Name Construct & Structure

Organzing readable code

Khalil ***Stemmler***

Name, Construct & Structure

Organizing Readable Code

by Khalil Stemmler

This is version 1.0. It was published **June 16th, 2019**.

Newer versions can be downloaded from https://khalilstemmler.com/resources.

Found typos? Want to suggest an update? Reach out to me, I'm pretty receptive to feedback.

Check back @ khalilstemmler.com for revisions.

Table of Contents

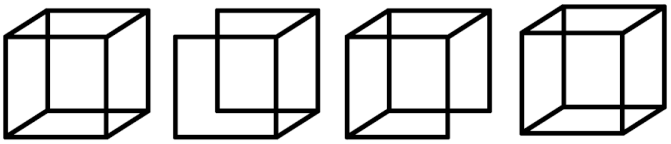
Nice to meet you!

What's up? I'm Khalil Stemmler from khalilstemmler.com. Thanks so much for downloading this free ebook. I sincerely hope it's useful to you.

My goal is to help developers learn best practices towards creating and maintaining large scale typescript & node.js applications. I frequently publish blog posts, guides, books, and courses for developers building enterprise software with JavaScript & TypeScript.

If you'd like to leave feedback on this book, suggest any other topics for me to cover, or simply just chat, check me on Twitter @stemmlerjs and see what I'm working on @ GitHub @stemmlerjs.

Introduction



Have you ever come back to an old codebase and felt like it was way too hard to understand what things are, what they do, and where they're located?

I came about realizing that I wanted to write a short ebook on organizing codebases for readability when I was tasked with adding new code to an old project I created for Univjobs a few years ago.

It took me **way longer** than I wanted it to have taken me to get things done. I found that I was spending a lot of time trying to figure out where things were and what they were responsible for.

Over the past year of writing code using Domain-Driven Design, I've paid a lot more attention to how I organize my code. Also over this time, I've experimented with a bunch of different approaches to code organization and noticed a few patterns for extremely readable code.

To summarize, it comes down to how we organize our files and folder with respect to the **names, constructs & structure.**

Naming files well, using well-understood technical constructs, and structuring our projects well are three ways to improve our ability to **change code quickly**.

Having **all three of these** when we're organizing codebases makes them highly readable.

The ultimate goal is to organize our projects to be readable & understandable so that we can rapidly make changes and add new features.

Organizing code more effectively

The fact of the matter is that we **spend more time reading code than we do writing code** (usually).

What does it mean when code is readable?

When code is readable, it means that we're very quickly able to determine:

- Name: what it is
- Construct: what it does
- Structure: where it's located & why it's there

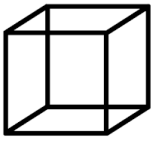
The faster we can discover **what it is**, **what it does** and **where it's located and why**, the more readable the code is (and the faster we can change or implement new things).

About this book

The approaches to organizing code in this book are what works for my team and I. You don't have to agree with all (or any) of it, but I hope that it at the very least prokes thought on how you can improve your own code organization to be more productive.

Name

What it is



Name

“What it is”

“There are only two hard things in Computer Science: cache invalidation and naming things”. -- Phil Karlton

Choosing names for **classes**, **files,** and **folders** can be incredibly hard sometimes. However, it’s kind of important to put some effort into it.

It’s important because the goal of naming things is to inform the reader as quickly as possible **what it is**.

How do we do that? How do we name things so that it quickly informs the reader **what it is**?

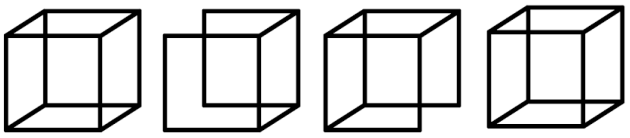
Tips for picking good names

Here are a few of the approaches that I use for most cases. Having these in my back pocket enables me to usually never spend more than a couple of seconds thinking about a name.

Approach #1: Name files by Domain + Construct

When naming **files,** I'll often choose names based on the **subdomain it belongs to** + the **construct type.**

You might be wondering, "what's a domain"?



definition

Domain

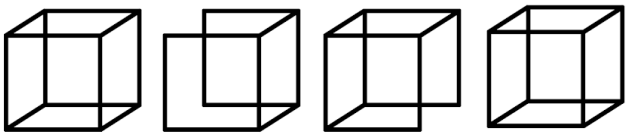
All applications are built up of several different *subdomains*.

If you're building a job board, you might have a subdomain for Users, Jobs, Notifications, Billing, etc.

These are the different components that all go together to make your app work.

Users	Jobs	Billing	Notifications
Authentication	Job	Customer	PushNotification
Role	Applicant	CreditCard	Email
Permission		Subscription	SMS

Secondly, we should understand what **constructs** are.



definition

Construct

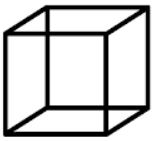
A construct is a general term I like to use to refer to the different tools in our software development toolbox that we use to build applications.

Tools like:

Controllers, Routes, Mappers, Repositories, Views, Models, etc

Some of these can be referred to as “patterns”, but loosely, I call everything a construct.

Since we understand what **subdomains** and **constructs** are now, we could use them to assemble names for our files by appending the **construct** type to the **subdomain** (or element from the subdomain - see the image on the next page).

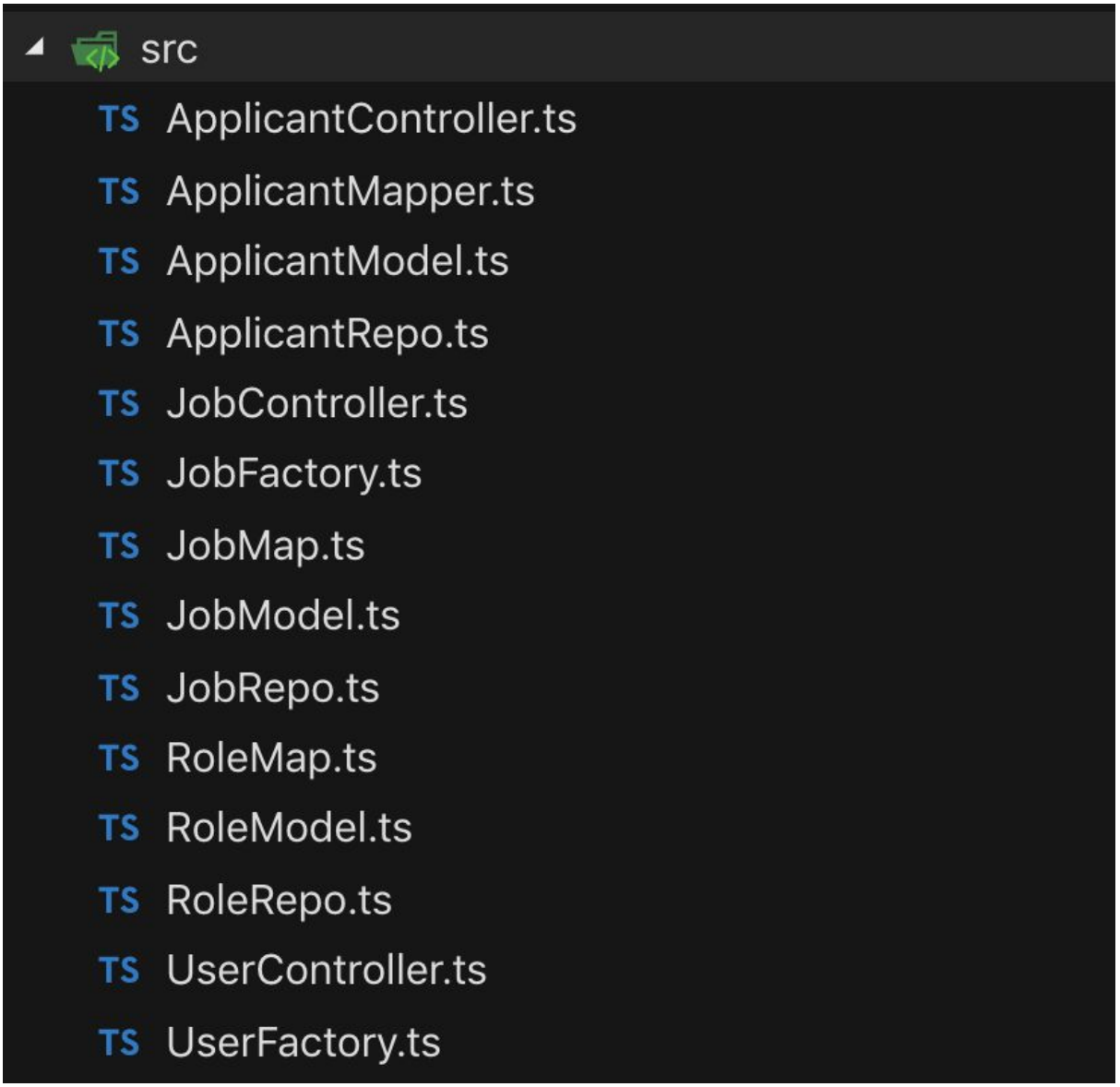


Choosing names

"Name by Domain + Construct"

Domain	Construct	=
User	Controller	UserController
	Mapper	UserMapper
	Repo	UserRepo
	Factory	UserFactory
	Model	UserModel
— Role	Mapper	RoleMapper
	Repo	RoleRepo
	Model	RoleModel
— ...	...	...
Job	Controller	JobController
	Mapper	JobMapper
	Repo	JobRepo
	Factory	JobFactory
	Model	JobModel
— Applicant	Controller	ApplicantController
	Mapper	ApplicantMapper
	Repo	ApplicantRepo
	Factory	ApplicantFactory
— ...	Model	ApplicantModel

Let’s see what some of that might look like in principle.



It’s a good start, but there’s a problem. **We’re not organizing our subdomains properly**.

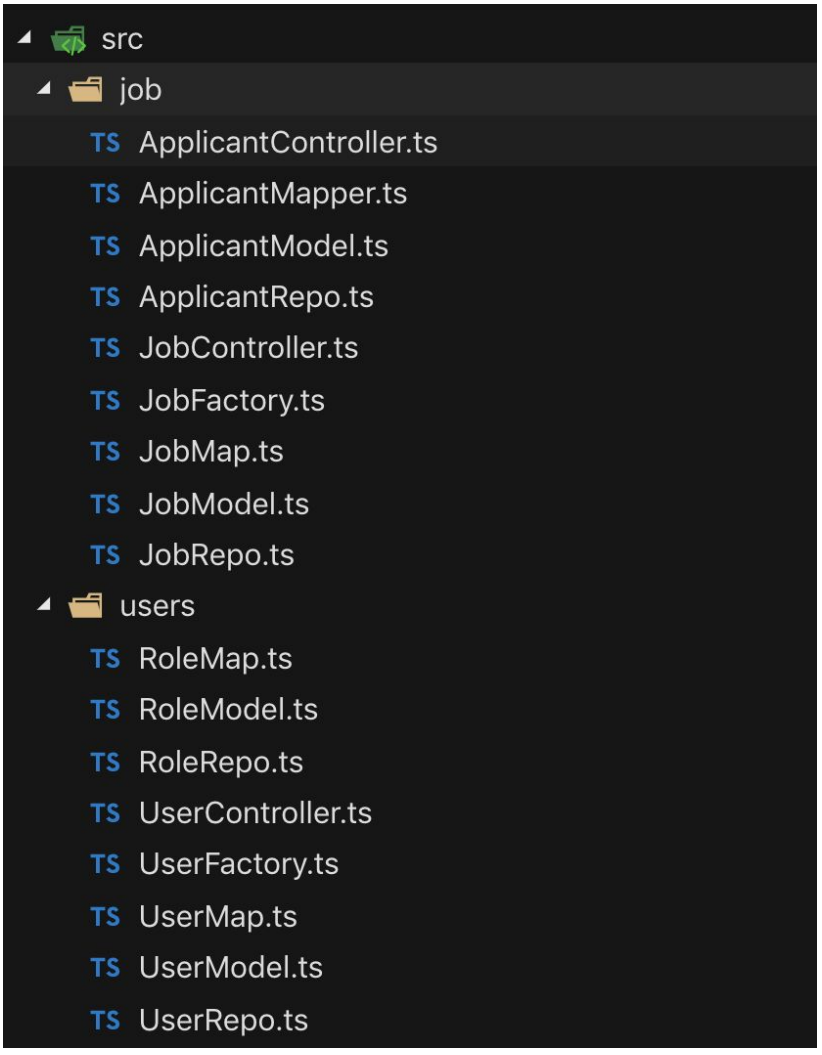
Things from the `Job` subdomain are mixed up with the `Users` subdomain. That’s definitely giving me clutter-y feels.

It’s time to think about naming folders.

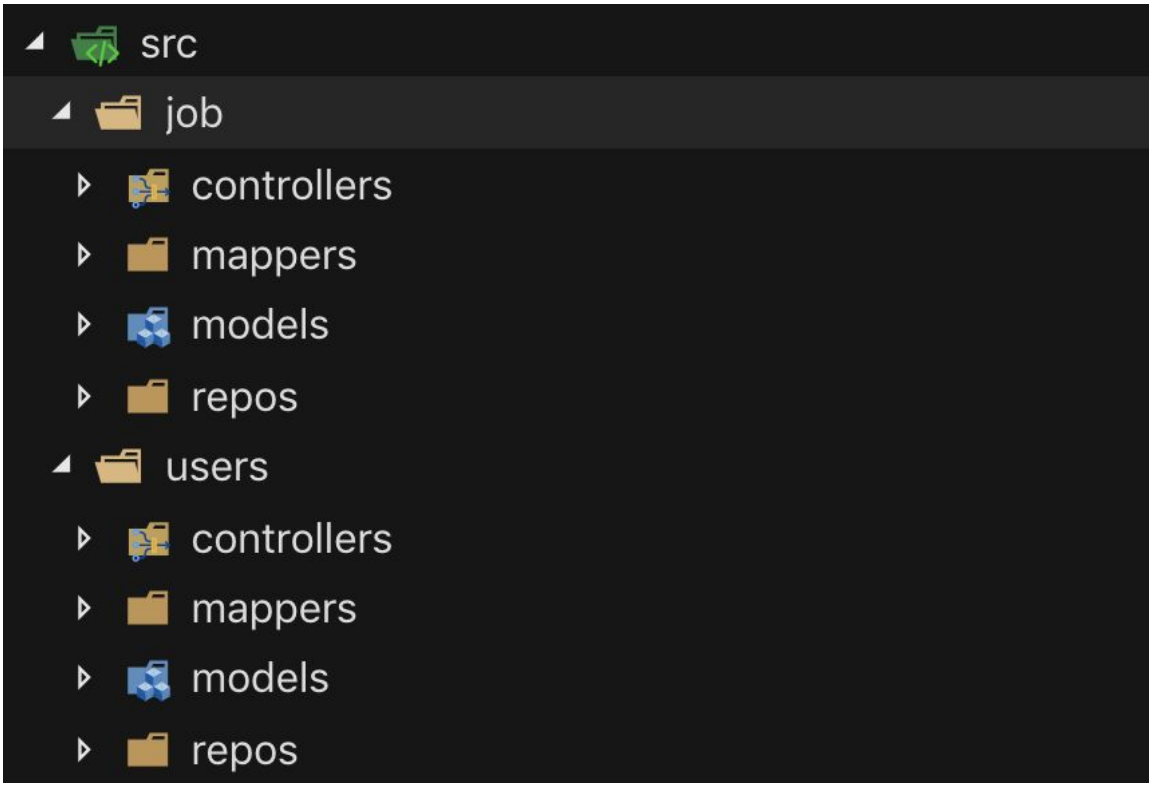
Approach #2: Name folders by Subdomain, and (optionally) Subdomain + Construct

Naming folders by the name of the **subdomain** is a great way to segment code so that you can focus on one particular part of your project at a time.

Here's a start.



That's better, but you might still feel a little bit discombobulated looking at this many files at one time. I like to take the next step of also creating sub-folders and naming them by the **construct type**.



I like this a lot. We talk about it more in **Part 3: Structure,** but this is called "Packaging by Component".

There's a construct for everything

If we can't decide on a good name for a file, it might be because *we don't know* what type of construct the class in our file actually is.

So beware of naming things "etc-manager" and "etc-implementor", because when we start to do that, files become harder to understand what they're **singularly responsible for** at first glance.

If you're finding that you're often not sure what type of construct to use, my suggestion is the following:

Learn Design Patterns

Design patterns tend to be pretty narrowly scoped solutions to common problems in software.

For example, when I started getting into **Domain-Driven Design** and realized that I needed to convert a `Job` model to different formats (to a Domain entity, to an ORM representation, to a DTO, etc), I went looking for a pattern. I discovered that the pattern was the *Data Mapper pattern*.

From then on, anytime I realize I need to do this particular thing, I name it an "`[Entity name from Domain]Mapper`".

So dig around a little bit! That's the best way to learn design patterns. By actually having problems that need to get solved, to looking for the correct tool for the job, to implementing it, you'll find that you'll find that you retain that information much deeper than if you simply went through the entire catalog of design patterns.

That being said, the best resources for to sift through are these books:

- Martin Fowler's book called Patterns of Enterprise Application Architecture
- GoF: Design Patterns Book
- PoSA: Pattern Oriented Software Architecture Book

Also, if you want to learn how to write cleaner code overall, check out "Clean Code" by Robert C. Martin (Uncle Bob). It's a great read.

Bad names

If you want to see an entire list of ways to **do evil and keep your job forever,** check out this repo: https://github.com/Droogans/unmaintainable-code

Look out for too many helper/utility/service classes

It's easy to call just about everything a "helper", "utility" or "service". Understanding if something should be a **helper** comes from having knowledge about the domain.

I've talked about this topic quite a bit here and here (anemic domain models).

Construct

What it does

Constructs are the tools (or patterns) in our software development toolbox that we plug together to build web applications (mappers, controllers, views, models, etc).

Each construct is responsible for one type of behavior

Mappers **map data** between domain objects, ORM models, dtos and the like.

Controllers **handle web requests**.

Domain Entities have zero infrastructure-layer dependencies and **describe the modeled attributes and capabilities of real-life entity**.

Repositories are responsible for **retrieving and persisting domain entities**.

In Angular, *RouteGuards* **guard routes** and allow for custom logic to be plugged in to **determine if a route can be routed to or not**.

Every construct should be singularly responsible for one thing. A *Mapper* should never also inherit the behaviours of a *Controller*, and vice-versa.

A construct instance should never be responsible for more than one subdomain element

Let's say we implemented a `JobMapper` like so:

```
export class JobMapper extends Mapper<Job> {
  public static toDomain (raw: any): Job {
    return Job.create({
      title: JobTitle.create(raw.user_name).getValue(),
      startDate: raw.start_date,
      description: JobDescription(raw.description).getValue(),
    }, new UniqueEntityID(raw.job_id)).getValue()
  }

  public static toPersistence (job: Job): any {
    return {
      job_id: job.id.toString(),
      title: job.title.value,
      description: job.description.value,
      start_date: job.startDate
    }
  }

  public static toDTO (job: Job): JobDTO {
    jobTitle: job.title.value,
    jobId: job.id.toString(),
    jobDescription: job.description.value,
    jobStartDate: job.startDate
  }
}
```

This `JobMapper` is pretty narrowly scoped. It's only responsible to mapping the `Job` domain entity. That's great.

Now, let's say that we needed to also return the `User` that the `Job` belonged to as a key called **ownedByUser** in the `JobDTO` like this:

```
interface JobDTO {
  jobTitle: string;
  jobId: string;
  jobDescription: string;
  jobStartDate: string;
  ownedByUser: UserDTO;
}
```

Where the `UserDTO` has the following properties:

```
interface UserDTO {
  email: string;
  firstName: string;
  lastName: string;
  profilePicture: string;
}
```

What should we do with our `toDTO()` method on the `JobMapper`?

Should we also define how we map `User`s to `UserDTO`s from within the `JobMapper`'s `toDTO()` method like this?

```
  public static toDTO (job: Job, user: User): JobDTO {
    jobTitle: job.title.value,
    jobId: job.id.toString(),
    jobDescription: job.description.value,
    jobStartDate: job.startDate,
    ownedByUser: {
      email: user.email.value,
      firstName: user.firstName.value,
      lastName: user.lastName.value,
      profilePicture: user.profilePicture.value
    }
  }
}
```

No. At this point, we should create another Mapper: `UserMapper`.

```
export class UserMapper extends Mapper<User> {
  // ...
  public static toDTO (user: User): UserDTO {
    email: user.email.value,
    firstName: user.firstName.value,
    lastName: user.lastName.value,
    profilePicture: user.profilePicture.value
  }
}
```

And then delegate that responsibility to the UserMapper in our JobDTO.

```
export class JobMapper extends Mapper<Job> {
  public static toDomain (raw: any): Job {
    return Job.create({
      title: JobTitle.create(raw.user_name).getValue(),
      startDate: raw.start_date,
      description: JobDescription(raw.description).getValue(),
    }, new UniqueEntityID(raw.job_id)).getValue()
  }

  public static toPersistence (job: Job): any {
    return {
      job_id: job.id.toString(),
      title: job.title.value,
      description: job.description.value,
      start_date: job.startDate
    }
  }

  public static toDTO (job: Job, user: User): JobDTO {
    jobTitle: job.title.value,
    jobId: job.id.toString(),
    jobDescription: job.description.value,
    jobStartDate: job.startDate,
    ownedByUser: UserMap.toDTO(user)
  }
}
```

Much better.
What we're doing here is performing quality control on our architectural boundaries.

Architectural Boundaries

If we have a construct named "`UserMap`" in an application that trades vinyl (we actually will soon 😃, check out this repo), we should be careful thinking about our **architectural boundaries** when we mention entities from other subdomains in our application, like "`vinyl`" or "`billing`", from the "`users`" subdomain.

Here's an example of a `UserRepo` interface from my Domain-Driven Design with TypeScript course.

```
interface IUserRepo {
  getVinylOwnedByUser(userId: string): Promise<Vinyl[]>;
}
```

Yes, this *is* a `UserRepo` in the `users` subdomain, but it seems like the observable responsibility is actually to retrieve `Vinyl`.

It would probably make more sense to create a `VinylRepo`.

These are the types of constant discussions that we need to have if we care about software design and architecture. We'll never get everything right the first time around.

Why should we care so much about architectural boundaries?

If we keep our code siloed in architectural boundaries, it leaves options open for us in the future to potentially do many different things if our project grows. Some of those things are to:

a) delegate ownership of parts of the project to separate teams
    i) If we keep boundaries at bay, this means we'll be able to break apart the application so that separate development teams could set up their own version control and testing pipelines.

b) microservice deployment
    i) If we keep boundaries at bay, we can not only separate our code logically, but physically too.
    ii) Code can be split into separate repos and communication between the deployments could take place over the network (HTTP, RabbitMQ, etc).

Either way, as codebases grow larger, and more developers join our company, we look for ways to improve productivity and at scale. Keeping an eye on boundaries can make transitions a whole lot less painful.

Structure

Where it belongs

Structure is most concerned with *where we place files* and *how we lay out our folders*. Structure is code organization and asks the question, “are things where they’re meant to be”?

When our first guesses towards finding a particular file are correct, it greatly improves productivity.

When we have to guess and flip through too many files and folders, our productivity tanks.

Project size

This is increasingly difficult as projects grow in size. When projects are small, it might be pragmatic to use a flat file structure and simple, yet generic names.

Take this file structure example of a script written to automate users into a MailChimp List.

How much can you tell about the project from this structure and the file names?

An assumption would be that the **mailchimp.js** does the bulk of the work. Maybe it's what actually connects to the MailChimp API. Not sure what **list.js** is responsible for, but since we have so few files, I wouldn't quarrel too much with this.

As soon as the project starts getting more complex, you can be sure I'll be looking for ways to be more declarative with these files and folder names.

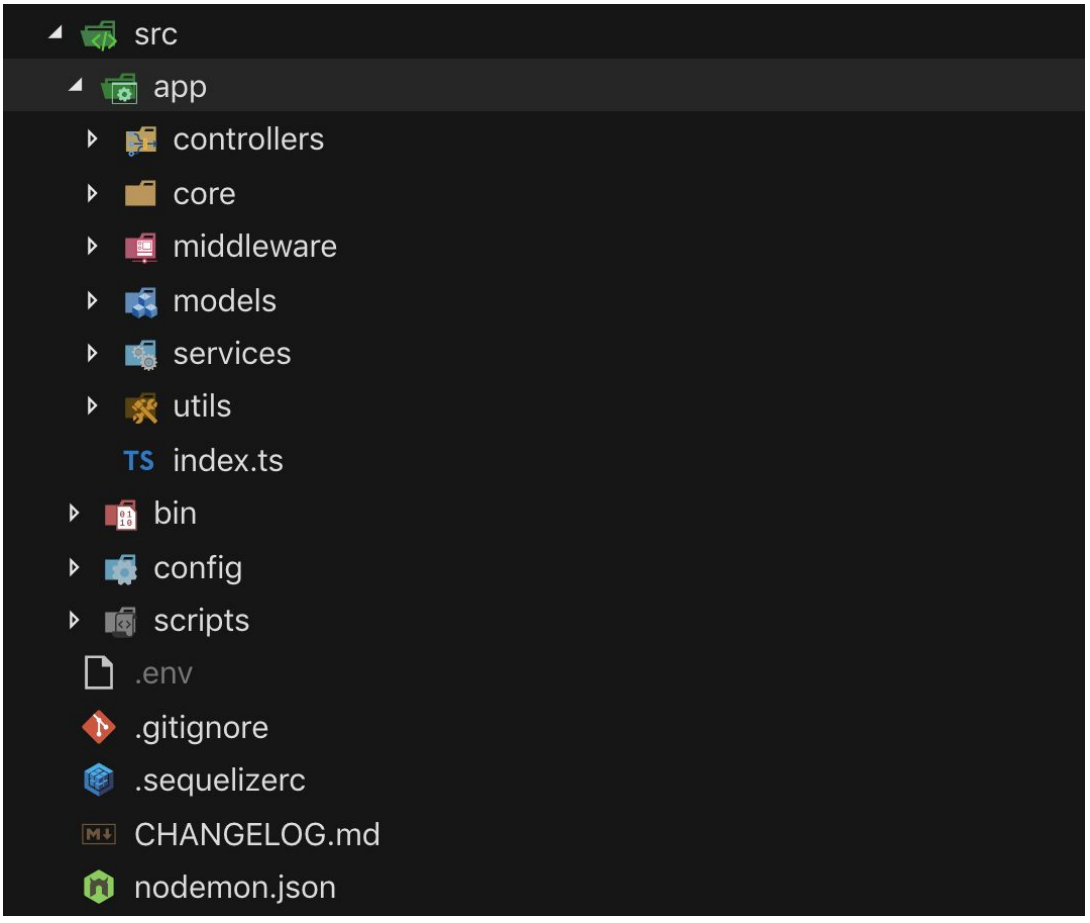
Package by Infrastructure

This is an important topic.

Most developers starting out, package their applications by infrastructure. This is the opposite of packaging your application by component.

Package by infrastructure is when we name our **top level folders** by the construct name.

To illustrate, here's the folder structure of the old project that I was working on that spawned my motivation to write this book.



If it's not obvious why this is **not useful**, let me make an analogy.

Imagine picking up a dictionary looking to find a particular word.

When you open the dictionary, it's not sorted by **alphabetical order**, but instead it's sorted by "the types of words".

So "long words", "short words", "scary words", "funny words"...

It would take you much longer and considerably more cognitive energy to find what you're looking for.

That's what we're doing here with **packaging by infrastructure**.

At a glance, this tells us nothing about the project

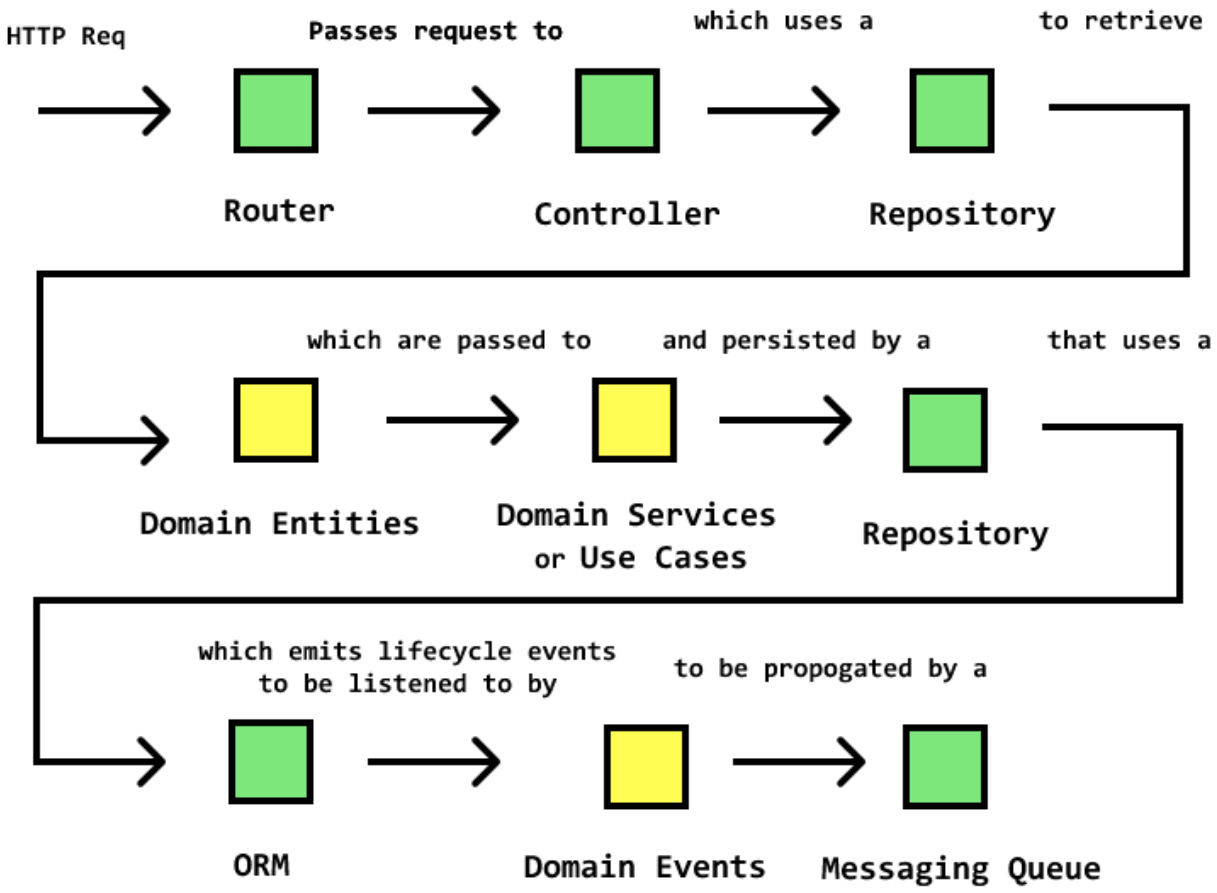
There's a term called **Screaming Architecture** from Uncle Bob's "Clean Architecture" book (I recommend it highly). It means that when we open a project's codebase, the code should *practically be screaming at us* what type of project we're looking at.

Within seconds, you should be able to determine: "hey, this is a healthcare application" or "this is a restaurant app".

Having code organized by ***construct*** at the top-level doesn't help us understand the project.

Features are developed vertically, not horizontally

When we develop features, it pretty much cuts across the entire stack. For example, let's say a POST request gets fired off by the client to **Create a User**. Here's how that might cut across several constructs.



Imagine that all of those constructs are in separate folders, away from each other.

That means, in order to develop one feature, you have to **flip between several layers of folders** to find the right construct to edit.

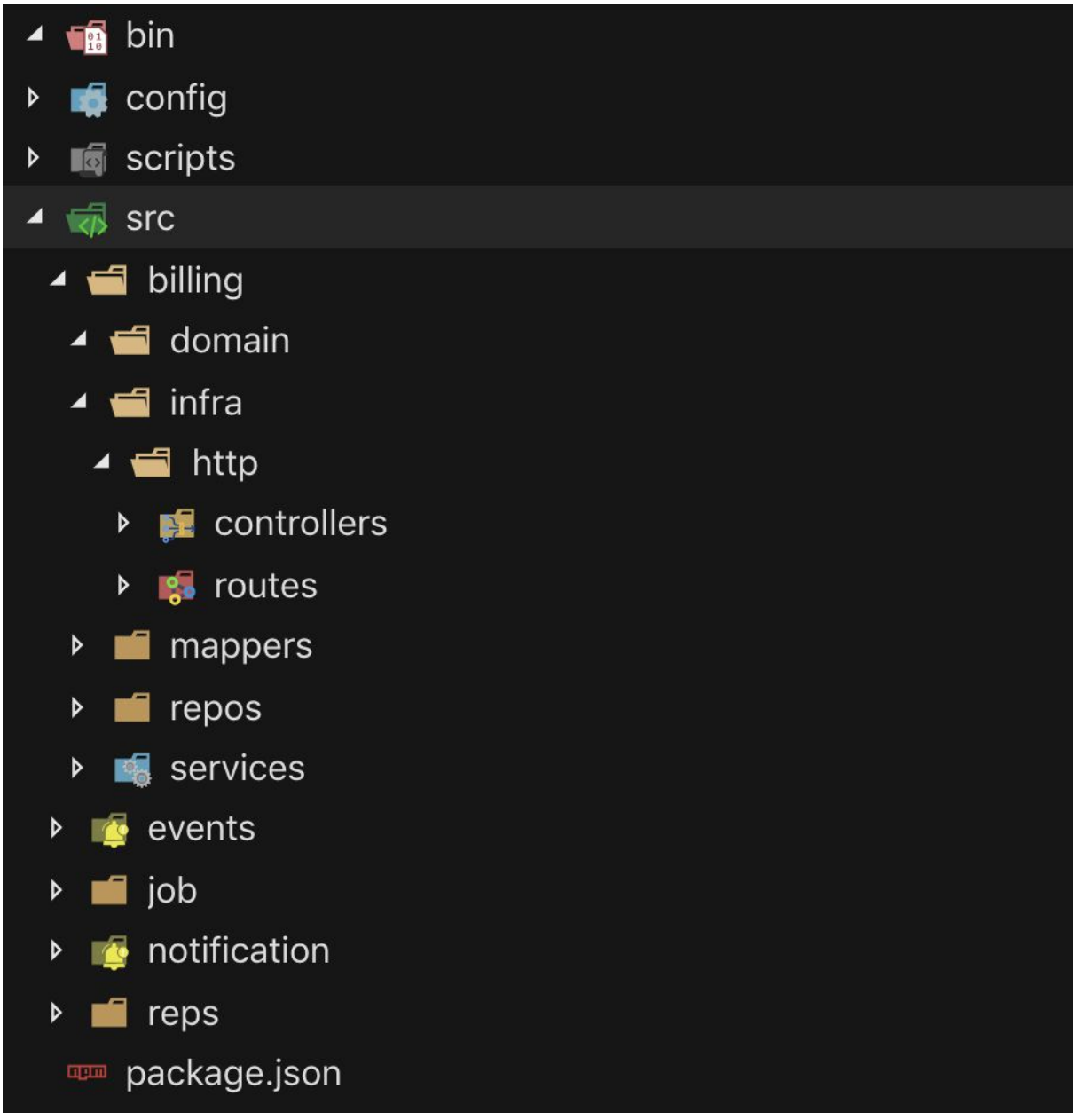
I believe we as humans can only remember at max 7 or so things at one time. So this approach isn't very human-friendly.

So let's go back to the human-friendly approach.

Package by component

In order words, packaging to be *brain-friendly*.

Here's that same project, but this time refactored to use package by component.



Doesn't that tell you so much more?

Package by component for everything

I used to think that this only really applied to backend code, but now also convinced this is the way to organize code **in general**.

Univjobs' React-Redux App Structure

The entirety of the Univjobs application is split into the **modular silos by "feature"** (or component, as we've been referring to it).

For example, the Create Job feature looks like this:

Everything related to a particular feature (container components, validation logic, styles, services, redux, models, etc) lives in a **single individual feature module**.

This has made it incredibly easy to identify where I need to go in the project to change code.

So here's my approach for good structure:

Tips for good structure

1. Start the first layer by identifying the features (or components or modules, whatever you want to call it).
2. The second layer in holds the constructs. Refer to the construct names here.

Nesting rules

Sometimes, we're in a feature that is split into several sub-features.

I'll repeat #1 until I'm at the lowest sub-feature and then implement #2.

Shared things

Be sure to identify when things are shared and place them in a shared module. For a good example of this, see the source code for khalilstemmler.com.

In Summary

Name, Construct and Structure all contribute to the readability of your project. As projects grow, it becomes harder to keep the same level of productivity up, so special attention to these three things becomes vital.

Names

What it is.

What to look out for

Look out for naming things too generally and assigning the code that lives inside of the files way too much responsibility.

For a guide on this, check out this article.

Constructs

What it does.

Structure

Where it belongs.

A real life project

I'm working on a Domain-Driven Design with TypeScript course where we build a **Vinyl Trading Enterprise application** called "White Label".

I'll be putting into practice what I've taught in this book, so if you want to see a real life example, check out the repo.