Extracted from:

Practical Microservices

Build Event-Driven Architectures with Event Sourcing and CQRS

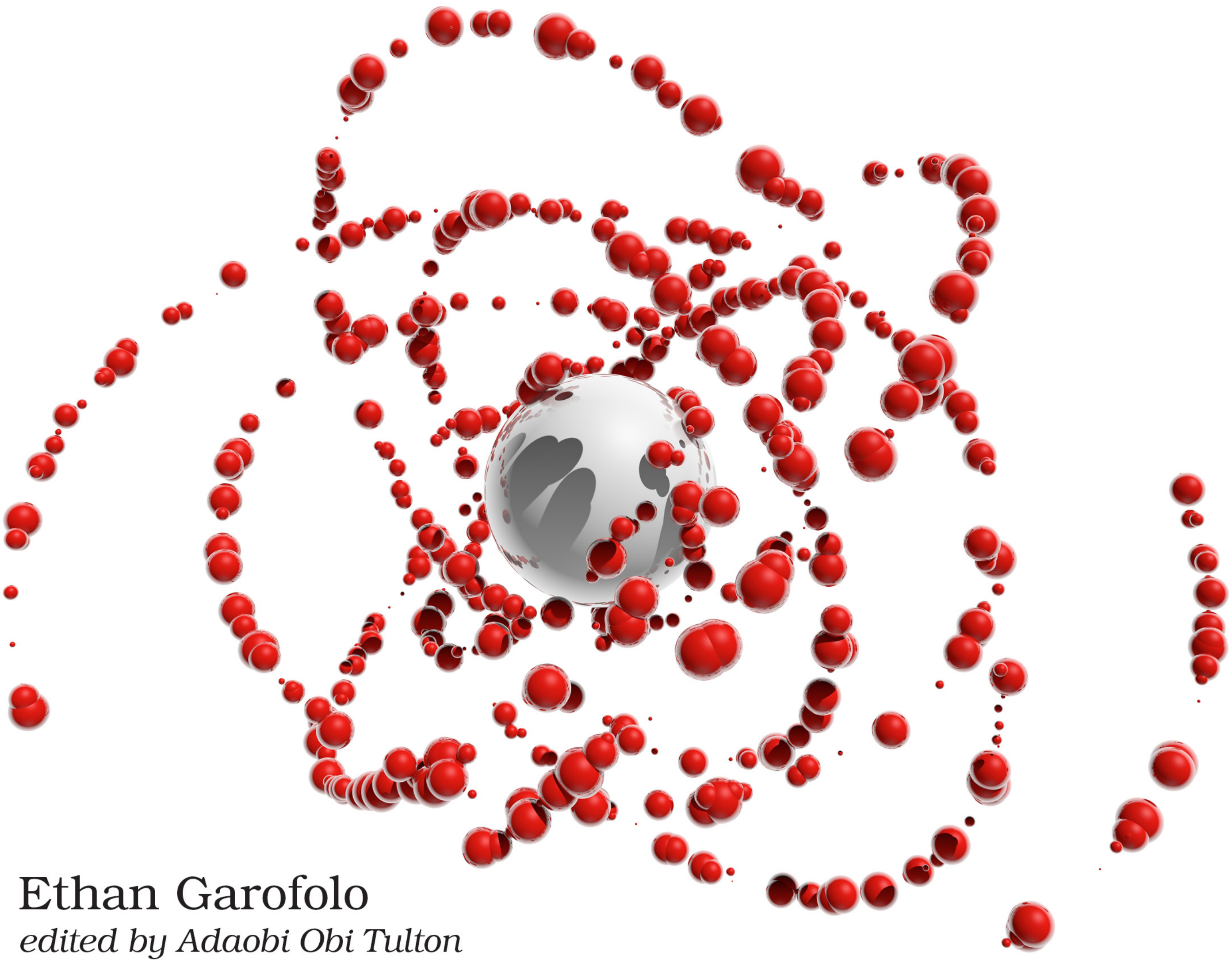
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Practical Microservices

Build Event-Driven Architectures with Event Sourcing and CQRS



Ethan Garofolo

edited by Adaobi Obi Tulton

Practical Microservices

Build Event-Driven Architectures
with Event Sourcing and CQRS

Ethan Garofolo

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Adaobi Obi Tulton
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

ISBN-13: 978-1-68050-645-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2020

For Julie, Sofia, Eva, and Angelo—

you are life and joy

Two roads diverged in a wood, and I—
I took the one less traveled by

➤ *Robert Frost*

CHAPTER 1

You Have a New Project

Congratulations! Today is your day. You've got a new project to get underway. A site full of videos; users to please—a system that will make future changes a breeze.

You're going to build Video Tutorials, the next-gen internet learning sensation. Content creators will publish videos, and the rest of the users will level up from watching those videos.

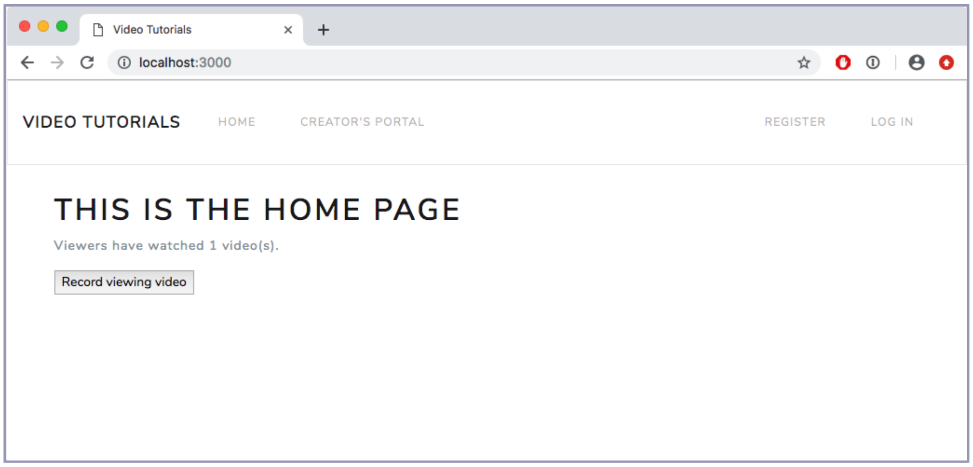
Some features of this system, like user registration and authentication, will seem familiar. Some features may be new, like transcoding videos. Through it all, rest assured that you have a top-notch business team to work with. They'll be busy discovering the benefits that our users want, and your job is to build a system that will support this platform for decades to come. Pieces of that system will necessarily evolve, so supporting that long-term change is going to be our main focus.

Kicking Off Video Tutorials

Let's start by getting the husk of this project off the ground. Our business team wants to support content creators by slicing and dicing video viewing metrics in all sorts of ways—some which they haven't even identified yet. The first metric they care about is a global count of video views.

Since we don't actually have any video content yet, we can just simulate video views with a button click. We're going to build enough server to serve a page like the screenshot on page 8. There's a button to click to simulate having viewed a video. To get this working we need:

- A basic project structure
- An HTTP route to GET this page
- An HTTP route to receive POSTs from the award-winning button



THIS IS THE HOME PAGE

Viewers have watched 1 video(s).

Let the fun begin.

Building the Bones

As mentioned in the book's introduction on page ?, our code samples are all in Node.js. It's a fine platform for evented architectures, and probably more importantly, if you've done web development, you can probably at least grok what is going on when you read JavaScript. In the *Application* layer—the layer of the system that users interact with—we use a library called Express[1] to route requests to the functions that handle them. This isn't a book specifically about programming in Node.js, but in case you're unfamiliar with Express, it's worth a little ink to introduce it.

Express is a "fast, unopinionated, minimalist web framework for Node.js," and we use it to map URLs to functions that handle them and to render HTML in response. We certainly could use single-page apps, but we want to keep the focus on microservices rather than JavaScript UI frameworks.

Our first job is to build a simple Express server, and an Express server is made up of some configuration, some middleware, and some routes:

first-pass/src/app/express/index.js
```
const express = require('express')
const { join } = require('path')

const mountMiddleware = require('./mount-middleware')
const mountRoutes = require('./mount-routes')

function createExpressApp ({ config, env }) {
```
(1)

1. https://expressjs.com/

```
2    const app = express()

     // Configure PUG
3    app.set('views', join(__dirname, '..'))
     app.set('view engine', 'pug')

4    mountMiddleware(app, env)
5    mountRoutes(app, config)

     return app
   }

   module.exports = createExpressApp
```

1. A typical Node.js file defines a top-level function and exports it. createExpressApp is that top-level function for this file, and we export it at the very bottom. Its return value is a configured Express application.

   To configure that Express application, this function receives some configuration. config has references to the rest of the pieces of our system, and env has all the environment variables our program was started with. If you're just dying to dive into these exciting files, they're at code/first-pass/src/config.js and code/first-pass/src/env.js, respectively, and we'll work through them on page 11.

2. This instantiates the Express application. Now we configure it.

4. This is where we mount middleware into the Express application. Middlewares are functions that get run on an incoming HTTP request and have the chance to do various setup and side effects before we get to the function that ultimately handles said request. As an example, we'll use a middleware to ensure users are authenticated on certain routes in Chapter 8, Authenticating Users, on page ?.

5. HTTP requests come into a server with a given URL, and you have to tell an Express server what to do for a given URL. That's what mounting the routes is for. You give Express a URL pattern and function to call to handle requests that go to that URL pattern. The file implementing this function is at code/first-pass/src/app/express/mount-routes.js. We don't have any routes quite yet, since we're just building the bones of the system right now, but we'll add our first route on page ? when we go to load the home page shown before on page 8.

That's the main structure of an Express application, and we won't have to touch this file for the rest of the project. We will add middlewares as we continue though, so that's where we turn next.

Mounting Middleware

Here is the start of our middleware:

first-pass/src/app/express/mount-middleware.js

```
Line 1 const express = require('express')
     - const { join } = require('path')
     -
     - const attachLocals = require('./attach-locals')
     5 const lastResortErrorHandler = require('./last-resort-error-handler')
     - const primeRequestContext = require('./prime-request-context')
     - function mountMiddleware (app, env) {
     -   app.use(lastResortErrorHandler)
     -   app.use(primeRequestContext)
    10   app.use(attachLocals)
     -   app.use(
     -     express.static(join(__dirname, '..', 'public'), { maxAge: 86400000 }))
     - }
     -
    15 module.exports = mountMiddleware
```

Middlewares in Express are functions that we run as part of the request/response cycle and that for the most part aren't meant to actually handle the request. We require three of our own, starting at line 4. Then we define the dependency-receiving function mountMiddleware that is also exported at the very end of the file. It receives app, the Express application, and env, our environment variables. We won't use the environment variables until Chapter 8, Authenticating Users, on page ?.

To actually mount a middleware, we call app.use, passing in the middleware function in question. We mount the first middleware, lastResortErrorHandler, at line 8. We follow this with primeRequestContext and attachLocals, ending the function with Express's built-in middleware. express.static serves static files. We use that for the CSS and JavaScript files we serve for the browser UI.

Let's write those custom middlewares we just included, starting with primeRequestContext:

first-pass/src/app/express/prime-request-context.js

```
const uuid = require('uuid/v4')

function primeRequestContext (req, res, next) {
  req.context = {
    traceId: uuid()
  }

  next()
}

module.exports = primeRequestContext
```

This middleware's job is to set up values that we'll want on every request. For now we use it to generate a traceId for each and every request. Even in Model-View-Controller (MVC) apps that model state as Create-Read-Update-Delete (CRUD) operations, having a traceId is a nice thing. We'll attach it to log statements so that we know which log statements belong together. We put these values onto req.context to namespace them all to a single property on req. We don't want to pollute the req object that Express hands us with a multitude of keys.

Next is attachLocals:

first-pass/src/app/express/attach-locals.js

```
function attachLocals (req, res, next) {
  res.locals.context = req.context
  next()
}

module.exports = attachLocals
```

We're rendering all of our UI on the server. This middleware makes the context we set up on the request available when rendering our UI.

Finally, there's lastResortErrorHandler:

first-pass/src/app/express/last-resort-error-handler.js

```
function lastResortErrorHandler (err, req, res, next) {
  const traceId = req.context ? req.context.traceId : 'none'
  console.error(traceId, err)

  res.status(500).send('error')
}

module.exports = lastResortErrorHandler
```

This is an error-handling middleware, identified by having four parameters in its signature. When nothing else manages to catch an error during a request, we at least catch it here and log it. We will be more sophisticated than this in our error handling—this is just our last resort.

With the middleware in place, we can dive into config and env.

Injecting Dependencies

We use a technique called dependency injection[2] in this system. Quickly stated, you can sum up dependency injection as "passing functions the things they need to do their job." This is in contrast to "having functions reach into the global namespace to get what they need to do their job." Dependency

2. https://www.youtube.com/watch?v=Z6vf6zC2DYQ

injection doesn't have anything to do with microservices, but it is how the code in this book is structured.

So, enter code/first-pass/src/config.js, and let's set up its shell:

first-pass/src/config.js
```
function createConfig ({ env }) {
  return {
    env,
  }
}

module.exports = createConfig
```

That sure doesn't do much yet. We'll flesh it out on page ? once we've finished with the home page application.

Next, let's write the file that ingests the runtime environment variables we'll use:

first-pass/src/env.js
```
module.exports = {
  appName: requireFromEnv('APP_NAME'),
  env: requireFromEnv('NODE_ENV'),
  port: parseInt(requireFromEnv('PORT'), 10),
  version: packageJson.version
}
```

This isn't the entire file, so check out the rest of it when you can. requireFromEnv is defined in the part that isn't printed here, and it checks if the given environment variable is present. If not, it exits the program and tells us why. When critical settings aren't present, we want to know about that ASAP.

What we've done here is locate every place where we read from the runtime environment in this one file. We don't have to wonder where we get environment settings from, and things that depend on environment settings don't realize that they do. This also isn't microservices-specific and is just the convention we use in this project.

We start with a few values. appName is a cosmetic name given to our running system. env tells if we're running in development, test, production, or whatever other environment we care to run. port is the port our HTTP server will listen on. version doesn't strictly originate from the environment, as we pull it out of the package.json file this project has (every Node.js project has a package.json).

Okay, with a barebones server built, let's start this puppy:

first-pass/src/index.js
```
1 const createExpressApp = require('./app/express')
  const createConfig = require('./config')
  const env = require('./env')
```

```
❷ const config = createConfig({ env })
  const app = createExpressApp({ config, env })

❸ function start () {
    app.listen(env.port, signalAppStart)
  }

  function signalAppStart () {
    console.log(`${env.appName} started`)
    console.table([['Port', env.port], ['Environment', env.env]])
  }

  module.exports = {
    app,
    config,
    start
  }
```

❶ Starting here we require the functions for building our Express app and config, as well as pulling in the environment.

❷ Then we instantiate config and the Express app.

❸ Finally, we define the start function that will be called to start the system. For now it calls the Express app's start function, passing it the port we want the HTTP server to listen on (env.port) and a callback function (signalApp-Start). This latter gets called when the HTTP server is listening, and it logs some of the settings from the environment. It's nice to have confirmation the server is running.

Lastly, we just need some code that calls this start function:

first-pass/src/bin/start-server.js
```
const { start } = require('../')

start()
```

It simply requires the file located at code/first-pass/src/index.js, the one we just wrote. It pulls out the start function and calls it.

Earlier we mentioned that every Node.js project has a package.json file at its root, and ours is no exception. A package.json file defines a key named "scripts", which are commands you can run using the npm command-line tool. If you have one called "start-dev-server", you can run it with npm run start-dev-server. Ours does define a script named "start-dev-server":

```
{
 "scripts": {
    "start-dev-server": "nodemon src/bin/start-server.js --color",
  }
}
```

nodemon[3] is a library that watches for changes to code files and then reruns the command passed to it. We tell it to run the file at src/bin/start-server.js, so every time that we make a change to the source code, it will restart the server for us.

Taking the Server for a Spin and Starting the Database

If you look in the package.json file, you'll also see that it defines dependencies and devDependencies. The former are other packages of Node code that we rely on in all situations, whereas the latter lists packages we use only in development. Both are installed when you run npm install, so do that now.

At this point, you can actually run this server. To do so, you'll need to have your PostgreSQL database set up. If you're comfortable doing that on your own, then by all means do so.

However, you can also use Docker,[4] which is what the rest of this book will assume. To use Docker, you'll need to install it, and we punt to the Docker docs[5] to explain how to do that for your platform. Then, in each code folder there is a docker-compose.yaml, which contains the necessary Docker configuration to run the databases for that folder. You can start the databases by running docker-compose rm -sf && docker-compose up. Go ahead and do that now.

If you do use your own PostgreSQL installation, you'll need to make the DATABASE_URL value match your database's setup in .env in the project's root directory.

Assuming that you have your database running, from the command line in the project's root folder, simply run npm run start-dev-server, and you should get output similar to the following:

```
$ npm run start-dev-server

> microservices-book@1.0.0 start first-pass
> nodemon src/bin/start-server.js --color

[nodemon] 1.17.5
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node src/bin/start-server.js --color`
Video Tutorials started
```

3. https://nodemon.io/
4. https://www.docker.com/
5. https://docs.docker.com/v17.09/engine/installation/

```
┌─────────┬───────────────┬───────────────┐
│ (index) │       0       │       1       │
├─────────┼───────────────┼───────────────┤
│    0    │    'Port'     │     3000      │
│    1    │ 'Environment' │ 'development' │
└─────────┴───────────────┴───────────────┘
```

Congratulations! You have an Express server running. Sure, it responds to exactly zero routes, but no one said you were *done*. Just that, you can start the system, and that's a milestone worth celebrating. Now we can get that incredible home page delivered to our users.