Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

Introduction to DDD Lite: When microservices in Go are not enough

JUL 1, 2020 · 20 MIN READ

Robert Laszczak
Tech Lead at **Karhoo**. Co-founder of **Three Dots Labs**. Creator of **Watermill**.

When I started working in Go, the community was not looking positively on techniques like DDD (Domain-Driven Design) and Clean Architecture. I heard multiple times: *"Don't do Java in Golang!*", *"I've seen that in Java, please don't!*".

These times, I already had almost 10 years of experience in PHP and Python. I've seen too many bad things already there. I remember all these "Eight-thousanders" (methods with +8k lines of code 😉) and applications that nobody wanted to maintain. I was checking old git history of these ugly monsters, and they were looking harmlessly at the beginning. But with time, small, innocent problems started to become more significant and more serious. **I've also seen how DDD and Clean Architecture solved these issues.**

Maybe Golang is different? Maybe writing microservices in Golang will fix this issue?

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

It was supposed to be so beautiful

Now, after exchanging experience with multiple people and having the ability to see a lot of codebases, my point of view is a bit cleaner than 3 years ago. Unfortunately, I'm now far from thinking that just using Golang and microservices will save us from all of these issues that I've encountered earlier. I started to actually have flashbacks from the old, bad times.

It's less visible because of the relatively younger codebase. It's less visible because of the Golang design. But I'm sure that with time, we will have more and more legacy Golang applications that nobody wants to maintain.

Fortunately, 3 years ago, despite to chilly reception I didn't give up. I decided to try to use DDD and related techniques that worked for me previously in Go. With Milosz we were leading teams for 3 years that were all successfully using DDD, Clean Architecture, and all related, not-popular-enough techniques in Golang. **They gave us the ability to develop our applications and products with constant velocity, regardless of the age of the code.**

It was obvious from the beginning, that **moving patterns 1:1 from other technologies will not work**. What is essential, we did not give up idiomatic Go code and microservices architecture - they fit together perfectly!

Today I would like to share with you first, most straightforward technique – DDD lite.

State of DDD in Golang

Before sitting to write this article, I checked a couple of articles about DDD in Go in Google. I will be brutal here: they are all missing the most critical points making DDD working. **If I imagine that I would read these articles without any DDD knowledge, I would not be encouraged to use them in my team. This superficial approach may also be the reason why DDD is still not socialized in the Go community.**

During this series, we try to show all essential techniques and do it in the most pragmatic way. Before describing any patterns, we start with a question: what does it give us? It's an excellent way to challenge our current thinking.

I'm sure that we can change the Go community reception of these techniques with this cycle of articles. We believe that they are the best way to implement complex business projects. **I believe that we will help to establish the position of Go as a great language for building not only infrastructure but also business software.**

You need to go slow, to go fast

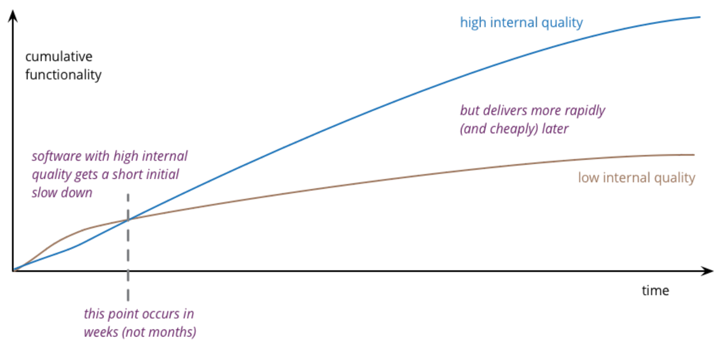
It may be tempting to implement the project you work on in the simplest way. It's even more tempting when you feel pressure from "the top". Are we using microservices, though? If it is needed, will we just rewrite the service? I heard that story multiple times, and it rarely ended with a happy end. 😉 **It's true that you will save some time with taking shortcuts. But onl in the short term.**

Let's consider the example of tests of any You can skip writing tests at the beginning of the project. You will obviously save some time and management will be happy. **Calculation seems to be simple – the project was delivered faster.**

But this shortcut is not worthwhile in the longer term. When the project grows, your team will start to be afraid of making any changes. In the end, the sum of time that you will spend will be higher than implementing tests from the beginning. **You will be slow down in the long term because of sacrificing quality for quick performance boost at the beginning.** On the other hand - if a project is not critical and needs to be created fast, you can skip tests. It should be a pragmatic decision, not just *"we know better, and we are not creating bugs"*.

The case is the same for DDD. When you want to use DDD, you will need a bit more time in the beginning, but long-term saving is enormous. However, not every project is complex enough to use advanced techniques like DDD.

There is no quality vs. speed tradeoff. If you want to go fast in the long term, you need to keep high quality.



'Is High Quality Software Worth the Cost?' from martinfowler.com

That's great, but do you have any evidence if that is workin

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

If you asked me this question two years ago, I would say: *"Well, I feel that's working better!"*. But just trusting my words may be not enough. 😉 **There are many tutorials showing some dumb ideas and claiming that they are working without any evidence – let's don't trust them blindly!**

Just to remind: if someone has a couple thousand Twitter followers, that alone is not a reason to trust them!

Fortunately, 2 years ago *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations* was released. In brief, this book is a description of what factors affect development teams' performance. But the reason this book became famous is that it's not just a set of not validated ideas – **it's based on scientific research.**

I was mostly interested with the part showing what allows teams to be top-performing teams. This book shows some obvious facts, like introducing DevOps, CI/CD, and loosely coupled architecture, which are all an essential factor in high-performing teams.

> If things like DevOps and CI/CD are not obvious to you, you can start with these books: The Phoenix Project and The DevOps Handbook.

What is Accelerate telling us about performing teams?

> We found that high performance is possible with all kinds of systems, provided that systems and the teams that build and maintain them—are loosely coupled.

> This key architectural property enables teams to easily test and deploy individual components or services even as the organization and the number of systems it operates grow—that is, it allows organizations to increase their productivity as they scale.

So let's use microservices, and we are done? I would not be writing this article if it was enough. 😉

> - Make large-scale changes to the design of their system without depending on other teams to make changes in their systems or creating significant work for other teams
> - Complete their work without communicating and coordinating with people outside their team
> - Deploy and release their product or service on demand, regardless of other services it depends upon
> - Do most of their testing on demand, without requiring an integrated test environment Perform deployments during normal business hours with negligible downtime

> Unfortunately, in real life, many so-called service-oriented architectures don't permit testing and deploying services independently of each other, and thus will not enable teams to achieve higher performance.

> […] employing the latest whizzy microservices architecture deployed on containers is no guarantee of higher performance if you ignore these characteristics. […] To achieve these characteristics, design systems are loosely coupled — that is, can be changed and validated independently of each other.

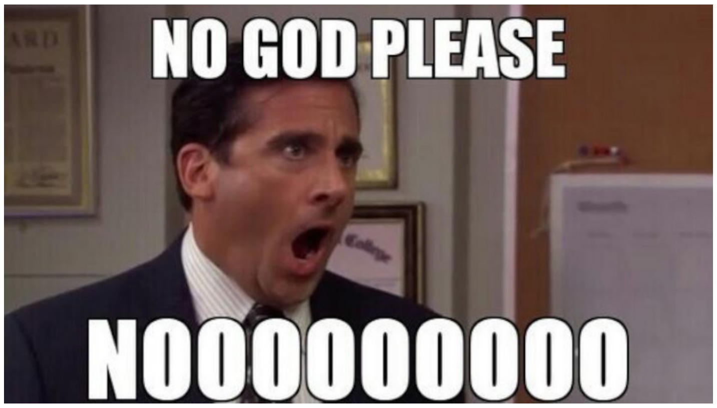
Using just microservices architecture and splitting services to small pieces is not enough. If it's done in a wrong way, it's adding extra complexity and slowing teams down. DDD can help us here.

I'm mentioning DDD term multiple times. What DDD actually is?

What is DDD (Domain-Driven Design)

Let's start with Wikipedia definition:

> Domain-driven design (DDD) is the concept that the structure and language of your code (class names, class methods, class variables) should match the business domain. For example, if your software processes loan applications, it might have classes such as LoanApplication and Customer, and methods such as AcceptOffer and Withdraw.



Well, it's not the perfect one. 😅 It's still missing some most important points.

It's also worth to mention, that DDD was introduced in 2003. That's pretty long time ago. Some distillation may be helpful to put DDD in the 2020 and Go context.

> If you are interested in some historical context on when DDD was created, you should check Tackling Complexity in the Heart of Software by the DDD creator - Eric Evans

Eric Evans - the creator of DDD.
Please print this and hang over the bed to receive +10 DDD blessing.

My simple DDD definition is: Ensure that you solve **valid problem** in the **optimal way**. After that **implement the solution in a way that your business will understand without any extra translation from technical language needed**.

How to achieve that?

Coding is a war, to win you need a strategy!

I like to say that *"5 days of coding can save 15 minutes of planning"*.

Before starting to write any code, you sh[illegible] ensure that you are solving a valid proble[illegible] may sound obvious, but in practice from my

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

experience, it's not as easy as it sounds. This is often the case that the solution created by engineers is not actually solving the problem that the business requested. A set of patterns that helps us in that field are named **Strategic DDD patterns**.

From my experience, DDD Strategic Patterns are often skipped. The reason is simple: we are all developers, and we like to write code rather than talk to the *"business people"*. 😉 Unfortunately, this approach when we are closed in a basement without talking to any business people has a lot of downsides. Lack of trust from the business, lack of knowledge of how the system works (from both business and engineering side), solving wrong problems – these are just some of the most common issues.

The good news is that in most cases it's caused by a lack of proper techniques like Event Storming. They can give both sides advantages. What is also surprising is that talking to the business may be one of the most enjoyable parts of the work!

Apart from that, we will start with patterns that apply to the code. They can give us **some** advantages of DDD. They will also be useful for you faster.
Without Strategic patterns, I'd say that you will just have 30% of advantages that DDD can give you. We will go back to Strategic patterns in the next articles.

DDD Lite in Go

After a pretty long introduction, it's finally ti touch some code! In this article, we will cover some basics of **Tactical Domain-Driven Design patterns in Go**. Please keep in mind that this is just the beginning. There will be a couple more articles needed to cover the entire topic.

One of the most crucial parts of Tactical DDD is trying to reflect the domain logic directly in the code.

But it's still some non-specific definition – and it's not needed at this point. I don't also want to start by describing what are *Value Objects, Entities, Aggregates*. Let's better start with practical examples.

Wild workouts

This is not just another article with dummy code snippets! ⚠

This post is part of a bigger series of artic
In the series, we show how to build **Go**

applications that are easy to develop, maintain, and fun to work with in the long term. We are doing it by sharing proven techniques based on many experiments that we did with the teams we are leading and on scientific research.

You can learn these patterns by building with us a fully functional example Go web application – **Wild Workouts**.

If it's the first article from the series that you are reading, you don't need to read all previous articles. Patterns presented in this article work standalone. Anyway, it's highly recommended – they are working the best if you will combine all of them!

There is one thing that we did in a totally different way in Wild Workouts than in all other guides – **we included some subtle issues to the initial implementation**. Have we lost our minds to do that? Not yet. 😉 From our observations, they are common for many Go projects. **In the long term, these small issues become critical and are stopping adding new features.**

In multiple articles we will refactor "Too modern" Wild Workouts, into an application that is easy to maintain in the long term and fun to work with. With refactoring in upcoming articles, you will easily understand the techniques we share.

If you are interested in how the Wild Workouts app is built, **how to run it loc with one command**, you should check "Too modern application" article.

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

> If you want to learn how you can **deploy your copy of Wild Workouts to Google Cloud** using just one command with **Terraform** you should check A complete Terraform setup of a serverless application on Google Cloud Run and Firebase.
>
> You can also check **the full list of 7 articles**.
>
> **Full source code** of Wild Workouts is available on GitHub.

I didn't mention yet, that especially for these articles, we created an entire application named Wild Workouts. What is interesting, we introduced some subtle issues in this application to have something to refactor. If Wild Workouts looks like an application that you are working on – better stay with us for a moment 😉.

Refactoring of `trainer` service

The first (micro)service that we will refactor is `trainer`. We will leave other services untouched now – we will go back to them later.

This service is responsible for keeping the trainer schedule and ensuring that we can have only one training scheduled in one hour. It also keeps the information about available hours (trainer's schedule).

The initial implementation was not the best, if it is not a lot of logic, some parts of the started to be messy. I have also some fe based on experience, that with time it wi worse. 😉

```
func (g GrpcServer) UpdateHour(ctx context.
    trainingTime, err := grpcTimestampToTime
    if err != nil {
        return nil, status.Error(codes.Invali
    }

    date, err := g.db.DateModel(ctx, trainin
    if err != nil {
        return nil, status.Error(codes.Intern
    }

    hour, found := date.FindHourInDate(train
    if !found {
        return nil, status.Error(codes.NotFou
    }

    if req.HasTrainingScheduled && !hour.Ava
        return nil, status.Error(codes.Failed
    }

    if req.Available && req.HasTrainingSched
        return nil, status.Error(codes.Failed
    }
    if !req.Available && !req.HasTrainingSch
        return nil, status.Error(codes.Failed
    }
    hour.Available = req.Available

    if hour.HasTrainingScheduled && hour.Has
        return nil, status.Error(codes.Failed
    }

    hour.HasTrainingScheduled = req.HasTrain
```

Full source: github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/grpc.go

Even if it's not the worst code ever, it reminds me what I've seen when I checked the git hist the code I worked on. I can imagine that after some time some new features will arrive and it will be much worse.

It's also hard to mock dependencies here, so there are also no unit tests.

The First Rule - reflect your business logic literally

While implementing your domain, you should stop thinking about structs like dummy data structures or "ORM like" entities with a list of setters and getters. You should instead think about them like **types with behavior.**

When you are talk with your business stakeholders, they say *"I'm scheduling training on 13:00"*, rather than *"I'm setting the attribute state to 'training scheduled' for hour 13:00."*.

They also don't say: *"you can't set attribute status to 'training_scheduled'"*. It is rather: *"You can't schedule training if the hour is not available"*. How to put it directly in the code?

```
func (h *Hour) ScheduleTraining() error {
    if !h.IsAvailable() {
        return ErrHourNotAvailable
    }

    h.availability = TrainingScheduled
    return nil
}
```

Full source: github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/domain/hour/availability.go

One question that can help us with implementation is: *"Will business understand my*

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

code without any extra translation of technical terms?". You can see in that snippet, that **even not technical person will be able to understand when you can schedule training**.

This approach's cost is not high and helps to tackle complexity to make rules much easier to understand. Even if the change is not big, we got rid of this wall of `if` s that would become much more complicated in the future.

We are also now able to easily add unit tests. What is good – we don't need to mock anything here. The tests are also a documentation that helps us understand how `Hour` behaves.

```
func TestHour_ScheduleTraining(t *testing.T
	h, err := hour.NewAvailableHour(validTra
	require.NoError(t, err)

	require.NoError(t, h.ScheduleTraining())

	assert.True(t, h.HasTrainingScheduled())
	assert.False(t, h.IsAvailable())
}

func TestHour_ScheduleTraining_with_not_ava
	h := newNotAvailableHour(t)
	assert.Equal(t, hour.ErrHourNotAvailable
}
```

Full source: github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/domain/hour/availability_test.go

Now, if anybody will ask the question "When schedule training", you can quickly answer that. In a bigger systems, the answer to this kir question is even less obvious – multiple times I spent hours trying to find all places where some objects were used in an unexpected way. The next rule will help us with that even more.

The Second Rule: always keep a valid state in the memory

> I recognize that my code will be used in ways I cannot anticipate, in ways it was not designed, and for longer than it was ever intended.
>
> *The Rugged Manifesto*

The world would be better if everyone would take this quote into account. I'm also not without fault here. 😉

From my observation, when you are sure that the object that you use is always valid, it helps to avoid a lot of `if` s and bugs. You will also feel much more confident knowing that you are not able to do anything stupid with the current code.

I have many flashbacks that I was afraid to make some change because I was not sure of the side effects of it. **Developing new features is much slower without confidence that you are correctly using the code!**

Our goal is to do validation in only one place (good DRY) and ensure that nobody can change the internal state of the `Hour` . The only public API of the object should be methods describing behaviors. No dumb getters and setters!. We to also put our types to separate package and make all attributes private.

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

```
type Hour struct {
    hour time.Time

    availability Availability
}

// ...

func NewAvailableHour(hour time.Time) (*Hou
    if err := validateTime(hour); err != nil
        return nil, err
    }

    return &Hour{
        hour:         hour,
        availability: Available,
    }, nil
}
```

Full source: github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/domain/hour/hour.go

We should also ensure that we are not breaking any rules inside of our type.

Bad example:

```
h := hour.NewAvailableHour("13:00")

if h.HasTrainingScheduled() {
    h.SetState(hour.Available)
} else {
    return errors.New("unable to cancel tra
}
```

Good example:

```
func (h *Hour) CancelTraining() error {
   if !h.HasTrainingScheduled() {
      return ErrNoTrainingScheduled
   }

   h.availability = Available
   return nil
}

// ...

h := hour.NewAvailableHour("13:00")
if err := h.CancelTraining(); err != nil {
    return err
}
```

The Third Rule - domain needs to be database agnostic

There are multiple schools here – some are telling that it's fine to have domain impacted by the database client. From our experience, keeping the domain strictly without any database influence works best.

The most important reasons are:

- domain types are not shaped by used database solution – they should be only shaped by business rules
- we can store data in the database in a more optimal way
- because of the Go design and lack of "ma like annotations, ORM's or any database solutions are affecting in even more signi way

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

Domain-First approach

If the project is complex enough, we can spend even 2-4 weeks to work on the domain layer, with just in-memory database implementation. In that case, we can explore the idea deeper and defer the decision to choose the database later. All our implementation is just based on unit tests.

We tried that approach a couple of times, and it always worked nicely. It is also a good idea to have some timebox here, to not spend too much time.

Please keep in mind that this approach requires a good relationship and a lot of trust from the business! **If your relationship with business is far from being good, Strategic DDD patterns will improve that. Been there, done that!**

To not make this article long, let's just introduce the Repository interface and assume that it's working. 😉 I will cover this topic more in-depth in the next article.

```
type Repository interface {
	GetOrCreateHour(ctx context.Context, tim
	UpdateHour(
		ctx context.Context,
		hourTime time.Time,
		updateFn func(h *Hour) (*Hour, error)
	) error
}
```

Full source: github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/domain/hour/repository.go

> You may ask why `UpdateHour` has `updateFn func(h *Hour) (*Hour, error)` – we will use that for handling transactions in a nice way. More in the article about repositories! 😉

Using domain objects

I did a small refactor of our gRPC endpoints, to provide an API that is more "behavior-oriented" rather than CRUD. It reflects better the new characteristic of the domain. From my experience, it's also much easier to maintain multiple, small methods than one, "god" method allowing us to update everything.

```
--- a/api/protobuf/trainer.proto
+++ b/api/protobuf/trainer.proto
@@ -6,7 +6,9 @@ import "google/protobuf/tim

 service TrainerService {
   rpc IsHourAvailable(IsHourAvailableReque
-  rpc UpdateHour(UpdateHourRequest) return
+  rpc ScheduleTraining(UpdateHourRequest)
+  rpc CancelTraining(UpdateHourRequest) re
+  rpc MakeHourAvailable(UpdateHourRequest)
 }

 message IsHourAvailableRequest {
@@ -19,9 +21,6 @@ message IsHourAvailableRe

 message UpdateHourRequest {
   google.protobuf.Timestamp time = 1;
-
-  bool has_training_scheduled = 2;
-  bool available = 3;
 }
```

```
message EmptyResponse {}
```

Full source: github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/commit/0249977c58a310d343ca2237c201b9ba016b148e#diff-15fd9ad3f3992b0210090109b82c5594

The implementation is now much simpler and easier to understand. We also have no logic here - just some orchestration. Our gRPC handler now has 18 lines and no domain logic!

```
func (g GrpcServer) MakeHourAvailable(ctx c
	trainingTime, err := protoTimestampToTim
	if err != nil {
		return nil, status.Error(codes.Invali
	}

	if err := g.hourRepository.UpdateHour(ct
		if err := h.MakeAvailable(); err != n
			return nil, err
		}

		return h, nil
	}); err != nil {
		return nil, status.Error(codes.Intern
	}

	return &trainer.EmptyResponse{}, nil
}
```

Full source: github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/grpc.go

No more Eight-thousanders

As I remember from the old-times, a lot Eight-thousanders was actually controllers with a lot of domain logic in HTTP controllers.

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

With hiding complexity inside of our domain types and keeping rules that I described, we are preventing uncontrolled growth in this place.

That's all for today

I don't want to make this article too long – let's go step by step!

If you can't wait, the entire working diff for the refactor is available on GitHub. In the next article, I'll cover one part from the diff that is not explained here: repositories.

Even if it's still the beginning, some simplifications in our code are visible.

The current implementation of the model is also not perfect – that's good! You will never implement the perfect model from the beginning. **It's better to be prepared to change this model easily, instead of wasting time to make it perfect.** After I added tests of the model and separated it from the rest of the application, I can change it without any fear.

Can I already put that I know DDD to my CV?

Not yet.

I needed 3 years after I heard about DDD t time when I connected all dots (It was before I heard about Go 😉). After that, I've seen w techniques that we will describe in the next articles are so important. But before connecting the dots, it will require some patience and trust that it will work. It is worth that! You will not need 3 years like me, but we currently planned about 10 articles on both strategic and tactical patterns. 😉 It's a lot of new features and parts to refactor in Wild Workouts left!

I know that nowadays a lot of people promise that you can become expert in some area after 10 minutes of an article or a video. The world would be beautiful if it would be possible, but in reality, it is not so simple.

Fortunately, a big part of the knowledge that we are sharing is universal and can be applied to multiple technologies, not only Go. You can treat these learnings as an investment in your career and mental health in the long term 😉. There is nothing better than solving the right problems without fighting with the unmaintainable code!

What is your experience with DDD in Go? Was it good or bad? Was it different from how we are doing that? Do you think that DDD may be useful in your project? Please let us know in the comments!

Do you have any colleagues that you think may be interested in this topic? Please share this article with them! Even if they are not working in Go. 😉

Follow @roblaszczak

Did you like this article?

We are working to create a lot of new useful content. We can let you know when a new post appears. Your e-mail will be safe and not shared with anyone.

Your name

E-mail

SUBSCRIBE

go golang ddd domain-driven design building-business-applications

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE

Three Dots Labs Blog Comment Policy

We welcome all relevant and respectful comments.
comments may be removed.

3 Comments **Three Dots Labs Blog**

Recommend 3 Tweet Share

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQU

Name

c_world_poster • 19 days ago

It is not doing java in go. It is plain and simpl
Too many people let hubris get in the way. M
business problems and write maintainable a
wheel. You are either using a framework, wri
dealing with the fact that you dont.

2 • Reply • Share ›

Rein Krul • 17 days ago

Good article on using Object Oriented progra
a far broader topic than discussed here, imh

On the gRPC API you refactored from CRUD
reflects the domain; sounds like the original
API took a REST-approach to gRPC. CRUD
symptomatic to REST. An API that better refl
natural to gRPC since well, it's RPC, Remote

That said, you *can* design a REST API that r
described in this article; have a POST to /sch
training, rather than POSTing an update to /t

• Reply • Share ›

© Three Dots Labs Cookies Policy

Three Dots Labs

Golang, Domain-Driven Design and Continuous Delivery.

Home

Tags

About us

Don't miss new posts. Subscribe to our newsletter!

Your name

E-mail

SUBSCRIBE