GO WITH THE DOMAIN

BUILDING MODERN BUSINESS SOFTWARE IN GO

ROBERT LASZCZAK

MIŁOSZ SMÓŁKA

Three Dots Labs

Hey there!

Hi Muralish!

This e-book copy has been prepared especially for you on 2022-06-20 and delivered to mcmuralishclint@outlook.com.

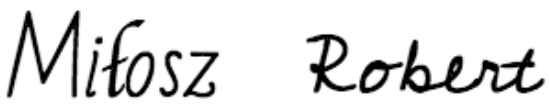
The e-book version: 20210903

Thank you for subscribing to our newsletter and we hope you enjoy!

If you want to share this book with your teammates (we hope so!), please let them know about our newsletter:

https://threedots.tech/newsletter

This will let us focus on creating new content. Thanks!



Contents

Cover design: Izabela Górecka

Chapter 1

Welcome on board!

Why business software

If you think "business software" is just enterprise payroll applications, think again. If you work on any web application or SaaS product, you write business software.

You've seen poor code before, and we did as well. What's worse, we've experienced how poorly designed codebases hold the product's rapid growth and the company's success.

We looked into different patterns and techniques on how to improve this. We applied them in practice within our teams and seen some work very well.

Why Go

Go's initial success happened mostly within low-level applications, like CLI tools and infrastructure software, like Kubernetes. However, Go 2020 Survey[1] shows that majority of Go developers write API/RPC services. 68% percent of respondents use Go for "web programming".

Some claim that Go is missing advanced features to become a productive tool. On the other hand, we feel simplicity is Go's strength. It's viable for writing business applications not despite the lack of features but because of it.

You don't want your business's most crucial code to be smart and using the latest hacks. You need it to be maintainable, extremely clear, and easy to change. We've found that using Go along advanced architectural patterns can give you that.

It's naive to think you will avoid technical debt just thanks to using Go. Any software project that's complex enough can become a Big Ball of Mud[2] if you don't think about its design beforehand. Spoiler alert: microservices don't reduce complexity.

[1] https://blog.golang.org/survey2020-results
[2] https://en.wikipedia.org/wiki/Big_ball_of_mud

How is this book different

In 2020, we started a new series on our blog focused on building business software.

We didn't want to write articles where the author makes bold claims, but the described techniques are hard to apply. Examples are great for learning, so we decided to create a real, open-source, and deployable web application that would help us show the patterns.

In the initial version of the application, we've hidden on purpose some anti-patterns. They seem innocent, are hard to spot, and can hurt your development speed later.

In the following posts from the series, we refactored the application several times, showing the anti-patterns and how to avoid them. We also introduced some essential techniques like DDD, CQRS, and Clean Architecture. The entire commit history is available for each article.

Posts from that blog series are *Go with the Domain's* foundation - edited and prepared for a better reading experience.

Who is this book for

We assume you have basic knowledge of Go and already worked on some projects. Ideally, you're looking for patterns that will help you design applications to not become legacy software in a few months.

Most of the ideas in this book shine in applications with complex business scenarios. Some make sense in simpler cases, and some will look like terrible over-engineering if used in small projects. Try to be pragmatic and choose the best tool for the job. We share some hints on this topic as well.

Who we are

We're Miłosz[3] and Robert[4], co-founders of Three Dots Labs[5]. We based this book on posts from our tech blog[6].

[3] https://twitter.com/m1_10sz
[4] https://twitter.com/roblaszczak
[5] https://threedotslabs.com/
[6] https://threedots.tech/

Chapter 2

Building a serverless application with Google Cloud Run and Firebase

Robert Laszczak

Welcome to the first chapter covering how to build business-oriented applications in Go! In this book, we want to show you how to build applications that are easy to develop, maintain, and fun to work with in the long term.

The idea of this book is to not focus too much on infrastructure and implementation details. But we need to have some base on which we can build later. In this chapter, we start by covering some basic tools from Google Cloud that can help us to do that.

Why serverless?

Running a Kubernetes cluster requires a lot of support from "DevOps teams". Let's skip the fact that DevOps is not a job title for now.

Figure 2.1: DevOps is a culture and mindset. It is not a job title! Slide from The gordian knot - Alberto Brandolini

Small applications that can be easily run on one virtual machine are now being deployed on super complex Kubernetes clusters. All these clusters require a lot of maintenance.

On the other hand, moving applications to containers has given us much flexibility in building and deploying them. It allowed us to do rapid **deployments of hundreds of microservices** with a lot of **autonomy**. But the **cost for that is high**.

Wouldn't it be great if any fully managed solution existed?

Maybe your company is already using a managed Kubernetes cluster. If so, you probably already know that even your managed cluster still requires a ton of "DevOps" support.

Maybe serverless? Well, splitting a big application to multiple, independent Lambdas (Cloud Functions) is a **great way to an unmaintainable cataclysm**.

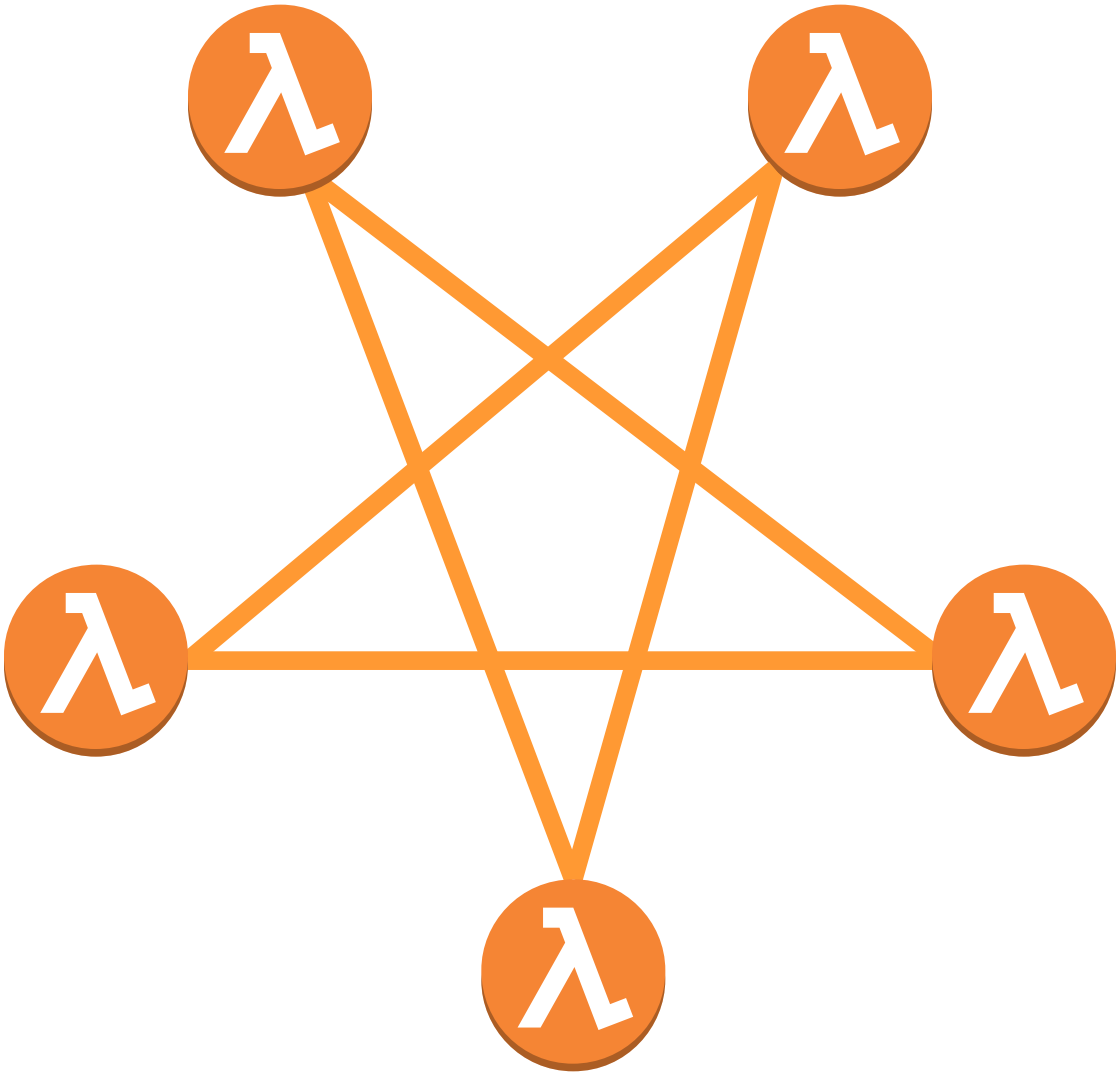


Figure 2.2: You should probably rethink your architecture if it can be used for summoning demons.

But wait, is it the only way to build serverless applications? No!

Google Cloud Run

The idea of Google Cloud Run is pretty simple - you **just need to provide a Docker container, and Google Cloud runs it**.
Inside this container, you can run an application written in any language that can expose a port with your HTTP or gRPC API.

You are not limited to synchronous processing – you can process Pub/Sub messages inside of this container.

And that's all that you need from the infrastructure side. Google Cloud does all the magic. Based on the traffic, the container will automatically scale up and down. Sounds like a perfect solution?

In practice, it is not so simple. There are many articles showing how to use Google Cloud Run, but they usually show **small bricks that may be used for building an application**.
It's hard to join all these bricks from multiple places to create a fully working project (been there, done that).

In most cases, these articles leave out the problem of vendor lock-in. The deployment method should be just an implementation detail. I've already covered this topic in the *Why using Microservices or Monolith can be just a detail?*[1] article in 2018.

But what is most important – using all the newest and most shiny technologies doesn't mean that your application will not become a hated legacy in the next 3 months.

Serverless solves only infrastructure challenges. It doesn't stop you from building an application that is hard to maintain. I even have the impression that it's the opposite – all these fancy applications sooner or later are the hardest to maintain.

For this book, we created a fully functional, real-life application. You can deploy this application with one command to Google Cloud using Terraform. You can run the local copy with one docker-compose command.

There is also one thing that we are doing differently than others. **We included some subtle issues, that from our observations are common in Go projects.** In the long term, these small issues become critical and stop us from adding new features.

Have we lost our minds? Not yet. **This approach will help you to understand what issues you can solve and what techniques can help.** It is also a kind of challenge for practices that we use. If something is not a problem, why should we use any technique to solve it?

Plan

In the next couple of chapters, we will cover all topics related to running the application on Google Cloud. In this part, we didn't add any issues or bad practices. **The first chapters may be a bit basic if you already have**

[1] https://threedots.tech/post/microservices-or-monolith-its-detail/

some experience in Go. We want to ensure that if you are just starting with Go, you will be able to follow more complex topics that come next.

Next, we will refactor parts of the application handling business logic. **This part will be much more complex.**

Running the project locally

The ability to run a project locally is **critical for efficient development**. It is very annoying when you are not able to check changes that you did in a simple and fast way.

It's much harder to achieve it for projects built from hundreds of microservices. Fortunately, our project has only 5 services. **In Wild Workouts we created Docker Compose with live code reloading for both frontend and backend.** For the frontend, we are using a container with `vue-cli-service serve` tool. For the backend, the situation is a bit more complex. In all containers, we are running the `reflex` tool. `reflex` listens for any code changes that trigger the recompilation of the service. If you are interested in details, you can find them in our *Go Docker dev environment with Go Modules and live code reloading*[2] blog post.

Requirements

The only requirements needed to run the project are Docker[3] and Docker Compose[4].

Running

```
git clone https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example.git && cd wild-workouts-go-ddd-example
```

And run Docker Compose:

```
docker-compose up
```

After downloading all JS and Go dependencies, you should see a message with the frontend address:

```
web_1             | $ vue-cli-service serve
web_1             |  INFO  Starting development server...
web_1             |  DONE  Compiled successfully in 6315ms11:18:26 AM
web_1             |
web_1             |
web_1             |  App running at:
web_1             |  - Local:   http://localhost:8080/
web_1             |
web_1             |  It seems you are running Vue CLI inside a container.
web_1             |  Access the dev server via http://localhost:<your container's external mapped port>/
web_1             |
web_1             |  Note that the development build is not optimized.
web_1             |  To create a production build, run yarn build.
```

[2] https://threedots.tech/post/go-docker-dev-environment-with-go-modules-and-live-code-reloading
[3] https://www.docker.com/
[4] https://docs.docker.com/compose/

Congratulations! Your local version of the Wild Workouts application is available at http://localhost:8080/.

There is also a public version available at https://threedotslabs-wildworkouts.web.app[5].

What Wild Workouts can do?

How often did you see tutorials without any real-life functionality? How often didn't patterns from these tutorials work in real projects? Probably too often. **Real life is not as simple as in tutorials.**

To avoid this problem, we created Wild Workouts application as a fully functional project. It's much harder to do shortcuts and skip extra complexity when an application needs to be complete. It makes all chapters longer, but there are no shortcuts here. **If you don't spend enough time in the beginning, you will lose much more time later during the implementation. Or even worse – you'll be fixing problems in a rush with the application already running on production.**

tl;dr

Wild Workouts is an application for personal gym trainers and attendees.

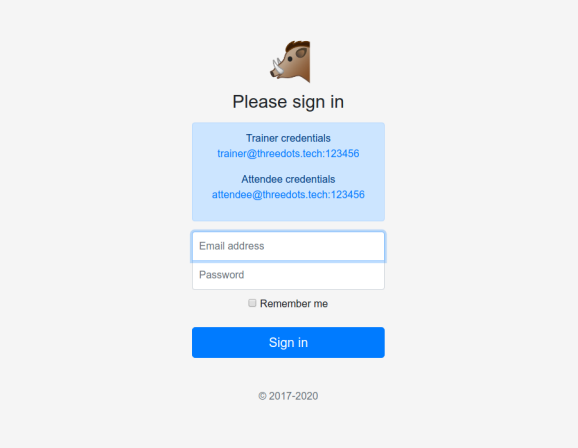


Figure 2.3: Login page

Trainers can set a schedule when they are available for the training.

Attendees can schedule training for provided dates.

Other functionalities are:

- management of "credits" (how many trainings the attendee can schedule)
- cancellation
  - if a training is canceled less than 24 hours before the time it begins, the attendee will not receive their credits back

[5]https://threedotslabs-wildworkouts.web.app/

Figure 2.4: Schedule

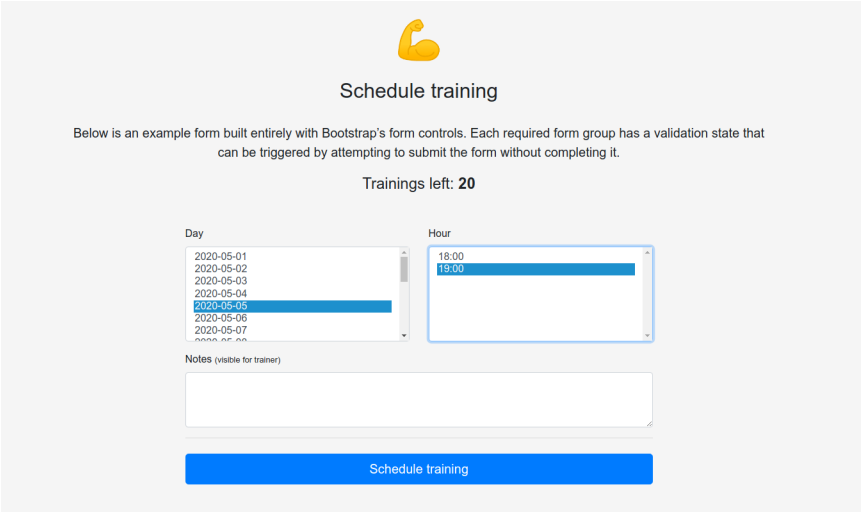


Figure 2.5: Schedule training

- training reschedule
  - if someone wants to reschedule a training less than 24 hours before the time it begins, it needs to be approved by the second participant (trainer or attendee)
- calendar view

Sounds simple. What can go wrong?

Frontend

If you are not interested in the frontend part, you can go straight to the Backend section.

I'm rather a backend engineer and to be honest I'm not the greatest JavaScript and frontend specialist. **But we can't have an application working end-to-end without a frontend!**

In this book, we focus on the backend part. I will give a high-level overview of what technologies I used on the frontend. **It will be nothing new to you if you have any basic frontend knowledge.** For more details, I recommend to check the source code in the web/ directory[6].

OpenAPI (Swagger) client

Nobody likes to keep API contracts up to date manually. It's annoying and counterproductive to keep multiple boring JSONs up-to-date. OpenAPI solves this problem with JavaScript HTTP client and Go HTTP server generated from the provided specification[7]. We will dive into details in the **Backend** part.

Figure 2.6: OpenAPI

Bootstrap

You probably already know Bootstrap[8], **the greatest friend of every backend engineer, like me**. Fighting with HTML and CSS is the part of frontend development that I dislike the most. Bootstrap provided me almost all building blocks needed for creating HTML of the application.

[6] https://bit.ly/3uw3xtB
[7] https://bit.ly/37D3hPE
[8] https://getbootstrap.com/

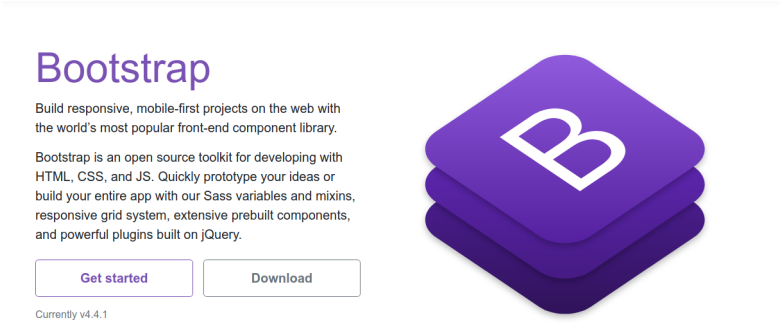


Figure 2.7: Bootstrap

Vue.js

After checking a couple of the most popular frontend frameworks, I decided to use Vue.js[9]. I really enjoyed the simplicity of this solution.

Figure 2.8: Vue.js

I was starting my journey as a full-stack developer in the pre-jQuery times. The frontend tooling made huge progress… but I'll stay with the backend for now.

Backend

The backend of Wild Workouts is built from 3 services.

- `trainer` – provides public HTTP[10] and internal gRPC endpoints[11] for managing trainer schedule
- `trainings` – provides public HTTP[12] for managing attendee trainings
- `users` – provides public HTTP endpoints[13] and internal gRPC endpoints[14] , manages credits and user data

If a service exposes 2 types of APIs, each of them is exposed in a separate process.

[9] https://vuejs.org/v2/guide/
[10] https://bit.ly/3aGUC0b
[11] https://bit.ly/3aGUC0b
[12] https://bit.ly/3upucIc
[13] https://bit.ly/3uh4rtA
[14] https://bit.ly/3uh4rtA

Public HTTP API

Most operations performed by applications are triggered by the public HTTP API. I've heard many times the question from newcomers to Go what framework they should use to create an HTTP service. **I always advise against using any kind of HTTP framework in Go. A simple router, like chi**[15] **is more than enough.** *chi* provides us only the lightweight glue to define what URLs and methods are supported by our API. Under the hood, it uses the Go standard library `http` package, so all related tools like middlewares are 100% compatible.

It may be a bit weird to not use a framework if you are coming from any language where Spring, Symfony, Django, or Express may be the obvious choice. It was weird for me too. Using any kind of framework in Go adds unnecessary complexity and will couple your project with this framework. KISS[16].

All the services are running the HTTP server in the same way. It sounds like a good idea to not copy it 3 times.

```
func RunHTTPServer(createHandler func(router chi.Router) http.Handler) {
    apiRouter := chi.NewRouter()
    setMiddlewares(apiRouter)

    rootRouter := chi.NewRouter()
    // we are mounting all APIs under /api path
    rootRouter.Mount("/api", createHandler(apiRouter))

    logrus.Info("Starting HTTP server")

    http.ListenAndServe(":"+os.Getenv("PORT"), rootRouter)
}
```

Source: http.go on GitHub[17]

chi provides us with a set of useful built-in HTTP middlewares, but we are not limited only to them. All middlewares compatible with Go standard library will work.

> **Note**
>
> Long story short – middlewares allow us to do anything before and after a request is executed (with access to the `http.Request`). Using HTTP middlewares gives us a lot of flexibility in building our custom HTTP server. We are building our server from multiple decoupled components that you can customize for your purposes.

```
func setMiddlewares(router *chi.Mux) {
    router.Use(middleware.RequestID)
    router.Use(middleware.RealIP)
    router.Use(logs.NewStructuredLogger(logrus.StandardLogger()))
    router.Use(middleware.Recoverer)

    addCorsMiddleware(router)
```

[15] https://github.com/go-chi/chi
[16] https://en.wikipedia.org/wiki/KISS_principle
[17] https://bit.ly/3bonCsL

```
    addAuthMiddleware(router)

    router.Use(
        middleware.SetHeader("X-Content-Type-Options", "nosniff"),
        middleware.SetHeader("X-Frame-Options", "deny"),
    )
    router.Use(middleware.NoCache)
}
```

Source: http.go on GitHub[18]

We have our framework almost ready now. :) It's time to use that. We can call `server.RunHTTPServer` in the `trainings` service.

```
package main
// ...
func main() {
    // ...
     server.RunHTTPServer(func(router chi.Router) http.Handler {
         return HandlerFromMux(HttpServer{firebaseDB, trainerClient, usersClient}, router)
     })
}
```

Source: main.go on GitHub[19]

`createHandler` needs to return `http.Handler`. In our case it is `HandlerFromMux` generated by oapi-codegen[20].

It provides us all the paths and query parameters from the OpenAPI specs[21].

```
// HandlerFromMux creates http.Handler with routing matching OpenAPI spec based on the provided mux.
func HandlerFromMux(si ServerInterface, r chi.Router) http.Handler {
    r.Group(func(r chi.Router) {
        r.Use(GetTrainingsCtx)
        r.Get("/trainings", si.GetTrainings)
    })
    r.Group(func(r chi.Router) {
        r.Use(CreateTrainingCtx)
        r.Post("/trainings", si.CreateTraining)
    })
// ...
```

Source: openapi_api.gen.go on GitHub[22]

```
# ...
paths:
  /trainings:
    get:
      operationId: getTrainings
```

[18] https://bit.ly/3aFCFzn
[19] https://bit.ly/3k63dNf
[20] https://github.com/deepmap/oapi-codegen
[21] https://bit.ly/37D3hPE
[22] https://bit.ly/3bqtTUN

```
      responses:
        '200':
          description: todo
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Trainings'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
# ...
```

Source: trainings.yml on GitHub[23]

If you want to make any changes to the OpenAPI spec, you need to regenerate Go server and JavaScript clients after. You need to run:

```
make openapi
```

Part of the generated code is `ServerInterface`. It contains all methods that need to be supported by the API. Implementation of server functionality is done by implementing that interface.

```
type ServerInterface interface {
   //  (GET /trainings)
   GetTrainings(w http.ResponseWriter, r *http.Request)
   //  (POST /trainings)
   CreateTraining(w http.ResponseWriter, r *http.Request)
    // ...
}
```

Source: openapi_api.gen.go on GitHub[24]

This is an example of how `trainings.HttpServer` is implemented:

```
package main

import (
   "net/http"

   "github.com/go-chi/render"
   "gitlab.com/threedotslabs/wild-workouts/pkg/internal/auth"
   "gitlab.com/threedotslabs/wild-workouts/pkg/internal/genproto/trainer"
   "gitlab.com/threedotslabs/wild-workouts/pkg/internal/genproto/users"
   "gitlab.com/threedotslabs/wild-workouts/pkg/internal/server/httperr"
)

type HttpServer struct {
```

[23] https://bit.ly/3pFKRDI
[24] https://bit.ly/3pInFol

```
    db            db
    trainerClient trainer.TrainerServiceClient
    usersClient   users.UsersServiceClient
}

func (h HttpServer) GetTrainings(w http.ResponseWriter, r *http.Request) {
    user, err := auth.UserFromCtx(r.Context())
    if err != nil {
        httperr.Unauthorised("no-user-found", err, w, r)
        return
    }

    trainings, err := h.db.GetTrainings(r.Context(), user)
    if err != nil {
        httperr.InternalError("cannot-get-trainings", err, w, r)
        return
    }

    trainingsResp := Trainings{trainings}

    render.Respond(w, r, trainingsResp)
}

// ...
```

Source: http.go on GitHub[25]

But HTTP paths are not the only thing generated from OpenAPI spec. **More importantly, it provides us also the models for responses and requests.** Models are, in most cases, much more complex than API paths and methods. **Generating them can save time, issues, and frustration during any API contract changes.**

```yaml
# ...
 schemas:
    Training:
      type: object
      required: [uuid, user, userUuid, notes, time, canBeCancelled, moveRequiresAccept]
      properties:
        uuid:
          type: string
          format: uuid
        user:
          type: string
          example: Mariusz Pudzianowski
        userUuid:
          type: string
          format: uuid
        notes:
          type: string
          example: "let's do leg day!"
        time:
```

[25] https://bit.ly/3ukpVWH

```
          type: string
          format: date-time
        canBeCancelled:
          type: boolean
        moveRequiresAccept:
          type: boolean
        proposedTime:
          type: string
          format: date-time
        moveProposedBy:
          type: string

    Trainings:
      type: object
      required: [trainings]
      properties:
        trainings:
          type: array
          items:
            $ref: '#/components/schemas/Training'
# ...
```

Source: trainings.yml on GitHub[26]

```
// Training defines model for Training.
type Training struct {
   CanBeCancelled     bool       `json:"canBeCancelled"`
   MoveProposedBy     *string    `json:"moveProposedBy,omitempty"`
   MoveRequiresAccept bool       `json:"moveRequiresAccept"`
   Notes              string     `json:"notes"`
   ProposedTime       *time.Time `json:"proposedTime,omitempty"`
   Time               time.Time  `json:"time"`
   User               string     `json:"user"`
   UserUuid           string     `json:"userUuid"`
   Uuid               string     `json:"uuid"`
}

// Trainings defines model for Trainings.
type Trainings struct {
   Trainings []Training `json:"trainings"`
}
```

Source: openapi_types.gen.go on GitHub[27]

Cloud Firestore database

All right, we have the HTTP API. But even the best API without any data and without the ability to save anything is useless.

[26] https://bit.ly/2P0KI1n
[27] https://bit.ly/3k7NYDj

If we want to build the application in **the most modern, scalable, and truly serverless way, Firestore is a natural choice**. We will have that out of the box. What is the cost of that?

Figure 2.9: Firestore

If we are talking about financial cost for the Europe multi-region[28] option we need to pay:

- $0.06 per 100,000 documents reads
- $0.18 per 100,000 documents writes
- $0.02 per 100,000 documents deletes
- $0.18/GiB of stored data/month

Sounds pretty cheap?

For comparison, let's take the cheapest Cloud SQL MySQL[29] `db-f1-micro` instance with shared Virtual CPU and `3 GB` of storage as a reference – it costs *$15.33/month*. The cheapest instance with high availability and with 1 non-shared Virtual CPU costs *$128.21/month*.

What is even better, in the free plan[30], you can store up to 1 GiB of data with 20k document writes per day.

Firestore is a NoSQL database, so we should not expect to build relational models in the SQL manner. Instead of that, we have a system of hierarchical collections[31]. In our case, the data model is pretty simple, so we have only one level of collections.

In contrast to many NoSQL databases, **Firestore offers ACID transactions on any operation**. It also works when updating multiple documents.

Firestore limitations

The important limitation may be the limit of *1 update / second / one document*. **It still means that you can update a lot of independent documents in parallel**. It is an important factor that you should consider while designing your database. In some cases, you should consider batching operations, different documents design, or using a different database. If data is changing often, maybe a key-value database would be a good choice?

From my experience, the limitation of 1 update per second per document is not a serious problem. In most cases, when I was using Firestore, we were updating a lot of independent documents. This is also true

[28] https://firebase.google.com/docs/firestore/pricing#europe
[29] https://cloud.google.com/sql/pricing#2nd-gen-pricing
[30] https://firebase.google.com/pricing
[31] https://firebase.google.com/docs/firestore/data-model

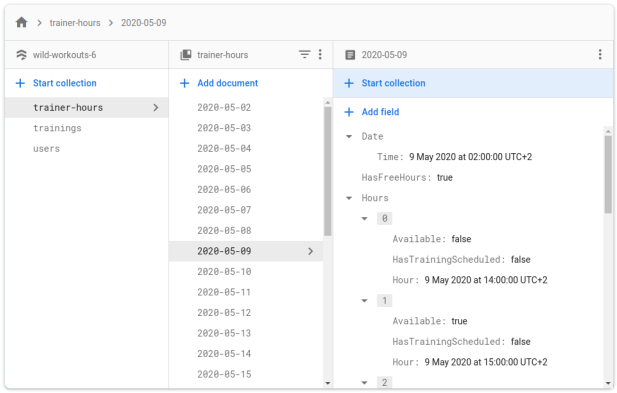


Figure 2.10: Firestore Console

when using Firestore for **event sourcing** - you will only use append operations. In Wild Workouts, we should also not have a problem with this limit.

> Note
>
> I have also observed that Firestore needs some time to warm up. In different words – if you want to insert 10 mln documents within one minute after you just set up a new project, **it may not work**. I guess this is related to some internal magic that handles scaleability.
> Fortunately, in the real world, it is not common to have traffic spikes from 0 to 10 mln writes / minute.

Running Firestore locally

Unfortunately Firestore emulator is not perfect.
I found some situations where the emulator was not 100% compatible with the real version. I also had some situations when I was doing an update and read of the same document in the transaction, and it caused a deadlock. From my point of view, this functionality is enough for local development.

The alternative may be to have a separate Google Cloud project for local development. My preference here is to have a local environment that is truly local and doesn't depend on any external services. It is also easier to set up and can be used later in Continuous Integration.

Since the end of May, the Firestore emulator provides a UI. It is added to the Docker Compose and is available at http://localhost:4000/. When I'm writing this, sub-collections are not displayed properly[32] in the UI in the emulator. Don't worry, for Wild Workouts it's not a problem.

Using Firestore

Apart from Firestore implementation, the code works in the same way locally and on the production. When we are using emulator locally, we need to run our application with env `FIRESTORE_EMULATOR_HOST` set to emulator

[32]https://github.com/firebase/firebase-tools-ui/issues/273

hostname (in our case `firestore:8787`). It is set in the `.env` file.

On production, all magic is done by Google Cloud under the hood, and no extra configuration is needed.

```
firebaseClient, err := firestore.NewClient(ctx, os.Getenv("GCP_PROJECT"))
if err != nil {
    panic(err)
}
```

Source: main.go on GitHub[33]

Here is the example of how I used Firestore client for querying trainer schedule. You can see how I used queries functionality[34] to get only dates from the queried dates interval.

```
package main

import (
    // ...
   "cloud.google.com/go/firestore"
   // ...
)

// ...

type db struct {
   firestoreClient *firestore.Client
}

func (d db) TrainerHoursCollection() *firestore.CollectionRef {
   return d.firestoreClient.Collection("trainer-hours")
}

// ...

func (d db) QueryDates(params *GetTrainerAvailableHoursParams, ctx context.Context) ([]Date, error) {
   iter := d.
      TrainerHoursCollection().
      Where("Date.Time", ">=", params.DateFrom).
      Where("Date.Time", "<=", params.DateTo).
      Documents(ctx)

   var dates []Date

   for {
      doc, err := iter.Next()
      if err == iterator.Done {
         break
      }
      if err != nil {
         return nil, err
```

[33] https://bit.ly/3bnhTDC
[34] https://firebase.google.com/docs/firestore/query-data/queries#simple_queries

```
        }

        date := Date{}
        if err := doc.DataTo(&date); err != nil {
            return nil, err
        }
        date = setDefaultAvailability(date)
        dates = append(dates, date)
    }

    return dates, nil
}
```

Source: firestore.go on GitHub[35]

There is also no extra data mapping needed. Firestore library can marshal any struct with public fields or `map[string]interface`. As long as there is nothing weird inside. You can find the entire specification of how conversion is done in cloud.google.com/go/firestore GoDoc[36].

```
type Date struct {
    Date         openapi_types.Date `json:"date"`
    HasFreeHours bool               `json:"hasFreeHours"`
    Hours        []Hour             `json:"hours"`
}
```

Source: openapi_types.gen.go on GitHub[37]

```
date := Date{}
if err := doc.DataTo(&date); err != nil {
    return nil, err
}
```

Source: firestore.go on GitHub[38]

Production deployment tl;dr

You can deploy your own version of Wild Workouts with one command:

```
> cd terraform/
> make

Fill all required parameters:
    project [current: wild-workouts project]:       # <----- put your Wild Workouts Google Cloud project name here (it
     ↪ will be created)
    user [current: email@gmail.com]:                # <----- put your Google (Gmail, G-suite etc.) e-mail here
    billing_account [current: My billing account]:  # <----- your billing account name, can be found here
     ↪ https://console.cloud.google.com/billing
```

[35] https://bit.ly/3btJPG9
[36] https://godoc.org/cloud.google.com/go/firestore#DocumentSnapshot.DataTo
[37] https://bit.ly/3pBnSd3
[38] https://bit.ly/3k7rPoM

```
    region [current: europe-west1]:
    firebase_location [current: europe-west]:

# it may take a couple of minutes...

The setup is almost done!

Now you need to enable Email/Password provider in the Firebase console.
To do this, visit https://console.firebase.google.com/u/0/project/[your-project]/authentication/providers

You can also downgrade the subscription plan to Spark (it's set to Blaze by default).
The Spark plan is completely free and has all features needed for running this project.

Congratulations! Your project should be available at: https://[your-project].web.app

If it's not, check if the build finished successfully:
↪  https://console.cloud.google.com/cloud-build/builds?project=[your-project]

If you need help, feel free to contact us at https://threedots.tech
```

We will describe the deployment in detail in the next chapters.

What's next?

In the next chapters, we will cover **internal gRPC communication between services** and **HTTP Firebase authentication**.

The code with gRPC communication and authentication is already on our GitHub. Feel free to read, run, and experiment with it.

Deployment and infrastructure

The infrastructure part is described in **Chapter 14**. It covers in details:

- Terraform
- Cloud Run
- CI/CD
- Firebase hosting

What is wrong with this application?!

After finishing all chapters describing the current application, we will start the part related with refactoring and adding new features to Wild Workouts.

We don't want to use many fancy techniques just to make our CVs look better. Our goal is to solve issues present in the application using Domain-Driven Design, Clean Architecture, CQRS, Event Storming, and Event Modeling.

We will do it by refactoring. It should be then visible **what issues are solved and how it makes the implementation more clean**. Maybe during this we will even see that some of these techniques are not useful in Go, who knows.

Chapter 3

gRPC communication on Google Cloud Run

Robert Laszczak

In this chapter we show how you can **build robust, internal communication between your services using gRPC**. We also cover some extra configuration required to set up authentication and TLS for the Cloud Run.

Why gRPC?

Let's imagine a story, that is true for many companies:

Meet Dave. Dave is working in a company that spent about 2 years on building their product from scratch. During this time, they were pretty successful in finding thousands of customers, who wanted to use their product. They started to develop this application during the biggest "boom" for microservices. It was an obvious choice for them to use that kind of architecture. Currently, they have more than 50 microservices using HTTP calls to communicate with each other.

Of course, Dave's company didn't do everything perfectly. The biggest pain is that currently all engineers are afraid to change anything in HTTP contracts. It is easy to make some changes that are not compatible or not returning valid data. It's not rare that the entire application is not working because of that. "Didn't we built microservices to avoid that?" – this is the question asked by scary voices in Dave's head every day.

Dave already proposed to use OpenAPI for generating HTTP server responses and clients. But he shortly found that he still can return invalid data from the API.

It doesn't matter if it (already) sounds familiar to you or not. The solution for Dave's company is simple and straightforward to implement. **You can easily achieve robust contracts between your services by using gRPC.**

The way of how servers and clients are generated from gRPC is way much stricter than OpenAPI. It's needless to

Figure 3.1: gRPC logo

say that it's infinitely better than the OpenAPI's client and server that are just copying the structures.

> Note
>
> It's important to remember that gRPC doesn't solve the **data quality problems**. In other words – you can still send data that is not empty, but doesn't make sense.
> It's important to ensure that data is valid on many levels, like the robust contract, contract testing, and end-to-end testing.

Another important "why" may be performance. You can find many studies that **gRPC may be even 10x faster than REST**. When your API is handling millions of millions of requests per second, it may be a case for cost optimization. In the context of applications like Wild Workouts, where traffic may be less than 10 requests/sec, **it doesn't matter**.

To be not biased in favor of using gRPC, I tried to find any reason not to use it for internal communication. I failed here:

- **the entry point is low**,
- adding a gRPC server doesn't need any extra infrastructure work – it works on top of HTTP/2,
- it workis with many languages like Java, C/C++, Python, C#, JS, and more[1],
- in theory, you are even able to use gRPC for the frontend communication[2] (I didn't test that),
- it's "Goish" – **the compiler ensures that you are not returning anything stupid.**

Sounds promising? Let's verify that with the implementation in Wild Workouts!

Generated server

Currently, we don't have a lot of gRPC endpoints in Wild Workouts. We can update trainer hours availability and user training balance (credits).

Let's check `Trainer gRPC service`. To define our gRPC server, we need to create `trainer.proto` file.

[1] https://grpc.io/about/#officially-supported-languages-and-platforms
[2] https://grpc.io/blog/state-of-grpc-web/

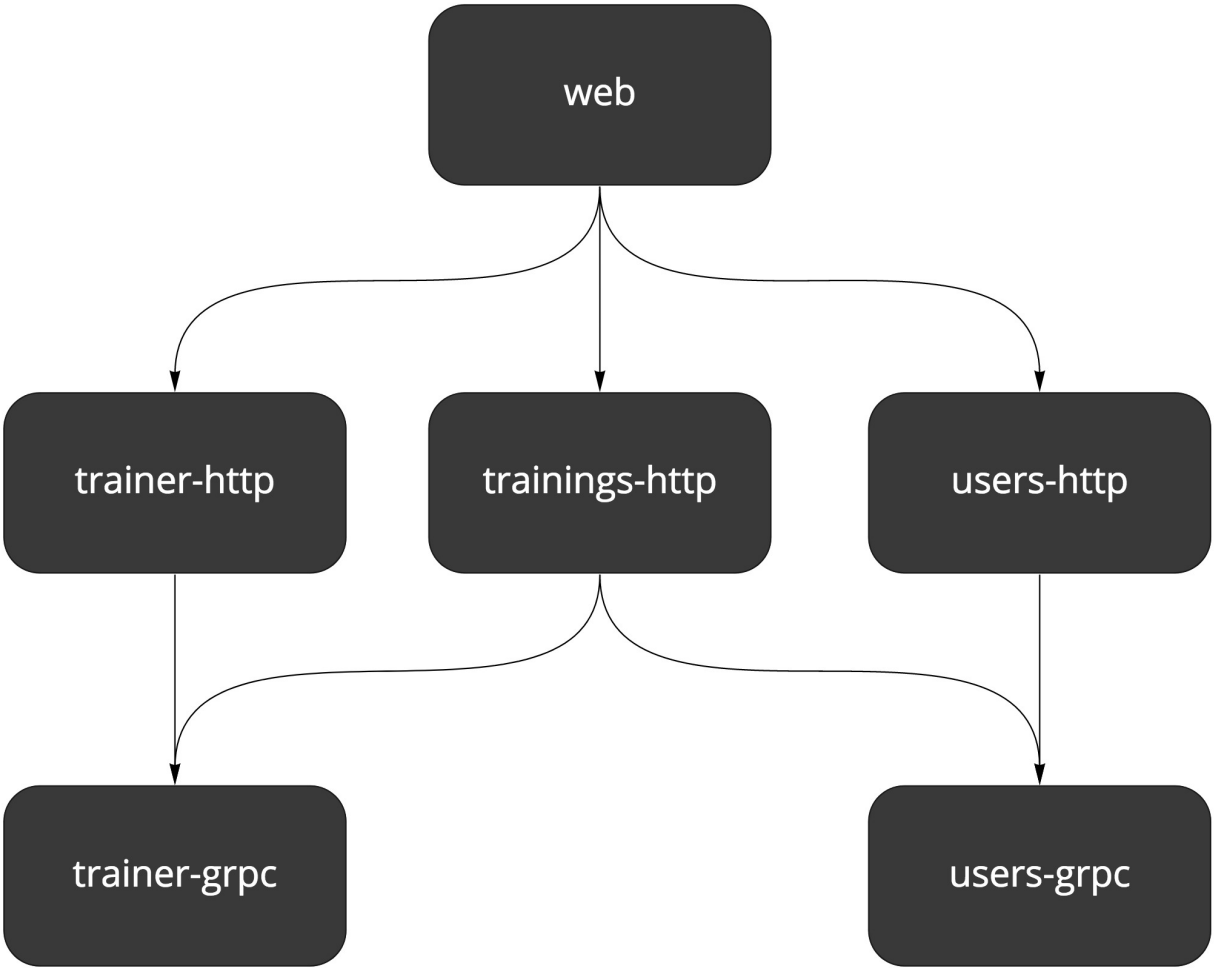


Figure 3.2: Architecture

```
syntax = "proto3";

package trainer;

import "google/protobuf/timestamp.proto";

service TrainerService {
  rpc IsHourAvailable(IsHourAvailableRequest) returns (IsHourAvailableResponse) {}
  rpc UpdateHour(UpdateHourRequest) returns (EmptyResponse) {}
}

message IsHourAvailableRequest {
  google.protobuf.Timestamp time = 1;
}

message IsHourAvailableResponse {
  bool is_available = 1;
}

message UpdateHourRequest {
  google.protobuf.Timestamp time = 1;

  bool has_training_scheduled = 2;
  bool available = 3;
}

message EmptyResponse {}
```

Source: trainer.proto on GitHub[3]

The `.proto` definition is converted into Go code by using *Protocol Buffer Compiler* (protoc).

```
.PHONY: proto
proto:
   protoc --go_out=plugins=grpc:internal/common/genproto/trainer -I api/protobuf api/protobuf/trainer.proto
   protoc --go_out=plugins=grpc:internal/common/genproto/users -I api/protobuf api/protobuf/users.proto
```

Source: Makefile on GitHub[4]

Note

To generate Go code from `.proto` you need to install protoc[a] and protoc Go Plugin[b].
A list of supported types can be found in Protocol Buffers Version 3 Language Specification[c]. More complex built-in types like Timestamp can be found in Well-Known Types list[d].

[a] https://grpc.io/docs/protoc-installation/
[b] https://grpc.io/docs/quickstart/go/
[c] https://developers.google.com/protocol-buffers/docs/reference/proto3-spec#fields
[d] https://developers.google.com/protocol-buffers/docs/reference/google.protobuf

[3] https://bit.ly/3blVhna
[4] https://bit.ly/3aEAsnE

This is how an example generated model looks like:

```
type UpdateHourRequest struct {
   Time                 *timestamp.Timestamp `protobuf:"bytes,1,opt,name=time,proto3" json:"time,omitempty"`
   HasTrainingScheduled bool
 ↪  `protobuf:"varint,2,opt,name=has_training_scheduled,json=hasTrainingScheduled,proto3"
 ↪  json:"has_training_scheduled,omitempty"`
   Available            bool                 `protobuf:"varint,3,opt,name=available,proto3"
 ↪  json:"available,omitempty"`
   XXX_NoUnkeyedLiteral struct{}             `json:"-"`
   // ... more proto garbage ;)
}
```

Source: trainer.pb.go on GitHub[5]

And the server:

```
type TrainerServiceServer interface {
   IsHourAvailable(context.Context, *IsHourAvailableRequest) (*IsHourAvailableResponse, error)
   UpdateHour(context.Context, *UpdateHourRequest) (*EmptyResponse, error)
}
```

Source: trainer.pb.go on GitHub[6]

The difference between HTTP and gRPC is that in gRPC we don't need to take care of what we should return and how to do that. If I would **compare the level of confidence with HTTP and gRPC, it would be like comparing Python and Go**. This way is much more strict, and it's impossible to return or receive any invalid values – **compiler will let us know about that**.

Protobuf also has built-in ability to handle deprecation of fields and handling backward-compatibility[7]. That's pretty helpful in an environment with many independent teams.

Protobuf vs gRPC

Protobuf (Protocol Buffers) is `Interface Definition Language` used by default for defining the service interface and the structure of the payload. Protobuf is also used for serializing these models to binary format.
You can find more details about gRPC and Protobuf on gRPC Concepts[a] page.

[a] https://grpc.io/docs/guides/concepts/

Implementing the server works in almost the same way as in HTTP generated by OpenAPI[8] – we need to implement an interface (`TrainerServiceServer` in that case).

```
type GrpcServer struct {
   db db
}
```

[5] https://bit.ly/37yjVjc
[6] https://bit.ly/3dwBjIN
[7] https://developers.google.com/protocol-buffers/docs/proto3#backwards-compatibility-issues
[8] See Chapter 2: Building a serverless application with Google Cloud Run and Firebase

```
func (g GrpcServer) IsHourAvailable(ctx context.Context, req *trainer.IsHourAvailableRequest)
↪  (*trainer.IsHourAvailableResponse, error) {
   timeToCheck, err := grpcTimestampToTime(req.Time)
   if err != nil {
      return nil, status.Error(codes.InvalidArgument, "unable to parse time")
   }

   model, err := g.db.DateModel(ctx, timeToCheck)
   if err != nil {
      return nil, status.Error(codes.Internal, fmt.Sprintf("unable to get data model: %s", err))
   }

   if hour, found := model.FindHourInDate(timeToCheck); found {
      return &trainer.IsHourAvailableResponse{IsAvailable: hour.Available && !hour.HasTrainingScheduled}, nil
   }

   return &trainer.IsHourAvailableResponse{IsAvailable: false}, nil
}
```

Source: grpc.go on GitHub[9]

As you can see, you cannot return anything else than `IsHourAvailableResponse`, and you can always be sure that you will receive `IsHourAvailableRequest`. In case of an error, you can return one of predefined error codes[10]. They are more up-to-date nowadays than HTTP status codes.

Starting the gRPC server is done in the same way as with HTTP server:

```
server.RunGRPCServer(func(server *grpc.Server) {
    svc := GrpcServer{firebaseDB}
    trainer.RegisterTrainerServiceServer(server, svc)
})
```

Source: main.go on GitHub[11]

Internal gRPC client

After our server is running, it's time to use it. First of all, we need to create a client instance. `trainer.NewTrainerServiceClient` is generated from `.proto`.

```
type TrainerServiceClient interface {
   IsHourAvailable(ctx context.Context, in *IsHourAvailableRequest, opts ...grpc.CallOption)
↪  (*IsHourAvailableResponse, error)
   UpdateHour(ctx context.Context, in *UpdateHourRequest, opts ...grpc.CallOption) (*EmptyResponse, error)
}

type trainerServiceClient struct {
```

[9] https://bit.ly/3pG9SPj
[10] https://godoc.org/google.golang.org/grpc/codes#Code
[11] https://bit.ly/2ZCKnDQ

```
    cc grpc.ClientConnInterface
}

func NewTrainerServiceClient(cc grpc.ClientConnInterface) TrainerServiceClient {
    return &trainerServiceClient{cc}
}
```

Source: trainer.pb.go on GitHub[12]

To make generated client work, we need to pass a couple of extra options. They will allow handling:

- authentication,
- TLS encryption,
- "service discovery" (we use hardcoded names of services provided by Terraform[13] via `TRAINER_GRPC_ADDR` env).

```
import (
    // ...
    "gitlab.com/threedotslabs/wild-workouts/pkg/internal/genproto/trainer"
    // ...
)

func NewTrainerClient() (client trainer.TrainerServiceClient, close func() error, err error) {
    grpcAddr := os.Getenv("TRAINER_GRPC_ADDR")
    if grpcAddr == "" {
        return nil, func() error { return nil }, errors.New("empty env TRAINER_GRPC_ADDR")
    }

    opts, err := grpcDialOpts(grpcAddr)
    if err != nil {
        return nil, func() error { return nil }, err
    }

    conn, err := grpc.Dial(grpcAddr, opts...)
    if err != nil {
        return nil, func() error { return nil }, err
    }

    return trainer.NewTrainerServiceClient(conn), conn.Close, nil
}
```

Source: grpc.go on GitHub[14]

After our client is created we can call any of its methods. In this example, we call `UpdateHour` while creating a training.

```
package main

import (
```

[12] https://bit.ly/3k8V5vo
[13] See Chapter 14: Setting up infrastructure with Terraform
[14] https://bit.ly/2M94vKJ

```
    // ...
    "github.com/pkg/errors"
    "github.com/golang/protobuf/ptypes"
    "gitlab.com/threedotslabs/wild-workouts/pkg/internal/genproto/trainer"
    // ...
)

type HttpServer struct {
    db            db
    trainerClient trainer.TrainerServiceClient
    usersClient   users.UsersServiceClient
}

// ...

func (h HttpServer) CreateTraining(w http.ResponseWriter, r *http.Request) {
    // ...
    timestamp, err := ptypes.TimestampProto(postTraining.Time)
    if err != nil {
        return errors.Wrap(err, "unable to convert time to proto timestamp")
    }
    _, err = h.trainerClient.UpdateHour(ctx, &trainer.UpdateHourRequest{
        Time:                timestamp,
        HasTrainingScheduled: true,
        Available:           false,
    })
    if err != nil {
        return errors.Wrap(err, "unable to update trainer hour")
    }
    // ...
}
```

Source: http.go on GitHub[15]

Cloud Run authentication & TLS

Authentication of the client is handled by Cloud Run out of the box.

The simpler (and recommended by us) way is using Terraform. We describe it in details in **Setting up infrastructure with Terraform** (Chapter 14).

One thing that doesn't work out of the box is sending authentication with the request. Did I already mention that standard gRPC transport is HTTP/2? For that reason, we can just use the old, good JWT for that.

To make it work, we need to implement `google.golang.org/grpc/credentials.PerRPCCredentials` interface. The implementation is based on the official guide from Google Cloud Documentation[16].

[15] https://bit.ly/3aDglGr
[16] https://cloud.google.com/run/docs/authenticating/service-to-service#go

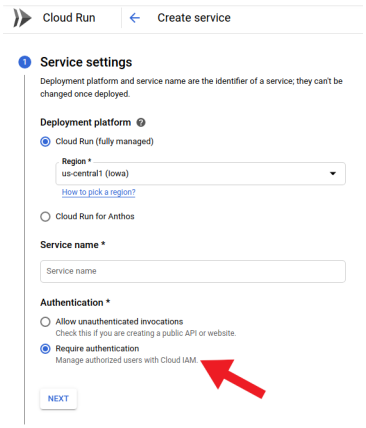


Figure 3.3: You can also enable Authentication from Cloud Run UI. You need to also grant the roles/run.invoker role to service's service account.

```go
type metadataServerToken struct {
   serviceURL string
}

func newMetadataServerToken(grpcAddr string) credentials.PerRPCCredentials {
   // based on https://cloud.google.com/run/docs/authenticating/service-to-service#go
   // service need to have https prefix without port
   serviceURL := "https://" + strings.Split(grpcAddr, ":")[0]

   return metadataServerToken{serviceURL}
}

// GetRequestMetadata is called on every request, so we are sure that token is always not expired
func (t metadataServerToken) GetRequestMetadata(ctx context.Context, in ...string) (map[string]string, error) {
   // based on https://cloud.google.com/run/docs/authenticating/service-to-service#go
   tokenURL := fmt.Sprintf("/instance/service-accounts/default/identity?audience=%s", t.serviceURL)
   idToken, err := metadata.Get(tokenURL)
   if err != nil {
      return nil, errors.Wrap(err, "cannot query id token for gRPC")
   }

   return map[string]string{
      "authorization": "Bearer " + idToken,
   }, nil
}

func (metadataServerToken) RequireTransportSecurity() bool {
   return true
}
```

Source: auth.go on GitHub[17]

[17] https://bit.ly/3k62Y4N

The last thing is passing it to the `[]grpc.DialOption` list passed when creating all gRPC clients.

It's also a good idea to ensure that the our server's certificate is valid with `grpc.WithTransportCredentials`.

Authentication and TLS encryption are disabled on the local Docker environment.

```
func grpcDialOpts(grpcAddr string) ([]grpc.DialOption, error) {
   if noTLS, _ := strconv.ParseBool(os.Getenv("GRPC_NO_TLS")); noTLS {
      return []grpc.DialOption{grpc.WithInsecure()}, nil
   }

   systemRoots, err := x509.SystemCertPool()
   if err != nil {
      return nil, errors.Wrap(err, "cannot load root CA cert")
   }
   creds := credentials.NewTLS(&tls.Config{
      RootCAs: systemRoots,
   })

   return []grpc.DialOption{
      grpc.WithTransportCredentials(creds),
      grpc.WithPerRPCCredentials(newMetadataServerToken(grpcAddr)),
   }, nil
}
```

Source: grpc.go on GitHub[18]

Are all the problems of internal communication solved?

A hammer is great for hammering nails but awful for cutting a tree. The same applies to gRPC or any other technique.

gRPC works great for synchronous communication, but not every process is synchronous by nature. Applying synchronous communication everywhere will end up creating a slow, unstable system. Currently, Wild Workouts doesn't have any flow that should be asynchronous. We will cover this topic deeper in the next chapters by implementing new features. In the meantime, you can check Watermill[19] library, which was also created by us. It helps with building asynchronous, event-driven applications the easy way.

What's next?

Having robust contracts doesn't mean that we are not introducing unnecessary internal communication. In some cases operations can be done in one service in a simpler and more pragmatic way.

It is not simple to avoid these issues. Fortunately, we know techniques that are successfully helping us with that. We will share that with you soon.

[18] https://bit.ly/2ZAMMz2
[19] http://watermill.io/

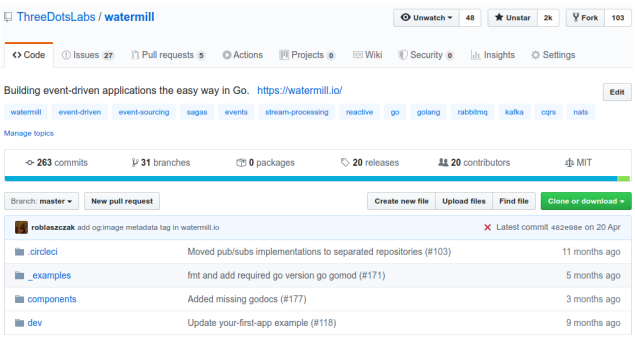


Figure 3.4: Watermill

Chapter 4

Authentication using Firebase

Robert Laszczak

We intentionally built the current version of the application to make it hard to maintain and develop in the future. In a subtle way.
In the next chapter we will start the refactoring process. **We will show you how subtle issues and shortcuts can become an issue in the long term. Even adding a new feature now may not be as easy, as developing greenfield features (even if the application is still pretty new).**

But before that, we have one important topic to cover – authentication. This is the part of the application that will not need refactoring.

You shouldn't reinvent the wheel

(I hope that) you are not creating a programming language, and a framework for every project you work on. Even if you do that, **besides losing a lot of time, it is not harmful. That case is not the same for authentication. Why?**

Let's imagine that you are working in a company implementing a cryptocurrency market. At the beginning of the project, you decided to build your own authentication mechanism. The first reason for that is your boss, who doesn't trust external authentication providers. The second reason is that you believe that it should be simple.

"You did it many times" – this is what strange voices in your head repeat every time.

Have you seen the *Mayday / Air Crash Investigation*[1] documentary TV series? In the Polish edition, every episode starts with more or less: *"Catastrophes are not a matter of coincidence, but series of unfortunate events"*. In programming, this sentence is surprisingly true.

In our hypothetical project, in the beginning, your company decided to introduce a service account that was able

[1] https://en.wikipedia.org/wiki/List_of_Mayday_episodes

to move funds between every account. Even from a wallet that, in theory, should be offline. Of course, temporarily. That doesn't sound like a good idea, but it simplified customer support from the very beginning.

With time everyone forgot about this feature. Until one day, when hackers found a bug in the authentication mechanism, that allowed them to hijack any account in our market. Including the service account.

Our boss and coworkers were less happy than hackers because they lost all the money. The problem would be not so big if we would lose "only" all company's money. In that case, we also lost all our customers' money .

Figure 4.1: Aannnd it's gone

This example may sound a bit extreme and unlikely to happen. Is it a rare scenario? OWASP Top Ten report[2] lists Broken Authentication at **second place**!

2. **Broken Authentication.** Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

()

You may say: *"My company is just selling toasters! Why should I care?"*. I'm pretty sure that you still care about your company's image. **Discreditation after an incident caused by hackers hijacking your customers' accounts is always embarrassing.**

Do you still feel that you can implement a perfect authentication? Even giants with hundreds of researchers and developers working just on authentication are not able to do that. In March 2020, a researcher found a Facebook vulnerability[3] that could allow hijacking anyone's Facebook account. Just a few months later, a bug in Apple Sign In mechanism allowed full account takeover of user accounts on that third party application[4].

[2]https://owasp.org/www-project-top-ten/
[3]https://latesthackingnews.com/2020/03/03/10-year-old-facebook-oauth-framework-flaw-discovered/
[4]https://bhavukjain.com/blog/2020/05/30/zeroday-signin-with-apple/

If you are still not convinced, it's worth to consider saving your time. I'm sure **customers of your company will be much happier about giving them a long-awaited feature. Not fancy, custom authentication** .

Using Firebase authentication

There are, of course, many solutions that can be used for implementing authentication. **In this chapter we want to go with the fastest and easiest way for our context.** We are also not considering any vendor lock-in problems – we will address that in the future .

Note

If you are looking for a simple solution, not dependent on any provider, you should check JWT[a] and `dgrijalva/jwt-go` library[b].

[a] https://jwt.io/
[b] https://github.com/dgrijalva/jwt-go

The Wild Workouts example application that we created for this ebook is now hosted on Firebase hosting. It's a natural choice to use Firebase Authentication[5]. There is one significant advantage of this solution – it works almost totally out of the box both from the backend and frontend side.

Note

Deployment of the project is described in detail in **Setting up infrastructure with Terraform** (Chapter 14). In **Building a serverless application with Google Cloud Run and Firebase** (Chapter 2) you can find deployment tl;dr at the end.
During the setup, don't forget about enabling Email/Password Sign-in provider in *Authentication / Sign-in method* tab in the Firebase Console[a]!

[a] https://console.firebase.google.com/project/_/authentication/providers

Frontend

The first thing that we need to do is the initialization of Firebase SDK[6].

Next, in the form on the main page, we call the `loginUser` function. This function calls `Auth.login`, `Auth.waitForAuthReady` and `Auth.getJwtToken`. The result is set to OpenAPI generated clients by `setApiClientsAuth`.

```
// ...
export function loginUser(login, password) {
    return Auth.login(login, password)
        .then(function () {
            return Auth.waitForAuthReady()
        })
```

[5] https://firebase.google.com/docs/auth
[6] https://firebase.google.com/docs/web/setup#add-sdks-initialize

```
        .then(function () {
            return Auth.getJwtToken(false)
        })
        .then(token => {
            setApiClientsAuth(token)
        })
// ...
```

Source: user.js on GitHub[7]

`Auth` is a class with two implementations: Firebase and mock. We will go through mock implementation later – let's focus on Firebase now.

To log-in we need to call `firebase.auth().signInWithEmailAndPassword`. If everything is fine, `auth().currentUser.getIdTok` returns our JWT token.

```
class FirebaseAuth {
    login(login, password) {
        return firebase.auth().signInWithEmailAndPassword(login, password)
    }

    waitForAuthReady() {
        return new Promise((resolve) => {
            firebase
                .auth()
                .onAuthStateChanged(function () {
                    resolve()
                });
        })
    }

    getJwtToken(required) {
        return new Promise((resolve, reject) => {
            if (!firebase.auth().currentUser) {
                if (required) {
                    reject('no user found')
                } else {
                    resolve(null)
                }
                return
            }

            firebase.auth().currentUser.getIdToken(false)
                .then(function (idToken) {
                    resolve(idToken)
                })
                .catch(function (error) {
                    reject(error)
                });
```

[7] https://bit.ly/3k5ivSq

```
        })
    }
```

Source: auth.js on GitHub[8]

> **Note**
>
> Firebase also provides out of the box support for logging in with most popular OAuth providers like Facebook, Gmail, or GitHub.

Next, we need to set `authentications['bearerAuth'].accessToken` of OpenAPI generated[9] clients attribute to the JWT token received from `Auth.getJwtToken(false)`. Now we need to set this token to OpenAPI clients and voilà! All our requests are now authenticated.

```
export function setApiClientsAuth(idToken) {
    usersClient.authentications['bearerAuth'].accessToken = idToken
    trainerClient.authentications['bearerAuth'].accessToken = idToken
    trainingsClient.authentications['bearerAuth'].accessToken = idToken
}
```

Source: auth.js on GitHub[10]

> **Note**
>
> If you are creating your own OpenAPI spec, it will not work without proper authentication definition[a]. In Wild Workouts' OpenAPI spec[b] it's already done.
>
> [a] https://swagger.io/docs/specification/authentication/bearer-authentication/
> [b] https://bit.ly/3qXbiqh

If you would like to know more, I would recommend you to check Firebase Auth API reference[11].

Backend

The last part is actually using this authentication in our HTTP server. I created a simple HTTP middleware that will do that task for us.

This middleware does three things:

1. Get token from HTTP header
2. Verify token with Firebase auth client
3. Save user data in credentials

```
package auth

import (
```

[8] https://bit.ly/3aFCDYh
[9] See Chapter 2: Building a serverless application with Google Cloud Run and Firebase
[10] https://bit.ly/3drjzyG
[11] https://firebase.google.com/docs/reference/js/firebase.auth

```
    "context"
    "net/http"
    "strings"

    "firebase.google.com/go/auth"
    "github.com/pkg/errors"
    "gitlab.com/threedotslabs/wild-workouts/pkg/internal/server/httperr"
)

type FirebaseHttpMiddleware struct {
    AuthClient *auth.Client
}

func (a FirebaseHttpMiddleware) Middleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ctx := r.Context()

        bearerToken := a.tokenFromHeader(r)
        if bearerToken == "" {
            httperr.Unauthorised("empty-bearer-token", nil, w, r)
            return
        }

        token, err := a.AuthClient.VerifyIDToken(ctx, bearerToken)
        if err != nil {
            httperr.Unauthorised("unable-to-verify-jwt", err, w, r)
            return
        }

        // it's always a good idea to use custom type as context value (in this case ctxKey)
        // because nobody from the outside of the package will be able to override/read this value
        ctx = context.WithValue(ctx, userContextKey, User{
            UUID:        token.UID,
            Email:       token.Claims["email"].(string),
            Role:        token.Claims["role"].(string),
            DisplayName: token.Claims["name"].(string),
        })
        r = r.WithContext(ctx)

        next.ServeHTTP(w, r)
    })
}

type ctxKey int

const (
    userContextKey ctxKey = iota
)

// ...

func UserFromCtx(ctx context.Context) (User, error) {
```

```
    u, ok := ctx.Value(userContextKey).(User)
    if ok {
        return u, nil
    }

    return User{}, NoUserInContextError
}
```

Source: http.go on GitHub[12]

User data can now be accessed in every HTTP request by using `auth.UserFromCtx` function.

```
func (h HttpServer) GetTrainings(w http.ResponseWriter, r *http.Request) {
    user, err := auth.UserFromCtx(r.Context())
    if err != nil {
        httperr.Unauthorised("no-user-found", err, w, r)
        return
    }

     // ...
```

Source: http.go on GitHub[13]

We can also limit access to some resources, based on the user role.

```
func (h HttpServer) MakeHourAvailable(w http.ResponseWriter, r *http.Request) {
    user, err := auth.UserFromCtx(r.Context())
    if err != nil {
        httperr.Unauthorised("no-user-found", err, w, r)
        return
    }

    if user.Role != "trainer" {
        httperr.Unauthorised("invalid-role", nil, w, r)
        return
    }
```

Source: http.go on GitHub[14]

Adding users

In our case, we add users during the start of the `users` service. You can also add them from the FirebaseUI. Unfortunately, via UI you cannot set all required data, like claims – you need to do it via API.

```
config := &firebase.Config{ProjectID: os.Getenv("GCP_PROJECT")}
firebaseApp, err := firebase.NewApp(context.Background(), config, opts...)
if err != nil {
    return nil, err
}
```

[12] https://bit.ly/2NoUda0
[13] https://bit.ly/2OOyhFy
[14] https://bit.ly/3k6hPMq

```go
authClient, err := firebaseApp.Auth(context.Background())
if err != nil {
    return nil, err
}

// ...

for _, user := range usersToCreate {
    userToCreate := (&auth.UserToCreate{}).
        Email(user.Email).
        Password("123456").
        DisplayName(user.DisplayName)

    createdUser, err := authClient.CreateUser(context.Background(), userToCreate)

    // ...

    err = authClient.SetCustomUserClaims(context.Background(), createdUser.UID, map[string]interface{}{
        "role": user.Role,
    })

    // ...
```

Source: fixtures.go on GitHub[15]

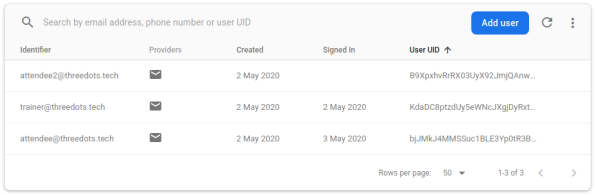


Figure 4.2: Firestore Console

Mock Authentication for local dev

There is high demand and a discission going around[16] support for Firebase Authentication emulator. Unfortunately, it doesn't exist yet. The situation is pretty similar to Firestore here – I want to be able to run my application locally without any external dependencies. As long as there is no emulator, there is no other way than to implement a simple mock implementation.

There is nothing really complicated in both backend and frontend implementation. Frontend's[17] `getJwtToken` is implemented by just generating JWT token with `mock secret`. Backend[18] instead of calling Firebase to verify the token is checking if JWT was generated with `mock secret`.

[15] https://bit.ly/3siayw5
[16] https://github.com/firebase/firebase-tools/issues/1677
[17] https://bit.ly/3pHyEi8
[18] https://bit.ly/3shJcGf

It gives us some confidence if our flow is implemented more or less correctly. But is there any option to test it with production Firebase locally?

Firebase Authentication for local dev

Mock authentication does not give us 100% confidence that the flow is working correctly. We should be able to test Firebase Authentication locally. To do that, you need to do some extra steps.

It is not straightforward. If you are not changing anything in the authentication, you can probably skip this part.

First of all, you need to generate a service account file into the repository.

> **Note**
>
> Remember to not share it with anyone! Don't worry, it's already in `.gitignore`.

```
gcloud auth login

gcloud iam service-accounts keys create service-account-file.json --project [YOUR PROJECT ID] --iam-account [YOUR
↪  PROJECT ID]@appspot.gserviceaccount.com
```

Next, you need to uncomment **all** lines linking service account in `docker-compose.yml`

```
     volumes:
       - ./pkg:/pkg
-#      - ./service-account-file.json:$SERVICE_ACCOUNT_FILE
+      - ./service-account-file.json:$SERVICE_ACCOUNT_FILE
     working_dir: /pkg/trainer
     ports:
       - "127.0.0.1:3000:$PORT"

       context: ./dev/docker/app
     volumes:
       - ./pkg:/pkg
-#      - ./service-account-file.json:$SERVICE_ACCOUNT_FILE
+      - ./service-account-file.json:$SERVICE_ACCOUNT_FILE
     working_dir: /pkg/trainer

@ ... do it for all services!
```

Source: docker-compose.yml on GitHub[19]

After that, you should set `GCP_PROJECT` to your project id, uncomment `SERIVCE_ACCOUNT_FILE` and set `MOCK_AUTH` to `false` in `.env`

```
-GCP_PROJECT=threedotslabs-cloudnative
+GCP_PROJECT=YOUR-PROJECT-ID

 PORT=3000
```

[19] https://bit.ly/37zdoVl

```
-FIRESTORE_PROJECT_ID=threedotslabs-cloudnative
+FIRESTORE_PROJECT_ID=YOUR-PROJECT-ID
 FIRESTORE_EMULATOR_HOST=firestore:8787
 # env used by karhoo/firestore-emulator container
-GCP_PROJECT_ID=threedotslabs-cloudnative
+GCP_PROJECT_ID=YOUR-PROJECT-ID

 TRAINER_GRPC_ADDR=trainer-grpc:3000
 USERS_GRPC_ADDR=users-grpc:3000

@ ...

 CORS_ALLOWED_ORIGINS=http://localhost:8080

-#SERVICE_ACCOUNT_FILE=/service-account-file.json
-MOCK_AUTH=true
+SERVICE_ACCOUNT_FILE=/service-account-file.json
+MOCK_AUTH=false
 LOCAL_ENV=true
```

Source: .env on GitHub[20]

```
-const MOCK_AUTH = process.env.NODE_ENV === 'development'
+const MOCK_AUTH = false
```

Source: auth.js on GitHub[21]

Now you need to go to Project Settings[22] (you can find it after clicking the gear icon right to *Project overview*) and update your Firebase settings in `web/public/__/firebase/init.json`.

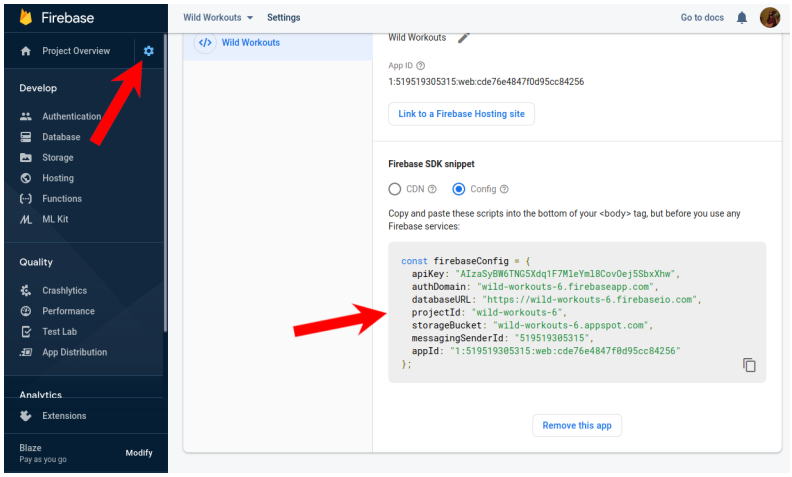


Figure 4.3: Firestore Console

[20] https://bit.ly/37PNRrp
[21] https://bit.ly/3k9K0uc
[22] https://console.firebase.google.com/project/_/settings/general

```
-  "apiKey": "",
-  "appId": "",
-  "authDomain": "",
-  "databaseURL": "",
-  "messagingSenderId": "",
-  "projectId": "",
-  "storageBucket": ""
+  "apiKey": "AIzaSyBW6TNG5Xdq1F7MleYml8CovOej5SbxXhw",
+  "authDomain": "wild-workouts-6.firebaseapp.com",
+  "databaseURL": "https://wild-workouts-6.firebaseio.com",
+  "projectId": "wild-workouts-6",
+  "storageBucket": "wild-workouts-6.appspot.com",
+  "messagingSenderId": "519519305315",
+  "appId": "1:519519305315cde76e4847f0d95cc84256"
 }
```

Source: init.json on GitHub[23]

The last thing is stopping and starting again docker-compose to reload envs and links.

And that's all

As you can see, the setup was really straightforward. We were able to save a lot of our time. **The situation could be more complicated if we wouldn't like to host our application on Firebase.** Our idea from the beginning was to prepare a simple base for future chapters.

We didn't exhaust the topic of authentication in this chapter. But at least for now, we don't plan to explore authentication deeper.

In this book we would like to show you how to build applications that are easy to develop, maintain, and fun to work with in the long term. Maybe you've seen examples of Domain-Driven Design or Clean Architecture in Go. Most of them are not done in a really pragmatic way that works in the language context.

In following chapters, we will show patterns that we successfully use in teams that we lead for a couple years. We will also show when applying them makes sense, and when it's over-engineering or CV-Driven Development. I'm sure that it will force you to rethink your point of view for these techniques.

[23] https://bit.ly/2NLBghq

Chapter 5

When to stay away from DRY

Miłosz Smółka

In this chapter, we begin refactoring of Wild Workouts. Previous chapters will give you more context, but reading them isn't necessary to understand this one.

Background story

Meet Susan, a software engineer, bored at her current job working with legacy enterprise software. Susan started looking for a new gig and found Wild Workouts startup that uses serverless Go microservices. This seemed like something fresh and modern, so after a smooth hiring process, she started her first day at the company.

There were just a few engineers in the team, so Susan's onboarding was blazing fast. Right on the first day, she was assigned her first task that was supposed to make her familiar with the application.

> We need to store each user's last IP address. It will enable new security features in the future, like extra confirmation when logging in from a new location. For now, we just want to keep it in the database.

Susan looked through the application for some time. She tried to understand what's happening in each service and where to add a new field to store the IP address. Finally, she discovered a `User`[1] structure that she could extend.

```
 // User defines model for User.
 type User struct {
        Balance     int     `json:"balance"`
        DisplayName string  `json:"displayName"`
        Role        string  `json:"role"`
+       LastIp      string  `json:"lastIp"`
 }
```

It didn't take Susan long to post her first pull request. Shortly, Dave, a senior engineer, added a comment during code review.

[1] https://bit.ly/2NUFZha

I don't think we're supposed to expose this field via the REST API.

Susan was surprised, as she was sure she updated the database model. Confused, she asked Dave if this is the correct place to add a new field.

Dave explained that Firestore, the application's database, stores documents marshaled from Go structures. So the `User` struct is compatible with both frontend responses and storage.

"*Thanks to this approach, you don't need to duplicate the code. It's enough to change the YAML definition*[2] *once and regenerate the file*", he said enthusiastically.

Grateful for the tip, Susan added one more change to hide the new field from the API response.

```
diff --git a/internal/users/http.go b/internal/users/http.go
index 9022e5d..cd8fbdc 100644
--- a/internal/users/http.go
+++ b/internal/users/http.go
@@ -27,5 +30,8 @@ func (h HttpServer) GetCurrentUser(w http.ResponseWriter, r *http.Request) {
        user.Role = authUser.Role
        user.DisplayName = authUser.DisplayName

+       // Don't expose the user's last IP externally
+       user.LastIp = nil
+
        render.Respond(w, r, user)
 }
```

One line was enough to fix this issue. That was quite a productive first day for Susan.

Second thoughts

Even though Susan's solution was approved and merged, something bothered her on her commute home. **Is it the right approach to keep the same struct for both API response and database model?** Don't we risk accidentally exposing user's private details if we keep extending it as the application grows? What if we'd like to change only the API response without changing the database fields?

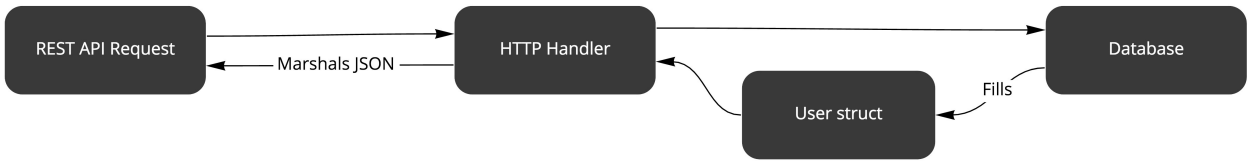


Figure 5.1: Original solution

What about splitting the two structures? That's what Susan would do at her old job, but maybe these were

[2] https://bit.ly/2NKI4Mp

enterprise patterns that should not be used in a Go microservice. Also, the team seemed very rigorous about the **Don't Repeat Yourself** principle.

Refactoring

The next day, Susan explained her doubts to Dave and asked for his opinion. At first, he didn't understand the concern and mentioned that maybe she needs to get used to the "Go way" of doing things.

Susan pointed to another piece of code in Wild Workouts that used a similar ad-hoc solution. She shared that, from her experience, such code can quickly get out of control.

```
    user, err := h.db.GetUser(r.Context(), authUser.UUID)
    if err != nil {
        httperr.InternalError("cannot-get-user", err, w, r)
        return
    }
    user.Role = authUser.Role
    user.DisplayName = authUser.DisplayName

    render.Respond(w, r, user)
```

It seems the HTTP handler modifies the user in-place.

Eventually, they agreed to discuss this again over a new PR. Shortly, Susan prepared a refactoring proposal.

```
diff --git a/internal/users/firestore.go b/internal/users/firestore.go
index 7f3fca0..670bfaa 100644
--- a/internal/users/firestore.go
+++ b/internal/users/firestore.go
@@ -9,6 +9,13 @@ import (
        "google.golang.org/grpc/status"
 )

+type UserModel struct {
+       Balance     int
+       DisplayName string
+       Role        string
+       LastIP      string
+}
+
diff --git a/internal/users/http.go b/internal/users/http.go
index 9022e5d..372b5ca 100644
--- a/internal/users/http.go
+++ b/internal/users/http.go
@@ -1,6 +1,7 @@
-       user.Role = authUser.Role
-       user.DisplayName = authUser.DisplayName

-       render.Respond(w, r, user)
+       userResponse := User{
+               DisplayName: authUser.DisplayName,
+               Balance:     user.Balance,
+               Role:        authUser.Role,
```

```
+       }
+
+       render.Respond(w, r, userResponse)
 }
```

Source: 14d9e7badcf5a91811059d377cfa847ec7b4592f on GitHub[3]

This time, Susan didn't touch the OpenAPI definition. After all, she wasn't supposed to introduce any changes to the REST API. Instead, she manually created another structure, just like `User`, but exclusive to the database model. Then, she extended it with a new field.

The new solution is a bit longer in code lines, but it removed code coupling between the REST API and the database layer (all without introducing another microservice). The next time someone wants to add a field, they can do it by updating the proper structure.

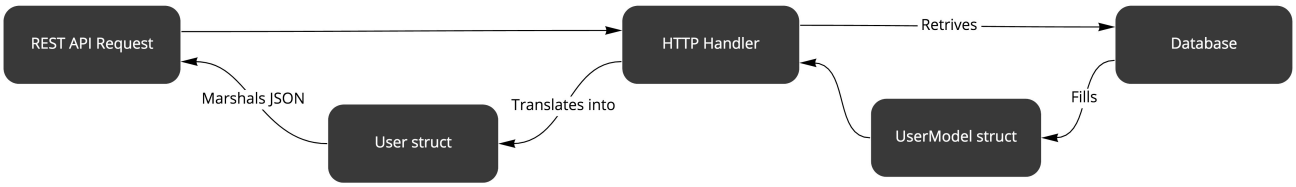


Figure 5.2: Refactored solution

The Clash of Principles

Dave's biggest concern was that the second solution breaks the DRY[4] principle and introduces boilerplate. On the other hand, Susan was afraid that the original approach violates the **Single Responsibility Principle** (the "S" in SOLID[5]). Who's right?

It's tough to come up with strict rules. Sometimes code duplication seems like a boilerplate, but it's one of the best tools to fight code coupling. It's helpful to ask yourself if the code using the common structure is likely to change together. If not, it's safe to assume duplication is the right choice.

Usually, DRY is better applied to behaviors, not data. For example, extracting common code to a separate function doesn't have the downsides we discussed so far.

What's the big deal?

Is such a minor change even "architecture"?

Susan introduced a little change that had consequences she didn't know about. It was obvious to other engineers, but not for a newcomer. I guess you also know **the feeling of being afraid to introduce changes in an unknown system because you can't know what it could trigger**.

[3] https://bit.ly/3aB4JUi
[4] https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
[5] https://en.wikipedia.org/wiki/SOLID

If you make many wrong decisions, even small, they tend to compound. Eventually, developers start to complain that it's hard to work with the application. The turning point is when someone mentions a "rewrite", and suddenly, you know you have a big problem.

> The alternative to good design is always bad design. There is no such thing as no design.
>
> Adam Judge ()

It's worth discussing architecture decisions before you stumble on issues. "No architecture" will just leave you with bad architecture.

Can microservices save you?

With all benefits that microservices gave us, a dangerous idea also appeared, preached by some "how-to build microservices" guides. **It says that microservices will simplify your application**. Because building large software projects is hard, some promise that you won't need to worry about it if you split your application into tiny chunks.

This idea sounds good on paper, but it misses the point of splitting software. How do you know where to put the boundaries? Will you just separate a service based on each database entity? REST endpoints? Features? **How do you ensure low coupling between services?**

The Distributed Monolith

If you start with poorly separated services, you're likely to end up with the same monolith you tried to avoid, with added network overhead and complex tooling to manage this mess (also known as a **distributed monolith**). You will just replace highly coupled modules with highly coupled services. And because now everyone runs a Kubernetes cluster, you may even think you're following the industry standards.

Even if you can rewrite a single service in one afternoon, can you as quickly change how services communicate with each other? What if they are owned by multiple teams, based on wrong boundaries? Consider how much simpler it is to refactor a single application.

Everything above doesn't void other benefits of using microservices, like independent deployments (crucial for Continuous Delivery) and easier horizontal scaling. Like with all patterns, make sure you use the right tool for the job, at the right time.

Note

We introduced similar issues on purpose in Wild Workouts. We will look into this in future chapters and discuss other splitting techniques.
You can find some of the ideas in Robert's post: *Why using Microservices or Monolith can be just a detail?*[a].

[a] https://threedots.tech/post/microservices-or-monolith-its-detail/

Does this all apply to Go?

Open-source Go projects are usually low-level applications and infrastructure tools. This was especially true in the early days, but it's still hard to find good examples of Go applications that handle **domain logic**.

By "domain logic", I don't mean financial applications or complex business software. **If you develop any kind of web application, there's a solid chance you have some complex domain cases you need to model somehow**.

Following some DDD examples can make you feel like you're no longer writing Go. I'm well aware that forcing OOP patterns straight from Java is not fun to work with. However, **there are some language-agnostic ideas that I think are worth considering.**

What's the alternative?

We spent the last few years exploring this topic. We really like the simplicity of Go, but also had success with ideas from Domain-Driven Design and Clean Architecture[6].

For some reason, developers didn't stop talking about technical debt and legacy software with the arrival of microservices. Instead of looking for the silver bullet, we prefer to use business-oriented patterns together with microservices in a pragmatic way.

We're not done with the refactoring of Wild Workouts yet. In **Clean Architecture** (Chapter 9), we will see how to introduce Clean Architecture to the project.

[6]https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Chapter 6

Domain-Driven Design Lite

Robert Laszczak

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 all looked harmless in 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?

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 chapter, 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.**

In this book, 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. 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 only in the short term.**

Let's consider the example of tests of any kind. 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 do not create 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.

That's great, but do you have any evidence it works?

If you asked me this question two years ago, I would say: *"Well, I feel that it works better!"*. But just trusting my words may be not enough.

There are many tutorials showing some dumb ideas and claiming that they work without any evidence – let's don't trust them blindly!

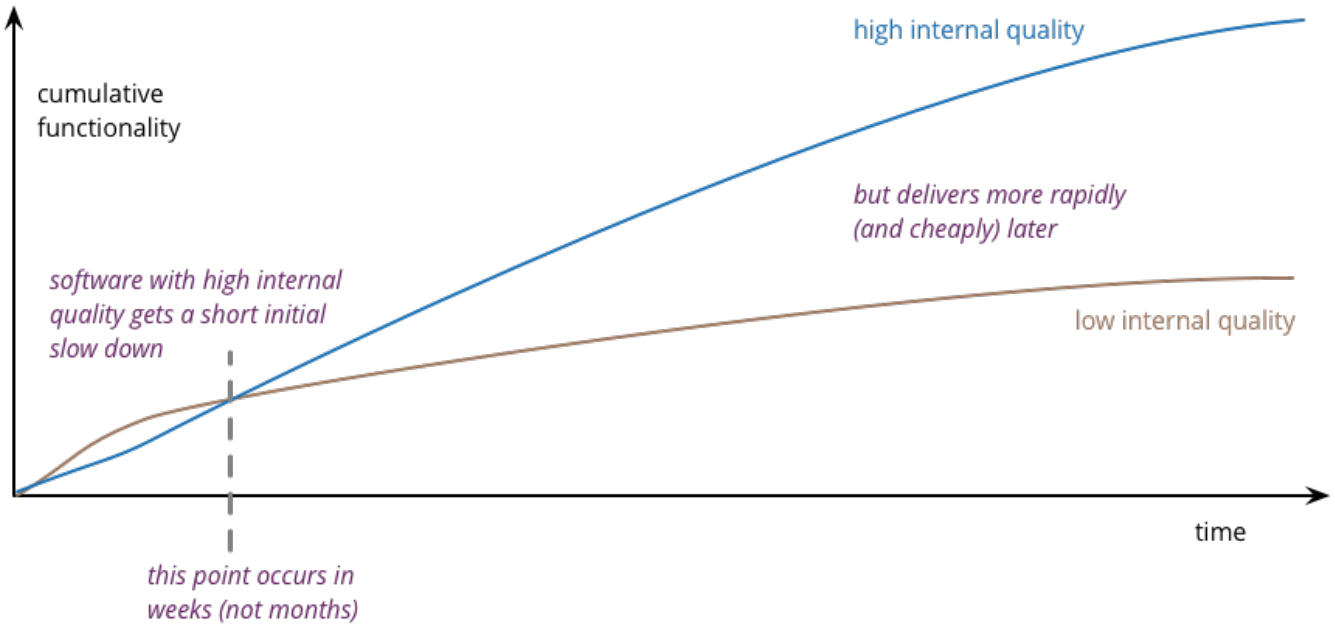


Figure 6.1: 'Is High Quality Software Worth the Cost?' from martinfowler.com

Just to remind: if someone has a couple thousand Twitter followers, that alone is not a reason to trust them[1]!

Fortunately, 2 years ago *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*[2] 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.

Note

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

[a] https://www.amazon.com/Phoenix-Project-DevOps-Helping-Business/dp/0988262592
[b] https://www.amazon.com/DevOps-Handbook-World-Class-Reliability-Organizations/dp/1942788002

What Accelerate tells us about top-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.

[1] https://en.wikipedia.org/wiki/Authority_bias
[2] https://www.amazon.com/Accelerate-Software-Performing-Technology-Organizations/dp/1942788339

> 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 chapter 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 adds extra complexity and slows 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.

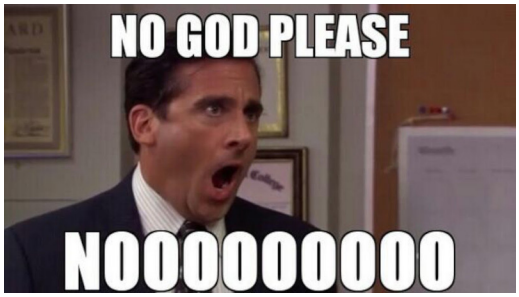


Figure 6.2: No, God please no!

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.

> **Note**
>
> If you are interested in some historical context on when DDD was created, you should check Tackling Complexity in the Heart of Software[a] by the DDD creator - Eric Evans
>
> [a] https://youtu.be/dnUFEg68ESM?t=1109

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 should ensure that you are solving a valid problem. It may sound obvious, but in practice from my 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 is 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 chapters.

DDD Lite in Go

After a pretty long introduction, it's finally time to touch some code! In this chapter, 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 chapters 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.

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. Even if it is not a lot of logic, some parts of the code started to be messy. I have also some feeling based on experience, that with time it will get worse.

```
func (g GrpcServer) UpdateHour(ctx context.Context, req *trainer.UpdateHourRequest) (*trainer.EmptyResponse, error) {
   trainingTime, err := grpcTimestampToTime(req.Time)
   if err != nil {
      return nil, status.Error(codes.InvalidArgument, "unable to parse time")
   }

   date, err := g.db.DateModel(ctx, trainingTime)
   if err != nil {
      return nil, status.Error(codes.Internal, fmt.Sprintf("unable to get data model: %s", err))
   }

   hour, found := date.FindHourInDate(trainingTime)
   if !found {
      return nil, status.Error(codes.NotFound, fmt.Sprintf("%s hour not found in schedule", trainingTime))
   }

   if req.HasTrainingScheduled && !hour.Available {
      return nil, status.Error(codes.FailedPrecondition, "hour is not available for training")
   }

   if req.Available && req.HasTrainingScheduled {
      return nil, status.Error(codes.FailedPrecondition, "cannot set hour as available when it have training
       ↪  scheduled")
   }
   if !req.Available && !req.HasTrainingScheduled {
      return nil, status.Error(codes.FailedPrecondition, "cannot set hour as unavailable when it have no training
       ↪  scheduled")
   }
   hour.Available = req.Available

   if hour.HasTrainingScheduled && hour.HasTrainingScheduled == req.HasTrainingScheduled {
      return nil, status.Error(codes.FailedPrecondition, fmt.Sprintf("hour HasTrainingScheduled is already %t",
       ↪  hour.HasTrainingScheduled))
   }

   hour.HasTrainingScheduled = req.HasTrainingScheduled
```

Source: grpc.go on GitHub[3]

Even if it's not the worst code ever, it reminds me what I've seen when I checked the git history of 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
}
```

Source: availability.go on GitHub[4]

One question that can help us with implementation is: *"Will business understand my 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(validTrainingHour())
   require.NoError(t, err)

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

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

[3] https://bit.ly/3qR2nWY
[4] https://bit.ly/3aBnS8A

```
func TestHour_ScheduleTraining_with_not_available(t *testing.T) {
   h := newNotAvailableHour(t)
   assert.Equal(t, hour.ErrHourNotAvailable, h.ScheduleTraining())
}
```

Source: availability_test.go on GitHub[5]

Now, if anybody will ask the question "When I can schedule training", you can quickly answer that. In a bigger systems, the answer to this kind of 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.

Testing Helpers

It's useful to have some helpers in tests for creating our domain entities. For example: `newExampleTrainingWithTime`, `newCanceledTraining` etc. It also makes our domain tests much more readable.

Custom asserts, like `assertTrainingsEquals` can also save a lot of duplication. github.com/google/go-cmp library is extremely useful for comparing complex structs. It allows us to compare our domain types with private field, skip some field validation[6] or implement custom validation functions[7].

```
func assertTrainingsEquals(t *testing.T, tr1, tr2 *training.Training) {
    cmpOpts := []cmp.Option{
        cmpRoundTimeOpt,
        cmp.AllowUnexported(
            training.UserType{},
            time.Time{},
            training.Training{},
        ),
    }

    assert.True(
        t,
        cmp.Equal(tr1, tr2, cmpOpts...),
        cmp.Diff(tr1, tr2, cmpOpts...),
    )
}
```

It's also a good idea to provide a `Must` version of constructors used often, so for example `MustNewUser`. In contrast with normal constructors, they will panic if parameters are not valid (for tests it's not a problem).

```
func NewUser(userUUID string, userType UserType) (User, error) {
    if userUUID == "" {
        return User{}, errors.New("missing user UUID")
    }
    if userType.IsZero() {
        return User{}, errors.New("missing user type")
    }
```

[5] https://bit.ly/3bofN6D
[6] https://godoc.org/github.com/google/go-cmp/cmp/cmpopts#IgnoreFields
[7] https://godoc.org/github.com/google/go-cmp/cmp#Comparer

```
    return User{userUUID: userUUID, userType: userType}, nil
}

func MustNewUser(userUUID string, userType UserType) User {
    u, err := NewUser(userUUID, userType)
    if err != nil {
        panic(err)
    }

    return u
}
```

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 (https://ruggedsoftware.org/)

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 need to also put our types to separate package and make all attributes private.

```
type Hour struct {
   hour time.Time

   availability Availability
}

// ...

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

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

Source: hour.go on GitHub[8]

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 training")
}
```

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 "magic" like annotations, ORM's or any database solutions are affecting in even more significant way

[8] https://bit.ly/3se9o4I

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 chapter long, let's just introduce the Repository interface and assume that it works.
I will cover this topic more in-depth in the next chapter.

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

Source: repository.go on GitHub[9]

Note

You may ask why `UpdateHour` has `updateFn func(h *Hour) (*Hour, error)` – we will use that for handling transactions nicely. You can learn more in **The Repository Pattern** (Chapter 7).

Using domain objects

I did a small refactor of our gRPC endpoints, to provide an API that is more "behavior-oriented" rather than CRUD[10]. 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/timestamp.proto";

 service TrainerService {
   rpc IsHourAvailable(IsHourAvailableRequest) returns (IsHourAvailableResponse) {}
-  rpc UpdateHour(UpdateHourRequest) returns (EmptyResponse) {}
+  rpc ScheduleTraining(UpdateHourRequest) returns (EmptyResponse) {}
+  rpc CancelTraining(UpdateHourRequest) returns (EmptyResponse) {}
```

[9] https://bit.ly/2NP1NuA
[10] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

```
+  rpc MakeHourAvailable(UpdateHourRequest) returns (EmptyResponse) {}
 }

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

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

 message EmptyResponse {}
```

Source: 0249977c58a310d343ca2237c201b9ba016b148e on GitHub[11]

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 context.Context, request *trainer.UpdateHourRequest)
 ↪  (*trainer.EmptyResponse, error) {
   trainingTime, err := protoTimestampToTime(request.Time)
   if err != nil {
      return nil, status.Error(codes.InvalidArgument, "unable to parse time")
   }

   if err := g.hourRepository.UpdateHour(ctx, trainingTime, func(h *hour.Hour) (*hour.Hour, error) {
      if err := h.MakeAvailable(); err != nil {
         return nil, err
      }

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

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

Source: grpc.go on GitHub[12]

No more Eight-thousanders

As I remember from the old-times, many Eight-thousanders were actually controllers with a lot of domain logic in HTTP controllers.
With hiding complexity inside of our domain types and keeping rules that I described, we can prevent uncontrolled growth in this place.

[11] https://bit.ly/3pFL4H0
[12] https://bit.ly/3pD36Kd

That's all for today

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

If you can't wait, the entire working diff for the refactor is available on GitHub[13]. In the next chapter, 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 to the time when I connected all the dots (It was before I heard about Go). After that, I've seen why all techniques that we will describe in the next chapters 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 chapters 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 share 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 unmaintainable code!

[13] https://bit.ly/3umRYES

Chapter 7

The Repository Pattern

Robert Laszczak

I've seen a lot of complicated code in my life. Pretty often, the reason of that complexity was application logic coupled with database logic. **Keeping logic of your application along with your database logic makes your application much more complex, hard to test, and maintain**.

There is already a proven and simple pattern that solves these issues. The pattern that allows you to **separate your application logic from database logic**. It allows you to **make your code simpler and easier to add new functionalities**. As a bonus, you can **defer important decision** of choosing database solution and schema. Another good side effect of this approach is out of the box **immunity for database vendor lock-in**. The pattern that I have in mind is *repository*.

When I'm going back in my memories to the applications I worked with, I remember that it was tough to understand how they worked. **I was always afraid to make any change there – you never know what unexpected, bad side effects it could have.** It's hard to understand the application logic when it's mixed with database implementation. It's also a source of duplication.

Some rescue here may be building end-to-end tests[1]. But it hides the problem instead of really solving it. Having a lot of E2E tests is slow, flaky, and hard to maintain. Sometimes they even prevent us from creating new functionality, rather than help.

In this chapter, I will teach you how to apply this pattern in **Go** in a pragmatic, elegant, and straightforward way. I will also deeply cover a topic that is often skipped - **clean transactions handling**.

To prove that I prepared 3 implementations: **Firestore, MySQL, and simple in-memory**.

Without too long introduction, let's jump to the practical examples!

[1] https://martinfowler.com/articles/microservice-testing/#testing-end-to-end-introduction

Repository interface

The idea of using the repository pattern is:

Let's abstract our database implementation by defining interaction with it by the interface. You need to be able to use this interface for any database implementation – that means that it should be free of any implementation details of any database.

Let's start with the refactoring of trainer[2] service. Currently, the service allows us to get information about hour availability via HTTP API[3] and via gRPC[4]. We can also change the availability of the hour via HTTP API[5] and gRPC[6].

In the previous chapter, we refactored `Hour` to use DDD Lite approach. Thanks to that, we don't need to think about keeping rules of when `Hour` can be updated. Our domain layer ensures that we can't do anything "stupid". We also don't need to think about any validation. We can just use the type and execute necessary operations.

We need to be able to get the current state of `Hour` from the database and save it. Also, in case when two people would like to schedule a training simultaneously, only one person should be able to schedule training for one hour.

Let's reflect our needs in the interface:

```
package hour

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

Source: repository.go on GitHub[7]

We will use `GetOrCreateHour` to get the data, and `UpdateHour` to save the data.

We define the interface in the same package as the `Hour` type. Thanks to that, we can avoid duplication if using this interface in many modules (from my experience, it may often be the case). It's also a similar pattern to `io.Writer`, where `io` package defines the interface, and all implementations are decupled in separate packages.

How to implement that interface?

[2] https://bit.ly/2NyKVbr
[3] https://bit.ly/2OXsgGB
[4] https://bit.ly/2OXsgGB
[5] https://bit.ly/3unIQzJ
[6] https://bit.ly/3unIQzJ
[7] https://bit.ly/3bo6QtT

Reading the data

Most database drivers can use the `ctx context.Context` for cancellation, tracing, etc. It's not specific to any database (it's a common Go concept), so you should not be afraid that you spoil the domain.

In most cases, we query data by using UUID or ID, rather than `time.Time`. In our case, it's okay – the hour is unique by design. I can imagine a situation that we would like to support multiple trainers – in this case, this assumption will not be valid. Change to UUID/ID would still be simple. But for now, YAGNI[8]!

For clarity – this is how the interface based on UUID may look like:

```
GetOrCreateHour(ctx context.Context, hourUUID string) (*Hour, error)
```

> **Note**
>
> You can find an example of a repository based on UUID in **Combining DDD, CQRS, and Clean Architecture** (Chapter 11).

How is the interface used in the application?

```
import (
    // ...
   "github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/domain/hour"
    // ...
)

type GrpcServer struct {
   hourRepository hour.Repository
}

// ...

func (g GrpcServer) IsHourAvailable(ctx context.Context, request *trainer.IsHourAvailableRequest)
 ↪  (*trainer.IsHourAvailableResponse, error) {
   trainingTime, err := protoTimestampToTime(request.Time)
   if err != nil {
      return nil, status.Error(codes.InvalidArgument, "unable to parse time")
   }

   h, err := g.hourRepository.GetOrCreateHour(ctx, trainingTime)
   if err != nil {
      return nil, status.Error(codes.Internal, err.Error())
   }

   return &trainer.IsHourAvailableResponse{IsAvailable: h.IsAvailable()}, nil
}
```

Source: grpc.go on GitHub[9]

[8] https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it
[9] https://bit.ly/2ZCdmaR

No rocket science! We get `hour.Hour` and check if it's available. Can you guess what database we use? No, and that is the point!

As I mentioned, we can avoid vendor lock-in and be able to easily swap the database. If you can swap the database, **it's a sign that you implemented the repository pattern correctly**. In practice, the situation when you change the database is rare. In case when you are using a solution that is not self-hosted (like Firestore), it's more important to mitigate the risk and avoid vendor lock-in.

The helpful side effect of that is that we can defer the decision of which database implementation we would like to use. I call this approach *Domain First*. I described it in depth in **Domain-Driven Design Lite** (Chapter 6). **Deferring the decision about the database for later can save some time at the beginning of the project. With more informations and context, we can also make a better decision.**

When we use the Domain-First approach, the first and simplest repository implementation may be in-memory implementation.

Example In-memory implementation

Our memory uses a simple map under the hood. `getOrCreateHour` has 5 lines (without a comment and one newline)!

```
type MemoryHourRepository struct {
   hours map[time.Time]hour.Hour
   lock  *sync.RWMutex

   hourFactory hour.Factory
}

func NewMemoryHourRepository(hourFactory hour.Factory) *MemoryHourRepository {
   if hourFactory.IsZero() {
      panic("missing hourFactory")
   }

   return &MemoryHourRepository{
      hours:       map[time.Time]hour.Hour{},
      lock:        &sync.RWMutex{},
      hourFactory: hourFactory,
   }
}

func (m MemoryHourRepository) GetOrCreateHour(_ context.Context, hourTime time.Time) (*hour.Hour, error) {
   m.lock.RLock()
   defer m.lock.RUnlock()

   return m.getOrCreateHour(hourTime)
}

func (m MemoryHourRepository) getOrCreateHour(hourTime time.Time) (*hour.Hour, error) {
   currentHour, ok := m.hours[hourTime]
   if !ok {
      return m.hourFactory.NewNotAvailableHour(hourTime)
```

```
    }

    // we don't store hours as pointers, but as values
    // thanks to that, we are sure that nobody can modify Hour without using UpdateHour
    return &currentHour, nil
}
```

Source: hour_memory_repository.go on GitHub[10]

Unfortunately, the memory implementation has some downsides. The biggest one is that it doesn't keep the data of the service after a restart. It can be enough for the functional pre-alpha version. To make our application production-ready, we need to have something a bit more persistent.

Example MySQL implementation

We already know how our model looks like[11] and how it behaves[12]. Based on that, let's define our SQL schema.

```
CREATE TABLE `hours`
(
    hour         TIMESTAMP                                                  NOT NULL,
    availability ENUM ('available', 'not_available', 'training_scheduled') NOT NULL,
    PRIMARY KEY (hour)
);
```

Source: schema.sql on GitHub[13]

When I work with SQL databases, my default choices are:

- sqlx[14] – for simpler data models, it provides useful functions that help with using structs to unmarshal query results. When the schema is more complex because of relations and multiple models, it's time for…
- SQLBoiler[15] - is excellent for more complex models with many fields and relations, it's based on code generation. Thanks to that, it's very fast, and you don't need to be afraid that you passed invalid `interface{}` instead of another `interface{}`. Generated code is based on the SQL schema, so you can avoid a lot of duplication.

We currently have only one table. `sqlx` will be more than enough . Let's reflect our DB model, with "transport type".

```
type mysqlHour struct {
    ID           string    `db:"id"`
    Hour         time.Time `db:"hour"`
    Availability string    `db:"availability"`
}
```

[10] https://bit.ly/2Mc636Q
[11] https://bit.ly/3btQqQD
[12] https://bit.ly/3btQqQD
[13] https://bit.ly/3qFUWCb
[14] https://github.com/jmoiron/sqlx
[15] https://github.com/volatiletech/sqlboiler

Source: hour_mysql_repository.go on GitHub[16]

> **Note**
>
> You may ask why not to add the `db` attribute to `hour.Hour`? From my experience, it's better to entirely separate domain types from the database. It's easier to test, we don't duplicate validation, and it doesn't introduce a lot of boilerplate.
> In case of any change in the schema, we can do it just in our repository implementation, not in the half of the project. Miłosz described a similar case in **When to stay away from DRY** (Chapter 5).
> I also described that rule deeper in **Domain-Driven Design Lite** (Chapter 6).

How can we use this struct?

```
// sqlContextGetter is an interface provided both by transaction and standard db connection
type sqlContextGetter interface {
   GetContext(ctx context.Context, dest interface{}, query string, args ...interface{}) error
}

func (m MySQLHourRepository) GetOrCreateHour(ctx context.Context, time time.Time) (*hour.Hour, error) {
    return m.getOrCreateHour(ctx, m.db, time, false)
}

func (m MySQLHourRepository) getOrCreateHour(
    ctx context.Context,
    db sqlContextGetter,
    hourTime time.Time,
    forUpdate bool,
) (*hour.Hour, error) {
    dbHour := mysqlHour{}

    query := "SELECT * FROM `hours` WHERE `hour` = ?"
    if forUpdate {
        query += " FOR UPDATE"
    }

    err := db.GetContext(ctx, &dbHour, query, hourTime.UTC())
    if errors.Is(err, sql.ErrNoRows) {
        // in reality this date exists, even if it's not persisted
        return m.hourFactory.NewNotAvailableHour(hourTime)
    } else if err != nil {
        return nil, errors.Wrap(err, "unable to get hour from db")
    }

    availability, err := hour.NewAvailabilityFromString(dbHour.Availability)
    if err != nil {
        return nil, err
    }
```

[16] https://bit.ly/2NoUdqw

```
    domainHour, err := m.hourFactory.UnmarshalHourFromDatabase(dbHour.Hour.Local(), availability)
    if err != nil {
        return nil, err
    }

    return domainHour, nil
}
```

Source: hour_mysql_repository.go on GitHub[17]

With the SQL implementation, it's simple because we don't need to keep backward compatibility. In previous chapters, we used Firestore as our primary database. Let's prepare the implementation based on that, keeping backward compatibility.

Firestore implementation

When you want to refactor a legacy application, abstracting the database may be a good starting point.

Sometimes, applications are built in a database-centric way. In our case, it's an HTTP Response-centric approach – our database models are based on Swagger generated models. In other words – our data models are based on Swagger data models that are returned by API. Does it stop us from abstracting the database? Of course not! It will need just some extra code to handle unmarshaling.

With Domain-First approach, our database model would be much better, like in the SQL implementation. But we are where we are. Let's cut this old legacy step by step. I also have the feeling that CQRS will help us with that.

> **Note**
>
> In practice, a migration of the data may be simple, as long as no other services are integrated directly via the database.
> Unfortunatly, it's an optimistic assumption when we work with a legacy response/database-centric or CRUD service...

```
func (f FirestoreHourRepository) GetOrCreateHour(ctx context.Context, time time.Time) (*hour.Hour, error) {
   date, err := f.getDateDTO(
      // getDateDTO should be used both for transactional and non transactional query,
      // the best way for that is to use closure
      func() (doc *firestore.DocumentSnapshot, err error) {
         return f.documentRef(time).Get(ctx)
      },
      time,
   )
   if err != nil {
      return nil, err
   }
```

[17] https://bit.ly/3uqtFGf

```
    hourFromDb, err := f.domainHourFromDateDTO(date, time)
    if err != nil {
        return nil, err
    }

    return hourFromDb, err
}
```

Source: hour_firestore_repository.go on GitHub[18]

```
// for now we are keeping backward comparability, because of that it's a bit messy and overcomplicated
// todo - we will clean it up later with CQRS :-)
func (f FirestoreHourRepository) domainHourFromDateDTO(date Date, hourTime time.Time) (*hour.Hour, error) {
    firebaseHour, found := findHourInDateDTO(date, hourTime)
    if !found {
        // in reality this date exists, even if it's not persisted
        return f.hourFactory.NewNotAvailableHour(hourTime)
    }

    availability, err := mapAvailabilityFromDTO(firebaseHour)
    if err != nil {
        return nil, err
    }

    return f.hourFactory.UnmarshalHourFromDatabase(firebaseHour.Hour.Local(), availability)
}
```

Source: hour_firestore_repository.go on GitHub[19]

Unfortunately, the Firebase interfaces for the transactional and non-transactional queries are not fully compatible. To avoid duplication, I created `getDateDTO` that can handle this difference by passing `getDocumentFn`.

```
func (f FirestoreHourRepository) getDateDTO(
    getDocumentFn func() (doc *firestore.DocumentSnapshot, err error),
    dateTime time.Time,
) (Date, error) {
    doc, err := getDocumentFn()
    if status.Code(err) == codes.NotFound {
        // in reality this date exists, even if it's not persisted
        return NewEmptyDateDTO(dateTime), nil
    }
    if err != nil {
        return Date{}, err
    }

    date := Date{}
    if err := doc.DataTo(&date); err != nil {
        return Date{}, errors.Wrap(err, "unable to unmarshal Date from Firestore")
    }
```

[18] https://bit.ly/2ZBYsBI
[19] https://bit.ly/3dwlnX3

```
    return date, nil
}
```

Source: hour_firestore_repository.go on GitHub[20]

Even if some extra code is needed, it's not bad. And at least it can be tested easily.

Updating the data

As I mentioned before – it's critical to be sure that **only one person can schedule a training in one hour**. To handle that scenario, we need to use **optimistic locking and transactions**. Even if *transactions* is a pretty common term, let's ensure that we are on the same page with *Optimistic Locking*.

> **Optimistic concurrency control** assumes that many transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.
>
> wikipedia.org (https://en.wikipedia.org/wiki/Optimistic_concurrency_control)

Technically transactions handling is not complicated. The biggest challenge that I had was a bit different – how to manage transactions in a clean way that does not affect the rest of the application too much, is not dependent on the implementation, and is explicit and fast.

I experimented with many ideas, like passing transaction via `context.Context`, handing transaction on HTTP/gRPC/message middlewares level, etc. All approaches that I tried had many major issues – they were a bit magical, not explicit, and slow in some cases.

Currently, my favorite one is an approach based on closure passed to the update function.

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

Source: repository.go on GitHub[21]

The basic idea is that we when we run `UpdateHour`, we need to provide `updateFn` that can update the provided hour.

So in practice in one transaction we:

[20] https://bit.ly/2M83z9p
[21] https://bit.ly/3bo6QtT

- get and provide all parameters for `updateFn` (`h *Hour` in our case) based on provided UUID or any other parameter (in our case `hourTime time.Time`)
- execute the closure (`updateFn` in our case)
- save return values (`*Hour` in our case, if needed we can return more)
- execute a rollback in case of an error returned from the closure

How is it used in practice?

```
func (g GrpcServer) MakeHourAvailable(ctx context.Context, request *trainer.UpdateHourRequest)
 ↪  (*trainer.EmptyResponse, error) {
   trainingTime, err := protoTimestampToTime(request.Time)
   if err != nil {
      return nil, status.Error(codes.InvalidArgument, "unable to parse time")
   }

   if err := g.hourRepository.UpdateHour(ctx, trainingTime, func(h *hour.Hour) (*hour.Hour, error) {
      if err := h.MakeAvailable(); err != nil {
         return nil, err
      }

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

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

Source: grpc.go on GitHub[22]

As you can see, we get `Hour` instance from some (unknown!) database. After that, we make this hour `Available`. If everything is fine, we save the hour by returning it. As part of **Domain-Driven Design Lite** (Chapter 6), **all validations were moved to the domain level, so here we are sure that we aren't doing anything "stupid". It also simplified this code a lot.**

```
+func (g GrpcServer) MakeHourAvailable(ctx context.Context, request *trainer.UpdateHourRequest)
 ↪  (*trainer.EmptyResponse, error) {
@ ...
-func (g GrpcServer) UpdateHour(ctx context.Context, req *trainer.UpdateHourRequest) (*trainer.EmptyResponse, error)
 ↪  {
-   trainingTime, err := grpcTimestampToTime(req.Time)
-   if err != nil {
-       return nil, status.Error(codes.InvalidArgument, "unable to parse time")
-   }
-
-   date, err := g.db.DateModel(ctx, trainingTime)
-   if err != nil {
-       return nil, status.Error(codes.Internal, fmt.Sprintf("unable to get data model: %s", err))
-   }
```

[22] https://bit.ly/3pD36Kd

```
-
-   hour, found := date.FindHourInDate(trainingTime)
-   if !found {
-       return nil, status.Error(codes.NotFound, fmt.Sprintf("%s hour not found in schedule", trainingTime))
-   }
-
-   if req.HasTrainingScheduled && !hour.Available {
-       return nil, status.Error(codes.FailedPrecondition, "hour is not available for training")
-   }
-
-   if req.Available && req.HasTrainingScheduled {
-       return nil, status.Error(codes.FailedPrecondition, "cannot set hour as available when it have training
 ↪ scheduled")
-   }
-   if !req.Available && !req.HasTrainingScheduled {
-       return nil, status.Error(codes.FailedPrecondition, "cannot set hour as unavailable when it have no training
 ↪ scheduled")
-   }
-   hour.Available = req.Available
-
-   if hour.HasTrainingScheduled && hour.HasTrainingScheduled == req.HasTrainingScheduled {
-       return nil, status.Error(codes.FailedPrecondition, fmt.Sprintf("hour HasTrainingScheduled is already %t",
 ↪ hour.HasTrainingScheduled))
-   }
-
-   hour.HasTrainingScheduled = req.HasTrainingScheduled
-   if err := g.db.SaveModel(ctx, date); err != nil {
-       return nil, status.Error(codes.Internal, fmt.Sprintf("failed to save date: %s", err))
-   }
-
-   return &trainer.EmptyResponse{}, nil
-}
```

Source: 0249977c58a310d343ca2237c201b9ba016b148e on GitHub[23]

In our case from `updateFn` we return only `(*Hour, error)` – **you can return more values if needed.** You can return event sourcing events, read models, etc.

We can also, in theory, use the same `hour.*Hour`, that we provide to `updateFn`. I decided to not do that. Using the returned value gives us more flexibility (we can replace a different instance of `hour.*Hour` if we want).

It's also nothing terrible to have multiple `UpdateHour`-like functions created with extra data to save. Under the hood, the implementation should re-use the same code without a lot of duplication.

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

[23] https://bit.ly/2ZCKjnL

```
    ) error

     UpdateHourWithMagic(
       ctx context.Context,
       hourTime time.Time,
       updateFn func(h *Hour) (*Hour, *Magic, error),
    ) error
}
```

How to implement it now?

In-memory transactions implementation

The memory implementation is again the simplest one. We need to get the current value, execute closure, and save the result.

What is essential, in the map, we store a copy instead of a pointer. Thanks to that, we are sure that without the "commit" (`m.hours[hourTime] = *updatedHour`) our values are not saved. We will double-check it in tests.

```
func (m *MemoryHourRepository) UpdateHour(
    _ context.Context,
    hourTime time.Time,
    updateFn func(h *hour.Hour) (*hour.Hour, error),
) error {
    m.lock.Lock()
    defer m.lock.Unlock()

    currentHour, err := m.getOrCreateHour(hourTime)
    if err != nil {
        return err
    }

    updatedHour, err := updateFn(currentHour)
    if err != nil {
        return err
    }

    m.hours[hourTime] = *updatedHour

    return nil
}
```

Source: hour_memory_repository.go on GitHub[24]

Firestore transactions implementation

Firestore implementation is a bit more complex, but again – it's related to backward compatibility. Functions `getDateDTO`, `domainHourFromDateDTO`, `updateHourInDataDTO` could be probably skipped when our data model would be better. Another reason to not use Database-centric/Response-centric approach!

[24] https://bit.ly/2M7RLnC

```go
func (f FirestoreHourRepository) UpdateHour(
   ctx context.Context,
   hourTime time.Time,
   updateFn func(h *hour.Hour) (*hour.Hour, error),
) error {
   err := f.firestoreClient.RunTransaction(ctx, func(ctx context.Context, transaction *firestore.Transaction) error {
      dateDocRef := f.documentRef(hourTime)

      firebaseDate, err := f.getDateDTO(
         // getDateDTO should be used both for transactional and non transactional query,
         // the best way for that is to use closure
         func() (doc *firestore.DocumentSnapshot, err error) {
            return transaction.Get(dateDocRef)
         },
         hourTime,
      )
      if err != nil {
         return err
      }

      hourFromDB, err := f.domainHourFromDateDTO(firebaseDate, hourTime)
      if err != nil {
         return err
      }

      updatedHour, err := updateFn(hourFromDB)
      if err != nil {
         return errors.Wrap(err, "unable to update hour")
      }
      updateHourInDataDTO(updatedHour, &firebaseDate)

      return transaction.Set(dateDocRef, firebaseDate)
   })

   return errors.Wrap(err, "firestore transaction failed")
}
```

Source: hour_firestore_repository.go on GitHub[25]

As you can see, we get `*hour.Hour`, call `updateFn`, and save results inside of `RunTransaction`.

Despite some extra complexity, this implementation is still clear, easy to understand and test.

MySQL transactions implementation

Let's compare it with MySQL implementation, where we designed models in a better way. Even if the way of implementation is similar, the biggest difference is a way of handling transactions.

In the SQL driver, the transaction is represented by `*db.Tx`. We use this particular object to call all queries and do

[25] https://bit.ly/3keLqU3

a rollback and commit. To ensure that we don't forget about closing the transaction, we do rollback and commit in the `defer`.

```
func (m MySQLHourRepository) UpdateHour(
    ctx context.Context,
    hourTime time.Time,
    updateFn func(h *hour.Hour) (*hour.Hour, error),
) (err error) {
    tx, err := m.db.Beginx()
    if err != nil {
        return errors.Wrap(err, "unable to start transaction")
    }

    // Defer is executed on function just before exit.
    // With defer, we are always sure that we will close our transaction properly.
    defer func() {
        // In `UpdateHour` we are using named return - `(err error)`.
        // Thanks to that that can check if function exits with error.
        //
        // Even if function exits without error, commit still can return error.
        // In that case we can override nil to err `err = m.finish...`.
        err = m.finishTransaction(err, tx)
    }()

    existingHour, err := m.getOrCreateHour(ctx, tx, hourTime, true)
    if err != nil {
        return err
    }

    updatedHour, err := updateFn(existingHour)
    if err != nil {
        return err
    }

    if err := m.upsertHour(tx, updatedHour); err != nil {
        return err
    }

    return nil
}
```

Source: hour_mysql_repository.go on GitHub[26]

In that case, we also get the hour by passing `forUpdate == true` to `getOrCreateHour`. This flag is adding `FOR UPDATE` statement to our query. The `FOR UPDATE` statement is critical because without that, we will allow parallel transactions to change the hour.

> `SELECT ... FOR UPDATE`
>
> For index records the search encounters, locks the rows and any associated index entries, the same as if

[26] https://bit.ly/3bi6pRG

> you issued an UPDATE statement for those rows. Other transactions are blocked from updating those rows.
>
> dev.mysql.com (https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-reads.html)

```
func (m MySQLHourRepository) getOrCreateHour(
   ctx context.Context,
   db sqlContextGetter,
   hourTime time.Time,
   forUpdate bool,
) (*hour.Hour, error) {
   dbHour := mysqlHour{}

   query := "SELECT * FROM `hours` WHERE `hour` = ?"
   if forUpdate {
      query += " FOR UPDATE"
   }

    // ...
```

Source: hour_mysql_repository.go on GitHub[27]

I never sleep well when I don't have an automatic test for code like that. Let's address it later.

`finishTransaction` is executed, when `UpdateHour` exits. When commit or rollback failed, we can also override the returned error.

```
// finishTransaction rollbacks transaction if error is provided.
// If err is nil transaction is committed.
//
// If the rollback fails, we are using multierr library to add error about rollback failure.
// If the commit fails, commit error is returned.
func (m MySQLHourRepository) finishTransaction(err error, tx *sqlx.Tx) error {
   if err != nil {
      if rollbackErr := tx.Rollback(); rollbackErr != nil {
         return multierr.Combine(err, rollbackErr)
      }

      return err
   } else {
      if commitErr := tx.Commit(); commitErr != nil {
         return errors.Wrap(err, "failed to commit tx")
      }

      return nil
   }
}
```

Source: hour_mysql_repository.go on GitHub[28]

[27] https://bit.ly/3sdIE46
[28] https://bit.ly/3dwAkbR

```go
// upsertHour updates hour if hour already exists in the database.
// If your doesn't exists, it's inserted.
func (m MySQLHourRepository) upsertHour(tx *sqlx.Tx, hourToUpdate *hour.Hour) error {
   updatedDbHour := mysqlHour{
      Hour:         hourToUpdate.Time().UTC(),
      Availability: hourToUpdate.Availability().String(),
   }

   _, err := tx.NamedExec(
      `INSERT INTO
         hours (hour, availability)
      VALUES
         (:hour, :availability)
      ON DUPLICATE KEY UPDATE
         availability = :availability`,
      updatedDbHour,
   )
   if err != nil {
      return errors.Wrap(err, "unable to upsert hour")
   }

   return nil
}
```

Source: hour_mysql_repository.go on GitHub[29]

Summary

Even if the repository approach adds a bit more code, it's totally worth making that investment. **In practice, you may spend 5 minutes more to do that, and the investment should pay you back shortly.**

In this chapter, we miss one essential part – tests. Now adding tests should be much easier, but it still may not be obvious how to do it properly.

I will cover tests in the next chapter. The entire diff of this refactoring, including tests, is available on GitHub[30].

And just to remind – you can also run the application with one command[31] and find the entire source code on GitHub[32]!

Another technique that works pretty well is Clean/Hexagonal architecture – Miłosz covers that in **Clean Architecture** (Chapter 9).

[29] https://bit.ly/3dwAgZF
[30] https://bit.ly/3qJQ13d
[31] See Chapter 2: Building a serverless application with Google Cloud Run and Firebase
[32] https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example

Chapter 8

High-Quality Database Integration Tests

Robert Laszczak

Did you ever hear about a project where changes were tested on customers that you don't like or countries that are not profitable? Or even worse – did you work on such project?

It's not enough to say that it's just not fair and not professional. It's also hard to develop anything new because you are afraid to make any change in your codebase.

In 2019 HackerRank Developer Skills Report[1] *Professional growth & learning* was marked as the most critical factor during looking for a new job. Do you think you can learn anything and grow when you test your application in that way?

It's all leading to frustration and burnout.

To develop your application easily and with confidence, you need to have a set of tests on multiple levels. **In this chapter, I will cover in practical examples how to implement high-quality database integration tests. I will also cover basic Go testing techniques, like test tables, assert functions, parallel execution, and black-box testing.**

What it actually means that test quality is high?

4 principles of high-quality tests

I prepared 4 rules that we need to pass to say that our integration tests quality is high.

1. Fast

Good tests **need to be fast. There is no compromise here.**

[1] https://research.hackerrank.com/developer-skills/2019#jobsearch3

Everybody hates long-running tests. Let's think about your teammates' time and mental health when they are waiting for test results. Both in CI and locally. It's terrifying.

When you wait for a long time, you will likely start to do anything else in the meantime. After the CI passes (hopefully), you will need to switch back to this task. Context switching is one of the biggest productivity killers. It's very exhausting for our brains. We are not robots.

I know that there are still some companies where tests can be executing for 24 hours. We don't want to follow this approach. You should be able to run your tests locally in less than 1 minute, ideally in **less than 10s**. I know that sometimes it may require some time investment. It's an investment with an excellent ROI *(Return Of Investment)*! In that case, you can really quickly check your changes. Also, deployment times, etc. are much shorter.

It's always worth trying to find quick wins that can reduce tests execution the most from my experience. Pareto principle (80/20 rule)[2] works here perfectly!

2. Testing enough scenarios on all levels

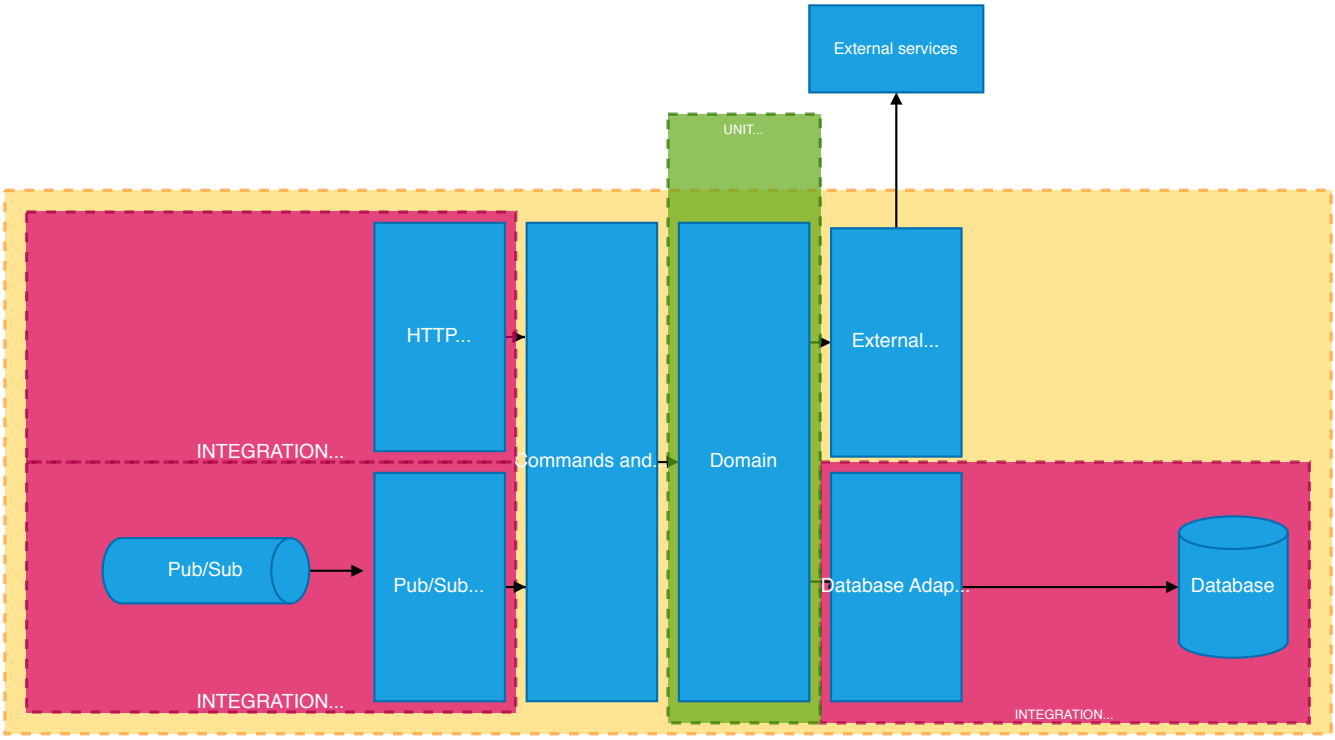
I hope that you already know that 100% test coverage is not the best idea (as long as it is not a simple/critical library).

It's always a good idea to ask yourself the question *"how easily can it break?"*. It's even more worth to ask this question if you feel that the test that you are implementing starts to look exactly as a function that you test. At the end we are not writing tests because tests are nice, but they should save our ass!

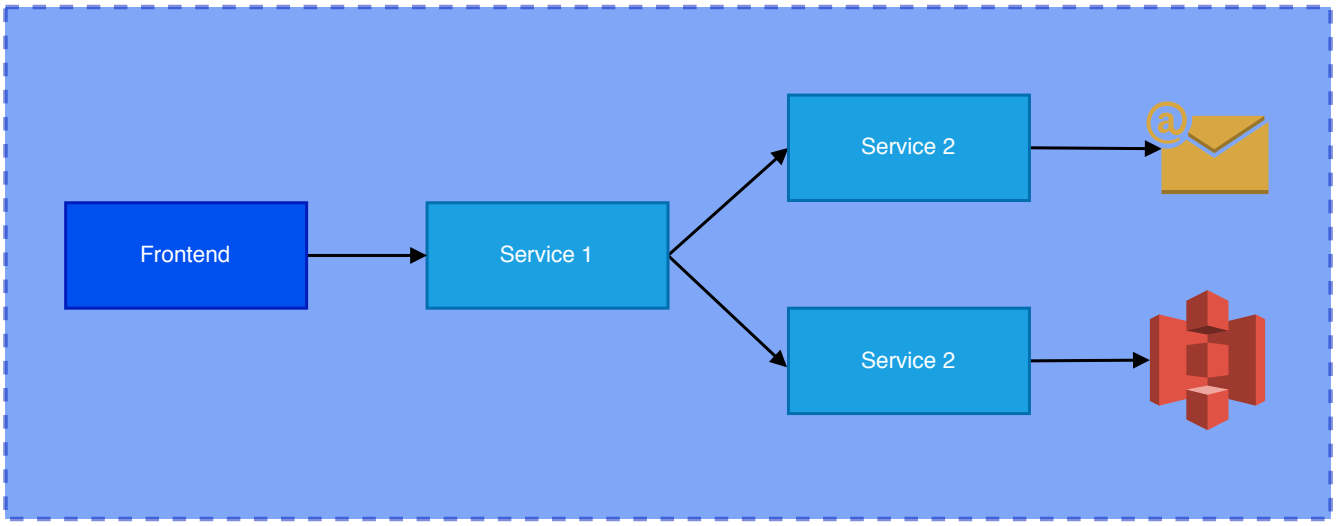
From my experience, **coverage like 70-80% is a pretty good result in Go**.

It's also not the best idea to cover everything with *component* or *end-to-end tests*. First – you will not be able to do that because of the inability to simulate some error scenarios, like rollbacks on the repository. Second – it will break the first rule. These tests will be slow.

[2] https://en.wikipedia.org/wiki/Pareto_principle



COMPONENT TESTS



end-to-end test

Tests on several layers should also overlap, so we will know that integration is done correctly.

You may think that solution for that is simple: the test pyramid! And that's true...sometimes. Especially in

applications that handle a lot of operations based on writes.

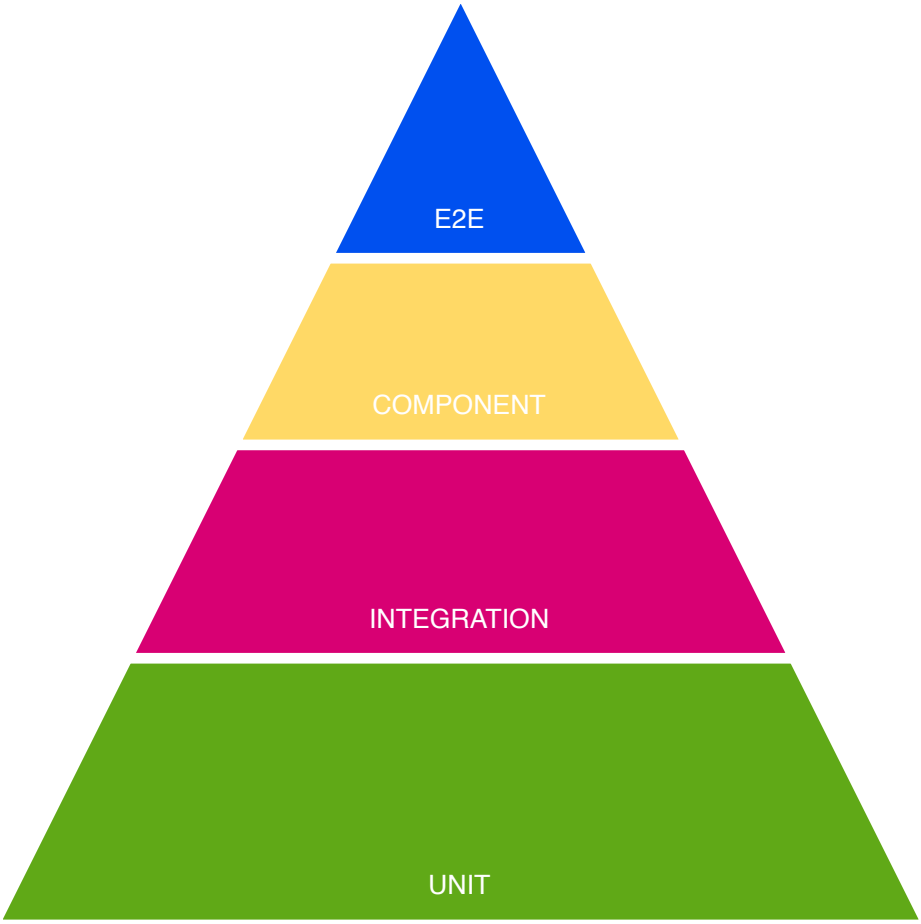


Figure 8.1: Testing Pyramid

But what, for example, about applications that aggregate data from multiple other services and expose the data via API? It has no complex logic of saving the data. Probably most of the code is related to the database operations. In this case, we should use **reversed test pyramid** (it actually looks more like a christmas tree). When big part of our application is connected to some infrastructure (for example: database) it's just hard to cover a lot of functionality with unit tests.

3. Tests need to be robust and deterministic

Do you know that feeling when you are doing some urgent fix, tests are passing locally, you push changes to the repository and … after 20 minutes they fail in the CI? It's incredibly frustrating. It also discourages us from adding new tests. It's also decreasing our trust in them.

You should fix that issue as fast as you can. In that case, Broken windows theory[3] is really valid.

[3]https://en.wikipedia.org/wiki/Broken_windows_theory

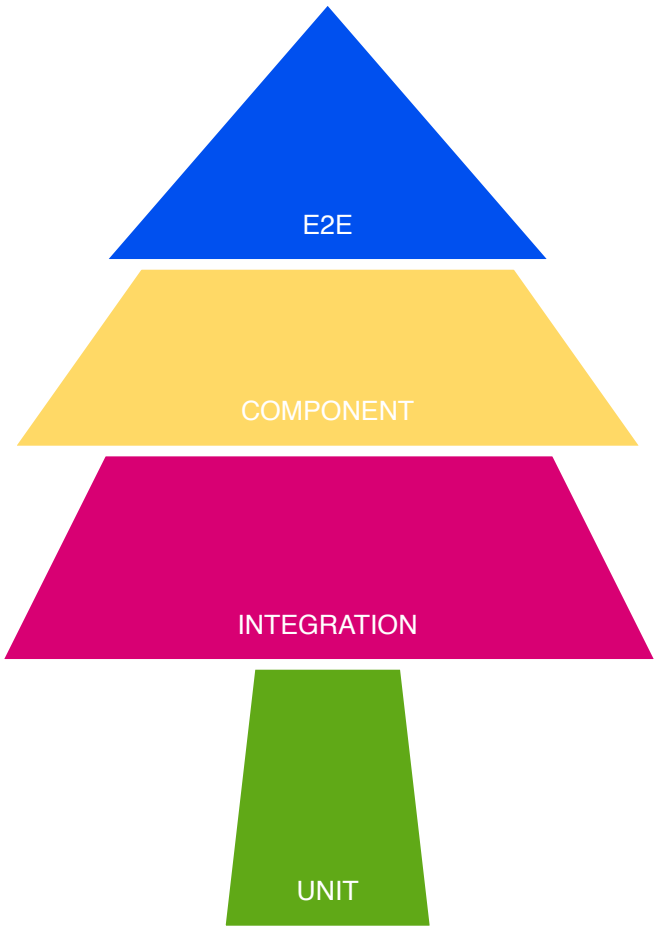


Figure 8.2: Christmas Tree

4. You should be able to execute most of the tests locally

Tests that you run locally should give you enough confidence that the feature that you developed or refactored is still working. **E2E tests should just double-check if everything is integrated correctly.**

You will have also much more confidence when contracts between services are robust because of using gRPC[4], protobuf, or OpenAPI.

This is a good reason to cover as much as we can at lower levels (starting with the lowest): unit, integration, and component tests. Only then E2E.

Implementation

We have some common theoretical ground. But nobody pays us for being the master of theory of programming. Let's go to some practical examples that you can implement in your project.

Let's start with the repository pattern that I described in the previous chapter.

The way how we can interact with our database is defined by the `hour.Repository` interface. It assumes that our repository implementation is stupid. All complex logic is handled by domain part of our application. **It should just save the data without any validations, etc. One of the significant advantages of that approach is the simplification of the repository and tests implementation.**

In the previous chapter I prepared three different database implementations: MySQL, Firebase, and in-memory. We will test all of them. All of them are fully compatible, so we can have just one test suite.

```
package hour

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

Source: repository.go on GitHub[5]

Because of multiple repository implementations, in our tests we iterate through a list of them.

> **Note**
>
> It's actually a pretty similar pattern to how we implemented tests in Watermill[a].
> All Pub/Sub implementations are passing the same test suite.
>
> [a] https://github.com/ThreeDotsLabs/watermill/blob/master/pubsub/tests/test_pubsub.go

[4] See Chapter 3: gRPC communication on Google Cloud Run
[5] https://bit.ly/3aBdRbt

All tests that we will write will be black-box tests. In other words – we will only cover public functions with tests. To ensure that, all our test packages have the `_test` suffix. That forces us to use only the public interface of the package. **It will pay back in the future with much more stable tests, that are not affected by any internal changes.** If you cannot write good black-box tests, you should consider if your public APIs are well designed.

All our repository tests are executed in parallel. Thanks to that, they take less than `200ms`. After adding multiple test cases, this time should not increase significantly.

```
package main_test

// ...

func TestRepository(t *testing.T) {
   rand.Seed(time.Now().UTC().UnixNano())

   repositories := createRepositories(t)

   for i := range repositories {
      // When you are looping over the slice and later using iterated value in goroutine (here because of
       ↪  t.Parallel()),
      // you need to always create variable scoped in loop body!
      // More info here: https://github.com/golang/go/wiki/CommonMistakes#using-goroutines-on-loop-iterator-variables
      r := repositories[i]

      t.Run(r.Name, func(t *testing.T) {
         // It's always a good idea to build all non-unit tests to be able to work in parallel.
         // Thanks to that, your tests will be always fast and you will not be afraid to add more tests because of
          ↪  slowdown.
         t.Parallel()

         t.Run("testUpdateHour", func(t *testing.T) {
            t.Parallel()
            testUpdateHour(t, r.Repository)
         })
         t.Run("testUpdateHour_parallel", func(t *testing.T) {
            t.Parallel()
            testUpdateHour_parallel(t, r.Repository)
         })
         t.Run("testHourRepository_update_existing", func(t *testing.T) {
            t.Parallel()
            testHourRepository_update_existing(t, r.Repository)
         })
         t.Run("testUpdateHour_rollback", func(t *testing.T) {
            t.Parallel()
            testUpdateHour_rollback(t, r.Repository)
         })
      })
   }
}
```

Source: hour_repository_test.go on GitHub[6]

When we have multiple tests, where we pass the same input and check the same output, it is a good idea to use a technique known as *test table*. The idea is simple: you should define a slice of inputs and expected outputs of the test and iterate over it to execute tests.

```
func testUpdateHour(t *testing.T, repository hour.Repository) {
   t.Helper()
   ctx := context.Background()

   testCases := []struct {
      Name       string
      CreateHour func(*testing.T) *hour.Hour
   }{
      {
         Name: "available_hour",
         CreateHour: func(t *testing.T) *hour.Hour {
            return newValidAvailableHour(t)
         },
      },
      {
         Name: "not_available_hour",
         CreateHour: func(t *testing.T) *hour.Hour {
            h := newValidAvailableHour(t)
            require.NoError(t, h.MakeNotAvailable())

            return h
         },
      },
      {
         Name: "hour_with_training",
         CreateHour: func(t *testing.T) *hour.Hour {
            h := newValidAvailableHour(t)
            require.NoError(t, h.ScheduleTraining())

            return h
         },
      },
   }

   for _, tc := range testCases {
      t.Run(tc.Name, func(t *testing.T) {
         newHour := tc.CreateHour(t)

         err := repository.UpdateHour(ctx, newHour.Time(), func(_ *hour.Hour) (*hour.Hour, error) {
            // UpdateHour provides us existing/new *hour.Hour,
            // but we are ignoring this hour and persisting result of `CreateHour`
            // we can assert this hour later in assertHourInRepository
            return newHour, nil
```

[6]https://bit.ly/3sj7H69

```
        })
        require.NoError(t, err)

        assertHourInRepository(ctx, t, repository, newHour)
    })
}
```

Source: hour_repository_test.go on GitHub[7]

You can see that we used a very popular `github.com/stretchr/testify`[8] library. It's significantly reducing boilerplate in tests by providing multiple helpers for asserts[9].

> require.NoError()
>
> When `assert.NoError` assert fails, tests execution is not interrupted.
> It's worth to mention that asserts from `require` package are stopping execution of the test when it fails. Because of that, it's often a good idea to use `require` for checking errors. In many cases, if some operation fails, it doesn't make sense to check anything later.
> When we assert multiple values, `assert` is a better choice, because you will receive more context.

If we have more specific data to assert, it's always a good idea to add some helpers. It removes a lot of duplication, and improves tests readability a lot!

```
func assertHourInRepository(ctx context.Context, t *testing.T, repo hour.Repository, hour *hour.Hour) {
    require.NotNil(t, hour)

    hourFromRepo, err := repo.GetOrCreateHour(ctx, hour.Time())
    require.NoError(t, err)

    assert.Equal(t, hour, hourFromRepo)
}
```

Source: hour_repository_test.go on GitHub[10]

Testing transactions

Mistakes taught me that I should not trust myself when implementing complex code. We can sometimes not understand the documentation or just introduce some stupid mistake. You can gain the confidence in two ways:

1. TDD - let's start with a test that will check if the transaction is working properly.
2. Let's start with the implementation and add tests later.

```
func testUpdateHour_rollback(t *testing.T, repository hour.Repository) {
    t.Helper()
    ctx := context.Background()
```

[7] https://bit.ly/3siawUZ
[8] https://github.com/stretchr/testify
[9] https://godoc.org/github.com/stretchr/testify/assert
[10] https://bit.ly/37uw4Wk

```
    hourTime := newValidHourTime()

    err := repository.UpdateHour(ctx, hourTime, func(h *hour.Hour) (*hour.Hour, error) {
        require.NoError(t, h.MakeAvailable())
        return h, nil
    })

    err = repository.UpdateHour(ctx, hourTime, func(h *hour.Hour) (*hour.Hour, error) {
        assert.True(t, h.IsAvailable())
        require.NoError(t, h.MakeNotAvailable())

        return h, errors.New("something went wrong")
    })
    require.Error(t, err)

    persistedHour, err := repository.GetOrCreateHour(ctx, hourTime)
    require.NoError(t, err)

    assert.True(t, persistedHour.IsAvailable(), "availability change was persisted, not rolled back")
}
```

Source: hour_repository_test.go on GitHub[11]

When I'm not using TDD, I try to be paranoid if test implementation is valid.

To be more confident, I use a technique that I call **tests sabotage**.

The method is pretty simple - let's break the implementation that we are testing and let's see if anything failed.

```
 func (m MySQLHourRepository) finishTransaction(err error, tx *sqlx.Tx) error {
-       if err != nil {
-               if rollbackErr := tx.Rollback(); rollbackErr != nil {
-                       return multierr.Combine(err, rollbackErr)
-               }
-
-               return err
-       } else {
-               if commitErr := tx.Commit(); commitErr != nil {
-                       return errors.Wrap(err, "failed to commit tx")
-               }
-
-               return nil
+       if commitErr := tx.Commit(); commitErr != nil {
+               return errors.Wrap(err, "failed to commit tx")
        }
+
+       return nil
 }
```

If your tests are passing after a change like that, I have bad news...

[11] https://bit.ly/2ZBXKUM

Testing database race conditions

Our applications are not working in the void. It can always be the case that two multiple clients may try to do the same operation, and only one can win!

In our case, the typical scenario is when two clients try to schedule a training at the same time. **We can have only one training scheduled in one hour.**

This constraint is achieved by optimistic locking (described in **The Repository Pattern** (Chapter 7)) and domain constraints (described in **Domain-Driven Design Lite** (Chapter 6)).

Let's verify if it is possible to schedule one hour more than once. The idea is simple: **let's create 20 goroutines, that we will release in one moment and try to schedule training.** We expect that exactly one worker should succeed.

```
func testUpdateHour_parallel(t *testing.T, repository hour.Repository) {
    // ...

    workersCount := 20
    workersDone := sync.WaitGroup{}
    workersDone.Add(workersCount)

    // closing startWorkers will unblock all workers at once,
    // thanks to that it will be more likely to have race condition
    startWorkers := make(chan struct{})
    // if training was successfully scheduled, number of the worker is sent to this channel
    trainingsScheduled := make(chan int, workersCount)

    // we are trying to do race condition, in practice only one worker should be able to finish transaction
    for worker := 0; worker < workersCount; worker++ {
        workerNum := worker

        go func() {
            defer workersDone.Done()
            <-startWorkers

            schedulingTraining := false

            err := repository.UpdateHour(ctx, hourTime, func(h *hour.Hour) (*hour.Hour, error) {
                // training is already scheduled, nothing to do there
                if h.HasTrainingScheduled() {
                    return h, nil
                }
                // training is not scheduled yet, so let's try to do that
                if err := h.ScheduleTraining(); err != nil {
                    return nil, err
                }

                schedulingTraining = true

                return h, nil
            })
```

```
            if schedulingTraining && err == nil {
                // training is only scheduled if UpdateHour didn't return an error
                trainingsScheduled <- workerNum
            }
        }()
    }

    close(startWorkers)

    // we are waiting, when all workers did the job
    workersDone.Wait()
    close(trainingsScheduled)

    var workersScheduledTraining []int

    for workerNum := range trainingsScheduled {
        workersScheduledTraining = append(workersScheduledTraining, workerNum)
    }

    assert.Len(t, workersScheduledTraining, 1, "only one worker should schedule training")
}
```

Source: hour_repository_test.go on GitHub[12]

It is also a good example that some use cases are easier to test in the integration test, not in acceptance or E2E level. Tests like that as E2E will be really heavy, and you will need to have more workers to be sure that they execute transactions simultaneously.

Making tests fast

If your tests can't be executed in parallel, they will be slow. Even on the best machine.

Is putting `t.Parallel()` enough? Well, we need to ensure that our tests are independent. In our case, **if two tests would try to edit the same hour, they can fail randomly.** This is a highly undesirable situation.

To achieve that, I created the `newValidHourTime()` function that provides a random hour that is unique in the current test run. In most applications, generating a unique UUID for your entities may be enough.

In some situations it may be less obvious, but still not impossible. I encourage you to spend some time to find the solution. Please treat it as the investment in your and your teammates' mental health .

```
// usedHours is storing hours used during the test,
// to ensure that within one test run we are not using the same hour
// (it should be not a problem between test runs)
var usedHours = sync.Map{}

func newValidHourTime() time.Time {
   for {
```

[12] https://bit.ly/3pKAVJu

```
        minTime := time.Now().AddDate(0, 0, 1)

        minTimestamp := minTime.Unix()
        maxTimestamp := minTime.AddDate(0, 0, testHourFactory.Config().MaxWeeksInTheFutureToSet*7).Unix()

        t := time.Unix(rand.Int63n(maxTimestamp-minTimestamp)+minTimestamp, 0).Truncate(time.Hour).Local()

        _, alreadyUsed := usedHours.LoadOrStore(t.Unix(), true)
        if !alreadyUsed {
            return t
        }
    }
}
```

Source: hour_repository_test.go on GitHub[13]

What is also good about making our tests independent, is no need for data cleanup. In my experience, doing data cleanup is always messy because:

- when it doesn't work correctly, it creates hard-to-debug issues in tests,
- it makes tests slower,
- it adds overhead to the development (you need to remember to update the cleanup function)
- it may make running tests in parallel harder.

It may also happen that we are not able to run tests in parallel. Two common examples are:

- pagination – if you iterate over pages, other tests can put something in-between and move "items" in the pages.
- global counters – like with pagination, other tests may affect the counter in an unexpected way.

In that case, it's worth to keep these tests as short as we can.

Please, don't use sleep in tests!

The last tip that makes tests flaky and slow is putting the sleep function in them. Please, **don't**! It's much better to synchronize your tests with channels or `sync.WaitGroup{}`. They are faster and more stable in that way.

If you really need to wait for something, it's better to use `assert.Eventually` instead of a sleep.

> `Eventually` asserts that given `condition` will be met in `waitFor` time, periodically checking target function each `tick`.

```
assert.Eventually(
    t,
    func() bool { return true }, // condition
    time.Second, // waitFor
    10*time.Millisecond, // tick
)
```

[13] https://bit.ly/2NN5YGV

godoc.org/github.com/stretchr/testify/assert (https://godoc.org/github.com/stretchr/testify/assert#Eventually)

Running

Now, when our tests are implemented, it's time to run them!

Before that, we need to start our container with Firebase and MySQL with `docker-compose up`.

I prepared `make test` command that runs tests in a consistent way (for example, `-race` flag). It can also be used in the CI.

```
$ make test

?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/common/auth [no test files]
?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/common/client   [no test files]
?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/common/genproto/trainer [no test files]
?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/common/genproto/users   [no test files]
?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/common/logs [no test files]
?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/common/server   [no test files]
?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/common/server/httperr   [no test files]
ok      github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer 0.172s
ok      github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/domain/hour 0.031s
?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainings   [no test files]
?       github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/users   [no test files]
```

Running one test and passing custom params

If you would like to pass some extra params, to have a verbose output (`-v`) or execute exact test (`-run`), you can pass it after `make test --`.

```
$ make test -- -v -run ^TestRepository/memory/testUpdateHour$

--- PASS: TestRepository (0.00s)
  --- PASS: TestRepository/memory (0.00s)
      --- PASS: TestRepository/memory/testUpdateHour (0.00s)
          --- PASS: TestRepository/memory/testUpdateHour/available_hour (0.00s)
          --- PASS: TestRepository/memory/testUpdateHour/not_available_hour (0.00s)
          --- PASS: TestRepository/memory/testUpdateHour/hour_with_training (0.00s)
PASS
```

If you are interested in how it is implemented, I'd recommend you check my Makefile magic[14].

Debugging

Sometimes our tests fail in an unclear way. In that case, it's useful to be able to easily check what data we have in our database.

[14] https://bit.ly/2ZCt0mO

For SQL databases my first choice for that are mycli for MySQL[15] and pgcli for PostgreSQL[16]. I've added `make mycli` command to Makefile, so you don't need to pass credentials all the time.

```
$ make mycli

mysql user@localhost:db> SELECT * from `hours`;
+---------------------+--------------------+
| hour                | availability       |
|---------------------+--------------------/
| 2020-08-31 15:00:00 | available          |
| 2020-09-13 19:00:00 | training_scheduled |
| 2022-07-19 19:00:00 | training_scheduled |
| 2023-03-19 14:00:00 | available          |
| 2023-08-05 03:00:00 | training_scheduled |
| 2024-01-17 07:00:00 | not_available      |
| 2024-02-07 15:00:00 | available          |
| 2024-05-07 18:00:00 | training_scheduled |
| 2024-05-22 09:00:00 | available          |
| 2025-03-04 15:00:00 | available          |
| 2025-04-15 08:00:00 | training_scheduled |
| 2026-05-22 09:00:00 | training_scheduled |
| 2028-01-24 18:00:00 | not_available      |
| 2028-07-09 00:00:00 | not_available      |
| 2029-09-23 15:00:00 | training_scheduled |
+---------------------+--------------------+
15 rows in set
Time: 0.025s
```

For Firestore, the emulator is exposing the UI at localhost:4000/firestore[17].

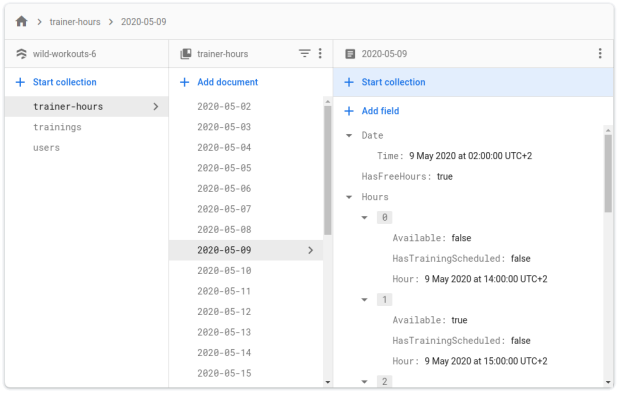


Figure 8.3: Firestore Console

[15] https://www.mycli.net/install
[16] https://www.pgcli.com/
[17] http://localhost:4000/firestore/

First step for having well-tested application

The biggest gap that we currently have is a lack of tests on the component and E2E level. Also, a big part of the application is not tested at all. We will fix that in the next chapters. We will also cover some topics that we skipped this time.

But before that, we have one topic that we need to cover earlier – Clean/Hexagonal architecture! This approach will help us organize our application a bit and make future refactoring and features easier to implement.

Just to remind, **the entire source code of Wild Workouts is available on GitHub**[18]. **You can run it locally and deploy to Google Cloud with one command.**

[18] https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/

Chapter 9

Clean Architecture

Miłosz Smółka

The authors of Accelerate[1] dedicate a whole chapter to software architecture and how it affects development performance. One thing that often comes up is designing applications to be "loosely coupled".

> The goal is for your architecture to support the ability of teams to get their work done—from design through to deployment—without requiring high-bandwidth communication between teams.
>
> Accelerate (https://itrevolution.com/book/accelerate/)

Note

If you haven't read Accelerate[a] yet, I highly recommend it. The book presents scientific evidence on methods leading to high performance in development teams. The approach I describe is not only based on our experiences but also mentioned throughout the book.

[a] https://itrevolution.com/book/accelerate/

While coupling seems mostly related to microservices across multiple teams, we find loosely coupled architecture just as useful for work within a team. Keeping architecture standards makes parallel work possible and helps onboard new team members.

You probably heard about the *"low coupling, high cohesion"* concept, but it's rarely obvious how to achieve it. The good news is, it's the main benefit of Clean Architecture.

The pattern is not only an excellent way to start a project but also helpful when refactoring a poorly designed application. I focus on the latter in this chapter. I show refactoring of a real application, so it should be clear how to apply similar changes in your projects.

There are also other benefits of this approach we noticed:

[1] https://itrevolution.com/book/accelerate/

- a standard structure, so it's easy to find your way in the project,
- faster development in the long term,
- mocking dependencies becomes trivial in unit tests,
- easy switching from prototypes to proper solutions (e.g., changing in-memory storage to an SQL database).

Clean Architecture

I had a hard time coming up with this chapter's title because the pattern comes in many flavors. There's Clean Architecture[2], Onion Architecture[3], Hexagonal Architecture[4], and Ports and Adapters.

We tried to use these patterns in Go in an idiomatic way during the last couple of years. It involved trying out some approaches, failing, changing them, and trying again.

We arrived at a mix of the ideas above, sometimes not strictly following the original patterns, but we found it works well in Go. I will show our approach with a refactoring of Wild Workouts[5], our example application.

I want to point out that the idea is not new at all. A big part of it is **abstracting away implementation details**, a standard in technology, especially software.

Another name for it is **separation of concerns**. The concept is so old now it exists on several levels. There are structures, namespaces, modules, packages, and even (micro)services. All meant to keep related things within a boundary. Sometimes, it feels like common sense:

- If you have to optimize an SQL query, you don't want to risk changing the display format.
- If you change an HTTP response format, you don't want to alter the database schema.

Our approach to Clean Architecture is two ideas combined: separating Ports and Adapters and limiting how code structures refer to each other.

Before We Start

Before introducing Clean Architecture in Wild Workouts, I refactored the project a bit. The changes come from patterns we shared in previous chapters.

The first one is using **separate models for database entities and HTTP responses**. I've introduced changes in the `users` service in **When to stay away from DRY** (Chapter 5). I applied the same pattern now in `trainer` and `trainings` as well. See the full commit on GitHub[6].

The second change follows **the Repository Pattern** that Robert introduced in **The Repository Pattern** (Chapter 7). My refactoring[7] moved database-related code in `trainings` to a separate structure.

[2] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
[3] https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
[4] https://web.archive.org/web/20180822100852/http://alistair.cockburn.us/Hexagonal+architecture
[5] https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example
[6] https://bit.ly/3kfl84g
[7] https://bit.ly/3bK1Pw7

Separating Ports and Adapters

Ports and Adapters can be called different names, like interfaces and infrastructure. At the core, the idea is to explicitly separate these two categories from the rest of your application code.

We take the code in these groups and place it in different packages. We refer to them as "layers". **The layers we usually use are adapters, ports, application, and domain.**

- **An adapter is how your application talks to the external world.** You have to **adapt** your internal structures to what the external API expects. Think SQL queries, HTTP or gRPC clients, file readers and writers, Pub/Sub message publishers.
- **A port is an input to your application**, and the only way the external world can reach it. It could be an HTTP or gRPC server, a CLI command, or a Pub/Sub message subscriber.
- **The application logic** is a thin layer that "glues together" other layers. It's also known as "use cases". If you read this code and can't tell what database it uses or what URL it calls, it's a good sign. Sometimes it's very short, and that's fine. Think about it as an orchestrator.
- If you also follow Domain-Driven Design[8], you can introduce **a domain layer that holds just the business logic**.

> Note
>
> If the idea of separating layers is still not clear, take a look at your smartphone. If you think about it, it uses similar concepts as well.
> You can control your smartphone using the physical buttons, the touchscreen, or voice assistant. Whether you press the "volume up" button, swipe the volume bar up, or say "Siri, volume up", the effect is the same. There are several entry points (**ports**) to the "change volume" **logic**.
> When you play some music, you can hear it coming from the speaker. If you plug in headphones, the audio will automatically change to them. Your music app doesn't care. It's not talking with the hardware directly, but using one of the **adapters** the OS provides.
> Can you imagine creating a mobile app that has to be aware of the headphones model connected to the smartphone? Including SQL queries directly inside the application logic is similar: it exposes the implementation details.

Let's start refactoring by introducing the layers in the `trainings` service. The project looks like this so far:

```
trainings/
  firestore.go
  go.mod
  go.sum
  http.go
  main.go
  openapi_api.gen.go
  openapi_types.gen.go
```

This part of refactoring is simple:

[8] See Chapter 6: Domain-Driven Design Lite

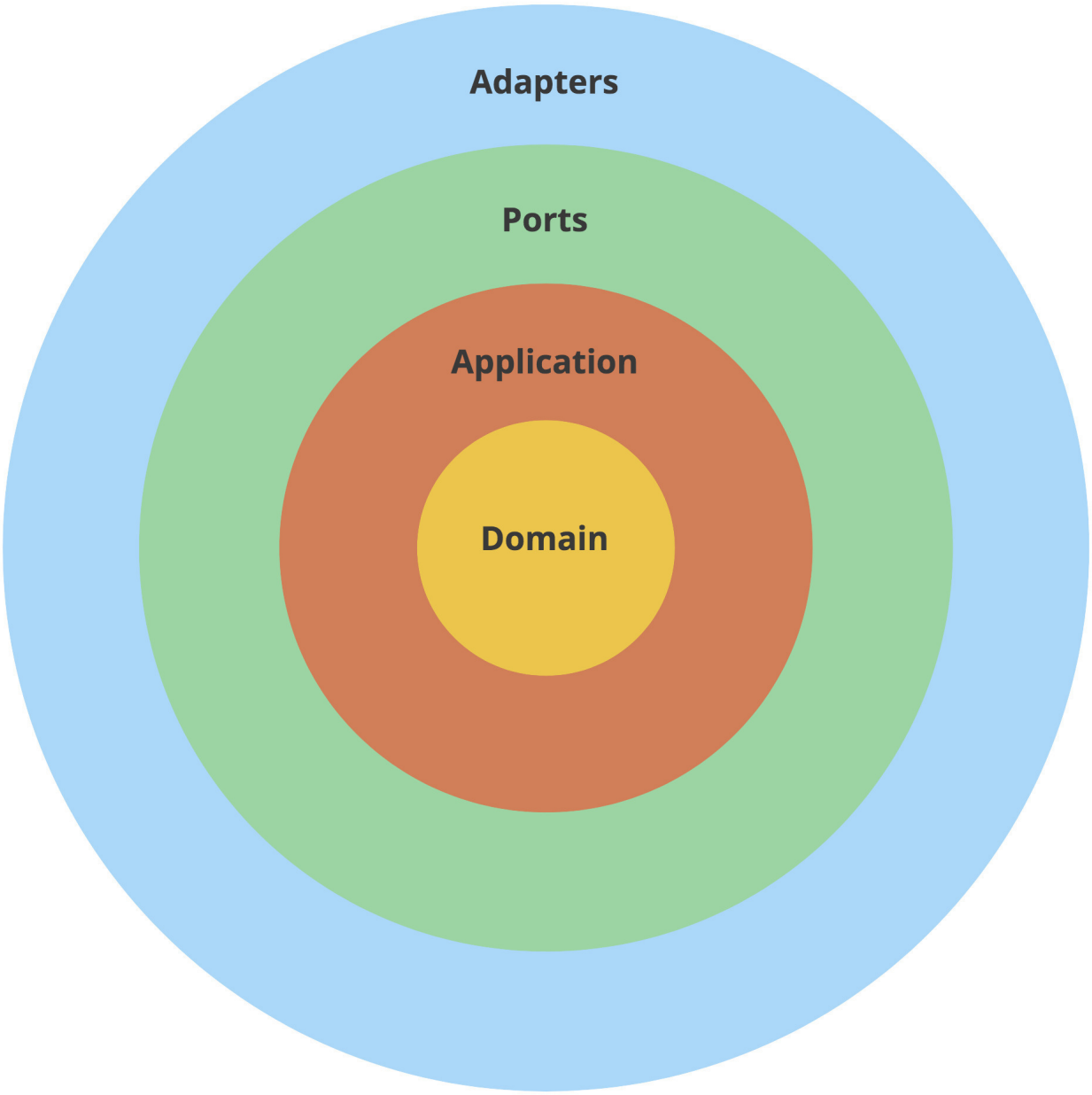


Figure 9.1: Clean Architecture layers

1. Create `ports`, `adapters`, and `app` directories.
2. Move each file to the proper directory.

```
trainings/
  adapters
     firestore.go
  app
  go.mod
  go.sum
  main.go
  ports
      http.go
      openapi_api.gen.go
      openapi_types.gen.go
```

I introduced similar packages in the `trainer` service. We won't make any changes to the `users` service this time. There's no application logic there, and overall, it's tiny. As with every technique, apply Clean Architecture where it makes sense.

> **Note**
>
> If the project grows in size, you may find it helpful to add another level of subdirectories. For example, `adapters/hour/mysql_repository.go` or `ports/http/hour_handler.go`.

You probably noticed there are no files in the `app` package. We now have to extract the application logic from HTTP handlers.

The Application Layer

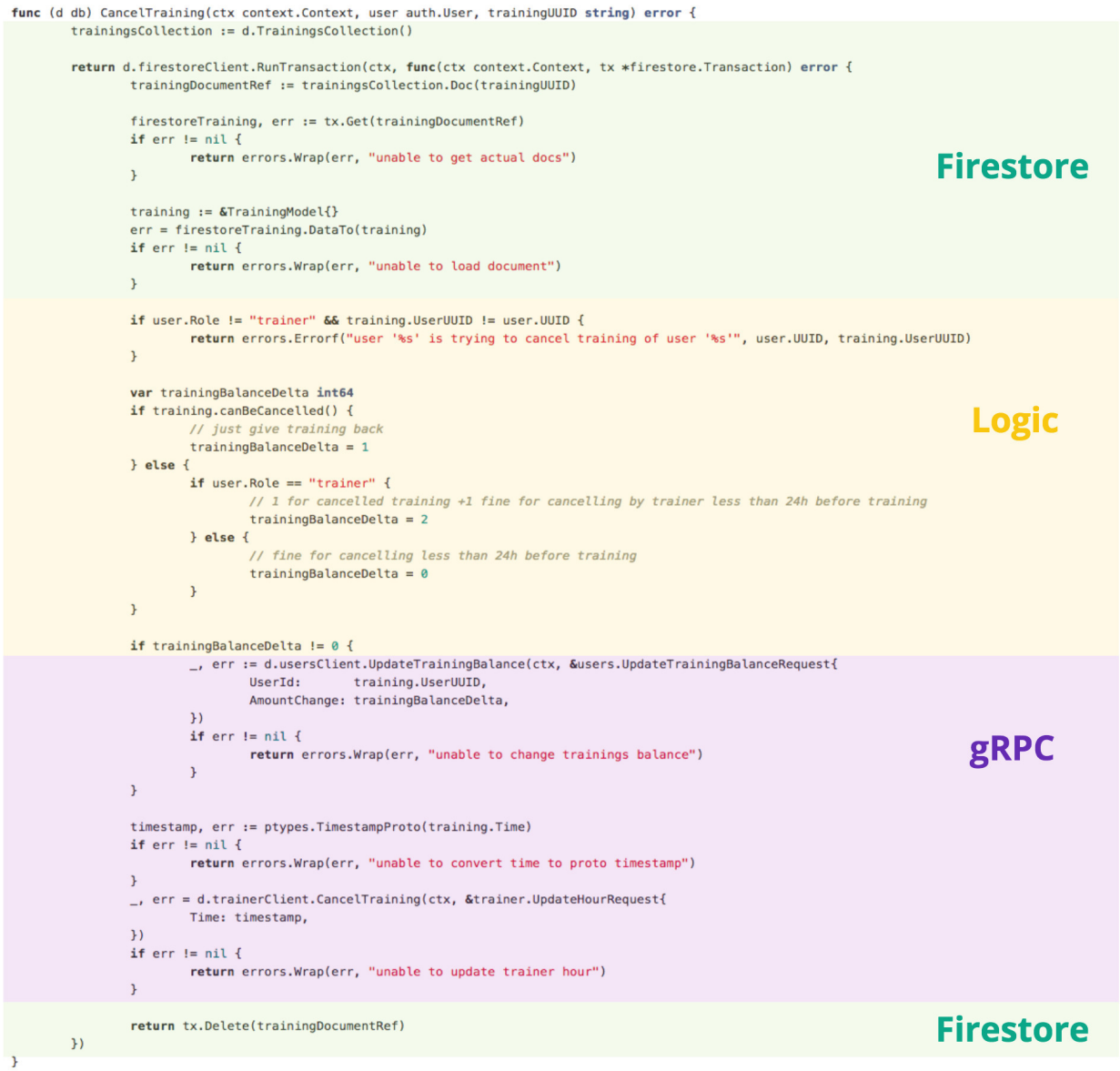
Let's see where our application logic lives. Take a look at the `CancelTraining` method in the `trainings` service.

```
func (h HttpServer) CancelTraining(w http.ResponseWriter, r *http.Request) {
    trainingUUID := r.Context().Value("trainingUUID").(string)

    user, err := auth.UserFromCtx(r.Context())
    if err != nil {
        httperr.Unauthorised("no-user-found", err, w, r)
        return
    }

    err = h.db.CancelTraining(r.Context(), user, trainingUUID)
    if err != nil {
        httperr.InternalError("cannot-update-training", err, w, r)
        return
    }
}
```

Source: http.go on GitHub[9]

[9] https://bit.ly/3ul3QXN

This method is the entry point to the application. There's not much logic there, so let's go deeper into the `db.CancelTraining` method.




```
func (d db) CancelTraining(ctx context.Context, user auth.User, trainingUUID string) error {
        trainingsCollection := d.TrainingsCollection()

        return d.firestoreClient.RunTransaction(ctx, func(ctx context.Context, tx *firestore.Transaction) error {
                trainingDocumentRef := trainingsCollection.Doc(trainingUUID)

                firestoreTraining, err := tx.Get(trainingDocumentRef)
                if err != nil {
                        return errors.Wrap(err, "unable to get actual docs")
                }

                training := &TrainingModel{}
                err = firestoreTraining.DataTo(training)
                if err != nil {
                        return errors.Wrap(err, "unable to load document")
                }

                if user.Role != "trainer" && training.UserUUID != user.UUID {
                        return errors.Errorf("user '%s' is trying to cancel training of user '%s'", user.UUID, training.UserUUID)
                }

                var trainingBalanceDelta int64
                if training.canBeCancelled() {
                        // just give training back
                        trainingBalanceDelta = 1
                } else {
                        if user.Role == "trainer" {
                                // 1 for cancelled training +1 fine for cancelling by trainer less than 24h before training
                                trainingBalanceDelta = 2
                        } else {
                                // fine for cancelling less than 24h before training
                                trainingBalanceDelta = 0
                        }
                }

                if trainingBalanceDelta != 0 {
                        _, err := d.usersClient.UpdateTrainingBalance(ctx, &users.UpdateTrainingBalanceRequest{
                                UserId:       training.UserUUID,
                                AmountChange: trainingBalanceDelta,
                        })
                        if err != nil {
                                return errors.Wrap(err, "unable to change trainings balance")
                        }
                }

                timestamp, err := ptypes.TimestampProto(training.Time)
                if err != nil {
                        return errors.Wrap(err, "unable to convert time to proto timestamp")
                }
                _, err = d.trainerClient.CancelTraining(ctx, &trainer.UpdateHourRequest{
                        Time: timestamp,
                })
                if err != nil {
                        return errors.Wrap(err, "unable to update trainer hour")
                }

                return tx.Delete(trainingDocumentRef)
        })
}
```

Inside the Firestore transaction, there's a lot of code that doesn't belong to database handling.

What's worse, the actual application logic inside this method uses the database model (`TrainingModel`) for decision making:

```
if training.canBeCancelled() {
    // ...
} else {
```

```
    // ...
}
```

Source: firestore.go on GitHub[10]

Mixing the business rules (like when a training can be canceled) with the database model slows down development, as the code becomes hard to understand and reason about. It's also difficult to test such logic.

To fix this, we add an intermediate `Training` type in the `app` layer:

```
type Training struct {
    UUID     string
    UserUUID string
    User     string

    Time  time.Time
    Notes string

    ProposedTime   *time.Time
    MoveProposedBy *string
}

func (t Training) CanBeCancelled() bool {
    return t.Time.Sub(time.Now()) > time.Hour*24
}

func (t Training) MoveRequiresAccept() bool {
    return !t.CanBeCancelled()
}
```

Source: training.go on GitHub[11]

It should now be clear on the first read when a training can be canceled. We can't tell how the training is stored in the database or the JSON format used in the HTTP API. That's a good sign.

We can now update the database layer methods to return this generic application type instead of the database-specific structure (`TrainingModel`). The mapping is trivial because the structs have the same fields (but from now on, they can evolve independently from each other).

```
t := TrainingModel{}
if err := doc.DataTo(&t); err != nil {
    return nil, err
}

trainings = append(trainings, app.Training(t))
```

Source: trainings_firestore_repository.go on GitHub[12]

[10] https://bit.ly/3usH5S2
[11] https://bit.ly/2P0KIOV
[12] https://bit.ly/3aFfBki

The Application Service

We then create a `TrainingsService` struct in the `app` package that will serve as the entry point to trainings application logic.

```
type TrainingService struct {
}

func (c TrainingService) CancelTraining(ctx context.Context, user auth.User, trainingUUID string) error {

}
```

So how do we call the database now? Let's try to replicate what was used so far in the HTTP handler.

```
type TrainingService struct {
    db adapters.DB
}

func (c TrainingService) CancelTraining(ctx context.Context, user auth.User, trainingUUID string) error {
    return c.db.CancelTraining(ctx, user, trainingUUID)
}
```

This code won't compile, though.

```
import cycle not allowed
package github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainings
        imports github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainings/adapters
        imports github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainings/app
        imports github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainings/adapters
```

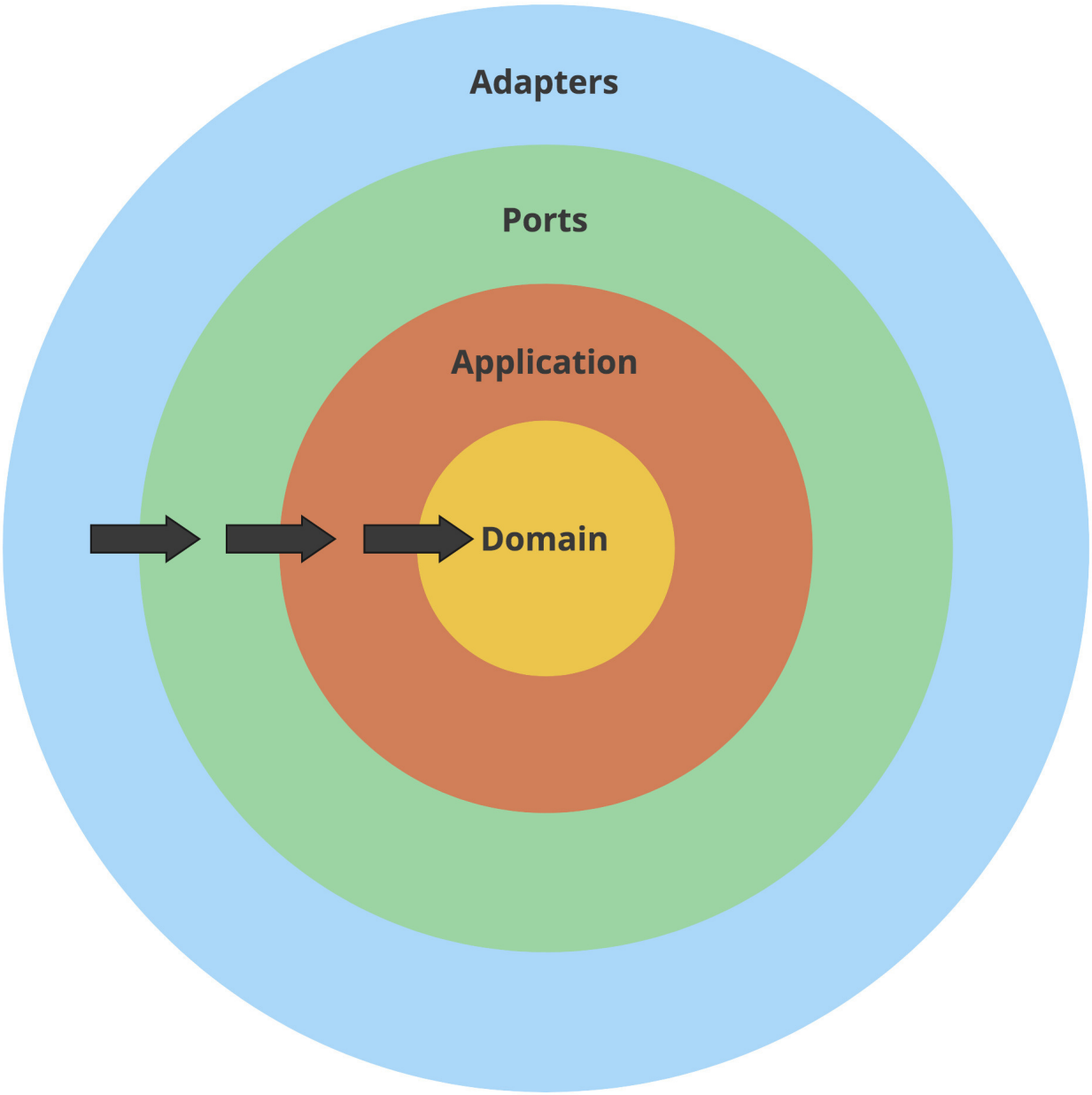
We need to **decide how the layers should refer to each other.**

The Dependency Inversion Principle

A clear separation between ports, adapters, and application logic is useful by itself. Clean Architecture improves it further with Dependency Inversion.

The rule states that **outer layers (implementation details) can refer to inner layers (abstractions), but not the other way around.** The inner layers should instead depend on interfaces.

- The **Domain** knows nothing about other layers whatsoever. It contains pure business logic.
- The **Application** can import domain but knows nothing about outer layers. **It has no idea whether it's being called by an HTTP request, a Pub/Sub handler, or a CLI command**.
- **Ports** can import inner layers. Ports are the entry points to the application, so they often execute application services or commands. However, they can't directly access **Adapters**.
- **Adapters** can import inner layers. Usually, they will operate on types found in **Application** and **Domain**, for example, retrieving them from the database.



Again, it's not a new idea. **The Dependency Inversion Principle is the "D" in SOLID**[13]. Do you think it applies only to OOP? It just happens that **Go interfaces make a perfect match with it**[14].

The principle solves the issue of how packages should refer to each other. The best way to do it is rarely obvious, especially in Go, where import cycles are forbidden. Perhaps that's why some developers claim it's best to avoid

[13]https://en.wikipedia.org/wiki/SOLID
[14]https://dave.cheney.net/2016/08/20/solid-go-design

"nesting" and keep all code in one package. **But packages exist for a reason, and that's separation of concerns.**

Going back to our example, how should we refer to the database layer?

Because the Go interfaces don't need to be explicitly implemented, we can **define them next to the code that needs them.**

So the application service defines: *"I need a way to cancel a training with given UUID. I don't care how you do it, but I trust you to do it right if you implement this interface".*

```
type trainingRepository interface {
    CancelTraining(ctx context.Context, user auth.User, trainingUUID string) error
}

type TrainingService struct {
    trainingRepository trainingRepository
}

func (c TrainingService) CancelTraining(ctx context.Context, user auth.User, trainingUUID string) error {
    return c.trainingRepository.CancelTraining(ctx, user, trainingUUID)
}
```

Source: training_service.go on GitHub[15]

The database method calls gRPC clients of `trainer` and `users` services. It's not the proper place, so we introduce two new interfaces that the service will use.

```
type userService interface {
    UpdateTrainingBalance(ctx context.Context, userID string, amountChange int) error
}

type trainerService interface {
    ScheduleTraining(ctx context.Context, trainingTime time.Time) error
    CancelTraining(ctx context.Context, trainingTime time.Time) error
}
```

Source: training_service.go on GitHub[16]

> **Note**
>
> Note that "user" and "trainer" in this context are not microservices, but application (business) concepts. It just happens that in this project, they live in the scope of microservices with the same names.

We move implementations of these interfaces to `adapters` as `UsersGrpc`[17] and `TrainerGrpc`[18]. As a bonus, the timestamp conversion now happens there as well, invisible to the application service.

[15] https://bit.ly/2NN67Kt
[16] https://bit.ly/3k7rPVO
[17] https://bit.ly/2P3KKFF
[18] https://bit.ly/2P3KKFF

Extracting the Application Logic

The code compiles, but our application service doesn't do much yet. Now is the time to extract the logic and put it in the proper place.

Finally, we can use the update function pattern from **The Repository Pattern** (Chapter 7) to extract the application logic out of the repository.

```
func (c TrainingService) CancelTraining(ctx context.Context, user auth.User, trainingUUID string) error {
    return c.repo.CancelTraining(ctx, trainingUUID, func(training Training) error {
        if user.Role != "trainer" && training.UserUUID != user.UUID {
            return errors.Errorf("user '%s' is trying to cancel training of user '%s'", user.UUID, training.UserUUID)
        }

        var trainingBalanceDelta int
        if training.CanBeCancelled() {
            // just give training back
            trainingBalanceDelta = 1
        } else {
            if user.Role == "trainer" {
                // 1 for cancelled training +1 fine for cancelling by trainer less than 24h before training
                trainingBalanceDelta = 2
            } else {
                // fine for cancelling less than 24h before training
                trainingBalanceDelta = 0
            }
        }

        if trainingBalanceDelta != 0 {
            err := c.userService.UpdateTrainingBalance(ctx, training.UserUUID, trainingBalanceDelta)
            if err != nil {
                return errors.Wrap(err, "unable to change trainings balance")
            }
        }

        err := c.trainerService.CancelTraining(ctx, training.Time)
        if err != nil {
            return errors.Wrap(err, "unable to cancel training")
        }

        return nil
    })
}
```

Source: training_service.go on GitHub[19]

The amount of logic suggests we might want to introduce a domain layer sometime in the future. For now, let's keep it as it is.

[19] https://bit.ly/3sfA8Su

I described the process for just a single `CancelTraining` method. Refer to the full diff[20] to see how I refactored all other methods.

Dependency Injection

How to tell the service which adapter to use? First, we define a simple constructor for the service.

```
func NewTrainingsService(
    repo trainingRepository,
    trainerService trainerService,
    userService userService,
) TrainingService {
    if repo == nil {
        panic("missing trainingRepository")
    }
    if trainerService == nil {
        panic("missing trainerService")
    }
    if userService == nil {
        panic("missing userService")
    }

    return TrainingService{
        repo:          repo,
        trainerService: trainerService,
        userService:   userService,
    }
}
```

Source: training_service.go on GitHub[21]

Then, in `main.go` we inject the adapter.

```
trainingsRepository := adapters.NewTrainingsFirestoreRepository(client)
trainerGrpc := adapters.NewTrainerGrpc(trainerClient)
usersGrpc := adapters.NewUsersGrpc(usersClient)

trainingsService := app.NewTrainingsService(trainingsRepository, trainerGrpc, usersGrpc)
```

Source: main.go on GitHub[22]

Using the `main` function is the most trivial way to inject dependencies. We'll look into the wire library[23] as the project becomes more complex in future chapters.

[20] https://bit.ly/2McYKM2
[21] https://bit.ly/2ZAMS9S
[22] https://bit.ly/3dxwJdt
[23] https://github.com/google/wire

Adding tests

Initially, the project had all layers mixed, and it wasn't possible to mock dependencies. The only way to test it was to use integration tests, with proper database and all services running.

While it's OK to cover some scenarios with such tests, they tend to be slower and not as fun to work with as unit tests. After introducing changes, I was able to cover `CancelTraining` with a unit tests suite[24].

I used the standard Go approach of table-driven tests to make all cases easy to read and understand.

```
{
    Name:     "return_training_balance_when_trainer_cancels",
    UserRole: "trainer",
    Training: app.Training{
        UserUUID: "trainer-id",
        Time:     time.Now().Add(48 * time.Hour),
    },
    ShouldUpdateBalance:   true,
    ExpectedBalanceChange: 1,
},
{
    Name:     "extra_training_balance_when_trainer_cancels_before_24h",
    UserRole: "trainer",
    Training: app.Training{
        UserUUID: "trainer-id",
        Time:     time.Now().Add(12 * time.Hour),
    },
    ShouldUpdateBalance:   true,
    ExpectedBalanceChange: 2,
},
```

Source: training_service_test.go on GitHub[25]

I didn't introduce any libraries for mocking. You can use them if you like, but your interfaces should usually be small enough to simply write dedicated mocks.

```
type trainerServiceMock struct {
    trainingsCancelled []time.Time
}

func (t *trainerServiceMock) CancelTraining(ctx context.Context, trainingTime time.Time) error {
    t.trainingsCancelled = append(t.trainingsCancelled, trainingTime)
    return nil
}
```

Source: training_service_test.go on GitHub[26]

Did you notice the unusually high number of not implemented methods in `repositoryMock`? That's because we use a single training service for all methods, so we need to implement the full interface, even when testing just one

[24] https://bit.ly/3soDvqh
[25] https://bit.ly/37uCGUv
[26] https://bit.ly/3k7rWRe

of them.

We'll improve it in **Basic CQRS** (Chapter 10).

What about the boilerplate?

You might be wondering if we didn't introduce too much boilerplate. The project indeed grew in size by code lines, but that by itself doesn't do any harm. **It's an investment in loose coupling[27] that will pay off as the project grows.**

Keeping everything in one package may seem easier at first, but having boundaries helps when you consider working in a team. If all your projects have a similar structure, onboarding new team members is straightforward. Consider how much harder it would be with all layers mixed (Mattermost's app package[28] is an example of this approach).

Handling application errors

One extra thing I've added is ports-agnostic errors with slugs[29]. They allow the application layer to return generic errors that can be handled by both HTTP and gRPC handlers.

```
if from.After(to) {
    return nil, errors.NewIncorrectInputError("date-from-after-date-to", "Date from after date to")
}
```

Source: hour_service.go on GitHub[30]

The error above translates to `401 Bad Request` HTTP response in ports. It includes a slug that can be translated on the frontend side and shown to the user. It's yet another pattern to avoid leaking implementation details to the application logic.

What else?

I encourage you to read through the full commit[31] to see how I refactored other parts of Wild Workouts.

You might be wondering how to enforce the correct usage of layers? Is it yet another thing to remember about in code reviews?

Luckily, it's possible to check the rules with static analysis. You can check your project with Robert's go-cleanarch[32] linter locally or include it in your CI pipeline.

With layers separated, we're ready to introduce more advanced patterns.

In the next chapter, we show how to improve the project by applying CQRS.

[27] See Chapter 5: When to stay away from DRY
[28] https://github.com/mattermost/mattermost-server/tree/master/app
[29] https://bit.ly/3bykevF
[30] https://bit.ly/3pBCSaL
[31] https://bit.ly/2McYKM2
[32] https://github.com/roblaszczak/go-cleanarch

If you'd like to read more on Clean Architecture, see Why using Microservices or Monolith can be just a detail?[33].

[33]https://threedots.tech/post/microservices-or-monolith-its-detail/

Chapter 10

Basic CQRS

Robert Laszczak

It's highly likely you know at least one service that:

- has one big, unmaintainable model that is hard to understand and change,
- or where work in parallel on new features is limited,
- or can't be scaled optimally.

But often, bad things come in threes. It's not uncommon to see services with all these problems.

What is an idea that comes to mind first for solving these issues? Let's split it into more microservices!

Unfortunately, without proper research and planning, the situation after blindly refactoring may be actually worse than before:

- **business logic and flow may become even harder to understand** – a complex logic is often easier to understand if it's in one place,
- **distributed transactions** – things are sometimes together for a reason; a big transaction in one database is much faster and less complex than distributed transaction across multiple services,
- **adding new changes may require extra coordination**, if one of the services is owned by another team.

To be totally clear – I'm not an enemy of microservices. **I'm just against blindly applying microservices in a way that introduces unneeded complexity and mess instead of making our lives easier.**

Another approach is using CQRS (Command Query Responsibility Segregation) with previously described Clean Architecture[1]. **It can solve the mentioned problems in a much simpler way.**

[1] See Chapter 9: Clean Architecture

Figure 10.1: Microservices are useful, but they will not solve all your issues…

Isn't CQRS a complex technique?

Isn't CQRS one of these C#/Java/über enterprise patterns that are hard to implement, and make a big mess in the code? A lot of books, presentations, and articles describe CQRS as a very complicated pattern. But it is not the case.

In practice, CQRS is a very simple pattern that doesn't require a lot of investment. It can be easily extended with more complex techniques like event-driven architecture, event-sourcing, or polyglot persistence. But they're not always needed. Even without applying any extra patterns, CQRS can offer better decoupling, and code structure that is easier to understand.

When to not use CQRS in Go? How to get all benefits from CQRS? You can learn all that in this chapter.

Like always, I will do it by refactoring Wild Workouts[2] application,

How to implement basic CQRS in Go

CQRS (Command Query Responsibility Segregation) was initially described by Greg Young[3]. **It has one simple assumption: instead of having one big model for reads and writes, you should have two separate models. One for writes and one for reads.** It also introduces concepts of *command* and *query*, and leads to splitting application services into two separate types: command and query handlers.

Command vs Query

In simplest words: **a Query should not modify anything, just return the data. A command is the opposite one: it should make changes in the system, but not return any data.** Thanks to that, our queries can be cached more efficiently, and we lower the complexity of commands.

It may sound like a serious constraint, but in practice, it is not. Most of the operations that we execute are reads or writes. Very rarely, both.

[2] https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example
[3] https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

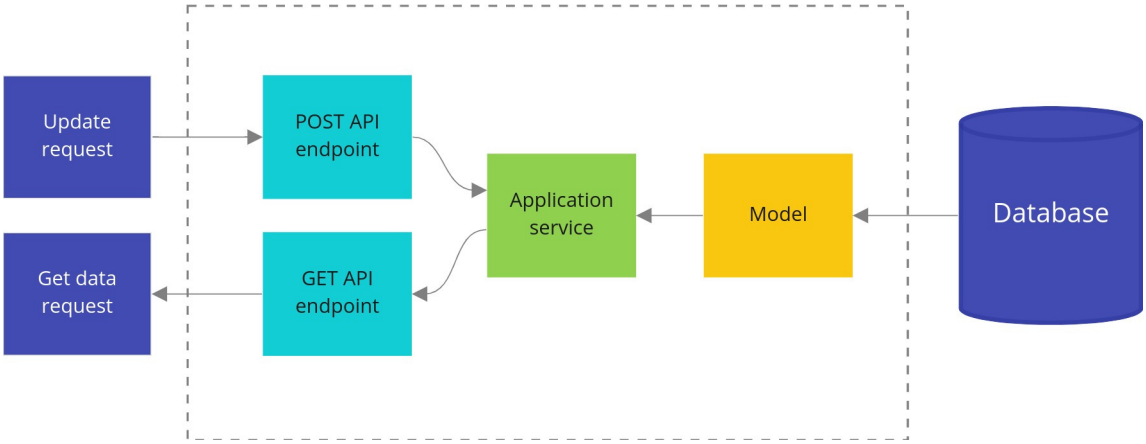


Figure 10.2: Standard, non-CQRS architecture

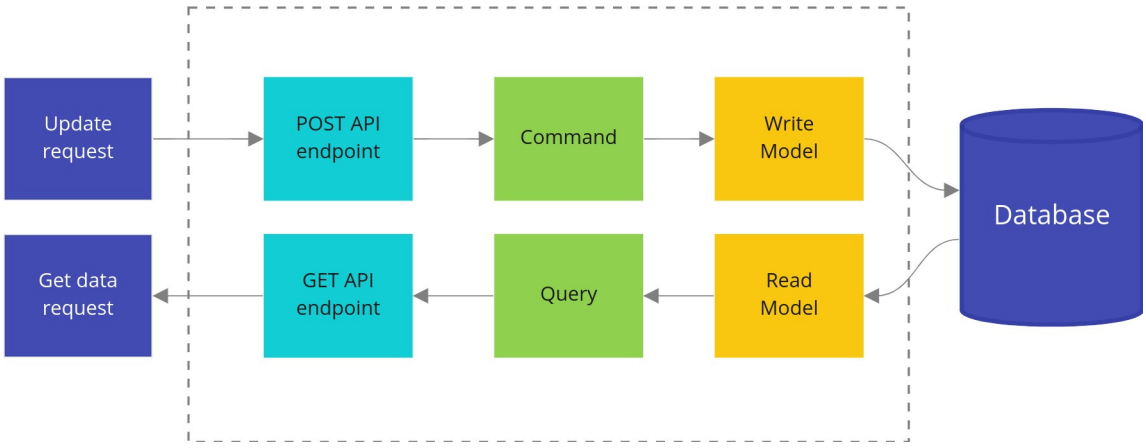


Figure 10.3: CQRS architecture

Of course, for a query, we don't consider side effects like logs, or metrics as modifying anything. For commands, it is also a perfectly normal thing to return an error.

> **Note**
>
> As with most rules, it is ok to break them... as long as you perfectly **understand why they were introduced and what tradeoffs** you make. In practice, you rarely need to break these rules. I will share examples at the end of the chapter.

How does the most basic implementation look in practice? In the previous chapter, Miłosz introduced an application service that executes application use cases. Let's start by cutting this service into separate command and query handlers.

ApproveTrainingReschedule command

Previously, the training reschedule was approved from the application service `TrainingService`.

```
- func (c TrainingService) ApproveTrainingReschedule(ctx context.Context, user auth.User, trainingUUID string) error
 ↪  {
-  return c.repo.ApproveTrainingReschedule(ctx, trainingUUID, func(training Training) (Training, error) {
-     if training.ProposedTime == nil {
-        return Training{}, errors.New("training has no proposed time")
-     }
-     if training.MoveProposedBy == nil {
-        return Training{}, errors.New("training has no MoveProposedBy")
-     }
-     if *training.MoveProposedBy == "trainer" && training.UserUUID != user.UUID {
-        return Training{}, errors.Errorf("user '%s' cannot approve reschedule of user '%s'", user.UUID,
 ↪  training.UserUUID)
-     }
-     if *training.MoveProposedBy == user.Role {
-        return Training{}, errors.New("reschedule cannot be accepted by requesting person")
-     }
-
-     training.Time = *training.ProposedTime
-     training.ProposedTime = nil
-
-     return training, nil
-  })
- }
```

Source: 8d9274811559399461aa9f6bf3829316b8ddfb63 on GitHub[4]

There were some magic validations there. They are now done in the domain layer. I also found out that we forgot to call the external `trainer` service to move the training. Oops. Let's refactor it to the CQRS approach.

[4] https://bit.ly/3dwArEj

Note

Because CQRS works best with applications following Domain-Driven Design, during refactoring towards CQRS I refactored existing models to DDD Lite as well. DDD Lite is described in more detail in **Domain-Driven Design Lite** (Chapter 6).

We start the implementation of a *command* with the command structure definition. That structure provides all data needed to execute this command. If a command has only one field, you can skip the structure and just pass it as a parameter.

It's a good idea to use types defined by domain in the command, like `training.User` in that case. We don't need to do any casting later, and we have type safety assured. **It can save us a lot of issues with string parameters passed in wrong order.**

```
package command

// ...

type ApproveTrainingReschedule struct {
   TrainingUUID string
   User         training.User
}
```

Source: approve_training_reschedule.go on GitHub[5]

The second part is a *command handler* that knows how to execute the command.

```
package command

// ...

type ApproveTrainingRescheduleHandler struct {
   repo           training.Repository
   userService    UserService
   trainerService TrainerService
}

// ...

func (h ApproveTrainingRescheduleHandler) Handle(ctx context.Context, cmd ApproveTrainingReschedule) (err error) {
    defer func() {
        logs.LogCommandExecution("ApproveTrainingReschedule", cmd, err)
    }()

    return h.repo.UpdateTraining(
        ctx,
        cmd.TrainingUUID,
        cmd.User,
        func(ctx context.Context, tr *training.Training) (*training.Training, error) {
```

[5]https://bit.ly/3dxwyij

```
            originalTrainingTime := tr.Time()

            if err := tr.ApproveReschedule(cmd.User.Type()); err != nil {
                return nil, err
            }

            err := h.trainerService.MoveTraining(ctx, tr.Time(), originalTrainingTime)
            if err != nil {
                return nil, err
            }

            return tr, nil
        },
    )
}
```

Source: approve_training_reschedule.go on GitHub[6]

The flow is much easier to understand now. You can clearly see that we approve a reschedule of a persisted `*training.Training`, and if it succeeds, we call the external `trainer` service. Thanks to techniques described in **Domain-Driven Design Lite** (Chapter 6), the command handler doesn't need to know when it can perform this operation. It's all handled by our domain layer.

This clear flow is even more visible in more complex commands. Fortunately, the current implementation is really straightforward. That's good. **Our goal is not to create complicated, but simple software.**

If CQRS is the standard way of building applications in your team, it also speeds up learning the service by your teammates who don't know it. You just need a list of available commands and queries, and to quickly take a look at how their execution works. Jumping like crazy through random places in code is not needed.

This is how it looks like in one of my team's most complex services:

You may ask - shouldn't it be cut to multiple services? **In practice, it would be a terrible idea.** A lot of operations here need to be transitionally consistent. Splitting it to separate services would involve a couple of distributed transactions (*Sagas*). It would make this flow much more complex, harder to maintain, and debug. It's not the best deal.

It's also worth mentioning that all of these operations are not very complex. **Complexity is scaling horizontally excellently here.** We will cover the extremely important topic of splitting microservices more in-depth soon. Did I already mention that we messed it up in Wild Workouts on purpose?

But let's go back to our command. It's time to use it in our HTTP port. It's available in `HttpServer` via injected `Application` structure, which contains all of our commands and queries handlers.

```
package app

import (
   "github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainings/app/command"
```

[6] https://bit.ly/3scHx4W

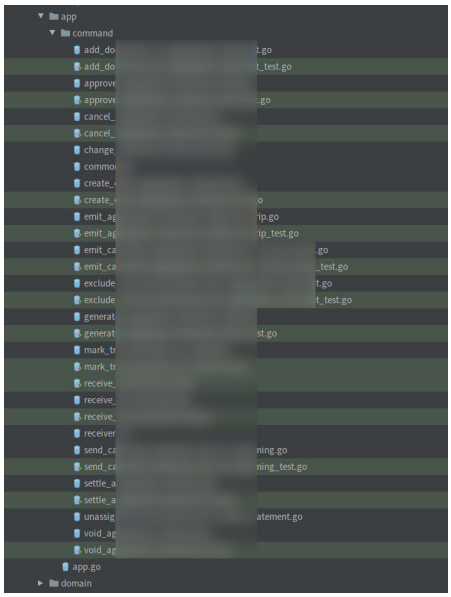


Figure 10.4: Example application layer of one service at Karhoo.

```
    "github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainings/app/query"
)

type Application struct {
    Commands Commands
    Queries  Queries
}

type Commands struct {
    ApproveTrainingReschedule command.ApproveTrainingRescheduleHandler
    CancelTraining            command.CancelTrainingHandler
    // ...
```

Source: app.go on GitHub[7]

```
type HttpServer struct {
    app app.Application
}

// ...

func (h HttpServer) ApproveRescheduleTraining(w http.ResponseWriter, r *http.Request) {
    trainingUUID := chi.URLParam(r, "trainingUUID")

    user, err := newDomainUserFromAuthUser(r.Context())
    if err != nil {
        httperr.RespondWithSlugError(err, w, r)
        return
```

[7] https://bit.ly/3dykLAq

```
    }

    err = h.app.Commands.ApproveTrainingReschedule.Handle(r.Context(), command.ApproveTrainingReschedule{
        User:         user,
        TrainingUUID: trainingUUID,
    })
    if err != nil {
        httperr.RespondWithSlugError(err, w, r)
        return
    }
}
```

Source: http.go on GitHub[8]

The command handler can be called in that way from any port: HTTP, gRPC, or CLI. It's also useful for executing migrations and loading fixtures[9] (we already do it in Wild Workouts).

RequestTrainingReschedule command

Some command handlers can be very simple.

```
func (h RequestTrainingRescheduleHandler) Handle(ctx context.Context, cmd RequestTrainingReschedule) (err error) {
    defer func() {
        logs.LogCommandExecution("RequestTrainingReschedule", cmd, err)
    }()

    return h.repo.UpdateTraining(
        ctx,
        cmd.TrainingUUID,
        cmd.User,
        func(ctx context.Context, tr *training.Training) (*training.Training, error) {
            if err := tr.UpdateNotes(cmd.NewNotes); err != nil {
                return nil, err
            }

            tr.ProposeReschedule(cmd.NewTime, cmd.User.Type())

            return tr, nil
        },
    )
}
```

Source: request_training_reschedule.go on GitHub[10]

It may be tempting to skip this layer for such simple cases to save some boilerplate. It's true, but you need to remember that **writing code is always much cheaper than the maintenance. Adding this simple type is 3 minutes of work. People who will read and extend this code later will appreciate that effort.**

[8] https://bit.ly/3sgPS7p
[9] https://bit.ly/2ZFmXOd
[10] https://bit.ly/3s6BgHM

AvailableHoursHandler query

Queries in the application layer are usually pretty boring. In the most common case, we need to write a *read model interface* (`AvailableHoursReadModel`) that defines how we can query the data.

Commands and queries are also a great place for all cross-cutting concerns[11], like logging and instrumentation. Thanks to putting that here, we are sure that performance is measured in the same way whether it's called from HTTP or gRPC port.

```
package query

// ...

type AvailableHoursHandler struct {
    readModel AvailableHoursReadModel
}

type AvailableHoursReadModel interface {
    AvailableHours(ctx context.Context, from time.Time, to time.Time) ([]Date, error)
}

// ...

type AvailableHours struct {
    From time.Time
    To   time.Time
}

func (h AvailableHoursHandler) Handle(ctx context.Context, query AvailableHours) (d []Date, err error) {
    start := time.Now()
    defer func() {
        logrus.
            WithError(err).
            WithField("duration", time.Since(start)).
            Debug("AvailableHoursHandler executed")
    }()

    if query.From.After(query.To) {
        return nil, errors.NewIncorrectInputError("date-from-after-date-to", "Date from after date to")
    }

    return h.readModel.AvailableHours(ctx, query.From, query.To)
}
```

Source: available_hours.go on GitHub[12]

We also need to define data types returned by the query. In our case, it's `query.Date`.

[11] https://en.wikipedia.org/wiki/Cross-cutting_concern
[12] https://bit.ly/3udkEAd

> **Note**
>
> To understand why we don't use structures generated from OpenAPI, see **When to stay away from DRY** (Chapter 5) and **Clean Architecture** (Chapter 9).

```
package query

import (
    "time"
)

type Date struct {
    Date         time.Time
    HasFreeHours bool
    Hours        []Hour
}

type Hour struct {
    Available            bool
    HasTrainingScheduled bool
    Hour                 time.Time
}
```

Source: types.go on GitHub[13]

Our query model is more complex than the domain `hour.Hour` type. It's a common scenario. Often, it's driven by the UI of the website, and it's more efficient to generate the most optimal responses on the backend side.

As the application grows, differences between domain and query models may become bigger. **Thanks to the separation and decoupling, we can independently make changes in both of them.** This is critical for keeping fast development in the long term.

```
package hour

type Hour struct {
    hour time.Time

    availability Availability
}
```

Source: hour.go on GitHub[14]

But from where `AvailableHoursReadModel` gets the data? For the application layer, it is fully transparent and not relevant. This allows us to add performance optimizations in the future, touching just one part of the application.

[13] https://bit.ly/3dAqvtg
[14] https://bit.ly/2M9PXut

Note

If you are not familiar with the concept of *ports and adapters*, I highly recommend reading **Clean Architecture** (Chapter 9).

In practice, the current implementation gets the data **from our write models database**. You can find the `AllTrainings`[15] read model implementation[16] and tests[17] for `DatesFirestoreRepository` in the adapters layer.

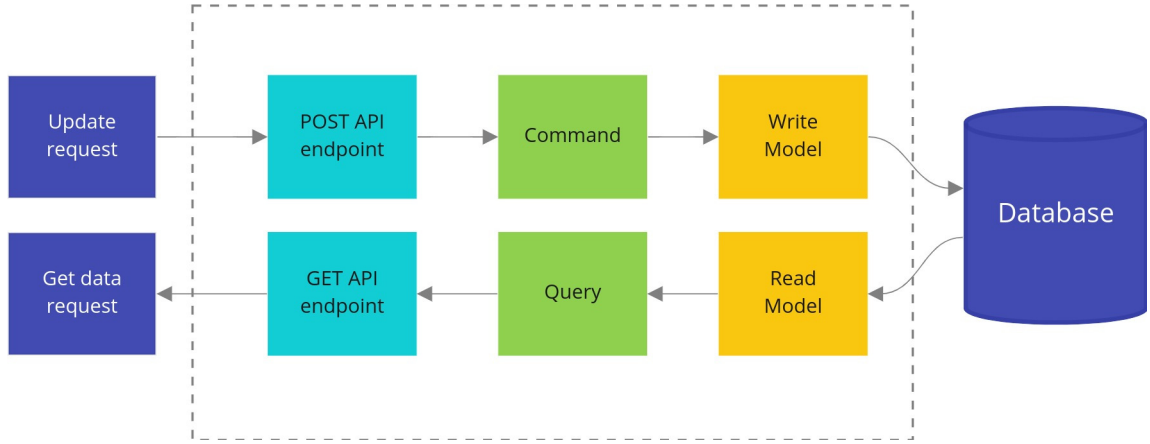


Figure 10.5: Data for our queries is currently queried from the same database where write models are stored.

If you read about CQRS earlier, it is often recommended to use a separate database built from events for queries. It may be a good idea, but in very specific cases. I will describe it in the *Future optimizations* section. In our case, it's sufficient to just get data from the write models database.

HourAvailabilityHandler query

We don't need to add a *read model* interface for every query. It's also fine to use the domain repository and pick the data that we need.

```
import (
    "context"
    "time"

    "github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/internal/trainer/domain/hour"
)

type HourAvailabilityHandler struct {
    hourRepo hour.Repository
}

func (h HourAvailabilityHandler) Handle(ctx context.Context, time time.Time) (bool, error) {
```

[15] https://bit.ly/3snBZoh
[16] https://bit.ly/3snBZoh
[17] https://bit.ly/3snBZoh

```
    hour, err := h.hourRepo.GetHour(ctx, time)
    if err != nil {
        return false, err
    }

    return hour.IsAvailable(), nil
}
```

Source: hour_availability.go on GitHub[18]

Naming

Naming is one of the most challenging and most essential parts of software development. In **Domain-Driven Design Lite** (Chapter 6) I described a rule that says you should stick to the language that is as close as it can be to how non-technical people (often referred to as "business") talk. It also applies to Commands and Queries names.

You should avoid names like "Create training" or "Delete training". **This is not how business and users understand your domain. You should instead use "Schedule training" and "Cancel training".**

We will cover this topic deeper in a chapter about Ubiquitous Language. Until then, just go to your business people and listen how they call operations. Think twice if any of your command names really need to start with "Create/Delete/Update".

Future optimizations

Basic CQRS gives some advantages like **better code organisation, decoupling, and simplifying models**. There is also one, even more important advantage. It is **the ability to extend CQRS with more powerful and complex** patterns.

Async commands

Some commands are slow by nature. They may be doing some external calls or some heavy computation. In that case, we can introduce *Asynchronous Command Bus*, which executes the command in the background.

Using asynchronous commands has some additional infrastructure requirements, like having a queue or a pub/sub. Fortunately, the Watermill[19] library can help you handle this in Go. You can find more details in the Watermill CQRS documentation[20]. (BTW We are the authors of Watermill as well Feel free to contact us if something's not clear there!)

A separate database for queries

Our current implementation uses the same database for reads (queries) and writes (commands). If we would need to provide more complex queries or have really fast reads, we could use the *polyglot persistence* technique. The idea

[18] https://bit.ly/3azs1Km
[19] https://github.com/ThreeDotsLabs/watermill
[20] https://watermill.io/docs/cqrs/?utm_source=introducing-cqrs-art

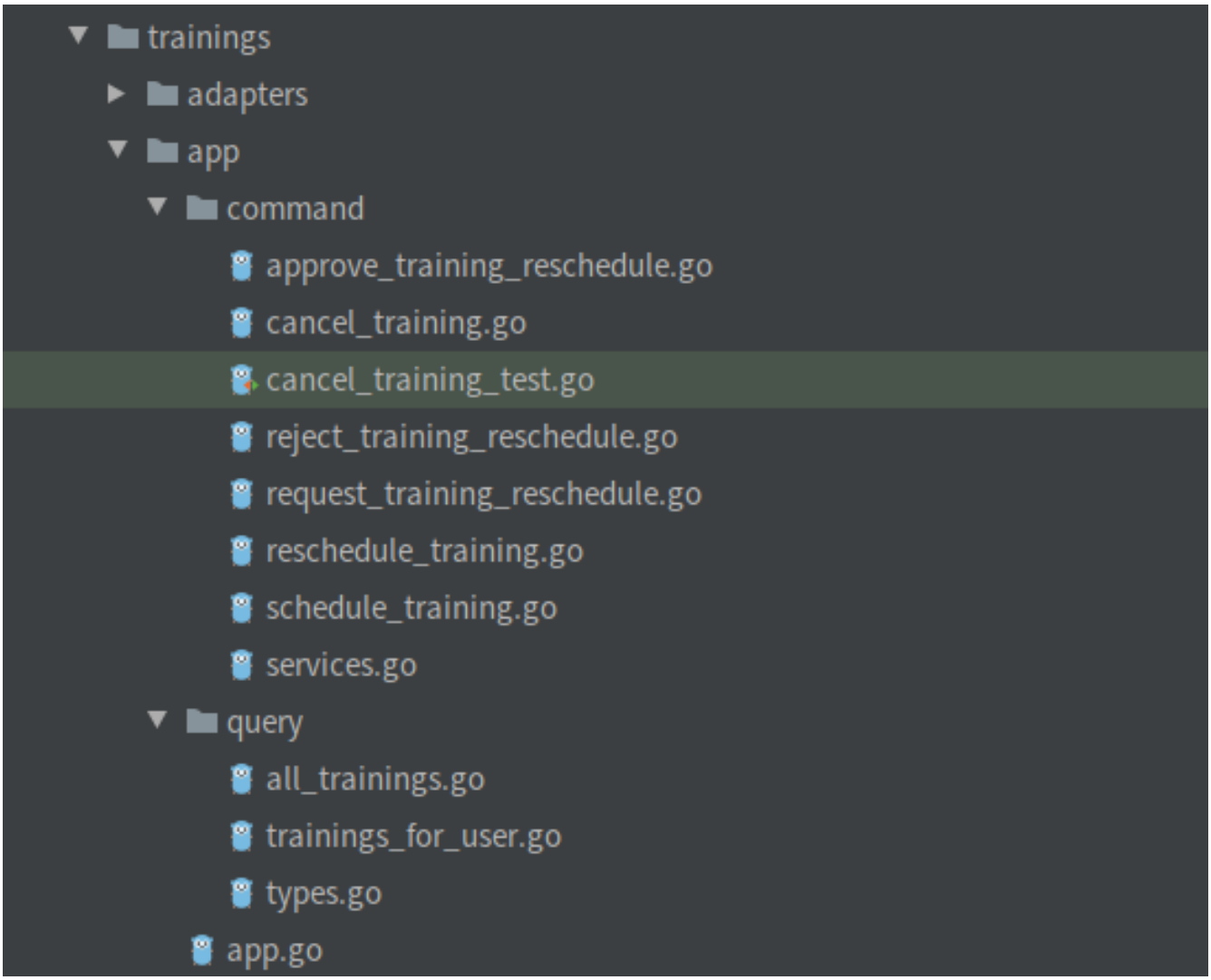


Figure 10.6: All commands and queries of the trainings service

is to duplicate queried data in a more optimal format in another database. For example, we could use Elastic to index some data that can be searched and filtered more easily.

Data synchronization, in this case, can be done via *events*. One of the most important implications of this approach is eventual consistency. You should ask yourself if it's an acceptable tradeoff in your system. If you are not sure, you can just start without polyglot persistence and migrate later. It's good to defer key decisions like this one.

An example implementation is described in the Watermill CQRS documentation[21] as well. Maybe with time, we will introduce it also in Wild Workouts, who knows?

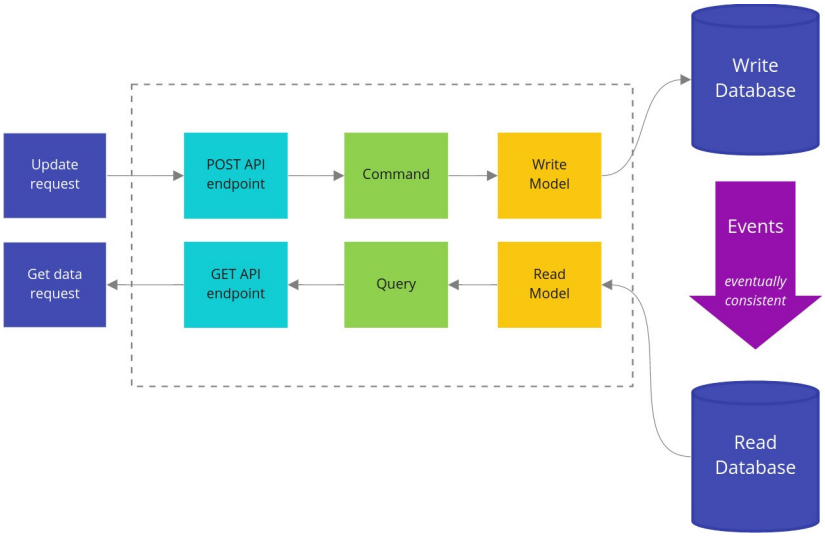


Figure 10.7: CQRS with polyglot persistence

Event-Sourcing

If you work in a domain with strict audit requirements, you should definitely check out the *event sourcing* technique. For example, I'm currently working in the financial domain, and *event sourcing* is our default persistence choice. It provides out-of-the-box audit and helps with reverting some bug implications.

CQRS is often described together with *event sourcing*. The reason is that by design in event-sourced systems, we don't store the model in a format ready for reads (queries), but just a list of events used by writes (commands). In other words, it's harder to provide any API responses.

Thanks to the separation of *command* and *query* models, it's not really a big problem. Our read models for queries live independently by design.

There are also a lot more advantages of event sourcing, that are visible in a financial systems. But let's leave it for another chapter. Until then, you can check the Ebook from Greg Young – Versioning in an Event Sourced System[22]. The same Greg Young who described CQRS.

[21] https://watermill.io/docs/cqrs/?utm_source=cqrs-art#building-a-read-model-with-the-event-handler
[22] https://leanpub.com/esversioning

When to not use CQRS?

CQRS is not a silver bullet that fits everywhere perfectly. A good example is authorization. You provide a login and a password, and in return, you get confirmation if you succeeded and maybe some token.

If your application is a simple CRUD that receives and returns the same data, it's also not the best case for CQRS. That's the reason why `users` microservice in Wild Workouts doesn't use Clean Architecture and CQRS. In simple, data-oriented services, these patterns usually don't make sense. On the other hand, you should keep an eye on services like that. If you notice the logic grows and development is painful, maybe it's time for some refactoring?

Returning created entity via API with CQRS

I know that some people have a problem with using CQRS for the REST API that returns the created entity as the response of a POST request. Isn't it against CQRS? Not really! You can solve it in two ways:

1. Call the command in the HTTP port and after it succeeds, call the query to get the data to return,
2. Instead of returning the created entity, return `204` HTTP code with header `content-location` set to the created resource URL.

The second approach is IMO better because it doesn't require to always query for the created entity (even if the client doesn't need this data). With the second approach, the client will only follow the link if it's needed. It can also be cached with that call.

The only question is how to get created entity's ID? A common practice is to provide the UUID of the entity to be created in the command.

This approach's advantage is that it will still work as expected if the command handler is asynchronous. In case you don't want to work with UUIDs, as a last resort you can return the ID from the handler – it won't be the end of the world.

```
cmd := command.ScheduleTraining{
    TrainingUUID: uuid.New().String(),
    UserUUID:     user.UUID,
    UserName:     user.DisplayName,
    TrainingTime: postTraining.Time,
    Notes:        postTraining.Notes,
}
err = h.app.Commands.ScheduleTraining.Handle(r.Context(), cmd)
if err != nil {
    httperr.RespondWithSlugError(err, w, r)
    return
}

w.Header().Set("content-location", "/trainings/" + cmd.TrainingUUID)
w.WriteHeader(http.StatusNoContent)
```

Source: http.go on GitHub[23]

[23] https://bit.ly/37PNVr9

You can now put CQRS in your resume!

We did it – we have a basic CQRS implementation in Wild Workouts. You should also have an idea of how you can extend the application in the future.

While preparing the code for this chapter, I also refactored the `trainer` service towards DDD. I will cover this in the next chapter. Although the entire diff of that refactoring is already available on our GitHub repository[24].

Having every command handler as a separate type also helps with testing, as it's easier to build dependencies for them. This part is covered by Miłosz in **Tests Architecture** (Chapter 12).

[24] https://bit.ly/3udkQiV

Chapter 11

Combining DDD, CQRS, and Clean Architecture

Robert Laszczak

In the previous chapters, we introduced techniques like DDD Lite, CQRS, and Clean Architecture. Even if using them alone is beneficial, they work the best together. Like Power Rangers. Unfortunately, it is not easy to use them together in a real project. **In this chapter, we will show how to connect DDD Lite, CQRS, and Clean Architecture in the most pragmatic and efficient way.**

Why should I care?

Working on a programming project is similar to planning and building a residential district. If you know that the district will be expanding in the near future, you need to keep space for future improvements. Even if at the beginning it may look like a waste of space. You should keep space for future facilities like residential blocks, hospitals, and temples. **Without that, you will be forced to destroy buildings and streets to make space for new buildings.** It's much better to think about that earlier.

The situation is the same with the code. If you know that the project will be developed for longer than 1 month, you should keep the long term in mind from the beginning. **You need to create your code in a way that will not block your future work. Even if at the beginning it may look like over-engineering and a lot of extra boilerplate, you need to keep in mind the long term.**

It doesn't mean that you need to plan every feature that you will implement in the future – it's actually the opposite one. This approach helps to adapt to new requirements or changing understanding of our domain. Big up front design is not needed here. It's critical in current times, when the world is changing really fast and who can't adapt to these changes can get simply out of business.

This is exactly what these patterns give you when they are combined – the ability to keep constant development speed. Without destroying and touching existing code too much.

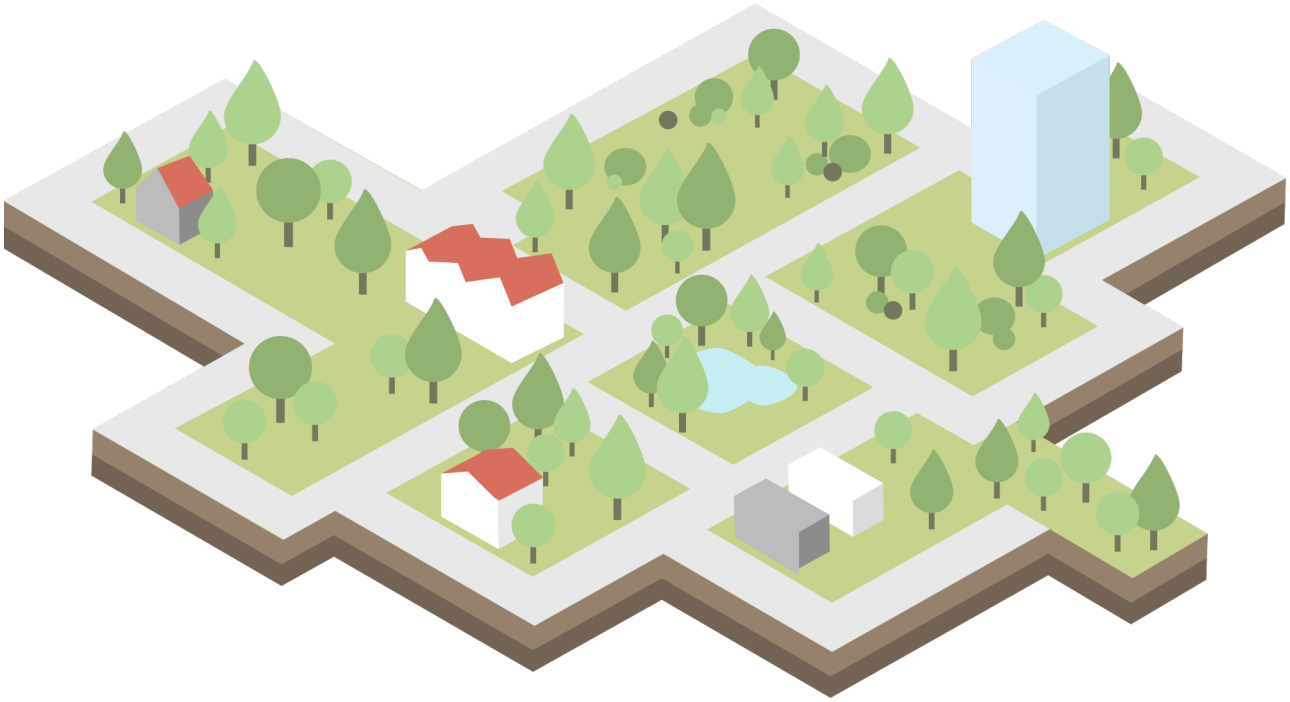


Figure 11.1: Empty district

Figure 11.2: Full district

Does it require more thinking and planning? Is it a more challenging way? Do you need to have extra knowledge to do that? Sure! **But the long term result is worth that!** Fortunately, you are in the right place to learn that.

But let's leave the theory behind us. Let's go to the code. In this chapter, we will skip reasonings for our design choices. We described these already in the previous chapters. If you did not read them yet, I recommend doing it – you will understand this chapter better.

Like in previous chapters, we will base our code on refactoring a real open-source project. This should make the examples more realistic and applicable to your projects.

Are you ready?

Let's refactor

Let's start our refactoring with the Domain-First[1] approach. We will start with introduction of a domain layer[2]. Thanks to that, we will be sure that implementation details do not affect our domain code. **We can also put all our efforts into understanding the business problem. Not on writing boring database queries and API endpoints.**

Domain-First approach works good for both rescue (refactoring) and greenfield projects.

To start building my domain layer, I needed to identify what the application is actually doing. This chapter will focus on refactoring of `trainings`[3] Wild Workouts microservice. I started with identifying use cases handled by the application. After previous refactoring to Clean Architecture[4]. When I work with a messy application, I look at RPC and HTTP endpoints to find supported use cases.

One of functionalities that I identified is the **approval of training reschedule**. In Wild Workouts, a training reschedule approval is required if it was requested less than 24h before its date. If a reschedule is requested by the attendee, the approval needs to be done by the trainer. When it's requested by the trainer, it needs to be accepted by the attendee.

```
- func (c TrainingService) ApproveTrainingReschedule(ctx context.Context, user auth.User, trainingUUID string) error
 ↪  {
-  return c.repo.ApproveTrainingReschedule(ctx, trainingUUID, func(training Training) (Training, error) {
-     if training.ProposedTime == nil {
-        return Training{}, errors.New("training has no proposed time")
-     }
-     if training.MoveProposedBy == nil {
-        return Training{}, errors.New("training has no MoveProposedBy")
-     }
-     if *training.MoveProposedBy == "trainer" && training.UserUUID != user.UUID {
-        return Training{}, errors.Errorf("user '%s' cannot approve reschedule of user '%s'", user.UUID,
 ↪  training.UserUUID)
-     }
-     if *training.MoveProposedBy == user.Role {
```

[1] See Chapter 6: Domain-Driven Design Lite
[2] See Chapter 9: Clean Architecture
[3] https://bit.ly/3bt5spP
[4] See Chapter 9: Clean Architecture

```
-         return Training{}, errors.New("reschedule cannot be accepted by requesting person")
-     }
-
-     training.Time = *training.ProposedTime
-     training.ProposedTime = nil
-
-     return training, nil
-  })
- }
```

Source: 8d9274811559399461aa9f6bf3829316b8ddfb63 on GitHub[5]

Start with the domain

Even if it doesn't look like the worst code you've seen in your life, functions like `ApproveTrainingReschedule` tend to get more complex over time. More complex functions mean more potential bugs during future development.

It's even more likely if we are new to the project, and we don't have the *"shaman knowledge"* about it. **You should always consider all the people who will work on the project after you, and make it resistant to be broken accidentally by them. That will help your project to not become the legacy that everybody is afraid to touch.** You probably hate that feeling when you are new to the project, and you are afraid to touch anything to not break the system.

It's not uncommon for people to change their job more often than every 2 years. That makes it even more critical for long-term project development.

If you don't believe that this code may become complex, I recommend checking the Git history of the worst place in the project you work on. In most cases, that worst code started with *"just a couple simple ifs"*. The more complex the code will be, the more difficult it will be to simplify it later. **We should be sensitive to emerging complexity and try to simplify it as soon as we can.**

`Training` domain entity

During analyzing the current use cases handled by the `trainings` microservice, I found that they are all related to a *training*. It is pretty natural to create a `Training` type to handle these operations.

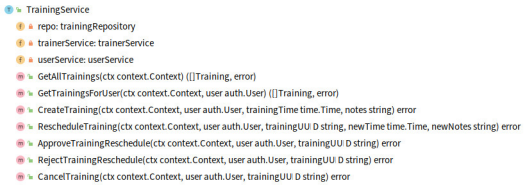


Figure 11.3: Methods of refactored TrainingService

[5] https://bit.ly/3udkQiV

noun == entity

Is it a valid approach to discover entities? Well, not really.
DDD provides tools that help us to model complex domains without guessing (*Strategic DDD Patterns*, *Aggregates*). We don't want to guess how our aggregates look like – we want to have tools to discover them.
Event Storming technique is extremely useful here… but it's a topic for an entire separate chapter.
The topic is complex enough to have a couple chapters about that. And this is what we will do shortly.
Does it mean that you should not use these techniques without Strategic DDD Patterns? Of course not! The current approach can be good enough for simpler projects. Unfortunately (or fortunately), not all projects are simple.

```
package training

// ...

type Training struct {
   uuid string

   userUUID string
   userName string

   time  time.Time
   notes string

   proposedNewTime time.Time
   moveProposedBy  UserType

   canceled bool
}
```

Source: training.go on GitHub[6]

All fields are private to provide encapsulation. This is critical to meet "always keep a valid state in the memory"[7] rule from the chapter about DDD Lite.

Thanks to the validation in the constructor and encapsulated fields, we are sure that `Training` is always valid. **Now, a person that doesn't have any knowledge about the project is not able to use it in a wrong way.**

The same rule applies to any methods provided by `Training`.

```
package training

func NewTraining(uuid string, userUUID string, userName string, trainingTime time.Time) (*Training, error) {
   if uuid == "" {
      return nil, errors.New("empty training uuid")
   }
```

[6] https://bit.ly/37x7Lal
[7] See Chapter 6: Domain-Driven Design Lite

```
    if userUUID == "" {
        return nil, errors.New("empty userUUID")
    }
    if userName == "" {
        return nil, errors.New("empty userName")
    }
    if trainingTime.IsZero() {
        return nil, errors.New("zero training time")
    }

    return &Training{
        uuid:     uuid,
        userUUID: userUUID,
        userName: userName,
        time:     trainingTime,
    }, nil
}
```

Source: training.go on GitHub[8]

Approve reschedule in the domain layer

As described in DDD Lite introduction[9], we build our domain with methods oriented on behaviours. Not on data. Let's model `ApproveReschedule` on our domain entity.

```
package training

// ...s

func (t *Training) IsRescheduleProposed() bool {
    return !t.moveProposedBy.IsZero() && !t.proposedNewTime.IsZero()
}

var ErrNoRescheduleRequested = errors.New("no training reschedule was requested yet")

func (t *Training) ApproveReschedule(userType UserType) error {
    if !t.IsRescheduleProposed() {
        return errors.WithStack(ErrNoRescheduleRequested)
    }

    if t.moveProposedBy == userType {
        return errors.Errorf(
            "trying to approve reschedule by the same user type which proposed reschedule (%s)",
            userType.String(),
        )
    }

    t.time = t.proposedNewTime
```

[8] https://bit.ly/3qEeQgP
[9] See Chapter 6: Domain-Driven Design Lite

```
    t.proposedNewTime = time.Time{}
    t.moveProposedBy = UserType{}

    return nil
}
```

Source: reschedule.go on GitHub[10]

Orchestrate with command

Now the application layer can be responsible only for the orchestration of the flow. There is no domain logic there. **We hide the entire business complexity in the domain layer. This was exactly our goal.**

For getting and saving a training, we use the Repository pattern[11].

```
package command

// ...

func (h ApproveTrainingRescheduleHandler) Handle(ctx context.Context, cmd ApproveTrainingReschedule) (err error) {
    defer func() {
        logs.LogCommandExecution("ApproveTrainingReschedule", cmd, err)
    }()

    return h.repo.UpdateTraining(
        ctx,
        cmd.TrainingUUID,
        cmd.User,
        func(ctx context.Context, tr *training.Training) (*training.Training, error) {
            originalTrainingTime := tr.Time()

            if err := tr.ApproveReschedule(cmd.User.Type()); err != nil {
                return nil, err
            }

            err := h.trainerService.MoveTraining(ctx, tr.Time(), originalTrainingTime)
            if err != nil {
                return nil, err
            }

            return tr, nil
        },
    )
}
```

Source: approve_training_reschedule.go on GitHub[12]

[10] https://bit.ly/2NLBncQ
[11] See Chapter 7: The Repository Pattern
[12] https://bit.ly/3k8VdLo

Refactoring of training cancelation

Let's now take a look at `CancelTraining` from `TrainingService`.

The domain logic is simple there: you can cancel a training up to 24h before its date. If it's less than 24h before the training, and you want to cancel it anyway:

- if you are the trainer, the attendee will have his training "back" plus one extra session (nobody likes to change plans on the same day!)
- if you are the attendee, you will lose this training

This is how the current implementation looks like:

```
- func (c TrainingService) CancelTraining(ctx context.Context, user auth.User, trainingUUID string) error {
-  return c.repo.CancelTraining(ctx, trainingUUID, func(training Training) error {
-     if user.Role != "trainer" && training.UserUUID != user.UUID {
-        return errors.Errorf("user '%s' is trying to cancel training of user '%s'", user.UUID, training.UserUUID)
-     }
-
-     var trainingBalanceDelta int
-     if training.CanBeCancelled() {
-        // just give training back
-        trainingBalanceDelta = 1
-     } else {
-        if user.Role == "trainer" {
-           // 1 for cancelled training +1 fine for cancelling by trainer less than 24h before training
-           trainingBalanceDelta = 2
-        } else {
-           // fine for cancelling less than 24h before training
-           trainingBalanceDelta = 0
-        }
-     }
-
-     if trainingBalanceDelta != 0 {
-        err := c.userService.UpdateTrainingBalance(ctx, training.UserUUID, trainingBalanceDelta)
-        if err != nil {
-           return errors.Wrap(err, "unable to change trainings balance")
-        }
-     }
-
-     err := c.trainerService.CancelTraining(ctx, training.Time)
-     if err != nil {
-        return errors.Wrap(err, "unable to cancel training")
-     }
-
-     return nil
-  })
- }
```

You can see some kind of "algorithm" for calculating training balance delta during cancelation. That's not a good sign in the application layer.

Logic like this one should live in our domain layer. **If you start to see some `if`'s related to logic in your**

application layer, you should think about how to move it to the domain layer. It will be easier to test and re-use in other places.

It may depend on the project, but often **domain logic is pretty stable after the initial development and can live unchanged for a long time**. It can survive moving between services, framework changes, library changes, and API changes. Thanks to that separation, we can do all these changes in a much safer and faster way.

Let's decompose the `CancelTraining` method to multiple, separated pieces. That will allow us to test and change them independently.

First of all, we need to handle cancelation logic and marking `Training` as canceled.

```
package training

func (t Training) CanBeCanceledForFree() bool {
   return t.time.Sub(time.Now()) >= time.Hour*24
}

var ErrTrainingAlreadyCanceled = errors.New("training is already canceled")

func (t *Training) Cancel() error {
   if t.IsCanceled() {
      return ErrTrainingAlreadyCanceled
   }

   t.canceled = true
   return nil
}
```

Source: cancel.go on GitHub[13]

Nothing really complicated here. That's good!

The second part that requires moving is the "algorithm" of calculating trainings balance after cancelation. In theory, we could put it to the `Cancel()` method, but IMO it would break the Single Responsibility Principle[14] and CQS[15]. And I like small functions.

But where to put it? Some object? A domain service? In some languages, like the one that starts with *J* and ends with *ava*, it would make sense. But in Go, it's good enough to just create a simple function.

```
package training

// CancelBalanceDelta return trainings balance delta that should be adjusted after training cancelation.
func CancelBalanceDelta(tr Training, cancelingUserType UserType) int {
   if tr.CanBeCanceledForFree() {
      // just give training back
      return 1
   }
```

[13] https://bit.ly/3dwBaoJ
[14] https://en.wikipedia.org/wiki/Single-responsibility_principle
[15] https://en.wikipedia.org/wiki/Command%E2%80%93query_separation

```
    switch cancelingUserType {
    case Trainer:
        // 1 for cancelled training +1 "fine" for cancelling by trainer less than 24h before training
        return 2
    case Attendee:
        // "fine" for cancelling less than 24h before training
        return 0
    default:
        panic(fmt.Sprintf("not supported user type %s", cancelingUserType))
    }
}
```

Source: cancel_balance.go on GitHub[16]

The code is now straightforward. **I can imagine that I could sit with any non-technical person and go through this code to explain how it works.**

What about tests? It may be a bit controversial, but IMO tests are redundant there. Test code would replicate the implementation of the function. Any change in the calculation algorithm will require copying the logic to the tests. I would not write a test there, but if you will sleep better at night – why not!

Moving CancelTraining to command

Our domain is ready, so let's now use it. We will do it in the same way as previously:

1. getting the entity from the repository,
2. orchestration of domain stuff,
3. calling external `trainer` service to cancel the training (this service is the point of truth of "trainer's calendar"),
4. returning entity to be saved in the database.

```
package command

// ...

func (h CancelTrainingHandler) Handle(ctx context.Context, cmd CancelTraining) (err error) {
    defer func() {
        logs.LogCommandExecution("CancelTrainingHandler", cmd, err)
    }()

    return h.repo.UpdateTraining(
        ctx,
        cmd.TrainingUUID,
        cmd.User,
        func(ctx context.Context, tr *training.Training) (*training.Training, error) {
            if err := tr.Cancel(); err != nil {
                return nil, err
            }
```

[16] https://bit.ly/3pInEAN

```
        if balanceDelta := training.CancelBalanceDelta(*tr, cmd.User.Type()); balanceDelta != 0 {
            err := h.userService.UpdateTrainingBalance(ctx, tr.UserUUID(), balanceDelta)
            if err != nil {
                return nil, errors.Wrap(err, "unable to change trainings balance")
            }
        }

        if err := h.trainerService.CancelTraining(ctx, tr.Time()); err != nil {
            return nil, errors.Wrap(err, "unable to cancel training")
        }

        return tr, nil
    },
  )
}
```

Source: cancel_training.go on GitHub[17]

Repository refactoring

The initial implementation of the repository was pretty tricky because of the custom method for every use case.

```
- type trainingRepository interface {
-  FindTrainingsForUser(ctx context.Context, user auth.User) ([]Training, error)
-  AllTrainings(ctx context.Context) ([]Training, error)
-  CreateTraining(ctx context.Context, training Training, createFn func() error) error
-  CancelTraining(ctx context.Context, trainingUUID string, deleteFn func(Training) error) error
-  RescheduleTraining(ctx context.Context, trainingUUID string, newTime time.Time, updateFn func(Training) (Training,
 ↪  error)) error
-  ApproveTrainingReschedule(ctx context.Context, trainingUUID string, updateFn func(Training) (Training, error))
 ↪  error
-  RejectTrainingReschedule(ctx context.Context, trainingUUID string, updateFn func(Training) (Training, error))
 ↪  error
- }
```

Thanks to introducing the `training.Training` entity, we can have a much simpler version, with one method for adding a new training and one for the update.

```
package training

// ...

type Repository interface {
    AddTraining(ctx context.Context, tr *Training) error

    GetTraining(ctx context.Context, trainingUUID string, user User) (*Training, error)

    UpdateTraining(
        ctx context.Context,
        trainingUUID string,
```

[17] https://bit.ly/3dtVKGs

```
      user User,
      updateFn func(ctx context.Context, tr *Training) (*Training, error),
   ) error
}
```

Source: repository.go on GitHub[18]

As in **The Repository Pattern** (Chapter 7), we implemented our repository using Firestore. We will also use Firestore in the current implementation. Please keep in mind that this is an implementation detail – you can use any database you want. In the previous chapter, we have shown example implementations using different databases.

```
package adapters

// ...

func (r TrainingsFirestoreRepository) UpdateTraining(
   ctx context.Context,
   trainingUUID string,
   user training.User,
   updateFn func(ctx context.Context, tr *training.Training) (*training.Training, error),
) error {
   trainingsCollection := r.trainingsCollection()

   return r.firestoreClient.RunTransaction(ctx, func(ctx context.Context, tx *firestore.Transaction) error {
      documentRef := trainingsCollection.Doc(trainingUUID)

      firestoreTraining, err := tx.Get(documentRef)
      if err != nil {
         return errors.Wrap(err, "unable to get actual docs")
      }

      tr, err := r.unmarshalTraining(firestoreTraining)
      if err != nil {
         return err
      }

      if err := training.CanUserSeeTraining(user, *tr); err != nil {
         return err
      }

      updatedTraining, err := updateFn(ctx, tr)
      if err != nil {
         return err
      }

      return tx.Set(documentRef, r.marshalTraining(updatedTraining))
   })
}
```

[18] https://bit.ly/3dykNIy

Source: trainings_firestore_repository.go on GitHub[19]

Connecting everything

How to use our code now? What about our ports layer? Thanks to the refactoring that Miłosz did in **Clean Architecture** (Chapter 9), our ports layer is decoupled from other layers. That's why, after this refactoring, it doesn't require almost any significant changes. We just call the application command instead of the application service.

```
package ports

// ...

type HttpServer struct {
   app app.Application
}

// ...

func (h HttpServer) CancelTraining(w http.ResponseWriter, r *http.Request) {
   trainingUUID := r.Context().Value("trainingUUID").(string)

   user, err := newDomainUserFromAuthUser(r.Context())
   if err != nil {
      httperr.RespondWithSlugError(err, w, r)
      return
   }

   err = h.app.Commands.CancelTraining.Handle(r.Context(), command.CancelTraining{
      TrainingUUID: trainingUUID,
      User:         user,
   })
   if err != nil {
      httperr.RespondWithSlugError(err, w, r)
      return
   }
}
```

Source: http.go on GitHub[20]

How to approach such refactoring in a real project?

It may not be obvious how to do such refactoring in a real project. It's hard to do a code review and agree on the team level on the refactoring direction.

From my experience, the best approach is Pair[21] or Mob[22] programming. Even if, at the beginning, you may feel

[19] https://bit.ly/3sfAdWi
[20] https://bit.ly/3bpwWwD
[21] https://en.wikipedia.org/wiki/Pair_programming
[22] https://en.wikipedia.org/wiki/Mob_programming

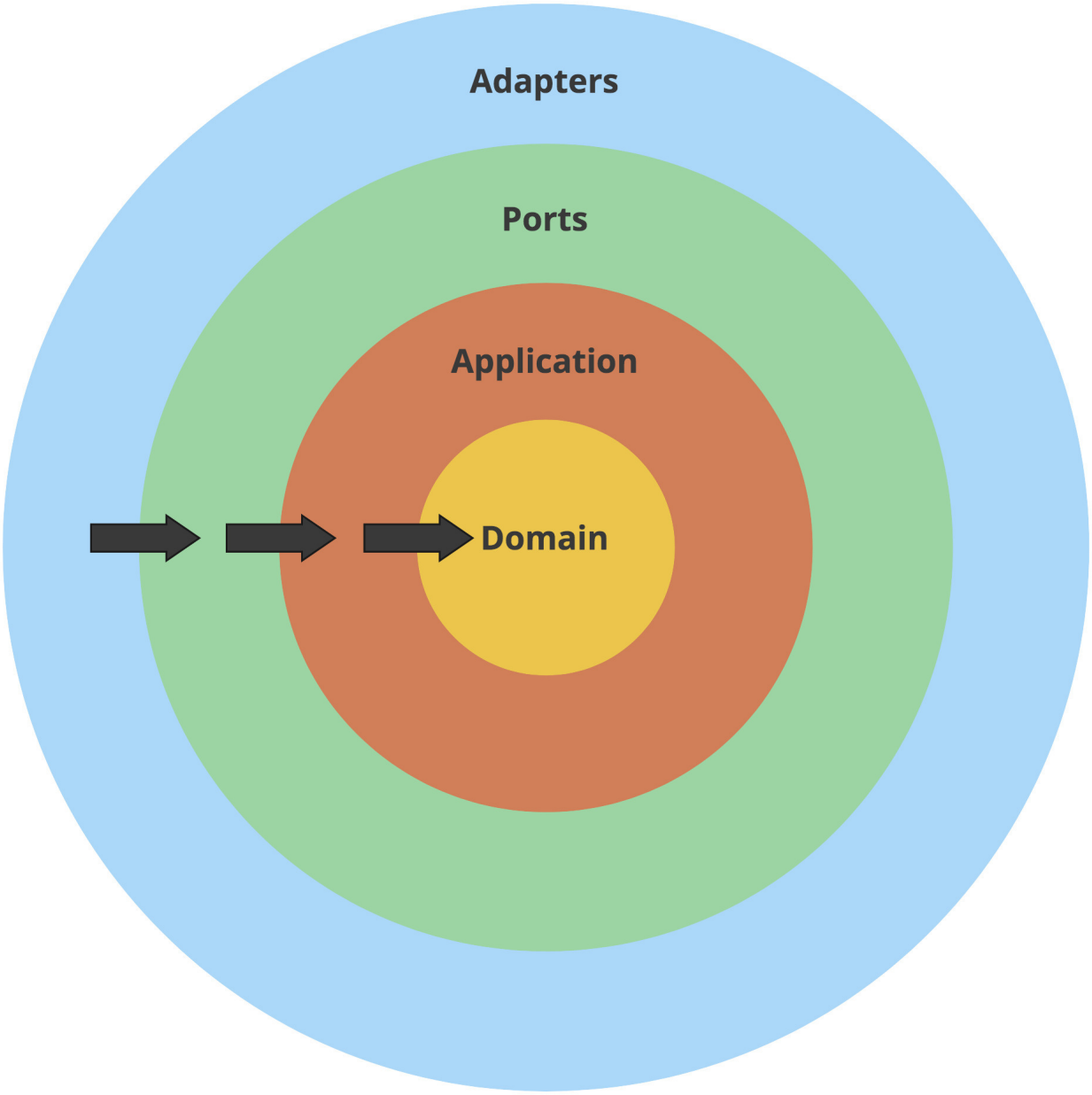


Figure 11.4: Clean/Hexagonal Architecture layers.

that it is a waste of time, the knowledge sharing and instant review will save a lot of time in the future. Thanks to great knowledge sharing, you can work much faster after the initial project or refactoring phase.

In this case, you should not consider the time lost for Mob/Pair programming. You should consider the time that you may lose because of not doing it. It will also help you finish the refactoring much faster because you will not need to wait for the decisions. You can agree on them immediately.

Mob and pair programming also work perfectly while implementing complex, greenfield projects. Knowledge sharing is especially important investment in that case. I've seen multiple times how this approach allowed to go very fast in the project in the long term.

When you are doing refactoring, it's also critical to agree on reasonable timeboxes. **And keep them.** You can quickly lose your stakeholders' trust when you spend an entire month on refactoring, and the improvement is not visible. It is also critical to integrate and deploy your refactoring as fast as you can. Perfectly, on a daily basis (if you can do it for non-refactoring work, I'm sure that you can do it for refactoring as well!). If your changes stay unmerged and undeployed for a longer time, it will increase the chance of breaking functionalities. It will also block any work in the refactored service or make changes harder to merge (it is not always possible to stop all other development around).

But when to know if the project is complex enough to use mob programming? Unfortunately, there is no magic formula for that. But there are questions that you should ask yourself: - do we understand the domain? - do we know how to implement that? - will it end up with a monstrous pull request that nobody will be able to review? - can we risk worse knowledge sharing while not doing mob/pair programming?

Summary

And we come to an end.

The entire diff for the refactoring is available on our Wild Workouts GitHub[23] (watch out, it's huge!).

I hope that after this chapter, you also see how all introduced patterns are working nicely together. If not yet, don't worry. **It took me 3 years to connect all the dots.** But it was worth the time spent. After I understood how everything is connected, I started to look at new projects in a totally different way. It allowed me and my teams to work more efficiently in the long-term.

It is also important to mention, that as all techniques, this combination is not a silver bullet. If you are creating project that is not complex and will be not touched any time soon after 1 month of development, probably it's enough to put everything to one `main` package. Just keep in mind, when this 1 month of development will become one year!

[23] https://bit.ly/3udkQiV

Chapter 12

Tests Architecture

Miłosz Smółka

Do you know the rare feeling when you develop a new application from scratch and can cover all lines with proper tests?

I said "rare" because most of the time, you will work with software with a long history, multiple contributors, and not so obvious testing approach. Even if the code uses good patterns, the test suite doesn't always follow.

Some projects have no modern development environment set up, so there are only unit tests for things that are easy to test. For example, they test single functions separately because it's hard to test the public API. The team needs to manually verify all changes, probably on some kind of staging environment. You know what happens when someone introduces changes and doesn't know they need to test it manually.

Other projects have no tests from the beginning. It allows quicker development by taking shortcuts, for example, keeping dependencies in the global state. When the team realizes the lack of tests causes bugs and slows them down, they decide to add them. But now, it's impossible to do it reasonably. So the team writes an end-to-end test suite with proper infrastructure in place.

End-to-end tests might give you some confidence, but **you don't want to maintain such a test suite**. It's hard to debug, takes a long time to test even the simplest change, and releasing the application takes hours. Introducing new tests is also not trivial in this scenario, so developers avoid it if they can.

I want to introduce some ideas that have worked for us so far and should help you avoid the scenarios above.

This chapter is not about which testing library is best or what tricks you can use (although I will show a few tips). It's closer to something I would call "test architecture". It's not only about "how", but also "where", "what", and "why".

There's been a lot of discussion on different types of tests, for example, the "test pyramid"[1] (Robert mentioned it in **High-Quality Database Integration Tests** (Chapter 8)). It's a helpful model to keep in mind. However, it's

[1] https://martinfowler.com/bliki/TestPyramid.html

also abstract, and you can't easily measure it. I want to take a more practical approach and show how to introduce a few kinds of tests in a Go project.

Why bother about tests?

But isn't the test code not as important as the rest of the application? Can't we just accept that keeping tests in good shape is hard and move on? Wouldn't it speed up the development?

If you've been following this series, you know we base all chapters on the Wild Workouts[2] application.

When I started writing this chapter, running tests locally didn't even work correctly for me, and this is a relatively new project. There's one reason this happened: we're not running tests in the CI pipeline.

It's a shock, but it seems **even a serverless, cloud-native application using the most popular, cutting edge technologies can be a mess in disguise.**

We know we should now add tests to the pipeline. It's common knowledge that this gives you the confidence to deploy the changes to production safely. However, there's also a cost.

Running tests will likely take a significant part of your pipeline's duration. If you don't approach their design and implementation with the same quality as the application code, you can realize it too late, with the pipeline taking one hour to pass and randomly failing on you. **Even if your application code is well designed, the tests can become a bottleneck of delivering the changes**.

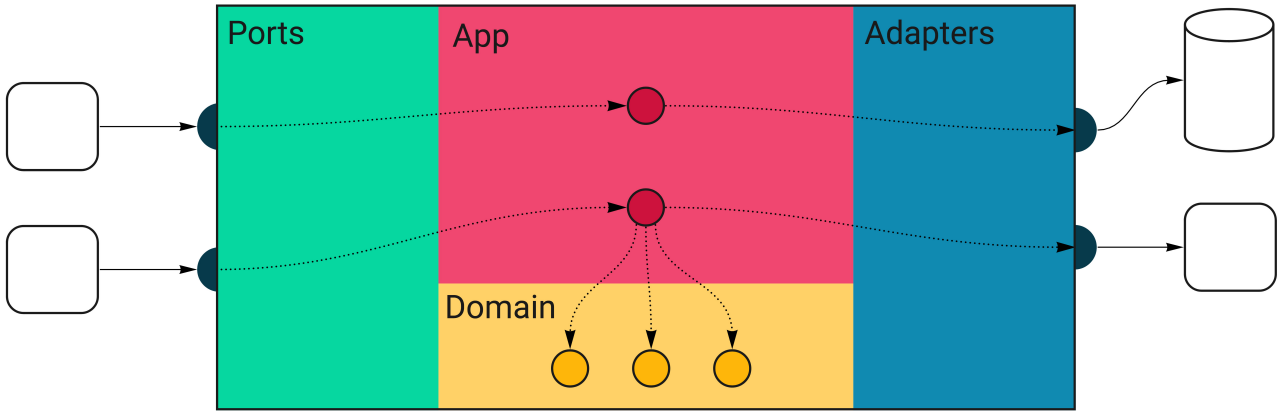
The Layers

We're now after a few refactoring sessions of the project. We introduced patterns like Repository[3]. With the solid separation of concerns, we can much easier reason about particular parts of the project.

Let's revisit the concept of layers we've introduced in previous chapters. If you didn't have a chance to read these earlier, I recommend doing so before you continue — it'll help you better understand this chapter.

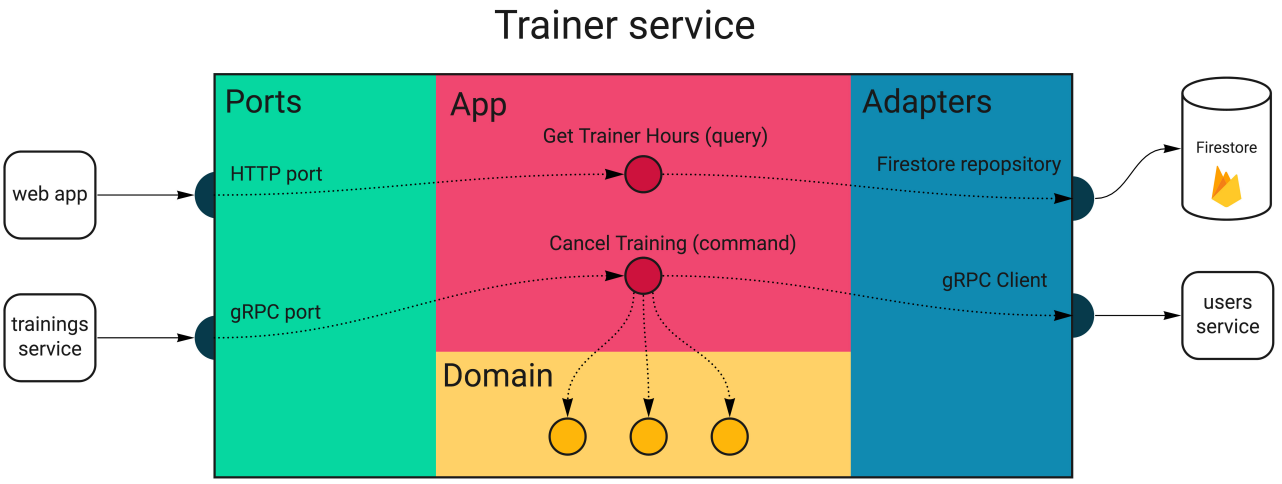
Take a look at a diagram that will help us understand the project's structure. Below is a generic service built with the approach used in Wild Workouts.

[2] https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example
[3] See Chapter 7: The Repository Pattern



All external inputs start on the left. The only entry point to the application is through the **Ports** layer (HTTP handlers, Pub/Sub message handlers). Ports execute relevant handlers in the **App** layer. Some of these will call the **Domain** code, and some will use **Adapters**, which are the only way out of the service. The adapters layer is where your database queries and HTTP clients live.

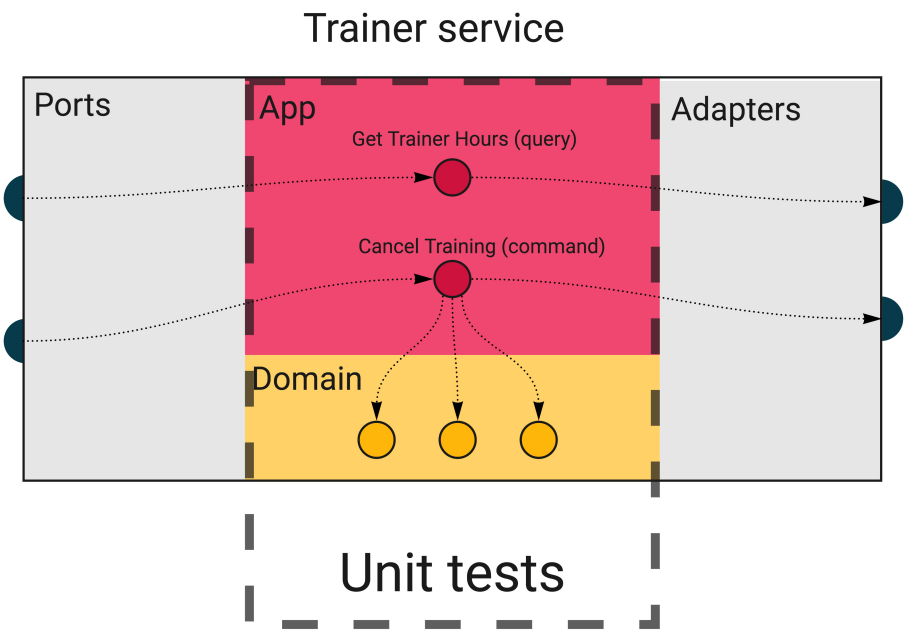
The diagram below shows the layers and flow of a part of the trainer service in Wild Workouts.



Let's now see what types of tests we would need to cover all of it.

Unit tests

We kick off with the inner layers and something everyone is familiar with: unit tests.



The domain layer is where the most complex logic of your service lives. However, **the tests here should be some of the simplest to write and running super fast.** There are no external dependencies in the domain, so you don't need any special infrastructure or mocks (except for really complex scenarios, but let's leave that for now).

As a rule of thumb, you should aim for high test coverage in the domain layer. Make sure you test only the exported code *(black-box testing)*. Adding the `_test` suffix to the package name is a great practice to enforce this.

The domain code is pure logic and straightforward to test, so it's the best place to check all corner cases. Table-driven tests are especially great for this.

```
func TestFactoryConfig_Validate(t *testing.T) {
    testCases := []struct {
        Name        string
        Config      hour.FactoryConfig
        ExpectedErr string
    }{
        {
            Name: "valid",
            Config: hour.FactoryConfig{
                MaxWeeksInTheFutureToSet: 10,
                MinUtcHour:               10,
                MaxUtcHour:               12,
            },
            ExpectedErr: "",
        },
        {
            Name: "equal_min_and_max_hour",
            Config: hour.FactoryConfig{
                MaxWeeksInTheFutureToSet: 10,
                MinUtcHour:               12,
```

```
                MaxUtcHour:              12,
            },
            ExpectedErr: "",
        },

        // ...
}

for _, c := range testCases {
        t.Run(c.Name, func(t *testing.T) {
            err := c.Config.Validate()

            if c.ExpectedErr != "" {
                assert.EqualError(t, err, c.ExpectedErr)
            } else {
                assert.NoError(t, err)
            }
}
```

Source: hour_test.go on GitHub[4]

We leave the domain and enter the application layer. After introducing CQRS[5], we've split it further into Commands and Queries.

Depending on your project, there could be nothing to test or some complex scenarios to cover. Most of the time, especially in queries, this code just glues together other layers. Testing this doesn't add any value. But if there's any complex orchestration in commands, it's another good case for unit tests.

> **Note**
>
> Watch out for complex logic living in the application layer. If you start testing business scenarios here, it's worth considering introducing the domain layer.
> On the other hand, it's the perfect place for orchestration — calling adapters and services in a particular order and passing the return values around. If you separate it like that, application tests should not break every time you change the domain code.

There are many external dependencies in the application's commands and queries, as opposed to the domain code. They will be trivial to mock if you follow Clean Architecture[6]. In most cases, a struct with a single method will make a perfect mock here.

> **Note**
>
> If you prefer to use mocking libraries or code-generated mocks, you can use them as well. Go lets you define and implement small interfaces, so we choose to define the mocks ourselves, as it's the simplest way.

[4] https://bit.ly/2NgvXqH
[5] See Chapter 10: Basic CQRS
[6] See Chapter 9: Clean Architecture

The snippet below shows how an application command is created with injected mocks.

```
func newDependencies() dependencies {
    repository := &repositoryMock{}
    trainerService := &trainerServiceMock{}
    userService := &userServiceMock{}

    return dependencies{
        repository:     repository,
        trainerService: trainerService,
        userService:    userService,
        handler:        command.NewCancelTrainingHandler(repository, userService, trainerService),
    }
}

// ...

    deps := newDependencies()

    tr := tc.TrainingConstructor()
    deps.repository.Trainings = map[string]training.Training{
        trainingUUID: *tr,
    }

    err := deps.handler.Handle(context.Background(), command.CancelTraining{
        TrainingUUID: trainingUUID,
        User:         training.MustNewUser(requestingUserID, tc.UserType),
    })
```

Source: cancel_training_test.go on GitHub[7]

Watch out for adding tests that don't check anything relevant, so you don't end up testing the mocks. Focus on the logic, and if there's none, skip the test altogether.

We've covered the two inner layers. I guess this didn't seem novel so far, as we're all familiar with unit tests. However, the Continuous Delivery Maturity Model[8] lists them only on the "base" maturity level. Let's now look into integration testing.

Integration tests

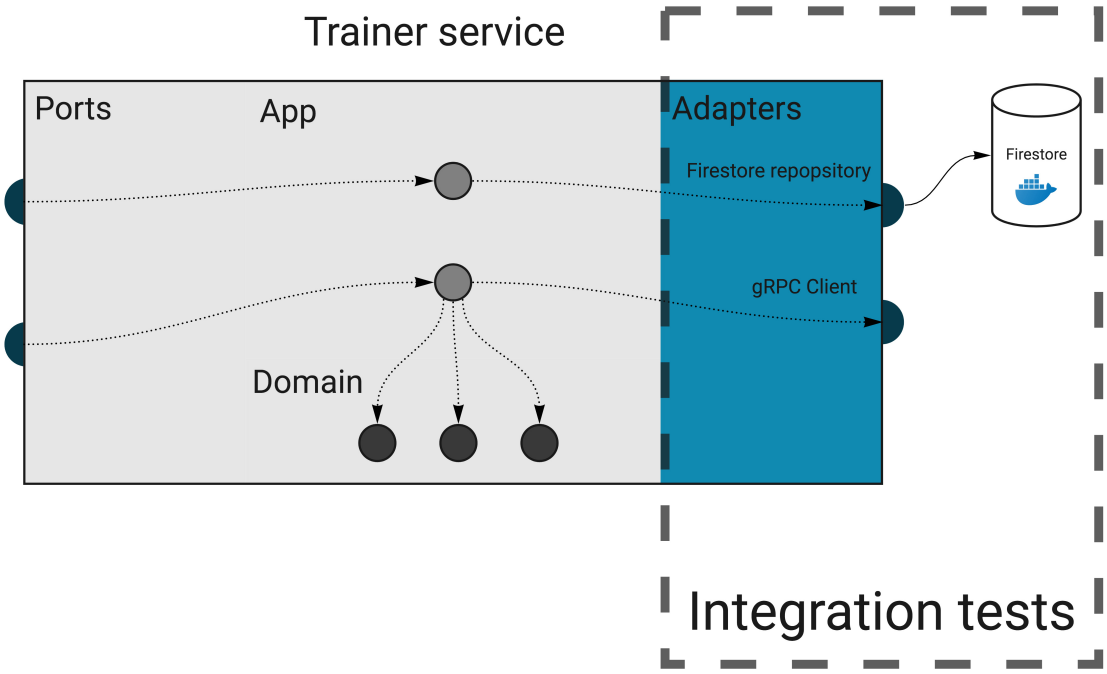
After reading this header, did you just imagine a long-running test that you have to retry several times to pass? And it's because of that 30-seconds sleep someone added that turns out to be too short when Jenkins is running under load?

There's no reason for integration tests to be slow and flaky. And practices like automatic retries and increasing sleep times should be absolutely out of the question.

[7] https://bit.ly/2NOwgsn
[8] https://www.infoq.com/articles/Continuous-Delivery-Maturity-Model/

In our context, **an integration test is a test that checks if an adapter works correctly with an external infrastructure.** Most of the time, this means testing database repositories.

These tests are not about checking whether the database works correctly, but whether you use it correctly (the **integration** part). It's also an excellent way to verify if you know how to use the database internals, like handling transactions.



Because we need real infrastructure, **integration tests are more challenging than unit tests to write and maintain.** Usually, we can use docker-compose to spin up all dependencies.

Note

Should we test our application with the Docker version of a database? The Docker image will almost always be slightly different than what we run on production. In some cases, like Firestore, there's only an emulator available, not the real database.
Indeed, Docker doesn't reflect the exact infrastructure you run on production. However, you're much more likely to mess up an SQL query in the code than to stumble on issues because of a minor configuration difference.
A good practice is to pin the image version to the same as running on the production. Using Docker won't give you 100% production parity, but it eliminates the "works on my machine" issues and tests your code with proper infrastructure.

Robert covered integration tests for databases in-depth in **High-Quality Database Integration Tests** (Chapter 8).

Keeping integration tests stable and fast

When dealing with network calls and databases, the test speed becomes super important. It's crucial to run tests in parallel, which can be enabled in Go by calling `t.Parallel()`. **It seems simple to do, but we have to make sure our tests support this behavior.**

For example, consider this trivial test scenario:

1. Check if the `trainings` collection is empty.
2. Call repository method that adds a training.
3. Check if there's one training in the collection.

If another test uses the same collection, you will get random fails because of the race condition. Sometimes, the collection will contain more than one training we've just added.

The simplest way out of this is never to assert things like a list length, but check it for the exact thing we're testing. For example, we could get all trainings, then iterate over the list to check if the expected ID is present.

Another approach is to isolate the tests somehow, so they can't interfere with each other. For example, each test case can work within a unique user's context (see component tests below).

Of course, both patterns are more complex than a simple length assertion. **When you stumble upon this issue for the first time, it may be tempting to give up and decide that "our integration tests don't need to run in parallel". Don't do this.** You will need to get creative sometimes, but it's not that much effort in the end. In return, your integration tests will be stable and running as fast as unit tests.

If you find yourself creating a new database before each run, it's another sign that you could rework tests to not interfere with each other.

Warning! A common, hard-to-spot, gotcha when iterating test cases.

When working with table-driven tests, you'll often see code like this:

```
for _, c := range testCases {
        t.Run(c.Name, func(t *testing.T) {
                // ...
        })
}
```

It's an idiomatic way to run tests over a slice of test cases. Let's say you now want to run each test case in parallel. The solution seems trivial:

```
for _, c := range testCases {
        t.Run(c.Name, func(t *testing.T) {
                t.Parallel()
                // ...
        })
}
```

Sadly, this won't work as expected.

The Common Mistakes[a] page on Go's GitHub wiki lists just two items, and both are actually about the same thing. So it seems there's only one mistake you should worry about in Go. However, this one is really hard to spot sometimes.

It's not obvious initially, but adding the parallel switch makes the parent test function not wait for the subtests spawned by `t.Run`. Because of this, you can't safely use the `c` loop variable inside the `func` closure. Running the tests like this will usually cause all subtests to work with the last test case, ignoring all others. **The worst part is the tests will pass, and you will see correct subtest names when running `go test` with the `-v` flag.** The only way to notice this issue is to change the code expecting tests to fail and see them pass instead.

As mentioned in the wiki, one way to fix this is to introduce a new scoped variable:

```
for _, c := range testCases {
        c := c
        t.Run(c.Name, func(t *testing.T) {
                t.Parallel()
                // ...
        })
}
```

It's just a matter of taste, but we don't like this approach, as it looks like some magic spell to anyone who doesn't know what this means. Instead, we choose the more verbose but obvious approach:

```
for i := range testCases {
        c := testCases[i]
        t.Run(c.Name, func(t *testing.T) {
                t.Parallel()
                // ...
        })
}
```

Even if you know about this behavior, it's dangerously easy to misuse it. What's worse, it seems that popular linters don't check this by default — if you know of a linter that does it well, please share in the comments. We made this mistake in the Watermill[b] library, which caused some of the tests not to run at all. You can see the fix in this commit[c].

[a] https://github.com/golang/go/wiki/CommonMistakes
[b] https://github.com/ThreeDotsLabs/watermill
[c] https://github.com/ThreeDotsLabs/watermill/commit/c72e26a67cb763ab3dd93ecf57a2b298fc81dd19

We covered the database repository with tests, but we also have a gRPC client adapter. How should we test this one?

It's similar to the application layer in this regard. If your test would duplicate the code it checks, it probably makes no sense to add it. It just becomes additional work when changing the code.

Let's consider the users service gRPC adapter:

```
func (s UsersGrpc) UpdateTrainingBalance(ctx context.Context, userID string, amountChange int) error {
    _, err := s.client.UpdateTrainingBalance(ctx, &users.UpdateTrainingBalanceRequest{
        UserId:       userID,
        AmountChange: int64(amountChange),
    })

    return err
}
```

Source: users_grpc.go on GitHub[9]

There's nothing interesting to test here. We could inject a mock `client` and check whether a proper method has been called. But this won't verify anything, and each change in the code will require a corresponding change in the test.

Component tests

So far, we've created mostly narrow, specialized tests for isolated parts of the application. Such tests work great for checking corner cases and specific scenarios, **but it doesn't mean each service works correctly**. It's easy enough to forget to call an application handler from a port. Also, unit tests alone won't help us make sure the application still works after a major refactoring.

Is it now the time to run end-to-end tests across all our services? Not yet.

As there is no standard of calling test types, I encourage you to follow Simon Stewart's advice from his Test Sizes post[10]. Create a table that will make it obvious for everyone on the team what to expect from a particular test. You can then cut all (unproductive) discussions on the topic.

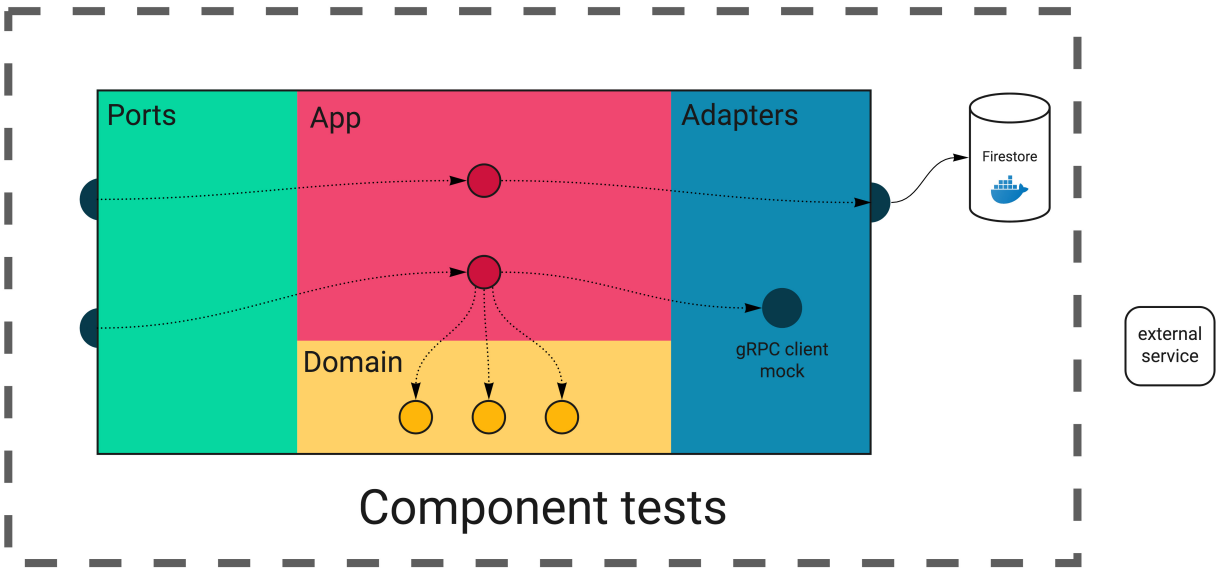
In our case, the table could look like this:

Feature	Unit	Integration	Component	End-to-End
Docker database	No	Yes	Yes	Yes
Use external systems	No	No	No	Yes
Focused on business cases	Depends on the tested code	No	Yes	Yes
Used mocks	Most dependencies	Usually none	External systems	None
Tested API	Go package	Go package	HTTP and gRPC	HTTP

[9] https://bit.ly/2OXVTaY
[10] https://testing.googleblog.com/2010/12/test-sizes.html

To ensure each service works correctly internally, **we introduce component tests to check all layers working together.** A component test covers a single service, isolated from other services in the application.

We will call real port handlers and use the infrastructure provided by Docker. However, we will **mock all adapters reaching external services**.



You might be wondering, why not test external services as well? After all, we could use Docker containers and test all of them together.

The issue is the complexity of testing several connected services. If you have just a couple of them, that can work well enough. But remember, you need to have the proper infrastructure in place for each service you spin up, including all databases it uses and all external services it calls. It can easily be tens of services in total, usually owned by multiple teams.

We'll come to this next in end-to-end tests. But for now, we add component tests because we need a fast way to know if a service works correctly.

To better understand why it's essential to have component tests in place, I suggest looking at some quotes from *Accelerate*[11].

> **Note**
>
> If you haven't heard about *Accelerate* yet, it's a book describing research on high-performing software teams. I recommend reading it to learn what can help your team get up to speed.

[11] https://itrevolution.com/book/accelerate/

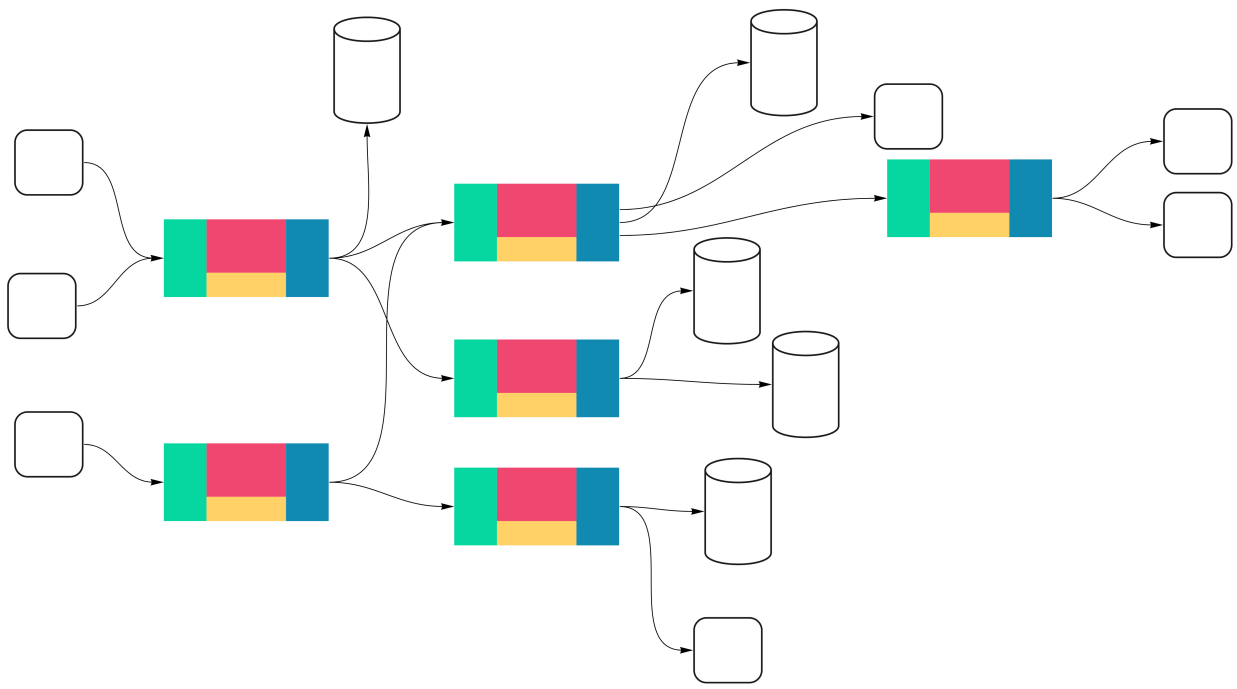


Figure 12.1: You don't want this as your primary testing approach.

According to the book, this is what the best software teams said about testability.

> We can do most of our testing without requiring an integrated environment. We can and do deploy or release our application independently of other applications/services it depends on.
>
> Accelerate (https://itrevolution.com/book/accelerate/)

Wait, weren't microservices supposed to fix teams being dependent on each other? If you think it's impossible to achieve this in the application you work on, it's likely because of poor architecture choices. You can fix it by applying strategic DDD patterns that we plan to introduce in future chapters.

Accelerate follows up with:

> 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.
>
> Accelerate (https://itrevolution.com/book/accelerate/)

We raise this point through the series: **using microservices doesn't make your application and teams less coupled by itself. Decoupling takes a conscious design of the application's architecture and at the system level.**

In component tests, our goal is to check the completeness of a single service in isolation, with all infrastructure it needs. We make sure the service accepts the API we agreed on and responds with expected results.

These tests are more challenging technically, but still relatively straightforward. We won't be running a real service binary because we need to mock some dependencies. We have to modify the way we start the service to make it possible.

> **Note**
>
> Once again, if you follow the Dependency Inversion Principle (just a reminder, it's part of SOLID), injecting mocks at the service level should be trivial.

I've introduced two constructors for our `app.Application` struct, which holds all commands and queries. The first one works just like before, setting up real gRPC clients and injecting them. The second replaces them with mocks.

```
func NewApplication(ctx context.Context) (app.Application, func()) {
    // ...

    trainerGrpc := adapters.NewTrainerGrpc(trainerClient)
    usersGrpc := adapters.NewUsersGrpc(usersClient)

    return newApplication(ctx, trainerGrpc, usersGrpc),
    // ...
}

func NewComponentTestApplication(ctx context.Context) app.Application {
    return newApplication(ctx, TrainerServiceMock{}, UserServiceMock{})
}
```

Source: service.go on GitHub[12]

We can now simply run the service in a separate goroutine.

We want to run just a single service instance for all tests, so we use the `TestMain` function. It's a simple way to insert a setup before running tests.

```
func startService() bool {
    app := NewComponentTestApplication(context.Background())

    trainingsHTTPAddr := os.Getenv("TRAININGS_HTTP_ADDR")
    go server.RunHTTPServerOnAddr(trainingsHTTPAddr, func(router chi.Router) http.Handler {
        return ports.HandlerFromMux(ports.NewHttpServer(app), router)
    })

    ok := tests.WaitForPort(trainingsHTTPAddr)
    if !ok {
        log.Println("Timed out waiting for trainings HTTP to come up")
    }

    return ok
}
```

[12] https://bit.ly/2Zzy0Zl

```
func TestMain(m *testing.M) {
    if !startService() {
        os.Exit(1)
    }

    os.Exit(m.Run())
}
```

Source: component_test.go on GitHub[13]

I've created the `WaitForPort` helper function that waits until the specified port is open or times out. It's crucial, as you need to ensure that the service has properly started. **Don't replace it with sleeps.** You will either add too much delay and make the test slow or make it too short, and it will fail randomly.

What to test in component tests? **Usually, the happy path should be enough. Don't check for corner cases there. Unit and integration tests should already cover these.** Make sure the correct payloads are processed, the storage works, and responses are correct.

Calling the ports

I use HTTP clients generated by `openapi-codegen`. Similar to the server part, this makes writing tests much easier. For example, you don't need to specify the whole REST path and worry about marshaling JSON each time.

Even though the generated clients save us a lot of boilerplate, I've still added the `tests/client.go` file with client wrappers for test purposes.

```
func (c TrainingsHTTPClient) CreateTraining(t *testing.T, note string, hour time.Time) string {
    response, err := c.client.CreateTrainingWithResponse(context.Background(),
↪   trainings.CreateTrainingJSONRequestBody{
        Notes: note,
        Time:  hour,
    })
    require.NoError(t, err)
    require.Equal(t, http.StatusNoContent, response.StatusCode())

    contentLocation := response.HTTPResponse.Header.Get("content-location")

    return lastPathElement(contentLocation)
}
```

Source: clients.go on GitHub[14]

The tests become even more readable, and it's easy to understand what's going on. **Making tests pass is easy enough, but it's much harder to understand them in a code review, reading between all asserts and mocks.** It's twice as important when modifying tests.

Instead of the snippet above, we can now use a single line that clearly states what's happening.

[13] https://bit.ly/3uidFpt
[14] https://bit.ly/2ZznyB3

```
trainingUUID := client.CreateTraining(t, "some note", hour)
```

Source: component_test.go on GitHub[15]

Other helper methods are `FakeAttendeeJWT` and `FakeTrainerJWT`. They generate valid authorization tokens with the chosen role.

As gRPC uses structs generated from protobuf, the clients are already straightforward to use.

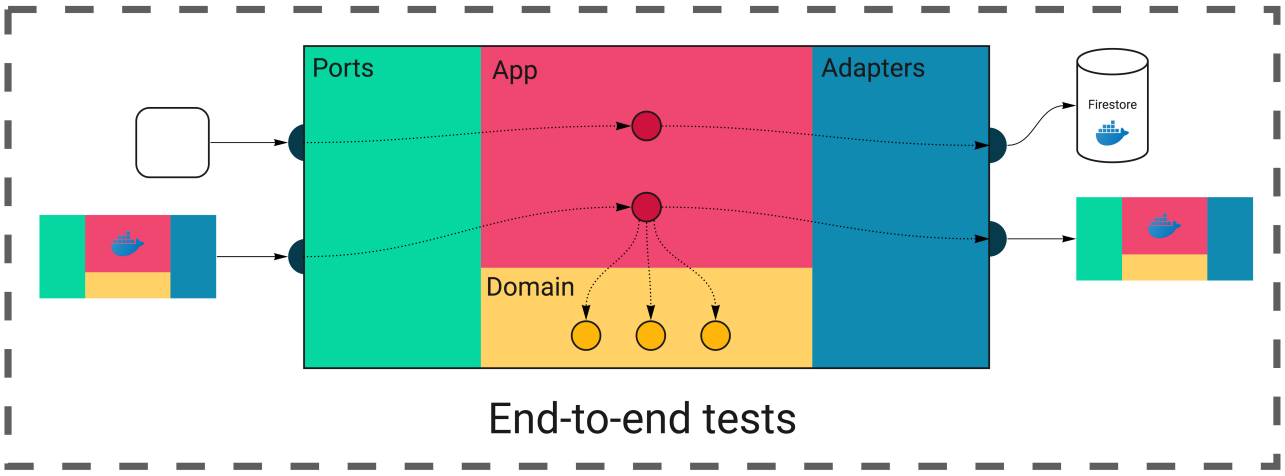
End-to-end tests

Finally, we come to the most dreaded part of our test suite. We will now leave all mocks behind.

End-to-end tests verify your whole system working together. They are slow, error-prone, and hard to maintain. You still need them, but make sure you build them well.

In Wild Workouts, it will be similar to running component tests, except we're going to spin up all services inside the docker-compose. We'll then verify a few critical paths calling just the HTTP endpoints, as this is what we expose to the external world.

> **Note**
>
> If you can't run the entire platform on docker-compose, you need to find a similar approach. It could be a separate Kubernetes cluster or namespace if you're already using it, or some kind of staging environment.



Now comes the part where you're going to have more questions than answers. Where should you keep end-to-end tests? Which team should own it? Where to run them? How often? Should they be part of the CI/CD pipeline or a separate thing ran from a cronjob?

I can't give you clear answers, as this heavily depends on your teams' structure, organizational culture, and CI/CD setup. As with most challenges, try an approach that seems the best and iterate over it until you're happy.

[15] https://bit.ly/2NIphS3

We're lucky to have just three services for the entire application, all using the same database. As the number of services and dependencies grows, your application will become harder to test this way.

Try to keep end-to-end tests short. **They should test if services correctly connect together, not the logic inside them.** These tests work as a double-check. They shouldn't fail most of the time, and if they do, it usually means someone has broken the contract.

> Note
>
> I said we'll use just HTTP endpoints, as this is what is exposed publicly. There's one exception: we're calling users service by gRPC to add trainings balance for our test attendee. As you would expect, this endpoint is not accessible to the public.

The code is very close to what we did in component tests. The main difference is we no longer spin up the service in a separate goroutine. Instead, all services are running inside docker-compose using the same binaries we deploy on production.

```
user := usersHTTPClient.GetCurrentUser(t)
originalBalance := user.Balance

_, err = usersGrpcClient.UpdateTrainingBalance(context.Background(), &users.UpdateTrainingBalanceRequest{
    UserId:       userID,
    AmountChange: 1,
})

require.NoError(t, err)
user = usersHTTPClient.GetCurrentUser(t)
require.Equal(t, originalBalance+1, user.Balance, "Attendee's balance should be updated")

trainingUUID := trainingsHTTPClient.CreateTraining(t, "some note", hour)

trainingsResponse := trainingsHTTPClient.GetTrainings(t)
require.Len(t, trainingsResponse.Trainings, 1)
require.Equal(t, trainingUUID, trainingsResponse.Trainings[0].Uuid, "Attendee should see the training")

user = usersHTTPClient.GetCurrentUser(t)
require.Equal(t, originalBalance, user.Balance, "Attendee's balance should be updated after a training is scheduled")
```

Source: e2e_test.go on GitHub[16]

Acceptance tests

You can often find acceptance tests defined as the next level after unit and integration tests. We see them as orthogonal to the technical aspects of tests. It's a test that focuses on a complete business feature instead of implementation details. As Martin Fowler puts it:

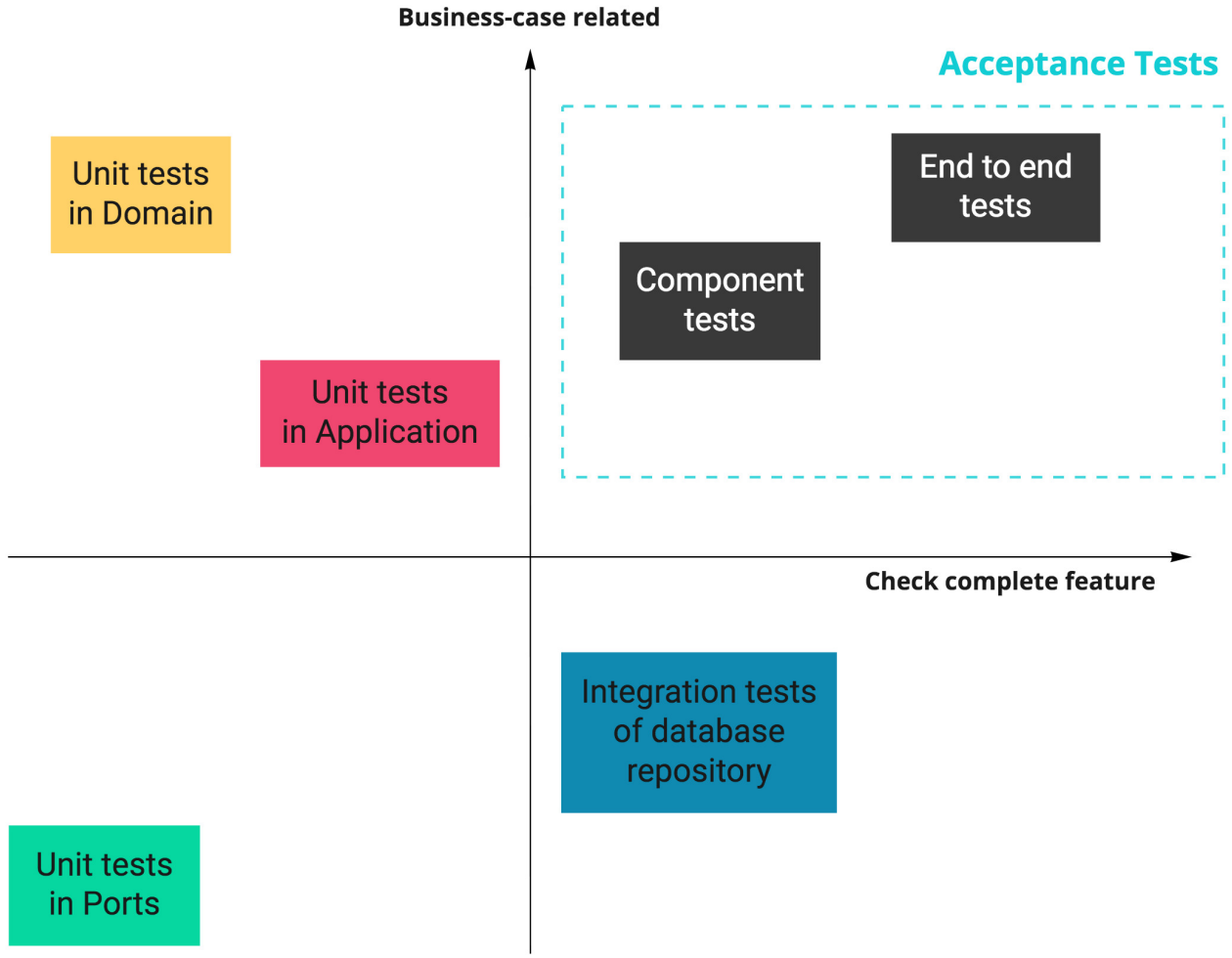
> Here's the thing: At one point you should make sure to test that your software works correctly from

[16] https://bit.ly/3uhiGPc

> a user's perspective, not just from a technical perspective. What you call these tests is really not that important. Having these tests, however, is. Pick a term, stick to it, and write those tests.
>
> Acceptance Tests — Do Your Features Work Correctly? (https://martinfowler.com/articles/practical-test-pyramid.html)

In our case, component tests and end-to-end tests can both be considered acceptance tests.

If you like, you can use the BDD[17] style for some of them — it makes them easier to read but adds some boilerplate.



Can we sleep well now?

> A service is not really tested until we break it in production.

[17] https://en.wikipedia.org/wiki/Behavior-driven_development

Jesse Robbins, according to The DevOps Handbook (https://itrevolution.com/the-devops-handbook/)

A reliable test suite will catch most of your bugs so that you can deliver consistently. But you want to be ready for outages anyway. We now enter the topic of monitoring, observability, and chaos engineering. However, this is not in the scope for today.

Just keep in mind that no test suite will give you total confidence. It's also crucial to have a straightforward process in place that allows quick rollbacks, reverts, and undoing migrations.

Moving on

If you look through the full commit[18], you might notice how we inject dependencies now is not very elegant. We're going to iterate over this in the future.

We have the test suite in place, but we still miss a critical part: it's not running in our Continuous Integration pipeline.

Also, coming up with proper docker-compose and environment variables for tests wasn't trivial. The current solution works but is not straightforward to understand if you don't know what's going on. And running the tests in an automated build will only make it more complicated.

We will cover these topics in future chapters.

[18] https://bit.ly/2Mj1vM2

Chapter 13

Repository Secure by Design

Robert Laszczak

Thanks to the tests and code review, you can make your project bug-free. Right? Well… actually, probably not. That would be too easy. These techniques lower the chance of bugs, but they can't eliminate them entirely. But does it mean we need to live with the risk of bugs until the end of our lives?

Over one year ago, I found a pretty interesting PR[1] in the `harbor` project. This was a fix for the issue that **allowed to create admin user by a regular user. This was obviously a severe security issue.** Of course, automated tests didn't found this bug earlier.

This is how the bugfix looks like:

```
        ua.RenderError(http.StatusBadRequest, "register error:"+err.Error())
        return
    }
+
+   if !ua.IsAdmin && user.HasAdminRole {
+       msg := "Non-admin cannot create an admin user."
+       log.Errorf(msg)
+       ua.SendForbiddenError(errors.New(msg))
+       return
+   }
+
    userExist, err := dao.UserExists(user, "username")
    if err != nil {
```

One `if` statement fixed the bug. Adding new tests also should ensure that there will be no regression in the future. Is it enough? **Did it secure the application from a similar bug in the future? I'm pretty sure it didn't.**

The problem becomes bigger in more complex systems with a big team working on them. What if someone is new to the project and forgets to put this `if` statement? Even if you don't hire new people currently, they may be hired

[1] https://github.com/goharbor/harbor/pull/8917/files

in the future. **You will probably be surprised how long the code you have written will live.** We should not trust people to use the code we've created in the way it's intended – they will not.

In some cases, the solution that will protect us from issues like that is good design. Good design should not allow using our code in an invalid way. Good design should guarantee that you can touch the existing code without any fear. People new to the project will feel safer introducing changes.

In this chapter, I'll show how I ensured that only allowed people would be able to see and edit a training. In our case, a training can only be seen by the training owner (an attendee) and the trainer. I will implement it in a way that doesn't allow to use our code in not intended way. By design.

Our current application assumes that a repository is the only way how we can access the data. Because of that, I will add authorization on the repository level. **By that, we are sure that it is impossible to access this data by unauthorized users.**

What is Repository tl;dr

If you had no chance to read our previous chapters, repository is a pattern that helps us to abstract database implementation from our application logic.
If you want to know more about its advantages and learn how to apply it in your project, read **The Repository Pattern** (Chapter 7).

But wait, is the repository the right place to manage authorization? Well, I can imagine that some people may be skeptical about that approach. Of course, we can start some philosophical discussion on what can be in the repository and what shouldn't. Also, the actual logic of who can see the training will be placed in the *domain* layer. I don't see any significant downsides, and the advantages are apparent. In my opinion, pragmatism should win here.

Tip

What is also interesting in this book, we focus on *business-oriented applications*. But even if the *Harbor* project is a pure system application, most of the presented patterns can be applied as well.
After introducing Clean Architecture[a] to our team, our teammate used this approach in his game to abstract rendering engine.
(Cheers, Mariusz, if you are reading that!)

[a] See Chapter 9: Clean Architecture

Show me the code, please!

To achieve our robust design, we need to implement three things: 1. Logic who can see the training (domain layer[2], 2. Functions used to get the training (`GetTraining` in the repository[3]), 3. Functions used to update the training

[2] https://bit.ly/3kdFQBf
[3] https://bit.ly/3qE1TDP

(`UpdateTraining` in the repository[4].

Domain layer

The first part is the logic responsible for deciding if someone can see the training. Because it is part of the domain logic (you can talk about who can see the training with your business or product team), it should go to the `domain` layer. It's implemented with `CanUserSeeTraining` function.

It is also acceptable to keep it on the repository level, but it's harder to re-use. I don't see any advantage of this approach – especially if putting it to the `domain` doesn't cost anything.

```
package training

// ...

type User struct {
    userUUID string
    userType UserType
}

// ...

type ForbiddenToSeeTrainingError struct {
    RequestingUserUUID string
    TrainingOwnerUUID  string
}

func (f ForbiddenToSeeTrainingError) Error() string {
    return fmt.Sprintf(
        "user '%s' can't see user '%s' training",
        f.RequestingUserUUID, f.TrainingOwnerUUID,
    )
}

func CanUserSeeTraining(user User, training Training) error {
    if user.Type() == Trainer {
        return nil
    }
    if user.UUID() == training.UserUUID() {
        return nil
    }

    return ForbiddenToSeeTrainingError{user.UUID(), training.UserUUID()}
}
```

Source: user.go on GitHub[5]

[4] https://bit.ly/2NqUhpB
[5] https://bit.ly/2NFEZgC

Repository

Now when we have the `CanUserSeeTraining` function, we need to use this function. Easy like that.

```
func (r TrainingsFirestoreRepository) GetTraining(
    ctx context.Context,
    trainingUUID string,
+   user training.User,
) (*training.Training, error) {
    firestoreTraining, err := r.trainingsCollection().Doc(trainingUUID).Get(ctx)

    if status.Code(err) == codes.NotFound {
        return nil, training.NotFoundError{trainingUUID}
    }
    if err != nil {
        return nil, errors.Wrap(err, "unable to get actual docs")
    }

    tr, err := r.unmarshalTraining(firestoreTraining)
    if err != nil {
        return nil, err
    }
+
+   if err := training.CanUserSeeTraining(user, *tr); err != nil {
+       return nil, err
+   }
+
    return tr, nil
}
```

Source: trainings_firestore_repository.go on GitHub[6]

Isn't it too simple? Our goal is to create a simple, not complex, design and code. **This is an excellent sign that it is deadly simple.**

We are changing `UpdateTraining` in the same way.

```
func (r TrainingsFirestoreRepository) UpdateTraining(
    ctx context.Context,
    trainingUUID string,
+   user training.User,
    updateFn func(ctx context.Context, tr *training.Training) (*training.Training, error),
) error {
    trainingsCollection := r.trainingsCollection()

    return r.firestoreClient.RunTransaction(ctx, func(ctx context.Context, tx *firestore.Transaction) error {
        documentRef := trainingsCollection.Doc(trainingUUID)

        firestoreTraining, err := tx.Get(documentRef)
        if err != nil {
            return errors.Wrap(err, "unable to get actual docs")
```

[6] https://bit.ly/3qE1TDP

```
        }

        tr, err := r.unmarshalTraining(firestoreTraining)
        if err != nil {
            return err
        }
+
+       if err := training.CanUserSeeTraining(user, *tr); err != nil {
+           return err
+       }
+
        updatedTraining, err := updateFn(ctx, tr)
        if err != nil {
            return err
        }

        return tx.Set(documentRef, r.marshalTraining(updatedTraining))
    })
}
```

Source: trainings_firestore_repository.go on GitHub[7]

And… that's all! Is there any way that someone can use this in a wrong way? As long as the `User` is valid – no.

This approach is similar to the method presented in **Domain-Driven Design Lite** (Chapter 6). It's all about creating code that we can't use in a wrong way.

This is how usage of `UpdateTraining` now looks like:

```
func (h ApproveTrainingRescheduleHandler) Handle(ctx context.Context, cmd ApproveTrainingReschedule) (err error) {
    defer func() {
        logs.LogCommandExecution("ApproveTrainingReschedule", cmd, err)
    }()

    return h.repo.UpdateTraining(
        ctx,
        cmd.TrainingUUID,
        cmd.User,
        func(ctx context.Context, tr *training.Training) (*training.Training, error) {
            originalTrainingTime := tr.Time()

            if err := tr.ApproveReschedule(cmd.User.Type()); err != nil {
                return nil, err
            }

            err := h.trainerService.MoveTraining(ctx, tr.Time(), originalTrainingTime)
            if err != nil {
                return nil, err
            }
```

[7] https://bit.ly/2NqUhpB

```
            return tr, nil
        },
    )
}
```

Source: approve_training_reschedule.go on GitHub[8]

Of course, there are still some rules if `Training` can be rescheduled, but this is handled by the `Training` domain type.

Handling collections

Even if this approach works perfectly for operating on a single training, you need to be sure that access to a collection of trainings is properly secured. There is no magic here:

```
func (r TrainingsFirestoreRepository) FindTrainingsForUser(ctx context.Context, userUUID string) ([]query.Training,
↪  error) {
    query := r.trainingsCollection().Query.
        Where("Time", ">=", time.Now().Add(-time.Hour*24)).
        Where("UserUuid", "==", userUUID).
        Where("Canceled", "==", false)

    iter := query.Documents(ctx)

    return r.trainingModelsToQuery(iter)
}
```

Source: trainings_firestore_repository.go on GitHub[9]

Doing it on the application layer with the `CanUserSeeTraining` function will be very expensive and slow. It's better to create a bit of logic duplication.

If this logic is more complex in your application, you can try to abstract it in the domain layer to the format that you can convert to query parameters in your database driver. I did it once, and it worked pretty nicely.

But in Wild Workouts, it will add unnecessary complexity - let's Keep It Simple, Stupid[10].

Handling internal updates

We often want to have endpoints that allow a developer or your company operations department to do some "backdoor" changes. The worst thing that you can do in this case is creating any kind of "fake user" and hacks.

It ends with a lot of `if` statements added to the code from my experience. It also obfuscates the audit log (if you have any). Instead of a "fake user", it's better to create a special role and explicitly define the role's permissions.

If you need repository methods that don't require any user (for Pub/Sub message handlers or migrations), it's better to create separate repository methods. In that case, naming is essential – we need to be sure that the person

[8] https://bit.ly/3scHx4W
[9] https://bit.ly/3pEIXDF
[10] https://en.wikipedia.org/wiki/KISS_principle

who uses that method knows the security implications.

From my experience, if updates are becoming much different for different actors, it's worth to even introduce a separate CQRS Commands[11] per actor. In our case it may be `UpdateTrainingByOperations`.

Passing authentication via `context.Context`

As far as I know, some people are passing authentication details via `context.Context`.

I highly recommend not passing anything required by your application to work correctly via `context.Context`. The reason is simple – when passing values via `context.Context`, we lose one of the most significant Go advantages – static typing. It also hides what exactly the input for your functions is.

If you need to pass values via context for some reason, it may be a symptom of a bad design somewhere in your service. Maybe the function is doing too much, and it's hard to pass all arguments there? Perhaps it's the time to decompose that?

And that's all for today!

As you see, the presented approach is straightforward to implement quickly.

I hope that it will help you with your project and give you more confidence in future development.

Do you see that it can help in your project? Do you think that it may help your colleagues? Don't forget to share it with them!

[11] See Chapter 10: Basic CQRS

Chapter 14

Setting up infrastructure with Terraform

Miłosz Smółka

We picked **Google Cloud Platform (GCP)** as the provider of all infrastructure parts of the project. We use **Cloud Run** for running Go services, **Firestore** as the database, **Cloud Build** as CI/CD, and **Firebase** for web hosting and authentication. The project is based on Terraform[1].

Infrastructure as Code

If you're not familiar with Terraform at all, it's a tool for keeping infrastructure configuration as text files. This technique is also known as "Infrastructure as Code". It's a broad topic, so I'll mention just a few benefits:

- **Storing infrastructure configuration in a repository**. This gives you all benefits that come with version control, like the history of commits, one source of truth, and code reviews.
- **Consistency over time**. Ideally, there are no manual changes to infrastructure other than those described in the configuration.
- **Multiple environments**. When you stop treating your servers as pets, it's much easier to create identical environments, sometimes even on demand.

Terraform 101

You can skip this part if you already know Terraform basics.

In Terraform, you define a set of **resources** in `.tf` files using HCL[2] syntax. A resource can be a database, network configuration, compute instance, or even a set of permissions. Available resource types depend on the used provider (e.g., AWS, GCP, Digital Ocean).

[1] https://www.terraform.io/
[2] https://github.com/hashicorp/hcl

All resources are defined in a **declarative** way. You don't write pseudocode that creates servers. Instead, you define the desired outcome. It's up to Terraform to come up with a way to configure what you specified. For example, it's smart enough to create a resource only if it doesn't exist.

Resources can refer to each other using the full name.

Besides resources, a project can also specify **input variables** to be filled by user and **output values**, that can be printed on the screen or stored in a file.

There are also **data sources** that don't create anything but read existing remote resources.

You can find all the available resources in the specific provider documentation. Here's an example for Google Cloud Platform[3].

Two basic commands that you need to know are `terraform plan` and `terraform apply`.

`apply` applies all resources defined in the current directory in all files with `.tf` extension. The `plan` command is a "dry-run" mode, printing out all changes that would be applied by `apply`.

After applying changes, you will find a `terraform.tfstate` file in the same directory. This file holds a local "state[4]" of your infrastructure.

Google Cloud Project

Our Terraform configuration creates a new GCP project. It's completely separated from your other projects and is easy to clean up.

Because some resources go beyond the free trier, you need to have a *Billing Account*. It can be an account with a linked credit card, but the $300 credit for new accounts is also fine.

The basic part of the project definition looks like this:

```
provider "google" {
  project = var.project
  region  = var.region
}

data "google_billing_account" "account" {
  display_name = var.billing_account
}

resource "google_project" "project" {
  name            = "Wild Workouts"
  project_id      = var.project
```

[3] https://www.terraform.io/docs/providers/google/index.html
[4] https://www.terraform.io/docs/state/index.html

```
  billing_account = data.google_billing_account.account.id
}

resource "google_project_iam_member" "owner" {
  role    = "roles/owner"
  member  = "user:${var.user}"

  depends_on = [google_project.project]
}
```

Source: project.tf on GitHub[5]

Let's consider what each *block* does:

1. Enables the GCP provider. Sets **project** name and chosen **region** from variables. These two fields are "inherited" by all resources unless overridden.
2. Finds a billing account with the display name provided by the variable.
3. Creates a new google project linked with the billing account. **Note the reference to the account ID.**
4. Adds your user as the owner of the project. **Note the string interpolation.**

Enabling APIs

On a fresh GCP project, you can't use most services right away. You have to first enable the API for each of them. You can enable an API by clicking a button in the GCP Console, or do the same in Terraform[6]:

```
resource "google_project_service" "compute" {
  service    = "[compute.googleapis.com](http://compute.googleapis.com/)"
  depends_on = [google_project.project]
}
```

Source: project.tf on GitHub[7]

We enable the following APIs:

- Cloud Source Repositories
- Cloud Build
- Container Registry
- Compute
- Cloud Run
- Firebase
- Firestore

[5] https://bit.ly/2OM8w8P
[6] https://www.terraform.io/docs/providers/google/r/google_project_service.html
[7] https://bit.ly/3qJjLNu

A note on dependencies

If you're wondering about the `depends_on` line, it's there to set an **explicit dependency** between the service and the project. Terraform detects dependencies between resources if they refer to each other. In the first snippet, you can see this with billing account that's referenced by the project:

```
data "google_billing_account" "account" {
  display_name = var.billing_account
}

resource "google_project" "project" {
  # ...
  billing_account = data.google_billing_account.account.id
}
```

In the `google_project_service` resource, we don't use `project` anywhere, because it's already set in the `provider` block. Instead, we use `depends_on` to specify explicit dependency. This line tells Terraform to wait with creating the resource until the project is correctly created.

Cloud Run

		Name ↑	Req/sec	Region	Authentication
☐	✓	trainer-grpc	0	europe-west1	
☐	✓	trainer-http	0	europe-west1	Allow unauthenticated
☐	✓	trainings-http	0	europe-west1	Allow unauthenticated
☐	✓	users-grpc	0	europe-west1	
☐	✓	users-http	0	europe-west1	Allow unauthenticated

Figure 14.1: Cloud Run services

In Cloud Run, a service is a set of Docker containers with a common endpoint, exposing a single port (HTTP or gRPC). Each service is automatically scaled, depending on the incoming traffic. You can choose the maximum number of containers and how many requests each container can handle.

It's also possible to connect services with Google Cloud Pub/Sub. Our project doesn't use it yet, but we will introduce it in the future versions.

You are charged only for the computing resources you use, so when a request is being processed or when the container starts.

Wild Workouts consists of 3 services: *trainer*, *trainings*, and *users*. We decided to serve public API with HTTP and internal API with gRPC.

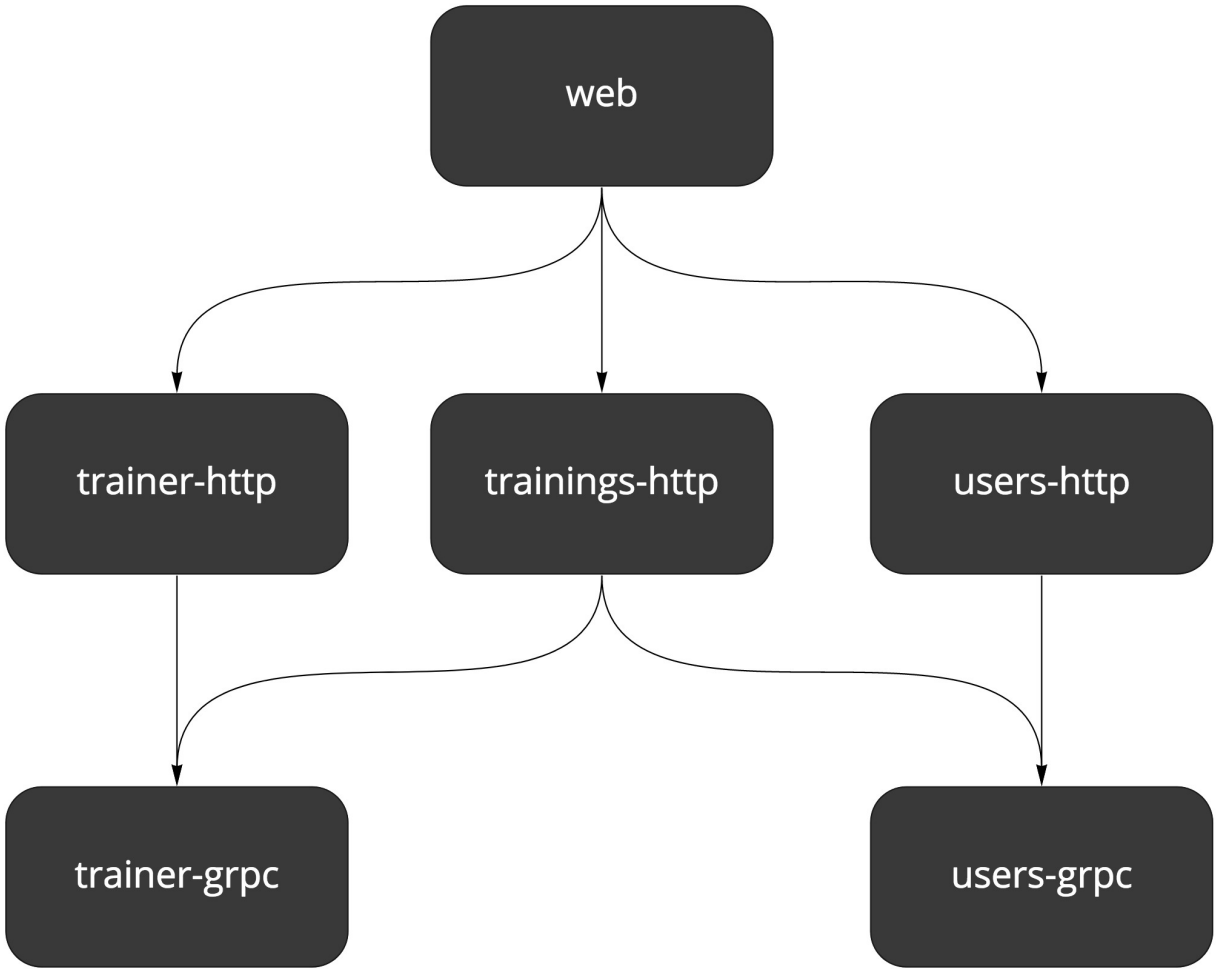


Figure 14.2: Services

Because you can't have two separate ports exposed from a single service, we have to expose two containers per service (except trainings, which doesn't have internal API at the moment).

We deploy 5 services in total, each with a similar configuration. Following the DRY[8] principle, the common Terraform configuration is encapsulated in the service module[9].

[8]https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
[9]https://bit.ly/3aJee3R

Note

A Module[a] in Terraform is a separate set of files in a subdirectory. Think of it as a container for a group of resources. It can have its input variables and output values.
Any module can call other modules by using the `module` block and passing a path to the directory in the `source` field. A single module can be called multiple times.
Resources defined in the main working directory are considered to be in a *root module*.

[a] https://www.terraform.io/docs/configuration/modules.html

The module holds a definition of a single generic Cloud Run service and is used multiple times in `cloud-run.tf`. It exposes several variables, e.g., name and type of the server (HTTP or gRPC).

A service exposing gRPC is the simpler one:

```
module cloud_run_trainer_grpc {
  source = "./service"

  project    = var.project
  location   = var.region
  dependency = null_resource.init_docker_images

  name     = "trainer"
  protocol = "grpc"
}
```

Source: cloud-run.tf on GitHub[10]

The `protocol` is passed to the `SERVER_TO_RUN` environment variable, which decides what server is run by the service.

Compare this with an HTTP service definition. We use the same module, and there are additional environment variables defined. We need to add them because public services contact internal services via the gRPC API and need to know their addresses.

```
module cloud_run_trainings_http {
  source = "./service"

  project    = var.project
  location   = var.region
  dependency = null_resource.init_docker_images

  name     = "trainings"
  protocol = "http"
  auth     = false
```

[10] https://bit.ly/3dwBuUt

```
  envs = [
    {
      name = "TRAINER_GRPC_ADDR"
      value = module.cloud_run_trainer_grpc.endpoint
    },
    {
      name  = "USERS_GRPC_ADDR"
      value = module.cloud_run_users_grpc.endpoint
    }
  ]
}
```

Source: cloud-run.tf on GitHub[11]

The reference: `module.cloud_run_trainer_grpc.endpoint` points to `endpoint` output defined in the `service` module:

```
output endpoint {
  value = "${trimprefix(google_cloud_run_service.service.status[0].url, "https://")}:443"
}
```

Source: outputs.tf on GitHub[12]

Using environment variables is an easy way to make services know each other. It would probably be best to implement some kind of service discovery with more complex connections and more services. Perhaps we will cover this in another chapter in the future.

If you're curious about the `dependency` variable see *In-depth* down below for details.

Cloud Run Permissions

Cloud Run services have authentication enabled by default. Setting the `auth = false` variable in the `service` module adds additional IAM policy for the service, making it accessible for the public. We do this just for the HTTP APIs.

```
data "google_iam_policy" "noauth" {
  binding {
    role = "roles/run.invoker"
    members = [
      "allUsers",
    ]
  }
```

[11] https://bit.ly/3btsm0e
[12] https://bit.ly/3udkNDL

```
}

resource "google_cloud_run_service_iam_policy" "noauth_policy" {
  count = var.auth ? 0 : 1

  location = google_cloud_run_service.service.location
  service  = google_cloud_run_service.service.name

  policy_data = data.google_iam_policy.noauth.policy_data
}
```

Source: service.tf on GitHub[13]

Note the following line:

```
count = var.auth ? 0 : 1
```

This line is Terraform's way of making an `if` statement. `count` defines how many copies of a resource Terraform should create. It skips resources with count equal `0`.

Firestore

We picked Firestore as the database for Wild Workouts. See our first post[14] for reasons behind this.

Firestore works in two modes - Native or Datastore. You have to decide early on, as the choice is permanent after your first write to the database.

You can pick the mode in GCP Console GUI, but we wanted to make the setup fully automated. In Terraform, there's `google_project_service` resource available, but it enables the Datastore mode. Sadly, we can't use it, since we'd like to use the Native mode.

A workaround is to use the `gcloud` command to enable it (note the `alpha` version).

```
gcloud alpha firestore databases create
```

To run this command, we use the null resource[15]. It's a special kind of resource that lets you run custom provisioners locally or on remote servers. A provisioner[16] is a command or other software making changes in the system.

We use local-exec provisioner[17], which is simply executing a bash command on the local system. In our case, it's one of the targets defined in `Makefile`.

```
resource "null_resource" "enable_firestore" {
  provisioner "local-exec" {
```

[13] https://bit.ly/3sq0aCD
[14] See Chapter 2: Building a serverless application with Google Cloud Run and Firebase
[15] https://www.terraform.io/docs/providers/null/resource.html
[16] https://www.terraform.io/docs/provisioners/index.html
[17] https://www.terraform.io/docs/provisioners/local-exec.html

```
    command = "make firestore"
  }

  depends_on = [google_firebase_project_location.default]
}
```

Source: firestore.tf on GitHub[18]

Firestore requires creating all composite indexes upfront. It's also available as a Terraform resource[19].

```
resource "google_firestore_index" "trainings_user_time" {
  collection = "trainings"

  fields {
    field_path = "UserUuid"
    order      = "ASCENDING"
  }

  fields {
    field_path = "Time"
    order      = "ASCENDING"
  }

  fields {
    field_path = "__name__"
    order      = "ASCENDING"
  }

  depends_on = [null_resource.enable_firestore]
}
```

Source: firestore.tf on GitHub[20]

Note the explicit `depends_on` that points to the `null_resource` creating database.

Firebase

Firebase provides us with frontend application hosting and authentication.

Currently, Terraform supports only part of Firebase API, and some of it is still in beta. That's why we need to enable the `google-beta` provider:

[18] https://bit.ly/2ZznMYV
[19] https://www.terraform.io/docs/providers/google/r/firestore_index.html
[20] https://bit.ly/3qE9iD9

```
provider "google-beta" {
  project     = var.project
  region      = var.region
  credentials = base64decode(google_service_account_key.firebase_key.private_key)
}
```

Source: firebase.tf on GitHub[21]

Then we define the project, project's location[22] and the web application.

```
resource "google_firebase_project" "default" {
  provider = google-beta

  depends_on = [
    google_project_service.firebase,
    google_project_iam_member.service_account_firebase_admin,
  ]
}

resource "google_firebase_project_location" "default" {
  provider = google-beta

  location_id = var.firebase_location

  depends_on = [
    google_firebase_project.default,
  ]
}

resource "google_firebase_web_app" "wild_workouts" {
  provider     = google-beta
  display_name = "Wild Workouts"

  depends_on = [google_firebase_project.default]
}
```

Source: firebase.tf on GitHub[23]

Authentication management still misses a Terraform API, so you have to enable it manually in the Firebase Console[24]. Firebase authentication is the only thing we found no way to automate.

[21] https://bit.ly/37zdkVB
[22] https://firebase.google.com/docs/projects/locations
[23] https://bit.ly/3bowwqm
[24] https://console.firebase.google.com/project/_/authentication/providers

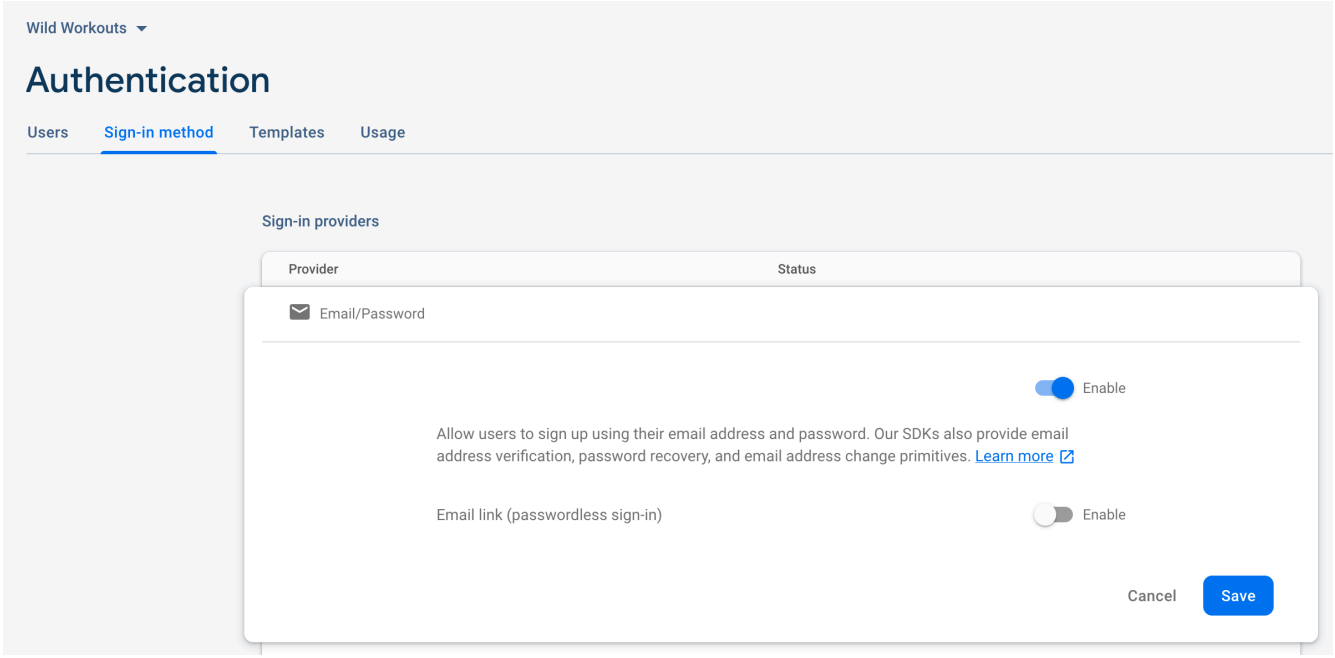


Figure 14.3: Cloud Run services

Firebase Routing

Firebase also handles public routing to services. Thanks to this the frontend application can call API with `/api/trainer` instead of `https://trainer-http-smned2eqeq-ew.a.run.app`[25]. These routes are defined in `web/firebase.json`.

```
"rewrites": [
    {
      "source": "/api/trainer{,/**}",
      "run": {
        "serviceId": "trainer-http",
        "region": "europe-west1"
      }
    },
    // ...
]
```

Source: firebase.json on GitHub[26]

[25] https://trainer-grpc-lmned2eqeq-ew.a.run.app/
[26] https://bit.ly/37wRbYf

Cloud Build

Cloud Build is our Continuous Delivery pipeline. It has to be enabled for a repository, so we define a trigger[27] in Terraform and the repository[28] itself.

```
resource "google_sourcerepo_repository" "wild_workouts" {
  name = var.repository_name

  depends_on = [
    google_project_service.source_repo,
  ]
}

resource "google_cloudbuild_trigger" "trigger" {
  trigger_template {
    branch_name = "master"
    repo_name   =  google_sourcerepo_repository.wild-workouts.name
  }

  filename = "cloudbuild.yaml"
}
```

Source: repo.tf on GitHub[29]

The build configuration has to be defined in the `cloudbuild.yaml`[30] file committed to the repository. We defined a couple of steps in the pipeline:

- Linting (go vet) - this step could be extended in future with running tests and all kinds of static checks and linters
- Building docker images
- Deploying docker images to Cloud Run
- Deploying web app to Firebase hosting

We keep several services in one repository, and their building and deployment are almost the same. To reduce repetition, there are a few helper bash scripts in the `scripts`[31] directory.

[27] https://www.terraform.io/docs/providers/google/r/cloudbuild_trigger.html
[28] https://www.terraform.io/docs/providers/google/r/sourcerepo_repository.html
[29] https://bit.ly/3k6hGZo
[30] https://bit.ly/3umPOoK
[31] https://bit.ly/3dGyiWv

Should I use a monorepo?

We decided to keep all services in one repository. We did this mainly because Wild Workouts is an example project, and it's much easier to set up everything this way. We can also easily share the common code (e.g., setting up gRPC and HTTP servers).

From our experience, using a single repository is a great starting point for most projects. It's essential, though, that all services are entirely isolated from each other. If there's a need, we could easily split them into separate repositories. We might show this in the future chapters in this book.

The biggest disadvantage of the current setup is that all services and the frontend application are deployed at the same time. That's usually not what you want when working with microservices with a bigger team. But it's probably acceptable when you're just starting out and creating an MVP.

As usual in CI/CD tools, the Build configuration[32] is defined in YAML. Let's see the whole configuration for just one service, to reduce some noise.

```
steps:
- id: trainer-lint
  name: golang
  entrypoint: ./scripts/lint.sh
  args: [trainer]

- id: trainer-docker
  name: gcr.io/cloud-builders/docker
  entrypoint: ./scripts/build-docker.sh
  args: ["trainer", "$PROJECT_ID"]
  waitFor: [trainer-lint]

- id: trainer-http-deploy
  name: gcr.io/cloud-builders/gcloud
  entrypoint: ./scripts/deploy.sh
  args: [trainer, http, "$PROJECT_ID"]
  waitFor: [trainer-docker]

- id: trainer-grpc-deploy
  name: gcr.io/cloud-builders/gcloud
  entrypoint: ./scripts/deploy.sh
  args: [trainer, grpc, "$PROJECT_ID"]
  waitFor: [trainer-docker]

options:
  env:
  - 'GO111MODULE=on'
  machineType: 'N1_HIGHCPU_8'

images:
- 'gcr.io/$PROJECT_ID/trainer'
```

[32] https://cloud.google.com/cloud-build/docs/build-config

Source: cloudbuild.yaml on GitHub[33]

A single step definition is pretty short:

- `id` is a unique identifier of a step. It can be used in `waitFor` array to specify dependencies between steps (steps are running in parallel by default).
- `name` is the name of a docker image that will be run for this step.
- `entrypoint` works like in docker images, so it's a command that is executed on the container's start. We use bash scripts to make the YAML definition short, but this can be any bash command.
- `args` will be passed to `entrypoint` as arguments.

In `options`, we override the machine type to make our builds run faster. There's also an environment variable to force the use of go modules.

`images` list defines docker images that should be pushed to Container Registry. In the example above, the docker image is built in the `trainer-docker` step.

Deploys

Deploys to Cloud Run are done with the `gcloud run deploy` command. Cloud Build builds the docker image in the previous step, so we can use the latest image on the registry.

```
- id: trainer-http-deploy
  name: gcr.io/cloud-builders/gcloud
  entrypoint: ./scripts/deploy.sh
  args: [trainer, http, "$PROJECT_ID"]
  waitFor: [trainer-docker]
```

Source: cloudbuild.yaml on GitHub[34]

```
gcloud run deploy "$service-$server_to_run" \
    --image "gcr.io/$project_id/$service" \
    --region europe-west1 \
    --platform managed
```

Source: deploy.sh on GitHub[35]

Frontend is deployed with the `firebase` command. There's no need to use a helper script there, as there's just one frontend application.

```
- name: gcr.io/$PROJECT_ID/firebase
  args: ['deploy', '--project=$PROJECT_ID']
  dir: web
  waitFor: [web-build]
```

Source: cloudbuild.yaml on GitHub[36]

[33] https://bit.ly/3uicMgI
[34] https://bit.ly/2OSRDtb
[35] https://bit.ly/3s9lJXR
[36] https://bit.ly/2ZBjCQo

This step uses `gcr.io/$PROJECT_ID/firebase`[37] docker image. It doesn't exist by default, so we use another `null_resource` to build it from cloud-builders-community[38]:

```
resource "null_resource" "firebase_builder" {
  provisioner "local-exec" {
    command = "make firebase_builder"
  }

  depends_on = [google_project_service.container_registry]
}
```

Source: repo.tf on GitHub[39]

Cloud Build Permissions

All required permissions are defined in `iam.tf`. Cloud Build needs permissions for Cloud Run[40] (to deploy backend services) and Firebase[41] (to deploy frontend application).

First of all, we define account names as local variables to make the file more readable. If you're wondering where these names come from, you can find them in the Cloud Build documentation[42].

```
locals {
  cloud_build_member = "serviceAccount:${google_project.project.number}@cloudbuild.gserviceaccount.com"
  compute_account    = "projects/${var.project}/serviceAccounts/${google_project.project.number}-compute@d
}
```

Source: iam.tf on GitHub[43]

We then define all permissions, as described in the documentation. For example, here's **Cloud Build member** with **Cloud Run Admin role**:

```
resource "google_project_iam_member" "cloud_run_admin" {
  role   = "roles/run.admin"
  member = local.cloud_build_member

  depends_on = [google_project_service.cloud_build]
}
```

Source: iam.tf on GitHub[44]

[37] http://gcr.io/$PROJECT_ID/firebase
[38] https://github.com/GoogleCloudPlatform/cloud-builders-community.git
[39] https://bit.ly/3dAqyVY
[40] https://cloud.google.com/cloud-build/docs/deploying-builds/deploy-cloud-run
[41] https://cloud.google.com/cloud-build/docs/deploying-builds/deploy-firebase
[42] https://cloud.google.com/cloud-build/docs/deploying-builds/deploy-cloud-run
[43] https://bit.ly/3pG9YXb
[44] https://bit.ly/3bi6qVK

Dockerfiles

We have a couple of Dockerfiles defined in the project in the `docker`[45] directory.

- `app`[46] - Go service image for local development. It uses reflex for hot code recompilation. See our post about local Go environment[47] to learn how it works.
- `app-prod`[48] - production image for Go services. It builds the Go binary for a given service and runs it.
- `web`[49] - frontend application image for local development.

Setup

Usually, you would execute a Terraform project with `terraform apply`. This command applies all resources in the current directory.

Our example is a bit more complex because it sets up the whole GCP project along with dependencies. Makefile orchestrates the setup.

Requirements:

- `git`, `gcloud`, and `terraform` installed on your system.
- Docker
- A GCP account with a Billing Account.

Authentication

There are two credentials that you need for the setup.

Start with logging in with `gcloud`:

```
gcloud auth login
```

Then you need to obtain **Application Default Credentials**:

```
gcloud auth application-default login
```

This stores your credentials in a well-known place, so Terraform can use it.

[45] https://bit.ly/3dDa7Io
[46] https://bit.ly/3smkjsT
[47] https://threedots.tech/post/go-docker-dev-environment-with-go-modules-and-live-code-reloading/
[48] https://bit.ly/3keqBYR
[49] https://bit.ly/3dIJd29

Security warning!

Authenticating like this is the easiest way and okay for local development, but you probably don't want to use it in a production setup.
Instead, consider creating a service account with only a subset of necessary permissions (depending on what your Terraform configuration does). You can then download the JSON key and store its path in the `GOOGLE_CLOUD_KEYFILE_JSON` environment variable.

Picking a Region

During the setup, you need to pick Cloud Run region[50] and Firebase location[51]. These two are not the same thing (see the lists on linked documentation).

We picked `europe-west1` as the default region for Cloud Run. It's hardcoded on the repository in two files:

- `scripts/deploy.sh`[52] - for deploying the Cloud Run services
- `web/firebase.json`[53] - for routing Cloud Run public APIs

If you're fine with the defaults, you don't need to change anything. Just pick Firebase location that's close, i.e., `europe-west`.

If you'd like to use another region, you need to update the files mentioned above. It's important to **commit your changes** on `master` before you run the setup. It's not enough to change them locally!

Run it!

Clone the project repository[54] and enter the `terraform` directory from the command line. A single `make` command should be enough to start. See the included README[55] for more details.

[50] https://cloud.google.com/run/docs/locations
[51] https://firebase.google.com/docs/projects/locations
[52] https://bit.ly/2ZFxl8K
[53] https://bit.ly/3dyYdzv
[54] https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example
[55] https://bit.ly/3qM21AW

Billing warning!

This project goes beyond the Google Cloud Platform free tier. You need to have a billing account to use it. You can use the $300 free credit given to new accounts or create a new account in the Billing section[a].
For pricing, see Cloud Run Pricing[b]. The best way to estimate your project cost is to use the official calculator[c].
Wild Workouts should cost you up to $1 monthly if you use it just occasionally. We recommend deleting the project after you're done playing with it.
If you want to run it continuously, you should know that most of the cost comes from Cloud Build and depends on the number of builds (triggered by new commits on master). You can downgrade the `machineType` in `cloudbuild.yaml` to reduce some costs at the expense of longer build times.
It's also a good idea to set up a Budget Alert in the Billing Account settings. You will then receive a notification if something triggers unexpected charges.

[a] https://console.cloud.google.com/billing/create
[b] https://cloud.google.com/run/pricing
[c] https://cloud.google.com/products/calculator

At the end of the setup, a new git remote will be added to your local repository, called `google`. The master branch will be pushed to this remote, triggering your first Cloud Build.

If you'd like to make any changes to the project, you need to push to the correct origin, i.e., `git push google`. You can also update the `origin` remote with `git remote set-url`.

Note

If you keep your code on GitHub, GitLab, or another platform, you can set up a mirror instead of hosting the repository on Cloud Source Repositories.

In-depth

Env variables

All possible variables are defined in the `vars.tf` file. If a variable has no default value and you don't provide it, Terraform won't let you apply anything.

There are several ways to supply variables for `terraform apply`. You can pass them individually with `-var` flag or all at once with `-var-file` flag. Terraform also looks for them in `TF_VAR_name` environment variables.

We've picked the last option for this project. We need to use the environment variables anyway in Makefile and bash scripts. This way, there's just one source of truth.

Make runs `set-envs.sh` at the beginning of the setup. It's just a helper script to ask the user for all the required variables. These are then saved to `.env` file. You can edit this file manually as well.

Note Terraform does not automatically read it. It's sourced just before running `apply`:

```
load_envs:=source ./.env
# ...

apply:
    ${load_envs} && terraform apply
```

Source: Makefile on GitHub[56]

Building docker images

Because we're setting up the project from scratch, we run into a dilemma: we need docker images in the project's Container Registry to create Cloud Run instances, but we can't create the images without a project.

You could set the image name in Terraform definition to an example image and then let Cloud Build overwrite it. But then each `terraform apply` would try to change them back to the original value.

Our solution is to build the images based on the "hello" image first and then deploy Cloud Run instances. Then Cloud Build builds and deploys proper images, but the name stays the same.

```
resource "null_resource" "init_docker_images" {
  provisioner "local-exec" {
    command = "make docker_images"
  }

  depends_on = [google_project_service.container_registry]
}
```

Source: docker-images.tf on GitHub[57]

Note the `depends_on` that points at Container Registry API. It ensures Terraform won't start building images until there's a registry where it can push them.

Destroying

You can delete the entire project with `make destroy`. If you take a look in the Makefile, there's something unusual before `terraform destroy`:

```
terraform state rm "google_project_iam_member.owner"
terraform state rm "google_project_service.container_registry"
terraform state rm "google_project_service.cloud_run"
```

Source: Makefile on GitHub[58]

These commands are a workaround for Terraform's lack of a "skip destroy" feature[59]. They remove some resources from Terraform's local state file. Terraform won't destroy these resources (as far as it's concerned, they don't exist

[56] https://bit.ly/2M7eqjU
[57] https://bit.ly/3qIqqaA
[58] https://bit.ly/3bpLVGP
[59] https://github.com/hashicorp/terraform/issues/23547

at this point).

We don't want to destroy the owner IAM policy, because if something goes wrong during the destroy, you will lock yourself out of the project and won't be able to access it.

The other two lines are related with enabled APIs — there's a possible race condition where some resources would still use these APIs during destroy. By removing them from the state, we avoid this issue.

A note on magic

Our goal was to make it as easy as possible to set up the whole project. Because of that, we had to cut some corners where Terraform is missing features, or an API is not yet available. There's also some Makefile and bash magic involved that's not always easy to understand.

I want to make this clear because you will likely encounter similar dilemmas in your projects. You will need to choose between fully-automated solutions glued together with alpha APIs or using plain Terraform with some manual steps documented in the project.

Both approaches have their place. For example, this project is straightforward to set up multiple times with the same configuration. So if you'd like to create the exact copy of the production environment, you can have a completely separate project working within minutes.

If you keep just one or two environments, you won't need to recreate them every week. It's then probably fine to stick to the available APIs and document the rest.

Is serverless the way to go?

This project is our first approach to "serverless" deployment, and at first, we had some concerns. Is this all just hype, or is serverless the future?

Wild Workouts is a fairly small project that might not show all there is to Cloud Run. Overall we found it quite simple to set up, and it nicely hides all complexities. It's also more natural to work with than Cloud Functions.

After the initial setup, there shouldn't be much infrastructure maintenance needed. You don't have to worry about keeping up a Docker registry or a Kubernetes cluster and can instead focus on creating the application.

On the other hand, it's also quite limited in features, as there are only a few protocols supported. It looks like a great fit for services with a REST API. The pricing model also seems reasonable, as you pay only for the used resources.

What about vendor lock-in?

The entire Terraform configuration is now tied to GCP, and there is no way around it. If we'd like to migrate the project to AWS, a new configuration will be needed.

However, there are some universal concepts. Services running in docker images can be deployed on any platform, whether it's a Kubernetes cluster or docker-compose. Most platforms also offer some kind of registry for images.

Is writing a new Terraform configuration all that's required to migrate out of Google? Not really, as the application itself is coupled to Firebase and Authentication offered by GCP. We will show a better approach to this problem in future chapters in this book.

What's next?

This chapter just scratches the surface of Terraform. There are many advanced topics you should take a look at if you consider using it. For example, the remote state[60] is necessary when working in a team.

[60]https://www.terraform.io/docs/state/remote.html

Chapter 15

Running integration tests in the CI/CD pipeline

Miłosz Smółka

This post is a direct follow-up to **Tests Architecture** (Chapter 12) where I've introduced new kinds of tests to our example project.

Wild Workouts[1] uses Google Cloud Build as CI/CD platform. It's configured in a **continuous deployment** manner, meaning the changes land on production as soon as the pipeline passes. If you consider our current setup, it's both brave and naive. We have no tests running there that could save us from obvious mistakes (the not-so-obvious mistakes can rarely be caught by tests, anyway).

In this chapter I will show how to run integration, component, and end-to-end tests on Google Cloud Build using docker-compose.

The current config

Let's take a look at the current `cloudbuild.yaml` file. While it's pretty simple, most steps are being run several times as we keep 3 microservices in a single repository. I focus on the backend part, so I will skip all config related to frontend deployment now.

```
steps:
  - id: trainer-lint
    name: golang
    entrypoint: ./scripts/lint.sh
    args: [trainer]
  - id: trainings-lint
    name: golang
```

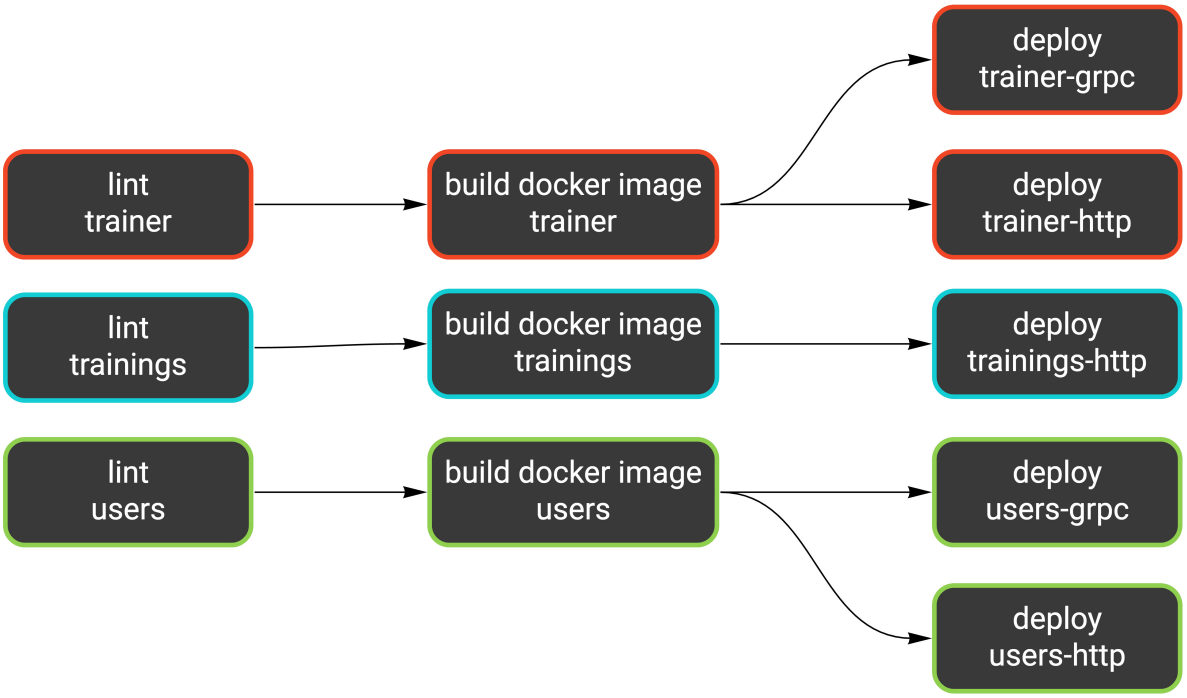
[1] https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example

```
    entrypoint: ./scripts/lint.sh
    args: [trainings]
  - id: users-lint
    name: golang
    entrypoint: ./scripts/lint.sh
    args: [users]

  - id: trainer-docker
    name: gcr.io/cloud-builders/docker
    entrypoint: ./scripts/build-docker.sh
    args: ["trainer", "$PROJECT_ID"]
    waitFor: [trainer-lint]
  - id: trainings-docker
    name: gcr.io/cloud-builders/docker
    entrypoint: ./scripts/build-docker.sh
    args: ["trainings", "$PROJECT_ID"]
    waitFor: [trainings-lint]
  - id: users-docker
    name: gcr.io/cloud-builders/docker
    entrypoint: ./scripts/build-docker.sh
    args: ["users", "$PROJECT_ID"]
    waitFor: [users-lint]

  - id: trainer-http-deploy
    name: gcr.io/cloud-builders/gcloud
    entrypoint: ./scripts/deploy.sh
    args: [trainer, http, "$PROJECT_ID"]
    waitFor: [trainer-docker]
  - id: trainer-grpc-deploy
    name: gcr.io/cloud-builders/gcloud
    entrypoint: ./scripts/deploy.sh
    args: [trainer, grpc, "$PROJECT_ID"]
    waitFor: [trainer-docker]
  - id: trainings-http-deploy
    name: gcr.io/cloud-builders/gcloud
    entrypoint: ./scripts/deploy.sh
    args: [trainings, http, "$PROJECT_ID"]
    waitFor: [trainings-docker]
  - id: users-http-deploy
    name: gcr.io/cloud-builders/gcloud
    entrypoint: ./scripts/deploy.sh
    args: [users, http, "$PROJECT_ID"]
    waitFor: [users-docker]
  - id: users-grpc-deploy
    name: gcr.io/cloud-builders/gcloud
    entrypoint: ./scripts/deploy.sh
    args: [users, grpc, "$PROJECT_ID"]
    waitFor: [users-docker]
```

Source: cloudbuild.yaml on GitHub[2]

Notice the `waitFor` key. It makes a step wait only for other specified steps. Some jobs can run in parallel this way.

Here's a more readable version of what's going on:



We have a similar workflow for each service: lint (static analysis), build the Docker image, and deploy it as one or two Cloud Run services.

Since our test suite is ready and works locally, we need to figure out how to plug it in the pipeline.

Docker Compose

We already have one docker-compose definition, and I would like to keep it this way. We will use it for:

- running the application locally,
- running tests locally,
- running tests in the CI.

These three targets have different needs. For example, when running the application locally, we want to have hot

[2] https://bit.ly/3u4O8PQ

code reloading. But that's pointless in the CI. On the other hand, we can't expose ports on `localhost` in the CI, which is the easiest way to reach the application in the local environment.

Luckily docker-compose is flexible enough to support all of these use cases. We will use a base `docker-compose.yml` file and an additional `docker-compose.ci.yml` file with overrides just for the CI. You can run it by passing both files using the `-f` flag (notice there's one flag for each file). Keys from the files will be merged in the provided order.

```
docker-compose -f docker-compose.yml -f docker-compose.ci.yml up -d
```

> **Note**
>
> Typically, docker-compose looks for the `docker-compose.yml` file in the current directory or parent directories. Using the `-f` flag disables this behavior, so only specified files are parsed.

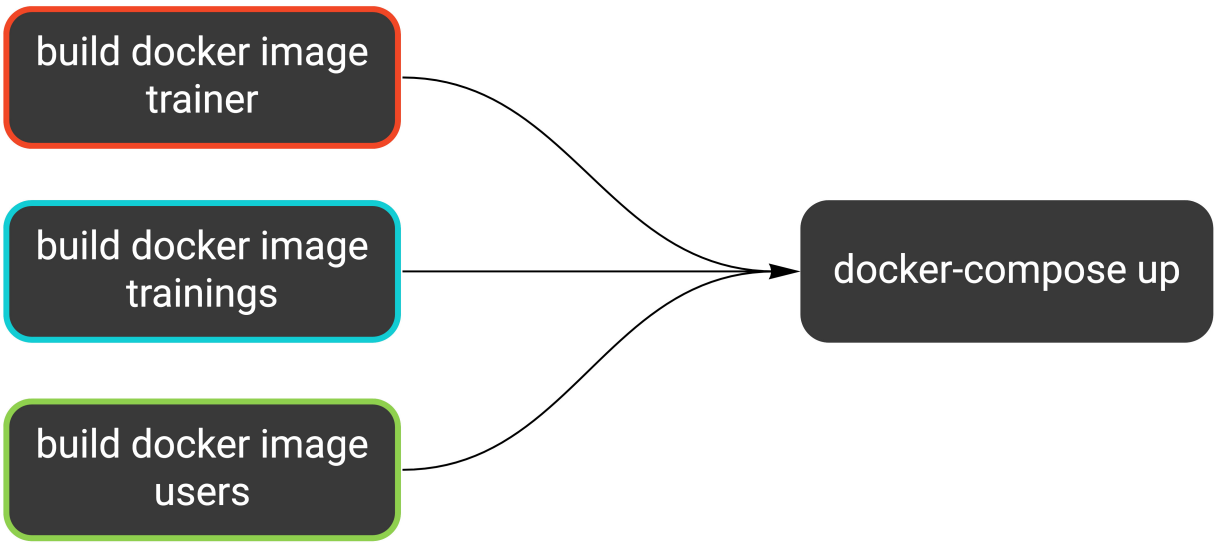
To run it on Cloud Build, we can use the `docker/compose` image.

```
- id: docker-compose
  name: 'docker/compose:1.19.0'
  args: ['-f', 'docker-compose.yml', '-f', 'docker-compose.ci.yml', 'up', '-d']
  env:
    - 'PROJECT_ID=$PROJECT_ID'
  waitFor: [trainer-docker, trainings-docker, users-docker]
```

Source: cloudbuild.yaml on GitHub[3]

Since we filled `waitFor` with proper step names, we can be sure the correct images are present. This is what we've just added:

[3] https://bit.ly/39xjcQb



The first override we add to `docker-compose.ci.yml` makes each service use docker images by the tag instead of building one from `docker/app/Dockerfile`. This ensures our tests check the same images we're going to deploy.

Note the `${PROJECT_ID}` variable in the `image` keys. This needs to be the production project, so we can't hardcode it in the repository. Cloud Build provides this variable in each step, so we just pass it to the `docker-compose up` command (see the definition above).

```
services:
  trainer-http:
    image: "gcr.io/${PROJECT_ID}/trainer"

  trainer-grpc:
    image: "gcr.io/${PROJECT_ID}/trainer"

  trainings-http:
    image: "gcr.io/${PROJECT_ID}/trainings"

  users-http:
    image: "gcr.io/${PROJECT_ID}/users"

  users-grpc:
    image: "gcr.io/${PROJECT_ID}/users"
```

Source: docker-compose.ci.yml on GitHub[4]

[4] https://bit.ly/2PqNMEr

Network

Many CI systems use Docker today, typically running each step inside a container with the chosen image. Using docker-compose in a CI is a bit trickier, as it usually means running Docker containers from within a Docker container.

On Google Cloud Build, all containers live inside the `cloudbuild` network[5]. Simply adding this network as the default one for our `docker-compose.ci.yml` is enough for CI steps to connect to the docker-compose services.

Here's the second part of our override file:

```
networks:
  default:
    external:
      name: cloudbuild
```

Source: docker-compose.ci.yml on GitHub[6]

Environment variables

Using environment variables as configuration seems simple at first, but it quickly becomes complex considering how many scenarios we need to handle. Let's try to list all of them:

- running the application locally,
- running component tests locally,
- running component tests in the CI,
- running end-to-end tests locally,
- running end-to-end tests in the CI.

I didn't include running the application on production, as it doesn't use docker-compose.

Why component and end-to-end tests are separate scenarios? The former spin up services on demand and the latter communicate with services already running within docker-compose. It means both types will use different endpoints to reach the services.

> **Note**
>
> For more details on component and end-to-end tests see **Tests Architecture** (Chapter 12). The TL;DR version is: we focus coverage on component tests, which don't include external services. End-to-end tests are there just to confirm the contract is not broken on a very high level and only for the most critical path. This is the key to decoupled services.

We already keep a base `.env` file that holds most variables. It's passed to each service in the docker-compose definition.

[5] https://cloud.google.com/build/docs/build-config#network
[6] https://bit.ly/3sHojEW

Additionally, docker-compose loads this file automatically when it finds it in the working directory. Thanks to this, we can use the variables inside the yaml definition as well.

```
services:
  trainer-http:
    build:
      context: docker/app
    ports:
      # The $PORT variable comes from the .env file
      - "127.0.0.1:3000:$PORT"
    env_file:
      # All variables from .env are passed to the service
      - .env
    # (part of the definition omitted)
```

Source: docker-compose.yml on GitHub[7]

We also need these variables loaded when running tests. That's pretty easy to do in bash:

```
source .env
# exactly the same thing
. .env
```

However, the variables in our `.env` file have no `export` prefix, so they won't be passed further to the applications running in the shell. We can't use the prefix because it wouldn't be compatible with the syntax docker-compose expects.

Additionally, we can't use a single file with all scenarios. We need to have some variable overrides, just like we did with the docker-compose definition. My idea is to keep one additional file for each scenario. It will be loaded together with the base `.env` file.

Let's see what's the difference between all scenarios. For clarity, I've included only users-http, but the idea will apply to all services.

Scenario	MySQL host	Firestore host	users-http address	File
Running locally	localhost	localhost	localhost:3002	.env[8]
Local component tests	localhost	localhost	localhost:5002	.test.env[9]
CI component tests	mysql	firestore-component-tests	localhost:5002	.test.ci.env[10]
Local end-to-end tests	-	-	localhost:3002	.e2e.env[11]
CI end-to-end tests	-	-	users-http:3000	.e2e.ci.env[12]

[7] https://bit.ly/31uSXWd
[8] https://bit.ly/2PhgMyu
[9] https://bit.ly/31zBMmi
[10] https://bit.ly/2Odc5VO
[11] https://bit.ly/3dq7TKH
[12] https://bit.ly/39wDwRQ

Services ran by docker-compose use ports 3000+, and component tests start services on ports 5000+. This way, both instances can run at the same time.

I've created a bash script that reads the variables and runs tests. **Please don't try to define such a complex scenario directly in the Makefile. Make is terrible at managing environment variables.** Your mental health is at stake.

The other reason for creating a dedicated script is that we keep 3 services in one repository and end-to-end tests in a separate directory. If I need to run the same command multiple times, I prefer calling a script with two variables rather than a long incantation of flags and arguments.

The third argument in favor of separate bash scripts is they can be linted with shellcheck[13].

```
#!/bin/bash
set -e

readonly service="$1"
readonly env_file="$2"

cd "./internal/$service"
env $(cat "../../.env" "../../$env_file" | grep -Ev '^#' | xargs) go test -count=1 ./...
```

Source: test.sh on GitHub[14]

The script runs `go test` in the given directory with environment variables loaded from `.env` and the specified file. The `env / xargs` trick passes all variables to the following command. Notice how we remove comments from the file with `grep`.

Testing cache

`go test` caches successful results, as long as the related files are not modified.
With tests that use Docker, it may happen that you will change something on the infrastructure level, like the `docker-compose` definition or some environment variables. `go test` won't detect this, and you can mistake a cached test for a successful one.
It's easy to get confused over this, and since our tests are fast anyway, we can disable the cache. The `-count=1` flag is an idiomatic (although not obvious) way to do it.

Running tests

I have the end-to-end tests running after tests for all the services passed. It should resemble the way you would usually run them. **Remember, end-to-end tests should work just as a double-check, and each service's own tests should have the most coverage.**

Because our end-to-end tests are small in scope, we can run them before deploying the services. If they were running

[13] https://www.shellcheck.net/
[14] https://bit.ly/2PlQtXG

for a long time, this could block our deployments. A better idea in this scenario would be to rely on each service's component tests and run the end-to-end suite in parallel.

```
- id: trainer-tests
  name: golang
  entrypoint: ./scripts/test.sh
  args: ["trainer", ".test.ci.env"]
  waitFor: [docker-compose]
- id: trainings-tests
  name: golang
  entrypoint: ./scripts/test.sh
  args: ["trainings", ".test.ci.env"]
  waitFor: [docker-compose]
- id: users-tests
  name: golang
  entrypoint: ./scripts/test.sh
  args: ["users", ".test.ci.env"]
  waitFor: [docker-compose]
- id: e2e-tests
  name: golang
  entrypoint: ./scripts/test.sh
  args: ["common", ".e2e.ci.env"]
  waitFor: [trainer-tests, trainings-tests, users-tests]
```

Source: cloudbuild.yaml on GitHub[15]

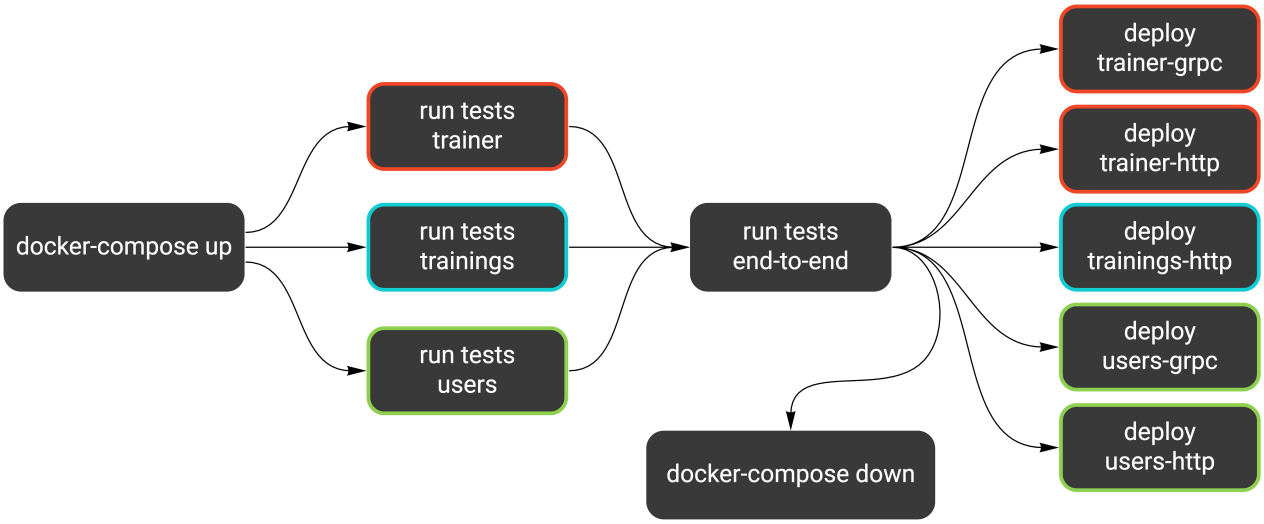
The last thing we add is running `docker-compose down` after all tests have passed. This is just a cleanup.

```
- id: docker-compose-down
  name: 'docker/compose:1.19.0'
  args: ['-f', 'docker-compose.yml', '-f', 'docker-compose.ci.yml', 'down']
  env:
    - 'PROJECT_ID=$PROJECT_ID'
  waitFor: [e2e-tests]
```

Source: cloudbuild.yaml on GitHub[16]

The second part of our pipeline looks like this now:

[15] https://bit.ly/31y3f84
[16] https://bit.ly/3cGGo0C



Here's how running the tests locally looks like (I've introduced this `make` target in the previous chapter). It's exactly the same commands as in the CI, just with different `.env` files.

```
test:
    @./scripts/test.sh common .e2e.env
    @./scripts/test.sh trainer .test.env
    @./scripts/test.sh trainings .test.env
    @./scripts/test.sh users .test.env
```

Source: Makefile on GitHub[17]

Separating tests

Looking at the table from the previous chapter, we could split tests into two groups, depending on whether they use Docker.

Feature	Unit	Integration	Component	End-to-End
Docker database	**No**	**Yes**	**Yes**	**Yes**

Unit tests are the only category not using a Docker database, while integration, component, and end-to-end tests do.

Even though we made all our tests fast and stable, setting up Docker infrastructure adds some overhead. It's helpful to run all unit tests separately, to have a first guard against mistakes.

We could use build tags to separate tests that don't use Docker. You can define the build tags in the first line of a file.

[17] https://bit.ly/3ftPIXp

```
// +build docker
```

We could now run unit tests separately from all tests. For example, the command below would run only tests that need Docker services:

```
go test -tags=docker ./...
```

Another way to separate tests is using the `-short` flag and checking `testing.Short()`[18] in each test case.

In the end, I decided to not introduce this separation. We made our tests stable and fast enough that running them all at once is not an issue. However, our project is pretty small, and the test suite covers just the critical paths. When it grows, it might be a good idea to introduce the build tags in component tests.

Digression: A short story about CI debugging

While I was introducing changes for this chapter, the initial test runs on Cloud Build kept failing. According to logs, tests couldn't reach services from docker-compose.

So I started debugging and added a simple bash script that would connect to the services via `telnet`.

To my surprise, connecting to `mysql:3306` worked correctly, but `firestore:8787` didn't, and the same for all Wild Workouts services.

I thought this is because docker-compose takes a long time to start, but any number of retries didn't help. Finally, I decided to try something crazy, and I've set up a reverse SSH tunnel from one of the containers in docker-compose.

This allowed me to SSH inside one of the containers while the build was still running. I then tried using `telnet` and `curl`, and they worked correctly for all services.

Finally, I spotted a bug in the bash script I've used.

```
readonly host="$1"
readonly port="$1"

# (some retries code)

telnet "$host" "$port"
```

The typo in variables definition caused the `telnet` command to run like this: `telnet $host $host`. So why it worked for MySQL? It turns out, telnet recognizes ports defined in `/etc/services`. So `telnet mysql mysql` got translated to `telnet mysql 3306` and worked fine, but it failed for any other service.

Why the tests failed though? Well, it turned out to be a totally different reason.

Originally, we connected to MySQL like this:

```
config := mysql.Config{
   Addr:      os.Getenv("MYSQL_ADDR"),
   User:      os.Getenv("MYSQL_USER"),
```

[18] https://golang.org/pkg/testing/#Short

```
    Passwd:    os.Getenv("MYSQL_PASSWORD"),
    DBName:    os.Getenv("MYSQL_DATABASE"),
    ParseTime: true, // with that parameter, we can use time.Time in mysqlHour.Hour
}

db, err := sqlx.Connect("mysql", config.FormatDSN())
if err != nil {
    return nil, errors.Wrap(err, "cannot connect to MySQL")
}
```

Source: hour_mysql_repository.go on GitHub[19]

I looked into environment variables, and all of them were filled correctly. After adding some `fmt.Println()` debugs, **I've found out the config's `Addr` part is completely ignored by the MySQL client because we didn't specify the `Net` field.** Why it worked for local tests? Because MySQL was exposed on `localhost`, which is the default address.

The other test failed to connect to one of Wild Workouts services, and it turned out to be because I've used incorrect port in the .env file.

Why I'm sharing this at all? I think it's a great example of how working with CI systems can often look like. **When multiple things can fail, it's easy to come to wrong conclusions and dig deep in the wrong direction.**

When in doubt, I like to reach for basic tools for investigating Linux issues, like `strace`, `curl`, or `telnet`. That's also why I did the reverse SSH tunnel, and I'm glad I did because it seems like a great way to debug issues inside the CI. I feel I will use it again sometime.

Summary

We managed to keep a single docker-compose definition for running tests locally and in the pipeline. The entire Cloud Build run from `git push` to production takes 4 minutes.

We've used a few clever hacks, but that's just regular stuff when dealing with CI. Sometimes you just can't avoid adding some bash magic to make things work.

In contrast to the domain code, hacks in your CI setup shouldn't hurt you too much, as long as there are only a few of them. Just make sure it's easy to understand what's going on, so you're not the only person with all the knowledge.

[19] https://bit.ly/2PlVyiT

Chapter 16

Introduction to Strategic DDD

Robert Laszczak

A couple of years ago, I worked in a SaaS company that suffered from **probably all possible issues with software development**. Code was so complex that adding simples changes could take months. All tasks and the scope of the project were defined by the project manager alone. Developers didn't understand what problem they were solving. Without an understanding the customer's expectations, many implemented functionalities were useless. The development team was also not able to propose better solutions.

Even though we had microservices, introducing one change often required changes in most of the services. The architecture was so tightly coupled that we were not able to deploy these "microservices" independently. The business didn't understand why adding *"one button"* may take two months. In the end, stakeholders didn't trust the development team anymore. We all were very frustrated. **But the situation was not hopeless.**

I was lucky enough to be a bit familiar with Domain-Driven Design. I was far from being an expert in that field at that time. But my knowledge was solid enough to help the company minimize and even eliminate a big part of mentioned problems.

Some time has passed, and these problems are not gone in other companies. Even if the solution for these problems exists and is not arcane knowledge. People seem to not be aware of that. Maybe it's because old techniques like GRASP *(1997)*, SOLID *(2000)* or DDD (Domain-Driven Design) *(2003)* are often forgotten or considered obsolete? It reminds me of the situation that happened in the historical Dark Ages, when the ancient knowledge was forgotten. Similarly, we can use the old ideas. They're still valid and can solve present-day issues, but they're often ignored. **It's like we live in Software Dark Ages.**

Another similarity is focusing on the wrong things. In the historical Dark Ages, religion put down science. In Software Dark Ages, infrastructure is putting down important software design techniques. I'm not claiming that religion is not important. Spiritual life is super important, but not if you are suffering from hunger and illness. It's the same case for the infrastructure. **Having awesome Kubernetes cluster and most fancy microservices infrastructure will not help you if your software design sucks.**

Figure 16.1: Let me try to put another flowered and useless tool in the production…

Software Dark Ages as a system problem

Software Dark Ages is a very strong self-perpetuating system. You can't fix the systemic problem without understanding the big picture. *System thinking* is a technique that helps to analyze such complex issues set. I used this technique to visualize the Software Dark Ages.

You can see one big loop of things that accelerate each other. Without intervention, problems become bigger and bigger. How can Domain-Driven Design fix that?

Unlike most famous programming gurus, we don't want you to just believe in our story. We could just make it up. **Fortunately, we can explain why it works by science.** More precisely, with the excellent *Accelerate: The Science of Lean Software and DevOps* book based on scientific research. The research described in the book mentions the characteristics of the best, and worst-performing teams. One of the most critical factors is loosely coupled architecture.

If you think that microservices may give you loosely coupled architecture – you are very wrong. I've seen microservices that are more coupled than a monolith multiple times. This is why we need something more than just infrastructure solutions. This is the time when Domain-Driven Design (DDD) comes into play.

> 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.
>
> (…) employing the latest whizzy microservices architecture deployed on containers is no guarantee of higher performance if you ignore these characteristics.
>
> (…)
>
> Architectural approaches that enable this strategy include the use of bounded contexts and APIs as a way to decouple large domains into smaller, more loosely coupled units, and the use of test doubles and virtualization as a way to test services or components in isolation.
>
> ()

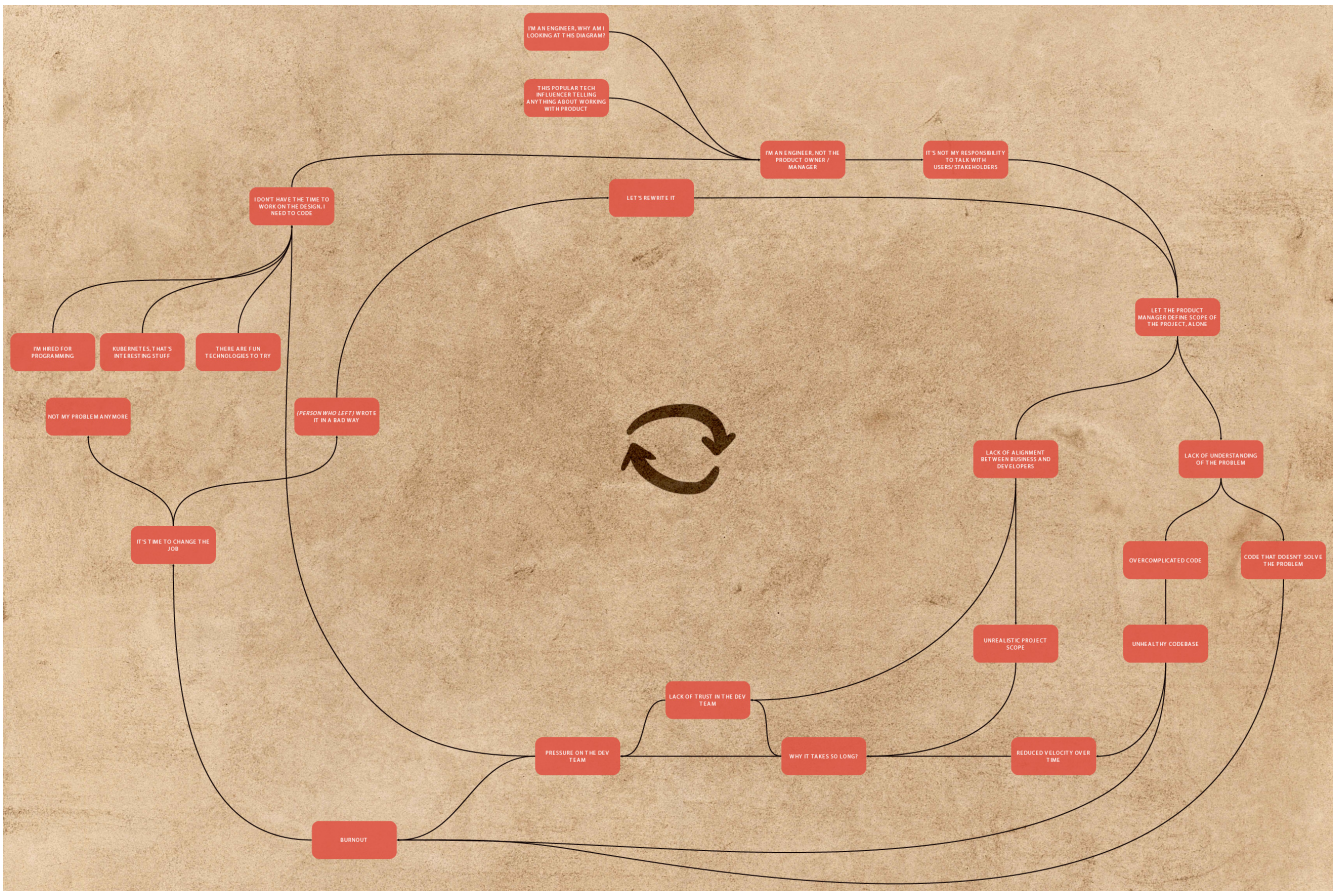


Figure 16.2: The ultimate guide to creating the worst team ever. See the full version.

DDD doesn't work

Maybe you know someone who tried DDD, and it didn't work for them?

Maybe you worked with a person who didn't understand it well, tried to force these techniques, and made everything too complex?

Maybe you've seen on Twitter that some famous software engineer said that DDD doesn't work?

Maybe for you, it's a legendary Holy Grail that someone claims work for them – but nobody has seen it yet.

Figure 16.3: We know what we are doing… kinda

Let's not forget that we are living in the Software Dark Ages. There's one problem with ideas from the previous epoch – there is a chance that some people may miss the initial point of DDD. That's not surprising in the context of 2003 when DDD was proposed for the first time.

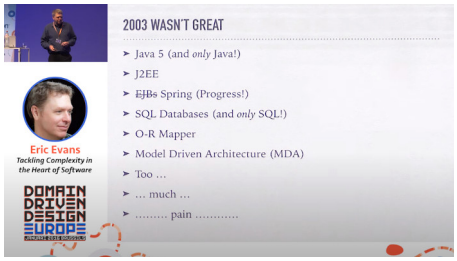


Figure 16.4: DDD in 2003

It's not easy to enter the DDD world. A lot of books and articles are missing the most important DDD points by oversimplifying them. They are also often explained in abstract examples that are detached from reality. It's also not rare to see too long and too complex examples that are impossible to understand.

Let me try to explain DDD in the simplest way.

From the Dark Ages to the Renaissance

DDD techniques can be split into two parts. Tactical and strategic patterns. Tactical Domain-Driven Design patterns are about **how** to implement the solution in the code. There is no rocket science in the Tactical DDD – it's all about Object-Oriented Programming good practices. But before writing the code, you need to know **what** to implement. That's where Strategic DDD patterns come into the game.

Many sources describing DDD spend most of the time covering tactical patterns. Sometimes, they even skip strategic patterns. **You can practice DDD by using just the strategic patterns**. In some projects using tactical DDD patterns is even overkill. Unfortunetly, most **people are doing the totally opposite thing. They use just tactical patterns without the strategic part. That's super horrifying.**

If anybody asked me if any silver bullet exists in product software engineering, I'd have only one candidate: *Domain-Driven Design Strategic Patterns*. Strategic DDD helps us to get the answer about:

- **what problem you are solving?**
- **will your solution meet stakeholders and users expectations?**
- **how complex is the project?**
- **what features are not necessary?**
- **how to separate services to support fast development in the long term?**

These questions are essential while implementing a new project, adding new functionality, or doing refactoring. **Strategic DDD patterns give us a way to answer these questions consistently and predictably.**

Some engineers tell me they are "just engineers". They don't care too much about who uses their software or why. They are just implementing, say, JIRA tasks – building services with some bytes on the input and some bytes on the output. Such a mindset leads to a big disconnection between engineers and their clients whose problems we, as engineers, are trying to solve. Without proper communication, it's much harder to create solutions that help clients in the right way. This is the goal in the end – not simply to process bytes.

It's tempting to spend a small amount of time on the planning phase of the project. To start coding as soon as possible and finish earlier. When anybody has such doubt, I like to say that *"With 5 days of coding, you can save 1 day of planning"*. The unasked questions will not magically disappear.

The best way to overcome the Software Dark Ages is to attack it from multiple angles. Let's see how DDD patterns can attack the system.

Event Storming

Event Storming is a game-changer for Strategic DDD patterns and software development in general. **I can't believe why it's not adopted by every team in the world yet.**

Event Storming is a workshop during which people with questions (often developers) meet with people with answers (often stakeholders). During the session, they can quickly explore complex business domains. In the beginning, you are focusing on building an entirely working flow based on *Domain Events* (orange sticky notes). Event Storming is a super flexible method. Thanks to that, you can verify if your solution meets expected requirements. You can also explore data flow, potential problems, or UX depending on the session's goal.

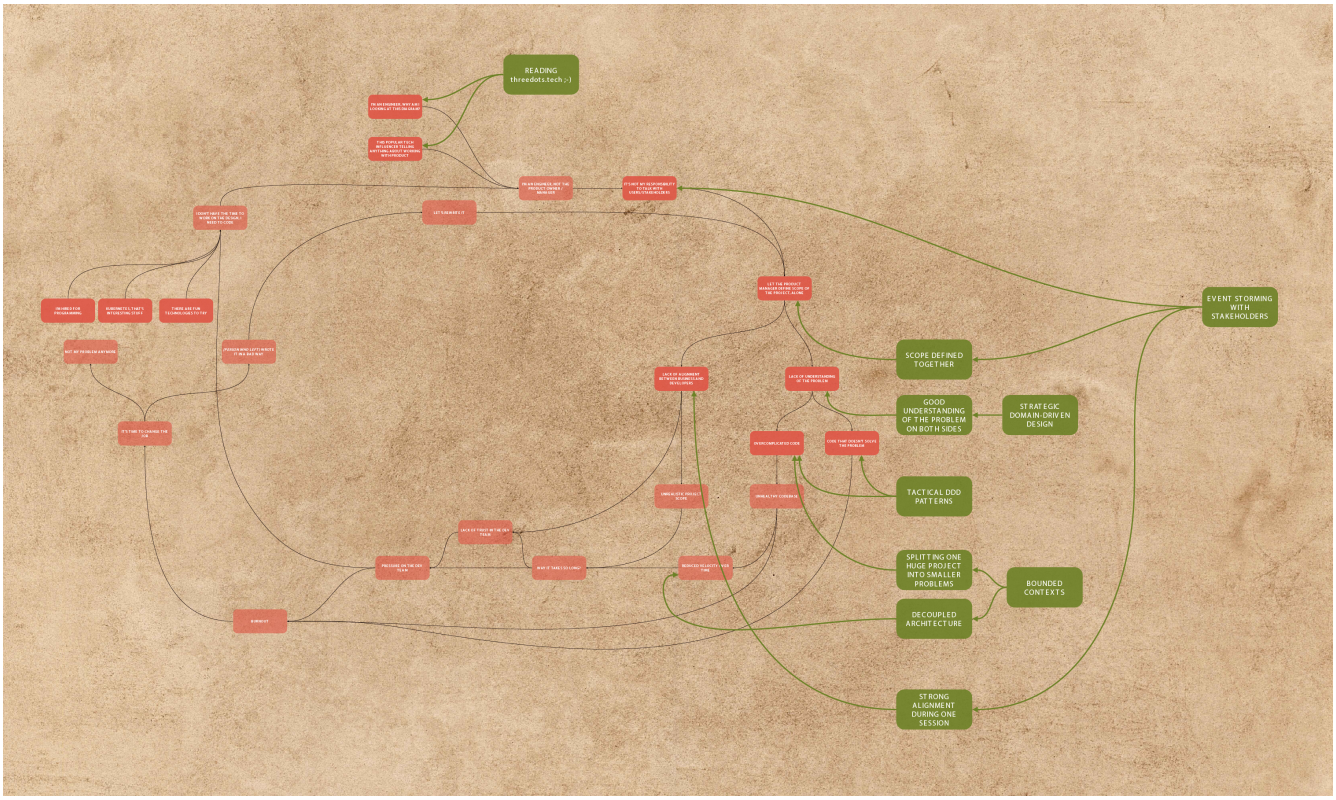
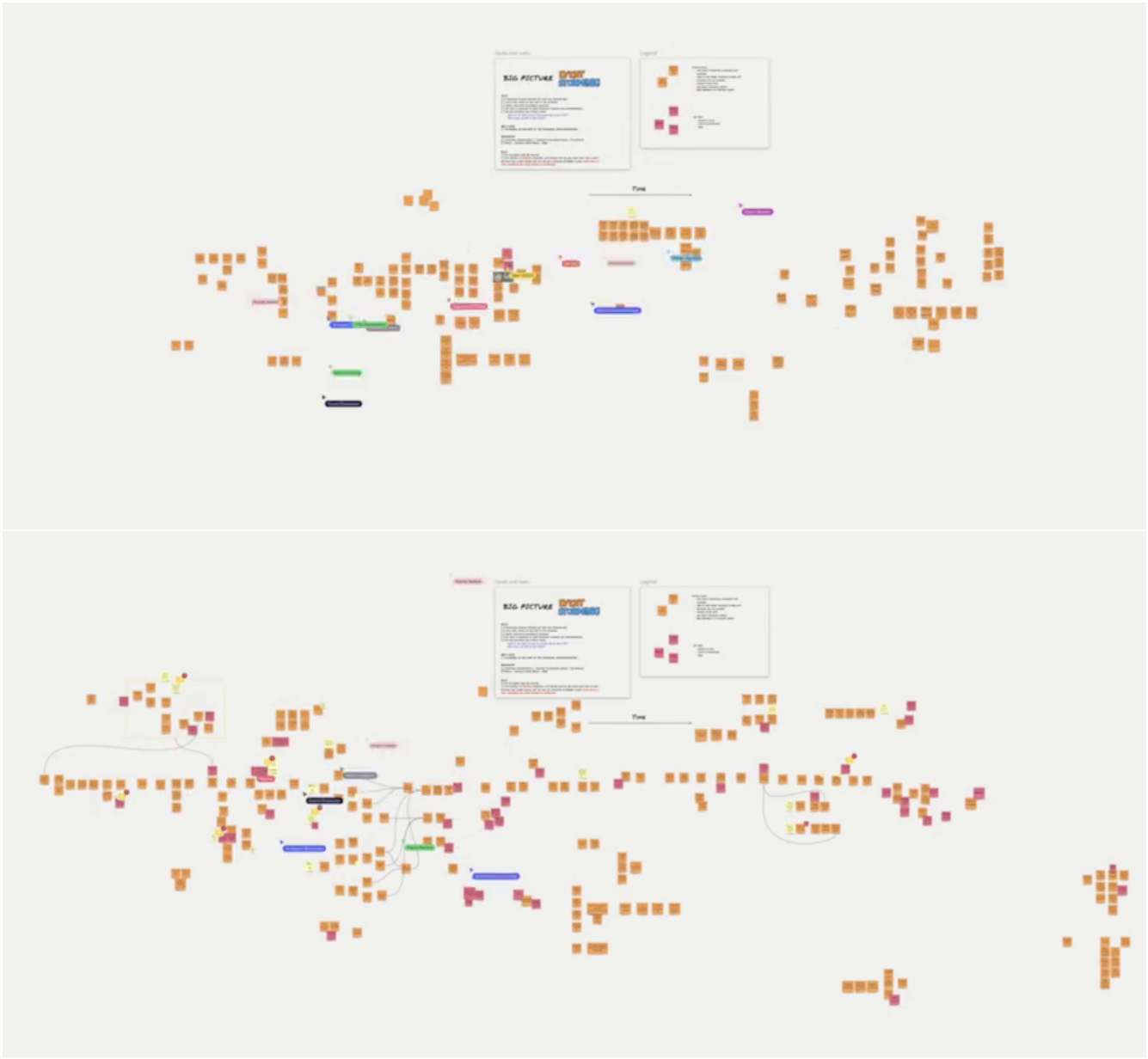


Figure 16.5: See the full version.



Verifying if the solution has no gaps and is about what users asked takes minutes. Introducing changes and verifying ideas in the developed and deployed code is hugely more expensive. Changing a sticky note on the board is extremely cheap.

A civil engineer or a rocket engineer can quickly see the result of a mistake. They can see that something is clearly wrong before finishing the building process. It is not so simple with software because it's not easily seen. Most of our critical decisions will not hurt anybody. Problems with features development and maintenance will not appear in a day.

Event Storming works when you are **planning both a big project or just a single story.** It's just a matter of how much time you would like to spend. When we are using it for one story, it can be something between 10 minutes

to a couple of hours. We tend to spend between one day to a couple of days of sessions for a bigger functionality.

After the session, you should have the correct answer for the questions about:

- **what problems are you trying to solve** – instead of guessing what may be useful for the end-user or assuming that *"we know everything better"*,
- **if stakeholders are happy with a proposed solution** – rather than verifying that half-year later with the implementation,
- **complexity of the problem is easily visible** – it makes clear why adding one button may require a ton of work,
- **initial idea of how you can split your microservices by responsibilities** – instead of blindly grouping "similar things".

In the end, you will end up with much more **trust from stakeholders** because you are planning a solution **together**. It's a much better approach than isolated coding in the basement.

What's excellent about Event Storming is that the outcome of a properly done session can be mapped directly to the code. It should help you avoid many discussions during development and speed up your work a lot.

Figure 16.6: Transforming Event Storming directly to the code.

You have to start with a clear purpose. Time can fly on a project, and before you know it, you have spent half a year on a project only to find that it's not useful to anyone. Have you experienced that? It happens more often than you might think, which is why some people lose trust in "engineers" and how we can end up as developers without any autonomy.

It's common to fear how much time we need to "lose" for running the session. **Thinking about time lost for running session is not the right approach. You should instead think about the benefits that you will lose if you will not run the session.** I heard the story when running one Event Storming session stopped the implementation of the project for a couple months. It may sound bad, but during the session the team found that current assumptions are totally invalid. Continuation of the project would lead to a complete failure. Even if the session may look time-consuming in the short term, the company avoided a couple of months of useless development.

Event Modeling

In 2018 Adam Dymitruk proposed the Event Modeling technique. The notation and idea are heavily based on the Event Storming technique but adds a couple of new features. It also puts an extra emphasis on the UX part of the session.

In general, these techniques are pretty compatible. Even if you stay with Event Storming, you may find some valuable approaches from Event Modeling that you may use.

You can read more about the technique on eventmodeling.org[1].

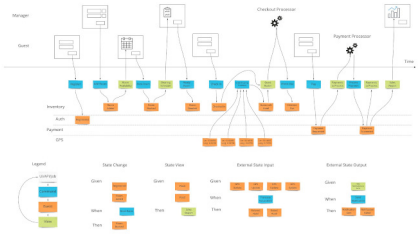


Figure 16.7: Event Modeling

Bounded Context and transaction boundaries (aggregates)

Bounded Context is another Strategic DDD pattern that helps us split big models into smaller, logical pieces.

It's a key for achieving **proper** services separation. If you need to touch half of the system to implement and test new functionality, your separation is wrong.

Alternative to the wrong separation is lack of separation. Often, a symptom of lack of separation is god objects (huge objects that know too much or do too much). In that case, changes will primarily affect one service. The cost of that approach is a higher risk of big system failure and higher complexity of changes.

In other words – in both cases, it will be harder to develop your project.

A great tool that helps with the discovery of *Bounded Context* and *Aggregates* is (of course) Event Storming.

As a result of the session, you can visually see how you should split your services and touchpoints between them.

[1] https://eventmodeling.org/posts/what-is-event-modeling/

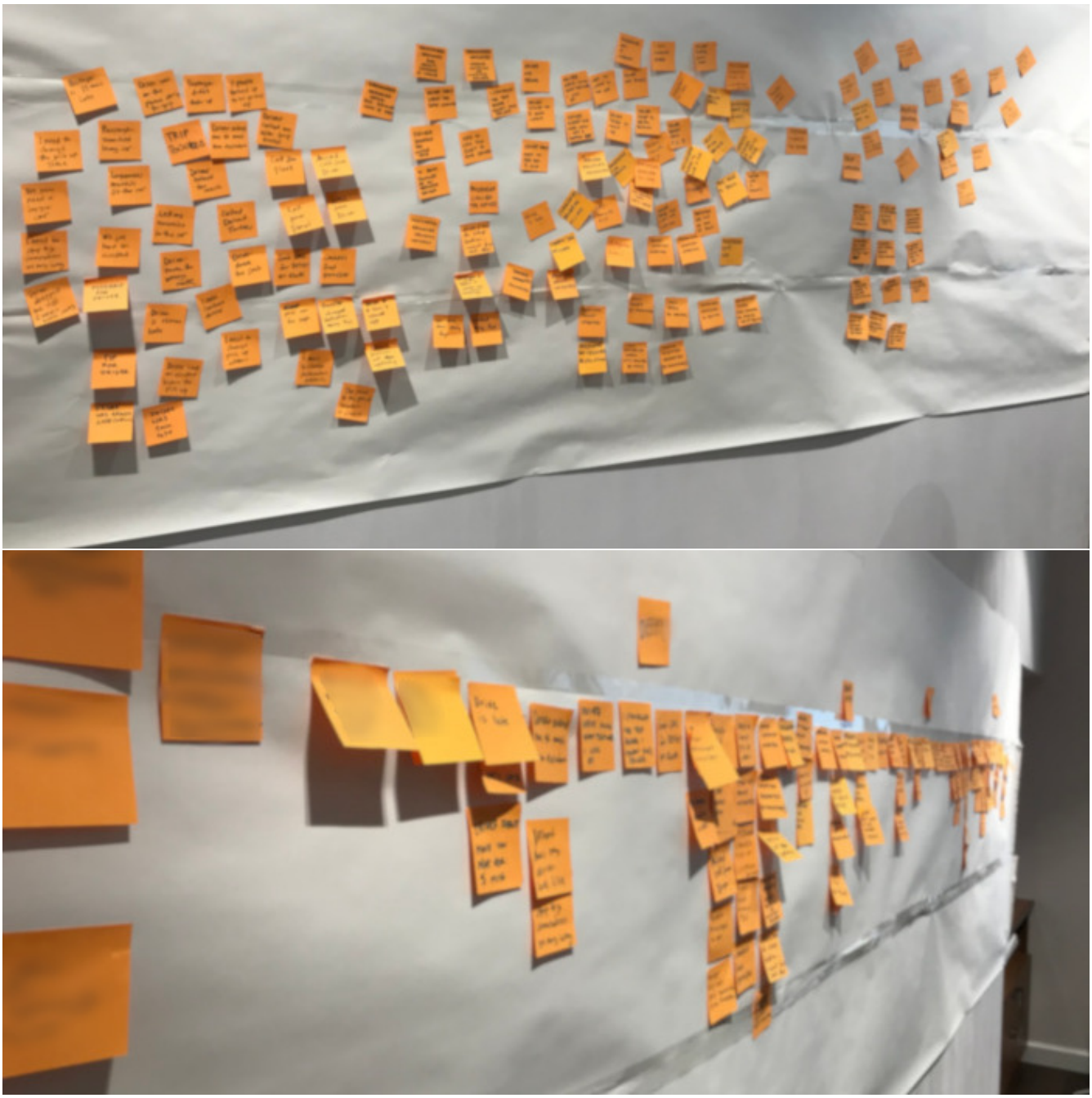
Figure 16.8: Event Storming artifact

Ubiquitous Language

Ubiquitous Language is a Strategic DDD pattern that covers building up a common language between developers, operations, stakeholders, and users. It is the **most underestimated Strategic DDD pattern.** Because who cares about the language, right?

It took me time to see **how many communication issues between developers and non-developers are because of using a different language.** And how painful it is. I'd encourage you to pay attention to that as well. Because of miscommunication, **developers aren't solving the right problem as nobody understands what's expected of them**.

Would you be surprised if I told you that Event Storming will help you develop Ubiquitous Language? Running the session together with stakeholders forces you to talk with them. **It's hard to build a solution together when you can't understand each other.** That's why it's critical to not miss your stakeholders at the workshop!



Does DDD solve all problems?

Even if DDD is great, it doesn't solve all problems that we have. It's important to manage your expectations. **Even if we use these techniques on a high level in my team, we still have doubts if the created design is good enough.** Sometimes we don't know how to approach the problem. Sometimes we go back from the code to the design phase. Sometimes we make bad decisions. **All of these are perfectly fine situations. There is no team in the world without these issues. It's better to assume that it will happen and not be**

surprised. But we know that without DDD, these issues would be much more significant.

While using DDD, you should pay special attention to avoid:

- Big Design Up Front,
- implementing code "for the future",
- trying to create anything perfect,

Instead, you should: - **Focus on delivering the MVP to the user in a short time** (by short, I mean rather 1 month than 6 months). - **If you need to implement something "for the future" because it will be harder to add it later** – that's a very bad sign. You should think about how to make it easy to add it later. - **Reconcile that even if you do your best, your design will not be perfect from the beginning** – it's much better to improve it over time.

For some teams, it may need a lot of work to go to this level. But I can promise you from my experience that it's possible. And the fun that you will have from delivering software again is worth it!

Note

If you feel that you should be a tech leader to propose such improvements – you're wrong! In the early days, when I was not a leader, I was already proposing many improvements in the teams in which I worked. You need to have good arguments with your teammates.
We always explain "why" the techniques work in our chapters. When you will use these arguments, they should be enough to convince them. If it will not work because your team is close-minded, it's a good reason to consider changing the job.

Software Renaissance

It's hard to go very deep with presented techniques within one chapter. My goal was rather to inspire you to question the status quo. **This is not how our industry should work. If you are not okay with the status quo, I hope that I inspired you to learn new techniques.** This is the best way to fight with the Software Dark Ages.

What about Strategic DDD patterns? This chapter is actually an introduction to the next part of our chapter series. **In the following months, we will deeply cover the most important Strategic DDD patterns** on a fully working example project.

Who are you in the Software Dark Ages?

Just an ordinary hick who blindly follows the rules imposed by others?

Inquisition, who will try to hate and stifle any unusual approach?

Alchemist, trying to create gold? Even if it's not scientifically backed?

Or maybe you are secretly reading forbidden books in your cellar? Maybe you are looking with us to finish the Software Dark Ages and to begin the Software Renaissance?

Chapter 17

Intermission

That's all in the book so far, but it's not the end yet!

We will keep the book up-to-date with every new article coming out on our blog. You can always download the most recent version using the same link from the newsletter.

Meanwhile, we'd love to hear from you. Please let us know if the chapters were easy to understand and if the ideas were helpful. You can reach us at **contact@threedotslabs.com**

Chapter 18

Event Storming *(Coming soon)*

This chapter is not completed yet. When the chapter will be ready, please use the same link that you used to download this copy.

Chapter 19

Bounded Context *(Coming soon)*

This chapter is not completed yet. When the chapter will be ready, please use the same link that you used to download this copy.

Chapter 20

Ubiquitous Language *(Coming soon)*

This chapter is not completed yet. When the chapter will be ready, please use the same link that you used to download this copy.

Chapter 21

Aggregate *(Coming soon)*

This chapter is not completed yet. When the chapter will be ready, please use the same link that you used to download this copy.

Chapter 22

Value Object *(Coming soon)*

This chapter is not completed yet. When the chapter will be ready, please use the same link that you used to download this copy.

Chapter 23

Dependency Injection *(Coming soon)*

This chapter is not completed yet. When the chapter will be ready, please use the same link that you used to download this