* will occur at some point).

Two new files.

```
    UpdateUserErrors.ts   # Holds all the unique types of application errors
    UpdateUserResult.ts   # Express the result as a functional error type
```

Lets create the namespace to represent errors for this use case.

updateUser/UpdateUserErrors.ts

```
import { UseCaseError } from "../../../../../shared/core/UseCaseError";
import { Result } from "../../../../../shared/core/Result";

export namespace UpdateUserErrors {

  export class UserNotFoundError extends Result<UseCaseError> {
    constructor (userId: string) {
      super(false, {
        message: `Couldn't find user id {${userId}} to update.`
      } as UseCaseError)
    }
  }

}
```

```
import { Either, Result } from "../../../../../shared/core/Result";
import { UpvotePostErrors } from "./UpvotePostErrors";
import { AppError } from "../../../../../shared/core/AppError";

export type UpdateUserResult = Either<
  UpdateUserErrors.UserNotFoundError |  // Specific use case error
  AppError.UnexpectedError |            // Global app error
  Result<any>,                          // Misc errors (value objects)
  Result<void>                          // Success!
>
```

**Error handling articles:** No idea what I'm doing? First read "Flexible Error Handling w/ the Result Class" and then read "Functional Error Handling with Express.js and DDD".

Then let's update the use case with the return type, represent the not found error, and represent the `void` success state

useCases/updateUser/UpdateUser.ts

```
export class UpdateUser implements UseCase<UpdateUserDTO, Promise<UpdateUserResult>> {
  private userRepo: IUserRepo;

  constructor (userRepo: IUserRepo) {
    this.userRepo = userRepo;
  }

  public async execute (request: UpdateUserDTO): Promise<UpdateUserResult> {
    let user: User;

    try {
      user = await this.userRepo.findUserById(request.userId);
    } catch (err) {
      return left(new UpdateUserErrors.UserNotFoundError(request.userId))
    }

    // Update logic goes here

    return right(Result.ok<void>())
  }
}
```

Fantastic. We're all set up with to write some update logic now.

the Repository Pattern using the Sequelize ORM [with Examples] - DDD w/ TypeScript".

## Handling update logic

In this particular scenario, we have a DTO with several keys that we'd like to update: `phone`, `email`, `address`.

Depending on the domain, each of these *could possibly* be modelled as simple value objects because they're more complex than simple `string` types and have validation rules to dictate what a valid instance of them looks like.

We can start by using each value object's factory method (which encapsulates its respective validation logic).

useCases/updateUser/UpdateUser.ts

```
import { IUserRepo } from '../repos/interfaces/userRepo'
import { UpdateUserDTO } from './UpdateUserDTO'
import { UpdateUserResult } from './UpdateUserResult'
```

```
export class UpdateUser implements UseCase<UpdateUserDTO, Promise<UpdateUserResult>> {
  private userRepo: IUserRepo;

  constructor (userRepo: IUserRepo) {
    this.userRepo = userRepo;
  }

  public async execute (request: UpdateUserDTO): Promise<UpdateUserResult> {
    let user: User;

    try {
      user = await this.userRepo.findUserById(request.userId);
    } catch (err) {
      return left(new UpdateUserErrors.UserNotFoundError(request.userId))
    }

    // Create value object instances
     const phoneOrError: Result<Phone> = Phone.create(request.phone);
     const emailOrError: Result<Email> = Email.create(request.email);
     const addressOrError: Result<Address> = Address.create(request.address);
    ...

    return right(Result.ok<void>())
  }
}
```

and `address` in a single request.

We can write our own null checks or use a utility like lodash's `has` function. It will return `true` if a property exists on an object.

useCases/updateUser/UpdateUser.ts

```
import { IUserRepo } from '../repos/interfaces/userRepo'
import { UpdateUserDTO } from './UpdateUserDTO'
import { UpdateUserResult } from './UpdateUserResult'
import { UpdateUserErrors } from './UpdateUserErrors'
import { Phone } from '../domain/phone'
import { Email } from '../domain/email'
import { Address } from '../domain/addres'
import { has } from 'lodash'

export class UpdateUser implements UseCase<UpdateUserDTO, Promise<UpdateUserResult>> {
  private userRepo: IUserRepo;

  constructor (userRepo: IUserRepo) {
    this.userRepo = userRepo;
  }

  public async execute (request: UpdateUserDTO): Promise<UpdateUserResult> {
      return left(new UpdateUserErrors.UserNotFoundError(request.userId))
    }

     if (has(request, 'phone')) {
       const phoneOrError: Result<Phone> = Phone.create(request.phone);
       // update
     }

     if (has(request, 'email')) {
       const emailOrError: Result<Email> = Email.create(request.email);
       // update
     }

     if (has(request, 'address')) {
       const addressOrError: Result<Address> = Address.create(request.address);
       // update
     }

    ...

    return right(Result.ok<void>())
  }
}
```

successfully create the value object, we can go ahead and update `user` with it.

And if we weren't able to create it, yet we included *some value* for that key in the request, that probably means that there was a validation error that didn't pass and we should let the client know about that (matches the `Result<any>` error type).

useCases/updateUser/UpdateUser.ts

```
...
export class UpdateUser implements UseCase<UpdateUserDTO, Promise<UpdateUserResult>> {
  ...
  public async execute (request: UpdateUserDTO): Promise<UpdateUserResult> {
    ...

    if (has(request, 'phone')) {
      const phoneOrError: Result<Phone> = Phone.create(request.phone);

      if (phoneOrError.isSuccess) {
        user.updatePhone(phoneOrError.getValue()) // This will be of Phone type.
      } else {
        return left(phoneOrError) // This will be a Result<any> return type.
      }
    }
}
```

We have a new method on the `user` aggregate called `updatePhone(phone: Phone)`.

domain/user.ts

```
...

export class User extends AggregateRoot<UserProps> {

  ...

  get phone (): Phone {
    return this.props.phone;
  }

  public updatePhone (phone: Phone): void {
    this.props.phone = phone;
  }

  ...

}
```

## Atomic Transactions

Lets say we wanted to update `phone` AND `address` in single transaction.

If `phone` was valid but `address` was **not**, should we let the transaction partially update the `user` aggregate or should the entire transaction fail?

It should fail, correct?

Things have the potential to get messy, code can get hard to debug, and things can be challenging to reason about if we were to allow partial transactions. A request to `UPDATE` the `user` has to be fully correct for it to succeed.

How do we do this?

Using our trusty `Result<T>` class and its `combine(results: Result<T>[])` method of course!~

```
...

export class UpdateUser implements UseCase<UpdateUserDTO, Promise<UpdateUserResult>> {
  private userRepo: IUserRepo;
  private changes: Result<T>[];

  constructor (userRepo: IUserRepo) {
    this.userRepo = userRepo;
    this.changes = [];
  }

  public addChange (result: Result<any>) : void {
    this.changes.push(result);
  }

  public getCombinedChangesResult (): Result<any> {
    return Result.combine(this.changes);
  }

  ...
}
```

design decision.

domain/user.ts

```
...

export class User extends AggregateRoot<UserProps> {

  ...

  get phone (): Phone {
    return this.props.phone;
  }

   public updatePhone (phone: Phone): Result<void> {
     this.props.phone = phone;
     return Result.ok<void>();
   }

  ...

}
```

sense upfront.

**Note:** After we finish this up, I'll show you an example of a real life aggregate that enforces class invariants that dictate *when* and *how* it's allowed to change. It makes much better use of the Result type.

Using this pattern, we can add each mutation against the `user` aggregate to the changes array.

useCases/updateUser/UpdateUser.ts

```
import { IUserRepo } from '../repos/interfaces/userRepo'
import { UpdateUserDTO } from './UpdateUserDTO'
import { UpdateUserResult } from './UpdateUserResult'
import { UpdateUserErrors } from './UpdateUserErrors'
import { Phone } from '../domain/phone'
import { Email } from '../domain/email'
import { Address } from '../domain/addres'
import { has } from 'lodash'

export class UpdateUser implements UseCase<UpdateUserDTO, Promise<UpdateUserResult>> {
  private userRepo: IUserRepo;
```

```
}

public addChange (result: Result<any>) : void {
  this.changes.push(result);
}

public getCombinedChangesResult (): Result<any> {
  return Result.combine(this.changes);
}

public async execute (request: UpdateUserDTO): Promise<UpdateUserResult> {
  let user: User;

  try {
    user = await this.userRepo.findUserById(request.userId);
  } catch (err) {
    return left(new UpdateUserErrors.UserNotFoundError(request.userId))
  }

  if (has(request, 'phone')) {
    const phoneOrError: Result<Phone> = Phone.create(request.phone);

    if (phoneOrError.isSuccess) {
      this.addChange(
        user.updatePhone(phoneOrError.getValue())
  if (has(request, 'email')) {
    const emailOrError: Result<Email> = Email.create(request.email);

    if (emailOrError.isSuccess) {
       this.addChange(
         user.updateEmail(emailOrError.getValue())
       )
    } else {
      return left(emailOrError)
    }
  }

  if (has(request, 'address')) {
    const addressOrError: Result<Address> = Address.create(request.address);

    if (addressOrError.isSuccess) {
       this.addChange(
         user.updateEmail(addressOrError.getValue())
       )
    } else {
      return left(addressOrError)
    }
  }
```

And then at the end, if all of the changes were successful, we can pass it to the repository to be saved.

useCases/updateUser/UpdateUser.ts

```
...

export class UpdateUser implements UseCase<UpdateUserDTO, Promise<UpdateUserResult>> {
  private userRepo: IUserRepo;
  private changes: Result<T>[];

  constructor (userRepo: IUserRepo) {
    this.userRepo = userRepo;
    this.changes = [];
  }

  public addChange (result: Result<any>) : void {
    this.changes.push(result);
  }

  public getCombinedChangesResult (): Result<any> {
    return Result.combine(this.changes);
    if (this.getCombinedChangesResult().isSuccess) {
      try {
        // Save!
        await this.userRepo.save(user);
      } catch (err) {
        return left (AppError.create(err))
      }
    }

    return right(Result.ok<void>())
  }
}
```

And the `UserRepo` knows whether to perform an `update` or a `create` because it will first check to see if the entity exists or not.

repos/implementations/sequelizeUserRepo.ts

```
export class SequelizeUserRepo implements UserRepo {

  ...
```

```
    if (!exists) {
      const rawSequelizeUser = await UserMap.toPersistence(user);
      await UserModel.create(rawSequelizeUser);
    } else {
      // Update!
    }

    return;
  }
}
```

For more on using Repositories to persist domain entities, read "Implementing DTOs, Mappers & the Repository Pattern using the Sequelize ORM [with Examples]".

## Extracting "change" functionality to a interface

Pretty much all `update` use cases need to use this kind of change functionality so I'd recommend extracting it into it's own separate class and then using composition to add it to use cases that need it.

```
import { Result } from "./Result";

// Use cases that need changes can implement this
export interface WithChanges {
  changes: Changes;
}

// Extracted into its own class
export class Changes {
  private changes: Result<any>[];

  constructor () {
    this.changes = [];
  }

  public addChange (result: Result<any>) : void {
    this.changes.push(result);
  }

  public getCombinedChangesResult (): Result<any> {
    return Result.combine(this.changes);
  }
}
```

```
import { IUserRepo } from '../repos/interfaces/userRepo'
import { UpdateUserDTO } from './UpdateUserDTO'
import { UpdateUserResult } from './UpdateUserResult'
import { UpdateUserErrors } from './UpdateUserErrors'
import { Phone } from '../domain/phone'
import { Email } from '../domain/email'
import { Address } from '../domain/addres'
import { has } from 'lodash'

export class UpdateUser implements UseCase<UpdateUserDTO, Promise<UpdateUserResult>>, Wi
  private userRepo: IUserRepo;
  public changes: Changes;

  constructor (userRepo: IUserRepo) {
    this.userRepo = userRepo;
    this.changes = new Changes();
  }

  public async execute (request: UpdateUserDTO): Promise<UpdateUserResult> {
    let user: User;

    try {
      user = await this.userRepo.findUserById(request.userId);
    } catch (err) {
      return left(new UpdateUserErrors.UserNotFoundError(request.userId))
    }
```

```
        this.changes.addChange(
          user.updatePhone(phoneOrError.getValue())
        )
      } else {
        return left(phoneOrError)
      }
    }

    ...

    if (this.changes.getCombinedChangesResult().isSuccess) {

      try {
        // Save!
        await this.userRepo.save(user);
      } catch (err) {
        return left (AppError.create(err))
      }
    }

    return right(Result.ok<void>())
  }
}
```

update use case.

Here's one from DDDForum.com, the Hackernews-inspired forum app built with TypeScript & DDD from solidbook.io.

A `Post` can have either a `link` or `text`, but not both.

And if you wish to change your `link` or `text`, you can only do so if the `Post` **doesn't yet have any comments.**

See that? In this scenario, there are business rules that dictate when it's OK for something to change. That's the power of DDD and model-driven design. The shift that should happen in your thinking is to prefer trying to represent those invalid error states as domain concepts, rather than than throwing untyped errors.

Here's a snippet from the `Post` aggregate.

domain/post.ts

```
export type UpdatePostTextOrLinkResult = Either<
  /**
   * This error means that we're trying to update a text post when the
   * post is actually a link post, or vice versa.
   */
  EditPostErrors.InvalidPostTypeOperationError |
  /**
   * This error means that the post is sealed due to there already being
   * comments
   */
  EditPostErrors.PostSealedError |
  Result<any>,
  Result<void>
>

class Post extends AggregateRoot<PostProps> {

  ...

  public updateText (postText: PostText): UpdatePostTextOrLinkResult {
    if (!this.isTextPost()) {
      return left(new EditPostErrors.InvalidPostTypeOperationError())
    }

    if (this.hasComments()) {
      return left(new EditPostErrors.PostSealedError())
      return left(Result.fail<any>(guardResult.message))
    }

    this.props.text = postText;
    return right(Result.ok<void>());
  }

  public updateLink (postLink: PostLink): UpdatePostTextOrLinkResult {
    if (!this.isLinkPost()) {
      return left(new EditPostErrors.InvalidPostTypeOperationError())
    }

    if (this.hasComments()) {
      return left(new EditPostErrors.PostSealedError())
    }

    const guardResult = Guard.againstNullOrUndefined(postLink, 'postLink');

    if (!guardResult.succeeded) {
      return left(Result.fail<any>(guardResult.message))
    }

    this.props.link = postLink;
    return right(Result.ok<void>());
```

khalilstemmler.com

## Discussion

Liked this? Sing it loud and proud 👨‍🎤.

Share on Twitter

## 2 Comments

Name

**B** *I* U 🔗 </>

Comment

Submit

**Praveen** 5 months ago

```
    user = await this.userRepo.findUserById(request.userId);
```

Here, user is of type User (Aggregate Root). But, if I use a database to persist, the object returned by the repo, say TypeOrm, will be of type Entity. So, how do I map Entity to the Aggregate root ?

> **Khalil Stemmler** 5 months ago
> Hey Praveen.
>
> In order to map an ORM object to a domain entity (like an Aggregate Root), try using the Mapper pattern from this article, "Implementing DTOs, Mappers & the Repository Pattern using the Sequelize ORM [with Examples] - DDD w/ TypeScript".

**Vicky** 5 months ago
Thanks for the post.

When we fetch User aggregate from the repo, how do you convert it to a domain entity again in the mapper.
Do you use the static method User.create() or new User().
If you use static method User.create() then you need all the props and the create validation runs again.

> **Khalil Stemmler** 5 months ago

## 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

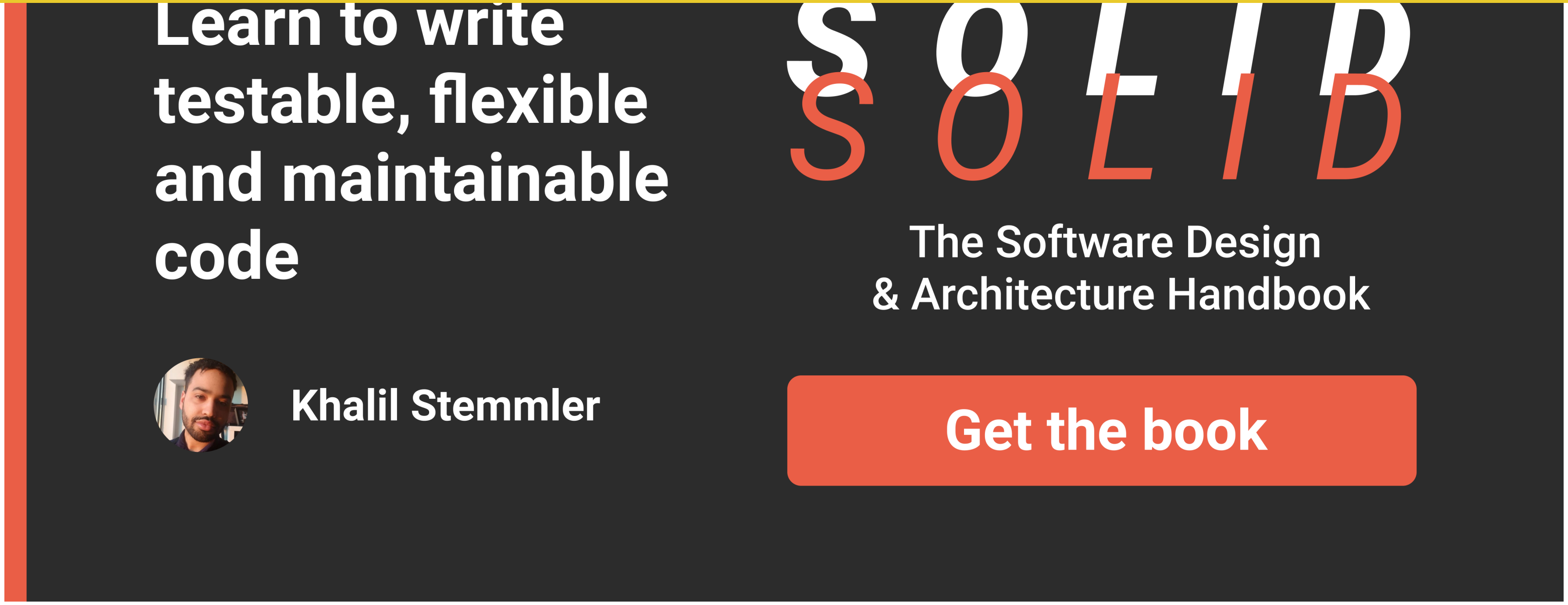
Get notified

## About the author

Khalil Stemmler,

Developer Advocate @ Apollo GraphQL ⚡

View more in Domain-Driven Design



## You may also enjoy...

A few more related articles

khalilstemmler.com

## How to Design & Persist Aggregates - Domain-Driven Design w/ TypeScript

Domain-Driven Design

ddd typescript software design aggregate root aggregate sequelize

In this article, you'll learn how identify the aggregate root and encapsulate a boundary around related entities. You'll also lear...

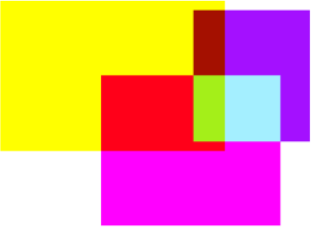
## Decoupling Logic with Domain Events [Guide] - Domain-Driven Design w/ TypeScript

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



## Handling Collections in Aggregates (0-to-Many, Many-to-Many) - Domain-Driven Design w/ TypeScript

Domain-Driven Design

ddd typescript software design aggregate one-to-many many-to-many

In this article, we discuss how we can use a few CQS principles to handle unbounded 0-to-many or many-to-many collections in aggre...

khalilstemmler.com

## Challenges in Aggregate Design #1 - Domain-Driven Design w/ TypeScript

Domain-Driven Design

ddd typescript software design aggregate

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