

Better APIs

*Quality, Stability,
Observability*

Everything you need to enhance the quality,
stability, and observability of applications.

Better APIs: Quality, Stability, Observability

Mikael Vesavuori

2024

Contents

Copyright	5
Found anything wrong?	6
Introduction	7
What will you learn?	7
Why should you care?	8
Prior art	11
Project resources	12
Workshop	13
Scenario	14
Workshop assignment	16
How to follow along	17
Commands	17
Prerequisites	17
My solution pitch	23
The application	24
Tech	25
Technical components	26
Architecture diagrams	27
Solution diagram	27
Deployment diagram	28
Microservices	29
API documentation	32
Fake user	32
Feature toggles	32
Weighing between a hardware- or a software-oriented approach	33
The benefits of hardware-segregated environments	33
The drawbacks of hardware-segregated environments	33
The benefits of a software-defined, dynamic environment	33
The drawbacks of a software-defined, dynamic environment	34
Yes, you can mix these patterns with a hardware-separated environment	34

Quality	35
Make your processes known	36
SOLID principles	37
Clean architecture	39
Baseline tooling and plugins	41
Continuous Integration and Continuous Deployment	42
Refactor continuously ("boy scout rule")	44
Trunk-Based Development	46
Test-Driven Development	48
Generate documentation	49
Unit testing (and more)	52
Creating coverage, fast	52
Test positive and negative states	53
Test automation in CI	54
Synthetic testing	56
Automated scans	57
Generate a software bill of materials	58
Open source license compliance	59
Release versioned software and produce release notes	61
Stability	62
Lifecycle management and roadmap	63
API schema	64
API schema validation	66
API client version using headers	68
Branch by abstraction	70
Beta functionality	73
Feature toggles ("feature flags")	75
Authorization and role-based access	77
Canary deployment	79
How we can do better	81
Observability	84
AWS baseline observability	85
Structured logger	87
Alerting	89
Service discoverability and metadata	91
Additional observability	93
Setting up Bunyan as a logger	94

Going even further with MikroLog, MikroTrace, and MikroMetric	94
References and resources	96
Books	96
Articles and online resources	96
Video	97
Thank you for reading this book	98

Copyright

© 2024 Mikael Vesavuori. All rights reserved.

No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including but not limited to physical copies, photocopying, recording, electronic books (eBooks), PDFs, digital downloads, or any other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law, such as under “fair use”.

Cover image adapts photographic material shared by Pawel Czerwinski on Unsplash. All relevant ownership of the original photograph remains with Pawel Czerwinski.

Published by Mikael Vesavuori on LeanPub and Gumroad.

Found anything wrong?

Writing technical books is challenging. While concepts and ideas may remain relevant for years, practical examples that rely on ever-changing technologies can become outdated quickly.

If you find anything incorrect, not working, or otherwise unusual, I'd greatly appreciate your feedback. I'll do my best to incorporate updates as soon as possible.

Find contact details on my website, mikaelvesavuori.se.

Introduction

“Better APIs: Quality, Stability, Observability” is a practical guide to how you can improve an API, combined with an example project (available on [GitHub](#)) and an assignment of sorts, if you’d want to try your hands at making a rock-solid, production-grade API.

Writing and maintaining APIs can be hard. While the cloud, serverless, and the microservices revolution made it easier and more convenient to set an API skeleton up, age-old issues like software quality — [SOLID](#), etc — and understanding the needs of the API consumers still persist.

Werner Vogels, legendary CTO of Amazon, [famously stated his API rules like this](#):

1. APIs are Forever
2. Never Break Backward Compatibility
3. Work Backwards [from](#) Customer Use Cases
4. Create APIs That are Self Describing and Have a Clear, Specific [↔](#)
[↔](#) Purpose
5. Create APIs with Explicit and Well-Documented Failure Modes
6. Avoid Leaking Implementation Details at All Costs

In practice, how can we begin moving towards those ideals?

What will you learn?

This book presents an [application](#) and a made-up (but “real-ish”) scenario that, taken together, practically demonstrate a range of techniques or methods, patterns, implementations, as well as tools, that all help enhance quality, stability, and observability of applications:

Quality means our applications are well-built, functional, safe and secure, maintainable, and are built to high standards.

Stability means that our application can withstand external pressure and internal change, without failing at predictably providing its key business values.

Observability means that we can understand, from the outputs of our application, what

it is doing and if it is behaving well. Or as [Charity Majors](#) writes:

Monitoring is for running and understanding other people's code (aka "your infrastructure").

Observability is for running and understanding *your* code – the code you write, change and ship every day; the code that solves your core business problems.

Of the three above concepts, *stability* is the most misunderstood one, and it will be the biggest and most pronounced component here. It will be impossible to reach what Vogels is pointing to, without addressing the need for stability.

Information

Caveat: No single example project or book can fully encompass all details involved in such a complex territory as this, but at least I will give it a try!

Why should you care?

You will be especially interested in this project if you have ever been involved in situations like the ones below, and want to have ideas for how to address them:

- You inherited something that is "impossible to work with or understand"
- Your team was unable to deliver new features because changes would mean breaking them for someone else
- You built something but don't know who your consumers are
- You document in Confluence or Sharepoint or something like that. If your document at all. You don't, since there is no allotted time for it. OK so maybe the API sometimes, but it's never up to date, really. To be honest, you tell people "ask me if you have questions, don't trust the docs".
- You looked up what [DORA metrics](#) are and laughed out the words "yeah not at my company, no sirree, never"

- You say that you are “autonomous” but someone always keeps freezing deployments to the shared staging environment so the only actual autonomous thing is localhost
- You say that you “implemented” continuous delivery, but it’s still too painful to integrate and release without a gatekeeper and crashing a hundred systems
- You have heard “we cannot have any confidence in our systems if we don’t do real, manual scheduled all-hands-on-deck testing with frozen versions of all external systems” so many times you actually have started to believe that ludicrous statement
- You wonder if things would be better with more infra and more environments, but start having nightmares when you do some back-of-the-napkin math on how many you’d need, not to mention the burden of supporting them logically

There are a million more of those, but you get the point; It’s a dark and strange place to be in.



Figure 1.1: Giovanni Battista Piranesi - Imaginary Prisons (1745-1761)

Giovanni Battista Piranesi - Imaginary Prisons (1745-1761)

While it's fun to throw around the old Piranesi drawing or [Happiness in Slavery](#) video as self-deprecating memes, I think it is Escher who brilliantly captures the “positive” side of this mess—That things become very, very strange when multiple realities and gravities are seen *together, at once*. Even if one “reality” is perfectly stable and sound, the work of architects is to support all the realities that need to work together.

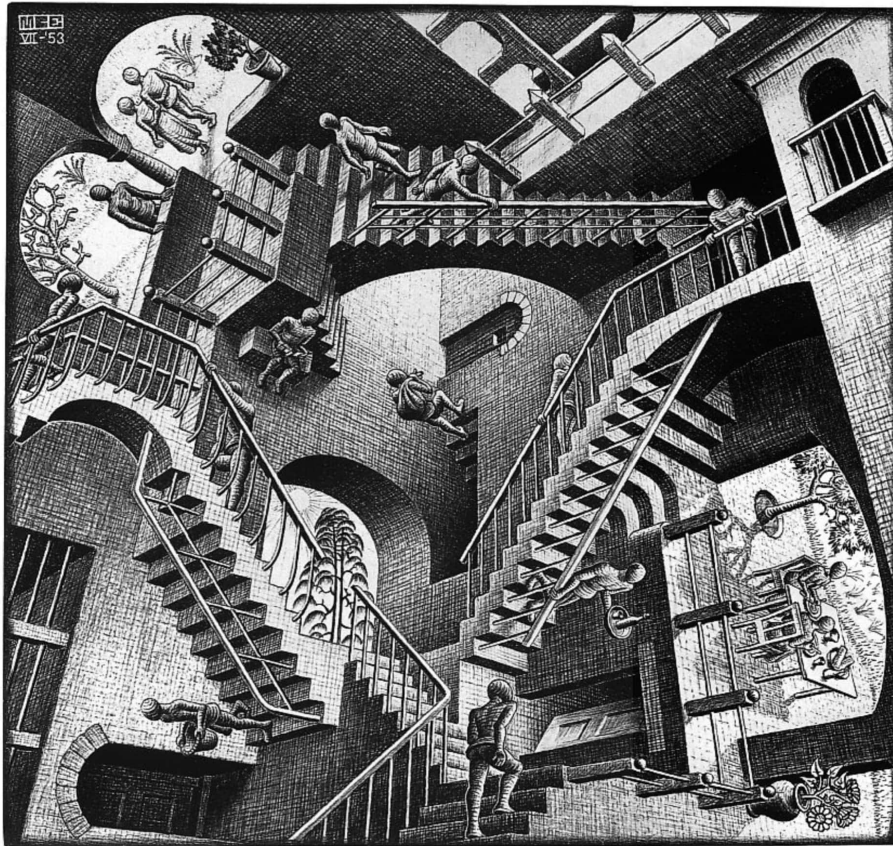


Figure 1.2: M.C. Escher - Relativity (1953)

M.C. Escher - Relativity (1953)

So: We need to get back some of that control so the totality does not look more like chaos than whatever it is that we are attempting to do.

At the end of the day, our work is about supporting the shared goals (business, organization, or personal goals, if it's a pet project) and letting us stay safe, sound, and happy working professionals while doing so.

Prior art

You might be interested in [microservices-testing-workshop](#) for a similar “workshop + demo” approach but focusing on testing an event-driven serverless application.

There’s also a previous piece of work you can look at: [multicloud-serverless-canary](#), which might pique your interest if you want to see more on the Azure and GCP side of CI and canaries.

Project resources

The application source code is available on [GitHub](#).

Feel free to check out the [generated static website on Cloudflare Pages](#) and the [API docs on Bump](#).

Workshop

This section outlines the prep work, resources, and high-level details on how I would approach creating an API that is qualitative, stable, and observable.

- Scenario
- Workshop assignment
- How to follow along
- My solution pitch
- The application

Scenario

You are supporting your friends who are making a new social online game by providing an API service that churns out fake users. For their initial MVP, they asked you to create a service that just returns hardcoded feedback; an object like so, { "name": "Someguy" ↵ ↵ "Someguyson" }. This was enough while they built their network code and first Non-Player Character engine.

The code in `src/FakeUserBasic/index.ts` is as simple as:

```
import {
  APIGatewayProxyResult,
} from "aws-lambda";

/**
 * @description The controller for our "fake user" service, in its ↵
 * ↵ basic or naive shape.
 */
export async function handler(
): Promise<APIGatewayProxyResult> {
  try {
    return {
      statusCode: 200,
      body: JSON.stringify({
        name: "Someguy Someguyson",
      }),
    };
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify(error),
    };
  }
}
```

Now, things are starting to become more involved:

- They are ready for a bit more detailed user data, meaning more fields.
- They are also considering pivoting the game to use cats, instead of humans.

- Of course, it's also ideal if the team can be sensibly isolated from any work on new features by individual contributors in the open-source community, who will need to have the work thoroughly documented and coded to a level that marks the high ambitions and makes it easy to contribute.
- Oh, and it would be good with a dedicated beta feature set, too.

For the near future, the **API must be able to handle all these use-cases**. It would also be perfect if the API can have **stable interfaces (for new clients and old), and not use several endpoints**, as the development overhead is already big for the two weekend game programmers.

How can you solve this situation?

Workshop assignment

If you want to, here's an example assignment for you to get going quickly.

Given the scenario, how would you approach addressing the requirements?

The workshop addresses typical skills needed of a technical lead or solution architect, prioritizing making considered architectural choices and being able to clearly communicate them.

This is possible to do two ways, either *with* or *without* code, as is needed based on the audience.

Suggested timeframe (no code): 90 minutes.

or

Suggested timeframe (with code): 3 hours. You can start from [src/FakeUserBasic/](#)↔↔ [index.ts](#).

The output should be a diagram or other visual artifact, and if also coding, a set of code that demonstrates a full or partial solution.

At the end of the period, present your solution proposal (~5-10 minutes), or if you are alone, go through what you've produced. Maybe even share on X, Reddit, LinkedIn...?

How to follow along

You can...

- **Go on a guided tour:** Grab a coffee, just read and follow along with links and references to the work.

or

- **Do this as a full-on workshop:** Clone the repo, run `npm install` and `npm start`, then read about the patterns and try it out in your own self-paced way.

Commands

The below commands are those I believe you will want to use. See `package.json` for more commands!

- `npm start`: Runs Serverless Framework in offline mode
- `npm test`: Tests code
- `npm run deploy`: Deploys code with Serverless Framework
- `npm run release`: Run [standard-version](#)
- `npm run build`: Package and build the code with Serverless Framework

Prerequisites

These only apply if you want to deploy code.

- Amazon Web Services (AWS) account with sufficient permissions so that you can deploy infrastructure. A naive but simple policy would be full rights for CloudWatch, Lambda, API Gateway, X-Ray, S3, and CodeDeploy.

- GitHub account to host your Git fork and for running CI with GitHub Action.
- Create a mock API payload on Mockachino.
- **Suggested:** For example a [Cloudflare](#) account for hosting your static documentation on [Cloudflare Pages](#).
- **Optional:** A [Bump](#) account to host your API description. You can remove the Bump section from `.github/workflows/main.yml` if you want to skip this.

All of the above services can be had for free!

1. Clone or fork the repo

Clone and fork the repo as you normally would.

Optional: Get a Bump account and token

*If you don't want to use Bump, go ahead and remove the part at the end of `.github/workflows/↔`
↔ `main.yml`.*

Go to the [Bump website](#) and create a free account and get your token (accessible under Automatic deployment, see the Your API key section).

Copy the token for later use.

2. Mockachino feature toggles

We will use [Mockachino](#) as a super-simple mock backend for our feature toggles. This way we can continuously change the values without having to redeploy a service or anything else.

It's really easy to set up. Go to the website and paste this payload into the HTTP ↔
↔ Response Body:

```
{  
  "error": {
```

```
    "enableBetaFeatures": false,
    "userGroup": "error"
  },
  "legacy": {
    "enableBetaFeatures": false,
    "userGroup": "legacy"
  },
  "beta": {
    "enableBetaFeatures": true,
    "userGroup": "beta"
  },
  "standard": {
    "enableBetaFeatures": false,
    "userGroup": "standard"
  },
  "dev": {
    "enableBetaFeatures": true,
    "userGroup": "dev"
  },
  "devNewFeature": {
    "enableBetaFeatures": true,
    "enableNewUserApi": true,
    "userGroup": "devNewFeature"
  },
  "qa": {
    "enableBetaFeatures": false,
    "userGroup": "qa"
  }
}
```

Change the path from the standard users to toggles. Click Create.

You will get a “space” in which you can administer and edit the mock API. You’ll see a link in the format https://www.mockachino.com/spaces/YOUR_RANDOM_ID.

Copy the endpoint, you’ll use it shortly.

3. First deployment to AWS using Serverless Framework

Install all dependencies with `npm install`, then set up [husky](#) pre-commits with `npm ↩ ↪ run prepare`.

For the next step you will need to be authenticated with AWS and have sufficient privileges to deploy the stack to AWS. Once you are authenticated, make the first deployment from your machine with `npm run deploy`.

We do this so that the dynamic endpoints are known to us; we have a logical dependency on these when it comes to our test automation.

Copy the endpoints to the functions.

4. Update references

Next, update the environment value in `serverless.yml` (around lines 35-36) to reflect your Mockachino endpoint:

```
environment:  
  TOGGLES_URL: https://www.mockachino.com/YOUR_RANDOM_ID/toggles
```

Next, also update the following files to reflect your Mockachino endpoint:

- `jest.env.js` (line 2)
- `tests/mocks/handlers.ts` (lines 11-12)

Continue by updating the following files to reflect your FakeUser endpoint on AWS:

- `api/schema.yml` (line 8)
- `tests/integration/index.ts` (lines 6-7)
- `tests/load/k6.js` (line 6)

If you have chosen to use Bump:

- Add your document name in the CI script `.github/workflows/main.yml` on line 113 (`doc: YOUR_DOC_NAME`)
- Update the reference to the Bump docs in `PROJECT.md` on line 43.

Optional: Continuous Integration (CI) on GitHub

If you connect this repository to GitHub you will be able to use GitHub Actions to run a sample CI script with all the tests, deployments, and stuff. The CI script acts as a template for how you can tie together all the build-time aspects in a simple way. It should be easily portable to whatever CI platform you might otherwise be running.

You'll need a few [secrets](#) set beforehand if you are going to use it:

- `AWS_ACCESS_KEY_ID`: Your AWS access key ID for a deployment user
- `AWS_SECRET_ACCESS_KEY`: Your AWS secret access key for a deployment user
- `FAKE_USER_ENDPOINT`: Your AWS endpoint for the FakeUser service, in the format `https://RANDOM.execute-api.REGION.amazonaws.com/shared/fakeUser` (known after the first deployment)
- `MOCKACHINO_ENDPOINT`: Your Mockachino endpoint for feature toggles, in the format `https://www.mockachino.com/RANDOM/toggles`
- `BUMP_TOKEN`: Your token for [Bump](#) which will hold your API docs (just skip if you don't want to use it; also remove it from the CI script in that case)

Optional: Deploy documentation to the web

If you have this repo in GitHub you can also very easily connect it through Cloudflare Pages to deploy the documentation as a website generated by TypeDoc.

You need to set the build command to `npm run build:hosting`, then the build output directory to `typedoc-docs`.

See the [Cloudflare Pages documentation](#) for more information.

You can certainly use something like [Netlify](#) if that's more up your alley.

5. Deploy the complete project

You can now deploy the project manually or through CI, now that all of the configurations are done.

Great work!

My solution pitch

We're going to cut down on environment sprawl by using a **single, dynamic environment** instead of multiple hardware-segregated environments (like dev, staging, prod setups) to deliver our stable, qualitative, and observable application.

The only other environment we'll use is a non-user-facing **CI environment for non-production testing**.

We'll use **feature toggles** and **canary releases** to do this *safely* in our single production environment, now being able to separate a (business-oriented) release from a mere technical deployment.

If these sound like **testing-in-production** strategies then—yep, they are!

To ensure functionality, we'll use **contract testing** and unit/integration/smoke/load testing to verify functionality across the time continuum: During development, before a deployment, after a deployment, and (optionally) under the application's lifetime, using **synthetic monitoring**.

The application will exhibit its dynamic nature by:

- Being able to run different sets of functionality based on if you are a “trusted” user or deliver a default feature set if you are not trusted.
- We'll be able to switch to a new implementation of an external dependency and verify that it works without breaking the original functionality.
- We can roll out the application over a period of time, and stop delivering it (and rollback) if we encounter errors.
- We top it off by using built-in observability in AWS to monitor our application and become alerted if something severe happens.

Later in this guide, you can read more about the implementation patterns and how they are organized into three areas (quality, stability, observability).

The application

The demo application provides an API that returns a “fake user”:

- The first version (`current`) of the API returns a hardcoded name.
- The second version (`beta`) includes a name and some other fields retrieved from an external API, plus a profile image (of a cat) from another external API.
- Finally, a new feature is developed on top of the second version, using a third external API. This feature is hidden under a feature toggle named `enableNewUserApi`.

More flows and details can be seen in the diagrams below.

The main software components are:

- **Microservice:** The `FakeUserBasic` service, if you want to do the workshop part and have some boilerplate code ready
- **Microservice:** The `FakeUser` service
- **Microservice:** The `FeatureToggles` service

Based on the user toggles, the service calls out to the following external APIs:

- **The Cat API** @ <https://thecatapi.com>
- **JSONPlaceholder** @ <https://jsonplaceholder.typicode.com/users>
- **RandomUser** @ <https://randomuser.me/api/>

The feature toggles are fetched, as has been stated previously, from Mockachino, where you will have to create an endpoint with the toggles payload.

Tech

This section presents the tech stack of this project and other purely technical concerns.

- **Technical components**
- **Architecture diagrams**
- **API documentation**
- **Weighing between a hardware- or a software-oriented approach**

Technical components

Services used are:

- [Amazon Web Services](#)
- Optional: [GitHub](#) and [GitHub Actions](#)
- Optional: [Bump](#)
- Optional: [Cloudflare Pages](#)

The infrastructure components are:

- [AWS API Gateway](#), to route incoming requests, handle request validation and authorize the user
- [AWS Lambda](#), for running our serverless microservices
- [AWS S3](#), for storing the functions

This project uses these primary libraries:

- [Serverless Framework](#) to handle packaging and deploying the microservices
- [TypeScript](#) as the language
- [Jest](#) to run unit tests
- [MSW](#) to mock external APIs
- [TripleCheck CLI](#) to run contract tests
- [TypeDoc](#) to generate documentation from the source code
- [Arkit](#) to generate architecture diagrams from the source code

Architecture diagrams

Diagrams that try to make sense of the various views of the solution.

Solution diagram

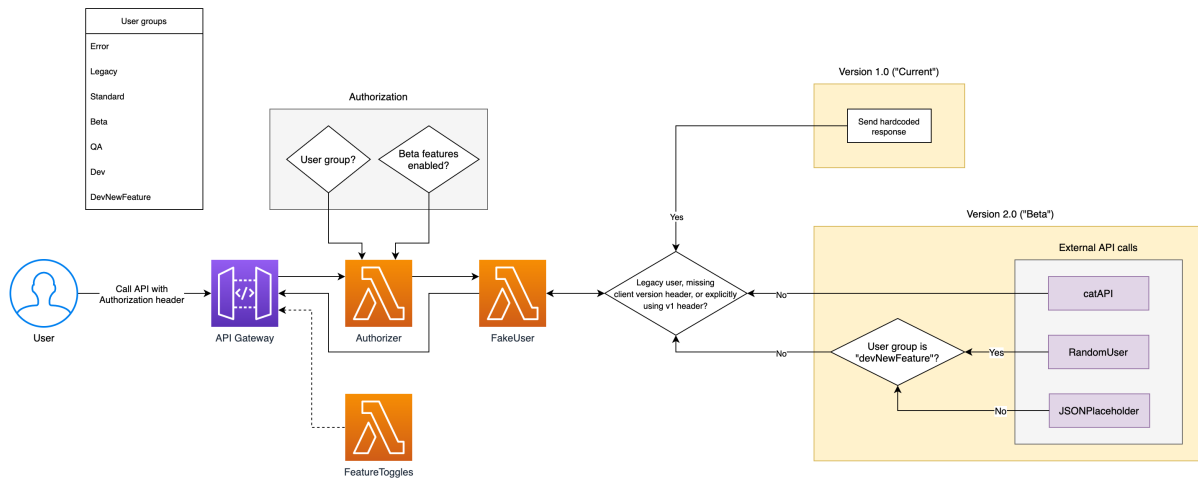


Figure 1.3: Architecture diagram

Deployment diagram

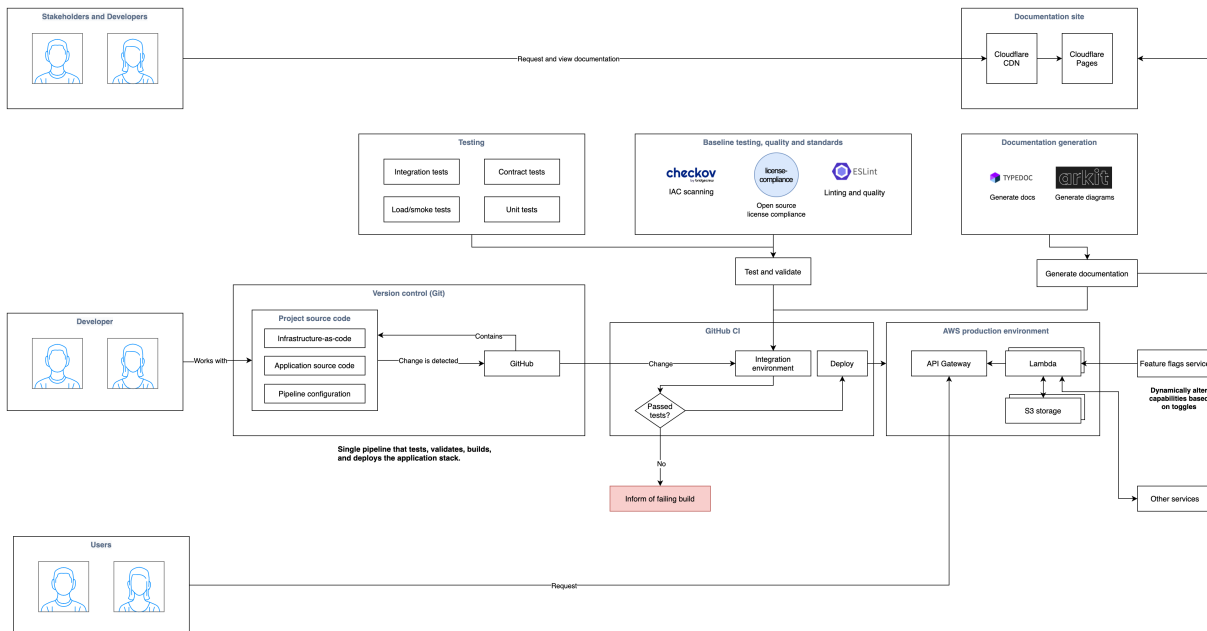


Figure 1.4: Deployment diagram

Microservices

FakeUser

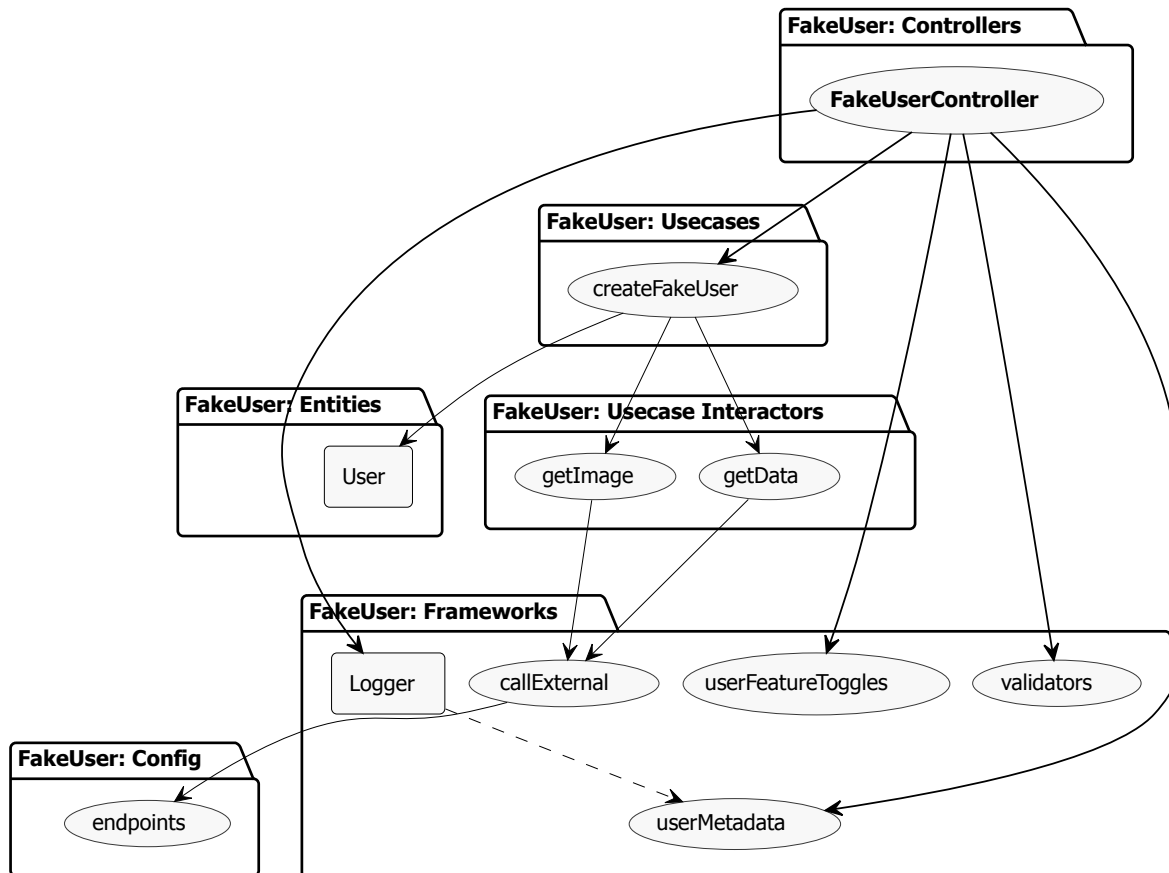


Figure 1.5: FakeUser

FakeUserBasic

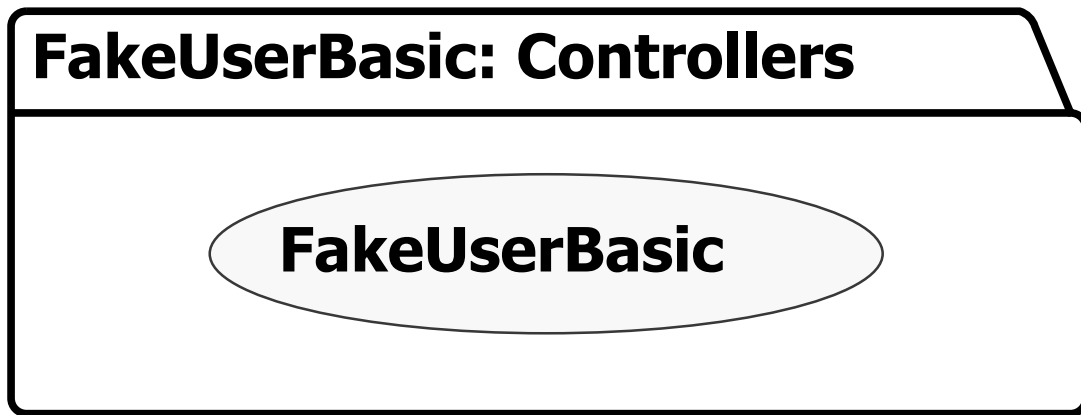


Figure 1.6: FakeUser

FeatureToggles

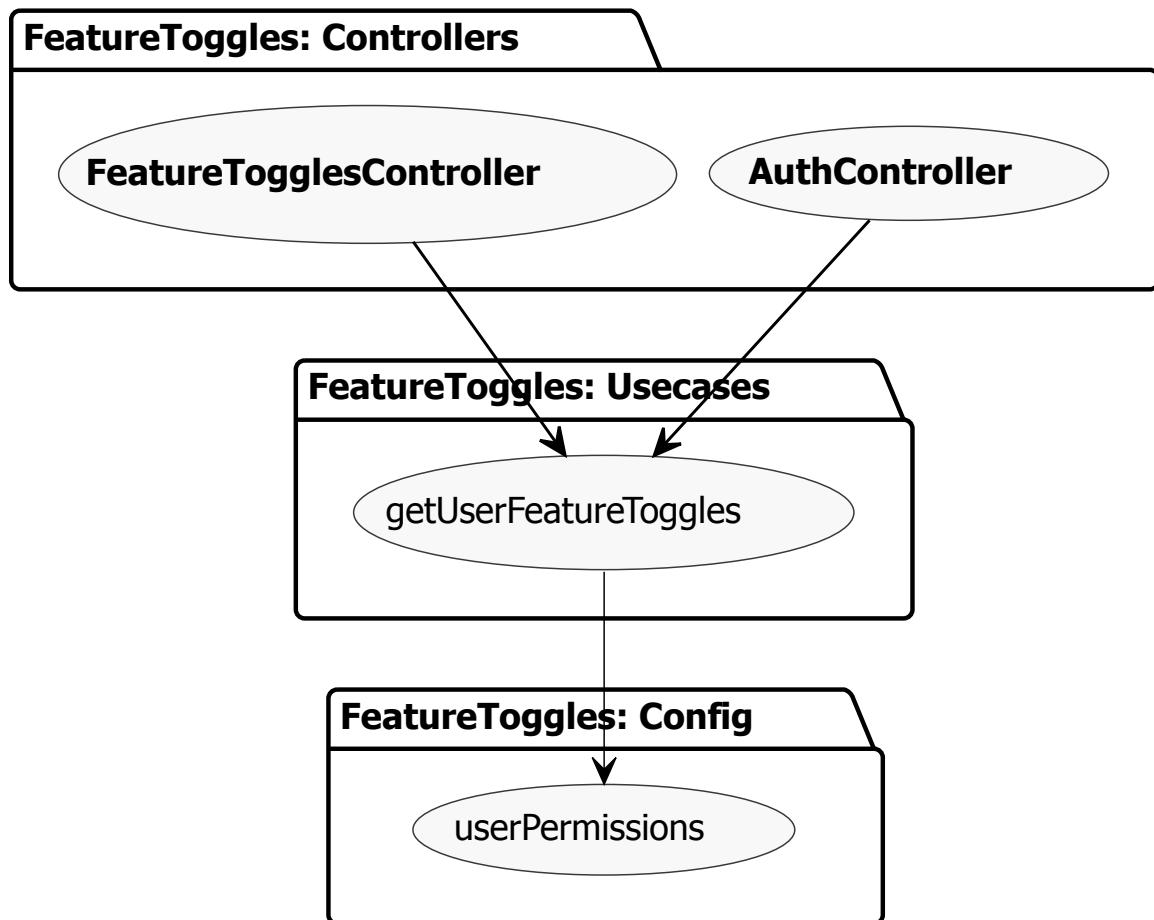


Figure 1.7: FeatureToggles

API documentation

These are API docs that apply to the full “finished” state.

Fake user

Get a fake user.

Required headers:

Authorization: erroruser@company.com | legacyuser@company.com | standarduser@company.com ↩

↩ .com | betauser@company.com | qauser@company.com | devuser@company.com | devnewfeatureuser@company.com

↩ .com

X-Client-Version: 1 | 2

```
GET {{API_URL_BASE}}/shared/fakeUser
```

Feature toggles

Send the user name (email) of a user to get their feature toggles. An unknown (or missing etc.) user will return default toggles.

```
POST {{API_URL_BASE}}/shared/featureToggles

{
  "userName": "devnewfeatureuser@company.com"
}
```

Weighing between a hardware- or a software-oriented approach

Let's go through pros and cons for a traditional vs "modern" approach to environments.

The benefits of hardware-segregated environments

- Very easy to understand conceptually
- Should result in very good separation of environments

The drawbacks of hardware-segregated environments

- You start expecting environmental parity between any other systems
- You start expecting that all systems are in similar, co-deployed stages
- The implicit reasoning starts becoming that you "should" or "can" only have a low degree of variability in configuration
- As a consequence, such "pre-baked" configuration tests may start becoming large-scale blocking, manual tests
- There may be significant cost overhead with a higher count of static environments
- There is most likely a significant complexity overhead with a higher count of static environments

The benefits of a software-defined, dynamic environment

- Less architectural complexity
- Cheaper with fewer environments
- No need to keep environments in sync

- Realistically promotes modern practices: “testing in production”, trunk-based development etc.
- Scales to more intricate and realistic scenarios (such as testing system X in mode A with customer type B in configuration D etc.)

The drawbacks of a software-defined, dynamic environment

- Can add solution complexity
- Will become harder to work with in the local scope (i.e. the actual code), and more so if there are many branches
- Requires some degree of cleanliness and pruning (governance even) to control, so things don’t grow out of hand

Yes, you can mix these patterns with a hardware-separated environment

You can certainly use the patterns seen in this project in a more “traditional” hardware-separated environment. However, the benefits become more pronounced as you also shed some of the overhead and weight of classical environments.

Quality

This section represents overall quality-enhancing activities that can be done to ensure your product is built with a solid engineering foundation.

- Make your processes known
- SOLID principles
- Clean architecture
- Baseline tooling and plugins
- Continuous Integration and Continuous Deployment
- Refactor continuously (“boy scout rule”)
- Trunk-Based Development
- Test-Driven Development
- Generate documentation
- Unit testing (and more)
- Test automation in CI
- Synthetic testing
- Automated scans
- Generate a software bill of materials
- Open source license compliance
- Release versioned software and produce release notes

Make your processes known

This is literally step zero.

Any non-trivial software development context requires some form of common ground to keep all of the work together, and for the team to correctly claim they have done their preparation work.

Example 1: See [PROJECT.md](#) which is the start page for the generated website. It uses a basic template structure to aid in a team providing the right information. The file provides (among other things) information on the project governance model with key roles in the project and their responsibilities, outlines the requirements process, and points out where and how to follow work on the tasks/requirements. This file is just an opinionated starter to ensure that base-level questions around the project/product are answered.

Example 2: [LICENSE.md](#) describes the license of this project. In an open-source project, a license should exist to make it clear what type of use is acceptable. For corporate projects, this is also valid so that you can clearly mark the software as proprietary (or open-source if you want to go that way!). [GitHub has a lot more on this](#), if you are interested.

Example 3: Reading [CONTRIBUTING.md](#) makes it clear how the team, and external parties, can contribute to the code base. This document also states some basic principles around code standards, code reviews, bug reporting, and similar “soft tech” issues. Don’t underestimate the need to be clear on expectations, whether these are highly detailed nitpicky bits or general guidance: Your project will do better with a good contribution document. See [Mozilla’s guide](#) for more information.

Example 4: For our [CODE_OF_CONDUCT.md](#), we are using the [Contributor Covenant](#), which is one standard to communicate baseline values and norms. Of course, enforcement and such are still in your hands. While it’s made for open source circles, something like this makes sense to have in corporate contexts as well.

Example 5: [SECURITY.md](#) describes how we address vulnerabilities and any reports of them, and how we are supposed to track them. Our tooling should use a risk-based remediation strategy based on [CVSS](#), which basically boils down to setting ourselves up to have an informed view on the actual risks we have, have these valued, prioritize the most dangerous risks, and adapt our mitigation based on the severity of them.

SOLID principles

The very first (technical) thing is to respect that good code, despite programming language, is good code even over time. Follow wise conventions like [SOLID](#) to guide your daily work.

Information

See for example this [Stack Overflow article](#) or [Khalil Stemmler's write-up](#) for a concise introduction.

The principles are:

1. The S or Single-responsibility principle: “There should never be more than one reason for a class to change.” (In other words, every class should have only one responsibility)
2. The O or Open-closed principle: “Software entities ... should be open for extension, but closed for modification.”
3. The L or Liskov substitution principle: “Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.”
4. The I or Interface segregation principle: “Many client-specific interfaces are better than one general-purpose interface.”
5. The D or Dependency inversion principle: “Depend upon abstractions, not concretions.”

Example: In our project, one example of the dependency inversion principle comes into play when calling the `betaVersion()` function in `src/FakeUser/controllers/FakeUserController` [↔](#) `.ts`, as we send in the toggles for it (and `createFakeUser()`) to use. Because this happens already in the controller or boot-strapping phase of the application, we ensure that the dynamic values (i.e. the toggles) are always present throughout the full call chain without the need to import them deeper inside the app.

```
/**
 * @description Handle the new (v2) beta version.
```

```
*/  
async function betaVersion(  
  toggles: Record<string, unknown>  
) : Promise<APIGatewayProxyResult> {  
  const response = await createFakeUser(toggles); // Run use case  
  return {  
    statusCode: 200,  
    body: JSON.stringify(response),  
  };  
}
```

The single responsibility principle should hopefully also be evident throughout most of the code.

Clean architecture

The second thing, and very much cross-functional in regards to quality and stability, is having an understandable, concise and powerful software architecture.

It is not enough for code to work.

— From [Clean Code](#) by Robert Martin.

Example 1: You can see a clear taxonomy for how the overall project and the microservices are organized by simply browsing the folder structure and seeing how code is linked together. Let's look at the FakeUser service:

```
FakeUser └──  
config └──  
contracts └──  
controllers └──  
entities └──  
frameworks └──  
usecases
```

What we're seeing is a somewhat simplified *Clean Architecture* structure. One of several tenets of Robert Martin's [Clean Architecture concept](#) is to produce [acyclic code](#). You can see that there are no cyclical relations in the Arkit diagrams (contained the Technology chapter). This, among other touches, means that our code is easy to understand, easy to test and debug and that it is easy to make stable, almost entirely by just logically organizing the code!

Like Martin, I'm also taking cues from [Domain Driven Design](#), where we use the “entity” concept to refer to “rich domain models”, as opposed to anemic domain models:

In Martin's seminal [Patterns of Enterprise Application Architecture] book (2002), a Domain Model is defined as **an object model of the domain that incorporates both behavior and data**'. This clearly sets it apart from Entity Objects, which are object representations of only the data stored in a database (relational or not), while the behavior is located in separate classes instead. Note that this divide is not really a layering, it's just procedures working with pure data structures.

— Source: [Codecentric blog](#)

Information

Read more about the Anemic Domain Model anti-pattern on Martin Fowler's site.

Example 2: While the examples in the code part of this project may be a bit contrived (bear in mind that they need to balance simplicity with meaningful examples), you can see how the User entity (at [src/FakeUser/entities/User.ts](#)) has not just data, but also business logic and internal validation on all the different operations it can perform. There is no need to leak such internal detail anywhere else; the only thing we add to that scenario is that we externalize the validation logic so that those functions can be independently tested (for obvious reasons private class methods are not as easily testable).

Information

To keep it short here, I'll just refer to [Robert Martin's original post on clean architecture](#) and [Clean architecture for the rest of us](#) for more details. Also, see [Khalil Stemmler's article on how CA and DDD intersect](#) if that floats your boat.

Clean architecture isn't a revolutionary concept: it's just the best and most logical realization (I feel) so far for questions around code organization that have lingered for decades.

Baseline tooling and plugins

The collective impact of several (on their own) small tools can make the difference between misery and joy very tangible.

This project uses two very common—but hugely effective—tools, namely [ESLint](#) and [Prettier](#). These two ensure that you have a baseline, pluggable way of ensuring similar standards (and automation of them) across a team. I’d not write many lines without those tools around.

Really, one of the very first things you want to make sure of is that the code looks and reads the same, regardless of who wrote it. Using these tools, now you can.

Success

Don’t forget to enable “fix on save”! Also, consider the VS Code plugins for [ESLint](#) and [Prettier](#) if you are using VS Code.

When it comes to more IDE-centric plugins in the security department, I highly recommend the [Snyk Vulnerability Scanner](#) (the successor to [vulncost](#)) for [Visual Studio Code](#). Other nice ones include:

- [Abracadabra, refactor this!](#)
- [Checkov](#)
- [DevSkim](#)
- [OpenAPI \(Swagger\) Editor](#)

Example: You’ll see that this project has configuration files such as `.eslintrc` and `.prettierrc` laying around.

Continuous Integration and Continuous Deployment

Welcome to the beating heart of Agile and DevOps.

We're at the very basics of classic [agile](#) (and [extreme programming](#)) when we state that Continuous Integration (or [CI](#)) and Continuous Delivery or Deployment ([CD](#)) are things to strive for.

However, even 20 years later it still seems like these notions are pretty far away in many organizations. To be frank, then, the problem is as [Atlassian writes](#), “agile isn’t agile without continuous delivery”.

Information

For more on the relation between delivery and deployment (and more), [read this article by Atlassian](#).

Thankfully, there's beginning to be some standardization around what high performance in technology delivery actually means, driven primarily by the [DORA research program](#) and related books like [Accelerate](#), by Nicole Forsgren, Jez Humble, and Gene Kim.

Success

You can take a simple test, called the [DORA DevOps Quick Check](#) to check your DORA performance metrics.

When it comes to our practices, we can make meaningful steps towards this by opting for [smaller releases](#) and having a [limit on work-in-progress](#). This keeps the focus tighter, features smaller, and the release cadence flowing smoother and faster. Everyone wins.

Example: You can see that we have CI running in GitHub with our script located at `.github/workflows/main.yml`. Every commit runs the full pipeline and deploys new code to AWS. The other “soft practices”, are somewhat less valid for a single author and can't be easily demonstrated.

CI/CD is nowadays 20% a technology problem (good and easy tooling is available in abundance) and 80% a people and process problem. Using DORA metrics and CI/CD, you can start setting a high mark on the objective delivery improvements that these tools and practices help enable when compared to traditional waterfall teams. This is of course contingent on you having a situation that allows such flexibility of operating (potentially) differently than your organization already does things.

Refactor continuously (“boy scout rule”)

First, what is *refactoring* exactly? [In the words of Martin Fowler](#), who wrote [one of the definitive books on the subject](#),

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. Each transformation (called a “refactoring”) does little, but a sequence of these transformations can produce a significant restructuring. Since each refactoring is small, it’s less likely to go wrong. The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring.

It takes perseverance, good communication, and business folks that truly understand software to get dedicated time for improving the solutions we work on. Most people will unfortunately not work in teams with dedicated refactoring time. What to do, then?

Rather than think of “change” as a drastic, singular, large-scale, and long-term event, it’s better to see change as a stream of small, manageable events that we can shape. Robert Martin wrote about the notion that every change in a codebase should also include some form of improvement in his classic book, [Clean Code](#): The *boy scout rule* which in the engineering context means “always leave the code better than you found it”.

Information

Read a [short summary here](#) and a [longer article on continuous refactoring here](#).

Moreover, the “boy scout rule” is definitely colored by other (at the time) contemporary management ideas like [kaizen](#) that work well in agile/lean contexts.

But what about our early work, when we are just starting on a new feature or product? In that case, I personally love, and truly resonate with, [Martin’s notion to start with “degenerate tests”](#) (also in the book, [Clean Craftsmanship](#)):

We begin with the degenerate tests. We return an empty list if the input list is empty, or if the number of requested elements is zero. [...] Note that I am following the rule of gradually increasing complexity. Rather than worrying about the whole problem of random selection, I'm first focusing on tests that describe the periphery of the problem.

We call this: “Don’t go for the Gold”. Gradually increase the complexity of your tests by staying away from the center of the algorithm for as long as possible. Deal with the degenerate, trivial, and simple administrative tasks first.

Practically it means that we start coding and testing from a perspective where the boundary functionality *works*, but has no complexity or elegance. Then, bit by bit, we add necessary complexity (dealing with harder problems more realistically) while subtracting unintended complexity (using SOLID principles, etc.) so we end up with something that worked early on, yet *evolved* into a coherent and good piece of work.

It would be wrong to assume that all code necessarily has this evolutionary spiral into something better—in fact, I think it’s correct to say that most code, unfortunately, grows worse. Again, we must remember that all code is a liability. It is the efficient pruning and nurturing, methodically done, that allows code to actually grow better with time. Refactoring is the key to this, and we should do it as early and often as possible, ideally within minutes of our first working code.

Example: Hard to point to something “post-fact”, but every single bit has been continuously enhanced and refactored (sometimes removed) since starting this project. This very book has, as well, gone from a README file to becoming a full Gitbook project!

Information

Go ahead and check out [Refactoring.guru](https://refactoring.guru) for lots of ways to approach making practical code improvements. Also, see the reference list at the end for even more materials.

Trunk-Based Development

There are two main patterns for developer teams to work together using version control. One is to use feature branches, where either a developer or a group of developers create a branch usually from trunk (also known as main or mainline), and then work in isolation on that branch until the feature they are building is complete. When the team considers the feature ready to go, they merge the feature branch back to the trunk.

The second pattern is known as trunk-based development, where each developer divides their own work into small batches and merges that work into the trunk at least once (and potentially several times) a day. The key difference between these approaches is scope. Feature branches typically involve multiple developers and take days or even weeks of work. In contrast, branches in trunk-based development typically last no more than a few hours, with many developers merging their individual changes into the trunk frequently.

— From Google’s article [DevOps tech: Trunk-based development](#)

For me, Trunk Based Development (TBD) captures an essential truth: That most branching models are just too complicated, and have too many adverse effects when it comes to merging code and staying agile. Even if certain models (IMHO) balance those aspects fairly well, like [GitHub Flow](#) (not to be mixed up with [GitFlow](#)!), nothing is simpler and more core to the agile values than TBD: Just push to main (or master if you aren’t up with the times) and trust your tooling to handle it.

Danger

Note that even Vincent Driessen, the original conceiver of GitFlow, nowadays [actively discourages the use of GitFlow in modern circumstances](#).

[Trunk Based Development](#) is worth reading about, if nothing else because it seems misunderstood by some.

Example 1: I can’t easily point to some evidence here, but the full history of this project (both the code and the book/guide) has been handled this way.

The pros and cons of TBD are of course only truly, fully visible when seen together with other practices and tools you have. There needs to be certain maturity before doing this while remaining safe. This project should represent perfectly valid conditions for which TBD can be used instead of less agile strategies.

Success

You can usually set up various branching strategies and restrictions in your CI tool, to effectively require TBD as part of the workflow.

Example 2: To truly support TBD, I've added [husky](#) to run pre-commit hooks for various activities, such as testing. This way we get to know even before the code is “sent off” if it's in shape to reach the CI stage.

Read more about Git workflow anti-patterns at [Jason Swett's blog](#).

Test-Driven Development

The red, green, refactor approach helps developers compartmentalize their focus into three phases:

- Red — think about *what* you want to develop.
- Green — think about *how* to make your tests pass.
- Refactor — think about *how* to improve your existing implementation.

— From [Codecademy](#)

Information

Read a more [complete introduction to TDD in TypeScript here](#).

[Test-driven development](#) is a practice that a lot of people swear by. I’m a 50/50 person myself—sometimes it makes sense to me, sometimes I just write the tests after I feel I’m out of the weeds when it comes to the first implementation. No need to be a fundamentalist; be a pragmatist! The important thing is that *there are tests*, not so much how and when they came. I’d still note that for this advice, I will assume that you have some kind of rigid standards—it’s just too easy to skip the tests!

However, in case you want to be a good-spirited TDD crusader then I’ve made it easy to do so.

Example: Just run `npm run test:unit:watch` and Jest will watch your tests and source code. You can also modify what Jest “watches” in the interactive dialog.

Generate documentation

Documentation... Love it or hate it, but professionally produced software absolutely requires documentation at least as good as the software itself.

Some of the different types of docs we might need to produce include:

- **Customer-facing documentation:** Guides, tutorials, how-to's...
- **API documentation**
- **Technical documentation:** Class diagrams, deployment diagrams, dependency graphs...
- **Code documentation:** Comments...
- **Project/product documentation:** Internal stuff like tickets, roadmaps, call sheets...

So being lax about *what* types of docs we are talking about is maybe not too helpful. But we can slice the cake differently, and try to bucket the above into two major categories:

- **Requires manual labor:** Intellectual work needed, not deterministic
- **Better when generated:** Context is complicated, dynamic, but deterministic

Warning

Just like code, documentation is a liability and a responsibility. The more of it you have, the higher the price of upkeep can be.

We should attempt to reach the point where as much as possible of the documentation can be generated and output through automation. Good candidates for automated document generation would be dependency graphs, some of our architecture views, and technical documentation from our comments, types, and code structure.

Now let's look at how we've done a bit in the code part of this project.

Example 1: Open up any TS file in the src folders. Here, we use the [JSDoc](#) standard for documenting source code. Since we are using TypeScript, the style I am using discards “params” instead mostly focusing on the descriptive and referential aspects of JSDoc.

Let’s see src/FakeUser/frameworks/callExternal.ts:

```
/**
 * @description Check if JSON is really a string.
 * @see https://stackoverflow.com/questions/3710204/how-to-check-if-a-↵
 * ↵ string-is-a-valid-json-string-in-javascript-without-using-try
 */
const isJsonString = (str: string): Record<string, unknown> | boolean ↵
    ↵ => {
  try {
    JSON.parse(str);
  } catch (e) {
    return false;
  }
  return true;
};
```

Example 2: Then, we use [TypeDoc](#) for generating docs from the source code. You can generate documentation into the typedoc-docs folder by running `npm run docs`. These are the documents uploaded to [the static website](#) as well. This brings us full circle on certain documentation being automatically updated and uploaded, as soon as we push any changes!

Example 3: For diagrams, we use [Arkit](#) for generating diagrams from your software architecture. As with the docs, these are also generated with `npm run docs` and presented on the published website.

Example 4: Taken together, this means that you can easily make rich documentation available in a central, visible location (like a static website in this case) in the CI stage. See [.github/workflows/main.yml](#) for the CI script itself. Note that Cloudflare Pages is set up outside of GitHub so you won’t see too much of that integration in the script.

Information

For deeper reading on the topic of documentation, I highly recommend:

- [Documenting APIs: A guide for technical writers and engineers](#)
- [Google: Technical Writing One](#)
- [Principles of technical documentation](#)

Unit testing (and more)

More code is bad. Less code is good. Only add code when the other options have been exhausted. This also applies to tests.

You are writing tests to increase your confidence in your system. If you are not confident about aspects of your system, do more testing. When you have to make compromises due to time and budget constraints, always prioritize testing the areas where you need the most confidence.

— Source: [Scott Logic](#)

Testing is fundamentally about building confidence in our code. We need to have enough tests to accurately be able to say that our code is covered for its critical use-cases and that we feel confident about the code (and tests!) we wrote.

Information

While there is no such thing as “untestable” code, in practice if something is hard as nails to test, then it’s probably the code (not the test) that needs refactoring.

Example: You’ll see that the tests under [tests/](#) are segmented into several wider categories. The tests under [tests/unit](#) are for the individual relevant layers such as controllers, entities, and frameworks.

Creating coverage, fast

If you are writing tests *after* having created the initial code—functions, classes, etc—then a good (and fast!) way to create baseline coverage and confidence is by writing tests for the high-level layers, such as the controllers. Some people call these fuller, outer-boundary tests “component tests” since they address full use cases that likely encapsulate multiple smaller functions.

Testing this way, you have the immediate benefit of understanding your code as an API, as this is the closest to—or even the exact same—code that will be “the real deal” going

out into production. These tests, therefore, tend to be very similar when writing for the integration and contract test use-cases: After all, they all try to address the same, or at least similar, things through more or less the exact same API.

Information

Spending even a bit of time raising the overall coverage to 90% or more will provide valuable confidence to you and your team. Remember that there are diminishing returns after a certain point, and you should feel comfortable about just leaving some things untested, especially so if the uncovered areas are not able to create meaningful problems.

Such coarse-grained API/integration/contract/component/unit tests also typically exercise a wider portion of your codebase, though normally not all of the error handling and such. *This is when you want to start testing on a finer level and not only at the very edge of your code.*

Test positive and negative states

For “frameworks” (such as utilities and deterministic calculations) these should be written so that they are very easy to test in isolation. These tend to be the easiest and least messy parts to test. On the other hand, though, they are functionally the inverse of broad tests: Less messy, but only covers a small subset of your overall codebase. Remember to test both “success” (positive) and “failure” (negative) states as far as logically possible and meaningful.

If you later discover unknown test cases that need to be added, to guard against those, we just add them to the unit test collection. Some people call these “regression tests”—though I never call anything a regression test, as the test collection overall acts as a regression suite—but it all leads to the same effect in the end.

And if there is something testers, QAs, and people in the testing sphere seem to love, it’s semantics. Screw the semantics and go for confidence instead, using all the tools you need to get there!

Test automation in CI

First, let's look at what I personally call “defensive testing”, that is, any typical testing that we do to test the resiliency and functionality of our solution.

We need to establish clear boundaries and expectations of where our work ends, and someone else's work begins. I really wish this were a technical topic with clear answers, but the more I work with people, the more I keep getting surprised by how little teams do to precisely define their boundaries.

Information

It's well worth understanding (and practically implementing!) the concept [bounded context](#) in your product/project.

Example 1: To not be overloaded with other's services (though we may be dependent on them) we can conduct API mocking to mock away external dependencies. For this purpose, we choose to use [Mock Service Worker](#). See [tests/mocks/handlers.ts](#). The generic pattern looks as easy as this ([tests/mocks/handlers.ts](#)):

```
rest.get(API_ENDPOINT, (req, res, ctx) => {  
  return res(ctx.status(200), ctx.json(apiResponse));  
});
```

Example 2: Next up we set up [contract testing](#) using [TripleCheck CLI](#). Contract testing means that we can easily verify if our assumptions of services and their interfaces are correct, but we skip verifying the exact semantics of the response. It's enough that the shape or syntax is right.

Information

For wider scale and bigger system landscapes, consider using the [TripleCheck broker](#) to be able to store and load contracts from a centralized source.

See [triplecheck.config.json](#) and the TripleCheck documentation linked above.

Example 3: During the CI stage we will deploy a complete, realistic stack with the most recent version. First, we'll do some basic [smoke tests](#) to verify we don't have a major malfunction on our hands. *In reality, smoke tests are just lighter types of integration tests.*

See [tests/synthetics/smoketest.sh](#); it's just a very basic curl call!

Example 4: When it comes to *actual* integration testing of the real service we'll do it after we've seen our smoke tests pass. My solution is a home-built thingy that makes some calls and evaluates the expected responses with the received data using [ajv](#).

See [tests/integration/index.ts](#).

Example 5: See [.github/workflows/main.yml](#) for the CI script. These scripts are not magic – get acquainted with them, and just as with regular code, make these easy to read and understand.

You'll see all the overall steps covered sequentially (I've cut out all non-name information):

```
- name: License check
- name: Scan for secrets
- name: Run Trivy vulnerability scanner
- name: Cache dependencies
- name: Install dependencies
- name: Test (units)
- name: Test (contracts)
- name: Test (IAC)
- name: Deploy services
- name: Bake time
- name: Smoke test
- name: Test (integrations)
- name: Test (load)
- name: Deploy API documentation
```

If that battery doesn't cover your needs you can just spend a bit of time extending it with your specifics. However, this script should provide a more than ample basis for production circumstances.

Synthetic testing

Synthetic testing sounds cool and intriguing and vaguely futuristic, but it's really no more than a directed stream of traffic from a non-human source, such as a computer.

Previously we used a smoke test—which is certainly a type of synthetic test—but there is another dimension I want to bring up as well: We can use synthetic testing to continuously verify that our solution *keeps* performing as expected. In that regard, we can think of it as a kind of “offensive testing” mechanism we submit our live code to.

Running synthetic traffic is something that can certainly stress the existing instance/server/-machine/environment, but in our case we could also leverage it *during the deployment window* (when rolling out a canary deployment) to more fully exercise the system, flushing out any issues. This would also give us a higher probability of hitting any issues—which we may not do during a (for example) 10-minute window with low organic traffic.

Example: We can use load testing to run various larger-scale synthetic traffic volumes as one-offs to stress the API. Under `tests/load/k6.js` you can see our `k6` script that we run in CI. The use-case we have here is to ensure that our system responds correctly when provided with a range of inputs rather than just doing the quick poke-and-feel we did with the smoke test.

Synthetics can be run with almost anything: Examples include a regular API client like [Insomnia](#), a load testing tool like `k6` or [Artillery](#), a SaaS product like [Checkly](#), or a managed tool like [AWS CloudWatch Synthetics](#). Even `curl` works fine! If you want to set something up to continuously run against your endpoint, I recommend using an easy-to-use tool like CloudWatch Canaries or Checkly, directing them to your endpoint.

Automated scans

In this age of DevOps, we don't want to forget the security portion of our responsibility. Perhaps the more proper term is DevSecOps, which is more and more making headway in the developer community.

To support Dev(Sec)Ops, a best practice is to use various types of scans to automate often boring, sometimes hard, sometimes also mandated requirements e.g. for compliance and security aspects.

Since there is no DevOps without automation the tooling we adopt needs to work both in CI and locally, and provide meaningful confidence and improvements to our delivery. [Git-Lab writes about their view on “5 benefits of automated security”](#), which they summarize as:

- Reduced human error.
- Early security intervention.
- Streamlined vulnerability triage.
- Repeatable security checks.
- Responsibility clarification.

This is all gold. We want this.

There exists a lot of tooling in this space these days. We're going to use some of the more well-known, free and open-source options.

Information

You can see the below tools running in the [CI script](#).

Example 1: We use [Trivy](#) to check for vulnerabilities in packages (among other places).

Example 2: Then, we use [Checkov](#) to scan for misconfigurations, and also create an infrastructure-as-code SBOM (“[software bill-of-materials](#)”).

Example 3: Finally we use [gitleaks](#) to detect hard-coded secrets in our repository.

Generate a software bill of materials

Answering the question “What went into making your software, besides swearing and broken deadlines?”

We need to understand what our software is composed of—this is called “software composition analysis” (SCA).

For certain cases (such as regulated industries) this is *extremely* important, down to the requirement of knowing each and every dependency and what they themselves are built out of... For our case, though, we are creating the SBOM to understand at “face value” what software (and risks) we are bundling together.

Example: We create a Software Bill of Materials (or SBOM) similarly to how we ran automated scans, except this time we do it for our packages using [Syft](#). This is, yet again, visible together with the other tools running in the [CI script](#).

Open source license compliance

Make sure you make all the open-source angels out there happy by complying with obligations and license restrictions.

Open source is fantastic, you don't need me to tell you about it! However, consuming (and sometimes redistributing) open source is not always a very trivial matter. Especially when we start having to comply with license obligations, like providing license files, attributing people, and so on.

Information

I've previously written on this topic on Medium, [Open source license compliance, the TL;DR version](#). Some other good resources for open source and licensing include:

- [TLDRLegal](#)
- [GitHub: Open Source Guides](#)
- [Google Open Source](#)

To deal with potential legal issues, we'll set up checks to allow only permissive, good, and well-established open-source licenses to be used in our own software.

Example: In `package.json` we have two scripts that run as part of the pre-commit hook and in CI:

```
"licenses": "npx license-compliance --direct --allow 'MIT;ISC;0BSD;BSD↔  
↔ -2-Clause;BSD-3-Clause;Apache-2.0;Unlicense;CC0-1.0'",  
"licenses:checker": "npx license-compatibility-checker",
```

These verify that we only use a set of allowed open-source licenses, using [license-compliance](#), and can also use [license-compatibility-checker](#) to check for compatibility between our license and our used ones.

Because we are doing server-side applications (i.e. a backend or API), we are not redis-

tributing any code, making our obligations easier to handle and less messy. Webpack will bundle all licenses as well, so we should be all set.

Release versioned software and produce release notes

Each release should be uniquely versioned. We should also keep release notes, but it's one of those things that can be easy to miss.

...I really wish this was something more enterprise teams did, but I find it to be less common than in the open-source or product development communities. It kind of makes sense, but this needn't be the case.

So, how do we make it easy “to do the right thing”? We'll add a tool to help us!

Example: Instead of writing manual release notes, we use [Standard Version](#) to output them for us from our commits into the [CHANGELOG.md](#) file. Practically, this works simply by running `npm run release`.

Information

The standard convention we want to use is called [semantic versioning](#), which is the format you've probably seen a million times with versions like `1.0.3` and `3.10.2`, representing MAJOR.MINOR.PATCH in the version number. Using this tool you get that automatically. For a Node project like the one we are using, you could simply update the number in `package.json` and then regenerate any files (`npm run build` or similar and build the docs) and you're all set prior to publishing on NPM or similar package libraries.

This should be easy and powerful enough to help set an example in this area.

Stability

If it's easy to overload tooling in the first portion of a product's lifecycle, we mustn't lose track of providing a stable experience, even as we work on the product.

This section brings out a battery of ways in which we can continue from the overall quality-enhancing practices to delivering without flinching.

- Lifecycle management and roadmap
- API schema
- API schema validation
- API client version using headers
- Branch by abstraction
- Beta functionality
- Feature toggles ("feature flags")
- Authorization and role-based access
- Canary deployment

Lifecycle management and roadmap

More and more companies and products are using *public roadmaps* (see for example [GitHub's public roadmap](#)). Public or not, ensuring that other enterprise teams and similar stakeholders know what you are doing should be considered a *bare minimum*.

We can follow a basic convention where we divide an API's lifecycle into design, lifetime↔↔, sunset, and deprecation phases.

Information

Refer to the pattern [Aggressive Obsolescence](#) and articles by [Nordic APIs](#) and [Stoplight](#).

Beyond these, we add the notion of being removed which is the point at which a given feature has been completely purged from the source code.

Example: We'd keep a roadmap like the below in our docs. Imagine that today is 25 November 2021 and see below what our codebase would represent at that point in time:

Feature	Group/Flag	Beta start	Stable	Sunset	Deprecated	Removed
Use hardcoded response	N/A	1 August 2021	8 August 2021	1 October 2021	15 November 2021	15 January 2022
Use RandomUser instead of JSONPlaceholder	Group devNewFeature	1 November 2021	15 January 2022	N/A	N/A	N/A

Figure 1.8: Table

This takes only a few minutes to set up, but already gives others clear visibility into the plans and current actions affecting your API.

API schema

Let's make an API schema, unless you want others to literally have to conduct black box penetration testing to understand your API.

An API schema describes our API in a standardized way. API schemas can be validated, tested, and linted to ensure that they correspond to given standards. It's important to understand that in most cases *the API schema is not the API itself*.

You can either:

- Write schemas by hand
- Generate schemas with the help of tooling
- Or go with services like [Stoplight](#), [Bump](#), [Readme](#), and API clients like [Insomnia](#) that sometimes have capabilities to design APIs, too.

When you actually have a schema, make sure to make it accessible and visible (that's our reason for using Bump in the code part of this book).

There are a few ways to think about schemas, like “[API design-first](#)”, in which we design the API and generate the actual code from the schema. Our way is more traditional since we create the code and keep the schema mostly as a representation of the implementation—However: A very important representation!

We use the [OpenAPI 3](#) standard. The approach is a manually constructed representation of our actual behavior. This is hardly the most forward-looking option available, but it's easy to understand, easy to get right, and lets us (for what it's worth) implement the API as we need while trying to stay true to the schema specification.

Information

Learn more about OpenAPI 3 over at [Swagger](#) and [Documenting APIs](#).

Example: See [api/schema.yml](#) for the OpenAPI 3 schema. Since our approach is manual, we have to implement any security and/or validations on our end in code. In our case,

this is both for in-going and outgoing data. Ingoing data can be seen handled at [src/↔ /FakeUser/controllers/FakeUserController.ts](#) in `checkInput()`, and outgoing data is handled in [src/FakeUser/entities/User.ts](#) and its various validation functions like `validateName()`.

```
/**
 * @description Check and validate input.
 */
function checkInput(event: APIGatewayProxyEvent): string {
  const clientVersion =
    event?.headers["X-Client-Version"] || event?.headers["x-client-↔
↔ version"];
  const isClientVersionValid = validateClientVersion(clientVersion || "↔
↔ ");
  const userId = event?.requestContext?.authorizer?.principalId;
  const isValidUser = validateUserId(userId || "");

  if (!isClientVersionValid || !isValidUser)
    throw new Error("Invalid client version or user!");

  return clientVersion || "";
}
```

Success

Strongly consider using security tooling like 42Crunch's [VS Code plugin for OpenAPI](#). Note also that because this is intended as a public API, the OAS security object is not present.

Information

For GraphQL, consider if something like [Apollo Studio](#) might be a way to cover this area for your needs.

API schema validation

AWS API Gateway offers [request \(schema\) validation](#). Schema validation is done with [JSON schemas](#) which are similar to, but ultimately not the same as, OpenAPI schemas.

These validators allow our gateway to respond to incorrect requests without us needing to do much of anything about them, other than provide the validator. Also, we have the benefit of our Lambda functions not running if the in-going input is not looking the way we expect it to.

Success

It can't be understated how important it is to actually use the capabilities of the cloud components/services we are using. It's *primitive and wrong* to have to do **basic** request validation in our application layer—use the built-in capabilities in API Gateway and similar services and do more business-oriented validation as needed in the application instead.

Once again, since we have a manual approach, any validation schemas need to be handled separately from our code and the OpenAPI schema.

Information

We only use validators for POST requests, which means the `FeatureToggles` function is in scope, but not the `FakeUser` function.

Example: See [api/FeatureToggles.validator.json](#). It's attached to our feature toggles function in `serverless.yml` on lines 96-98.

```
FeatureToggles:
  handler: src/FeatureToggles/controllers/FeatureTogglesController.↵
  ↵ handler
  description: Feature toggles
  events:
    - http:
```

```
method: POST
path: /featureToggles
request:
  schema:
    application/json: ${file(api/FeatureToggles.validator.json)}↵
↵ }
```

API client version using headers

One typical way to define expectations on an API is to use versioning. While there are several ways to do this—for example, refer to [this article from Nordic APIs](#)—we are going to use a header to decide which API backend to actually activate for the request.

Warning

[GraphQL is a different beast when it comes to versioning](#). See this section as REST-specific in its details, but a good idea overall as long as you adapt to what versioning means for your protocol.

Example: See `src/FakeUser/controllers/FakeUserController.ts` and note how the header is handled in `checkInput()`.

```
/**
 * @description Check and validate input.
 */
function checkInput(event: APIGatewayProxyEvent): string {
  const clientVersion =
    event?.headers["X-Client-Version"] || event?.headers["x-client-↵
    ↵ version"];
  const isClientVersionValid = validateClientVersion(clientVersion || "↵
    ↵ ");
  const userId = event?.requestContext?.authorizer?.principalId;
  const isValidUser = validateUserId(userId || "");

  if (!isClientVersionValid || !isValidUser)
    throw new Error("Invalid client version or user!");

  return clientVersion || "";
}
```

Danger

An alternative and perhaps more commonly used approach would be to deploy a new instance of the API—like `api.com/v2`—but of course, this would create further hardware segregation (having a separate v1 and v2 instance), which we want to avoid.

So, while it may be non-standard, in this context version 2 of the API represents the beta, meaning that version 1 represents the current (or “stable”, “old”) variant.

With this, we have created a way to dynamically define our response simply through a header, without resorting to separate codebases or separate deployments. No need for anything more complicated, as long as we handle this logic in a well-engineered way.

Branch by abstraction

Get rid of heavy-handed branching and just branch in code instead. But how?

How do we do better than using branches? Well... not using branches! But how to deal with changes bigger than we want to contain in a single commit?

“Branch by Abstraction” is a technique for making a large-scale change to a software system in gradual way that allows you to release the system regularly while the change is still in-progress.

— Source: [Martin Fowler: BranchByAbstraction](#)

[Paul Hammant](#) seems to be the originator, if not of the pattern, then at least of the term. He’s also clear on this being smarter than using multiple branches.

[Branch] by abstraction instead of by [code] branching in source control. And no, that doesn’t mean sprinkle conditionals into your source code, it means to use an abstraction concept that’s idiomatic for the programming language you are using.

— Source: [Branch By Abstraction?](#)

This pattern works especially well when making significant changes to existing code. I might be harsh here, but there might well be severe code smells already present, since abstracting this way should be easy with well-engineered and nicely separated code.

It’s all pretty simple, actually. The full eight steps are:

1. Introduce an abstraction to methodically chomp away at that time consuming non-functional change
2. Start with a single most depended on and least depending component
3. To not jeopardize anyone else’s throughput, work in a place in the code-base that is separate from the existing code
4. Methodically complete the work, temporarily duplicating tens or hundreds of components

5. Go live in a ‘dark deployment’ mode part complete as many times as needed
6. Scale up your CI infrastructure to guard old and new implementations
7. When ready, switch over in production (a toggle flip to end ‘dark deployment’ mode)
8. Lastly, delete old implementations and the abstraction itself (essential follow up work)

Example: While our example might be too lightweight, and involved “new” and not old code, we do have a “beta” version and a “current” version as two code paths (see [src/FakeUser/controllers/FakeUserController.ts](#), lines 37-44), abstracted in the controller.

Effectively, we are—even in this contrived example—making it easy and possible to work together with both new and old code without too much risk. Taking this thinking further, it’s possible to solve vastly more complex situations just as well.

This is `src/FakeUser/controllers/FakeUserController.ts`:

```
/**
 * Run current version for:
 * - Legacy users
 * - If missing version header
 * - If version header is explicitly set to an older version
 */
if (
  !clientVersion ||
  parseFloat(clientVersion) < BETA_VERSION ||
  toggles.userGroup === "legacy"
)
  return currentVersion();
// Run beta version for everyone else
else return await betaVersion(toggles);
```

If you need a hint, the encapsulation of the versions into their own “use-cases” makes it very easy to package completely different functionality into the same deployable artifact.

Information

Refer to the [Branch by Abstraction website](#) for more.

Beta functionality

Supporting beta features is not that hard. Let's do it!

We have built in a “beta functionality” concept that is propagated from feature toggles into our services. This is a catch-all for new features that we want to test, and which may not yet be ready for wider release. This means that our services need to have distinct checks for this, though.

As you can see in the next section on feature toggles, we can also use user groups to segment features, as well as classic, individual toggles that can be used on a per-user basis.

What gives? Aren't these beta features just like *any other* toggles? **Yes.** In this project, we define “beta features” as a *feature-level, wide bucket across user groups*, while user grouping is a dynamically wide bucket (basically just audience segmentation). This way, we can define granularly both user groups and on beta usage.

Example: See for example [src/FakeUser/usecases/createFakeUser.ts](#) that uses the provided toggles when calling the User entity which dynamically creates different types of users from this toggle.

```
import { UserData, UserDataExtended, User } from "../entities/User";

import { getData } from "../interactors/getData";
import { getImage } from "../interactors/getImage";

/**
 * @description This is where we orchestrate the work needed to fulfill↵
 * ↵ our use case "create a fake user".
 */
export async function createFakeUser(
  toggles: Record<string, unknown>
): Promise<UserData | UserDataExtended> {
  // Use of Cat API is same in all cases
  const user = new User(toggles.enableBetaFeatures as boolean | false);
  const imageResponse = await getImage("CatAPI");
  user.applyUserImageFromCatApi(imageResponse); // <-- Rich entity ↵
  ↵ object has dedicated functionality for differing data sources

  // Use code branching for new development feature
```

```
if (toggles.enableNewUserApi) {  
  const dataResponse = await getData("RandomUser");  
  user.applyUserDataFromRandomUser(dataResponse); // <-- Rich entity ↩  
  ↩ object has dedicated functionality for differing data sources  
}  
// Else return regular response  
else {  
  const dataResponse = await getData("JSONPlaceholder");  
  user.applyUserDataFromJsonPlaceholder(dataResponse); // <-- Rich ↩  
  ↩ entity object has dedicated functionality for differing data ↩  
  ↩ sources  
}  
  
return user.viewUserData();  
}
```

Feature toggles (“feature flags”)

It’s time to go from static configs to dynamic configurations and spend less time deploying.

The code part of this book uses a handcrafted, simple feature flags engine and a small toggle configuration.

While a technically trivial solution, this enables a dynamic configuration that can be detached from the source code itself. [In effect, this means we are one big step closer to separating a \(technical\) deployment from a \(business-oriented\) release.](#)

The feature function is located at [src/FeatureToggles](#). For the configuration part, we should put it on [Mockachino](#) so that we can approximate a more conventional feature toggles service, getting (and updating) flags without having to re-deploy our code. The set of feature toggles is ordered under each respective user group.

Unknown (or missing, non-existing, null...) users get the standard user group access.

Information

Note that different feature toggle services or implementations may differ in how they exactly apply toggles, and how they work. In our case here, we apply group-level toggles rather than request individual flags, if nothing else than for simplicity of demonstration.

Example: The full configuration you can use as your template looks like the one below. Notice how it’s segmented on the group level:

Let’s see `src/FeatureToggles/config/mock.toggles.json`:

```
{
  "error": {
    "enableBetaFeatures": false,
    "userGroup": "error"
  },
  "legacy": {
    "enableBetaFeatures": false,
```

```
    "userGroup": "legacy"
  },
  "beta": {
    "enableBetaFeatures": true,
    "userGroup": "beta"
  },
  "standard": {
    "enableBetaFeatures": false,
    "userGroup": "standard"
  },
  "dev": {
    "enableBetaFeatures": true,
    "userGroup": "dev"
  },
  "devNewFeature": {
    "enableBetaFeatures": true,
    "enableNewUserApi": true,
    "userGroup": "devNewFeature"
  },
  "qa": {
    "enableBetaFeatures": false,
    "userGroup": "qa"
  }
}
```

You can update the flags as you want, to get the effects you need, without requiring redeploying the code!

Information

Refer to [FeatureFlags.io](https://featureflags.io) and to [Unleash's documentation on activation strategies](#) for inspiration.

Authorization and role-based access

You'll often see tutorials and such talking about [authentication](#), which is about how we can verify that a person really is the person they claim to be. This tends to be mostly a technical exercise.

[Authorization](#), on the other hand, is knowing what this person is allowed to do.

Only trivial systems will require no authorization, so prepare to think about how you want your model to work and how to construct your permission sets. Rather than being a technical concern, this becomes more logical than anything else.

Warning

Authentication is entirely out of scope here, whereas trivial authorization is in scope.

Example 1: In [serverless.yml](#) (lines 60-62) we define an authorizer function, located at [src/FeatureToggles/controllers/AuthController.ts](#). Then on lines 70-74, we attach it to the FakeUser function. Now, the authorizer will run before the FakeUser function when called.

```
functions:
  Authorizer:
    handler: src/FeatureToggles/controllers/AuthController.handler
    description: ${self:service} authorizer
  FakeUser:
    handler: src/FakeUser/controllers/FakeUserController.handler
    description: Fake user
    events:
      - http:
          method: GET
          path: /fakeUser
          authorizer:
            name: Authorizer
            resultTtlInSeconds: 30 # See: https://forum.serverless.com/↔
            ↪ t/api-gateway-custom-authorizer-caching-problems/4695
            identitySource: method.request.header.Authorization
            type: request
```

Example 2: In our application we use a handcrafted [RBAC](#) (role-based access control) to attach a user group to each of the users. The user group is added as a flag in the subsequent call to the actual service.

In [src/FeatureToggles/usecases/getUserFeatureToggles](#) users are matched like so:

```
/**
 * @description Get user's authorization level keyed for their name. ↵
 * ↵ Fallback is "standard" features.
 */
function getUserAuthorizationLevel(user: string): string {
  const authorizationLevel = userPermissions[user];
  if (!authorizationLevel) return "standard";
  else return authorizationLevel;
}
```

And [src/FeatureToggles/config/userPermissions.ts](#):

```
export const userPermissions: Record<string, UserGroup> = {
  "erroruser@company.com": "error",
  "legacyuser@company.com": "legacy",
  "betauser@company.com": "beta",
  "standarduser@company.com": "standard",
  "devuser@company.com": "dev",
  "devnewfeatureuser@company.com": "devNewFeature",
  "qauser@company.com": "qa",
};
```

Information

For more full-fledged implementations, consider tools like [Permit.io](#) or [Oso](#) to authorize on the overall “access level”. [See for example this demo](#) using a similar serverless stack, or the plain [quick start version](#).

Canary deployment

Get ready for Friday deploys with canary deployments!

The *traditional* way to deploy software is as one huge chunk that becomes instantly activated whenever it's deployed to a machine. If the code was a complete failure, then you end up having zero time to verify and correct this before the failure is apparent to users.

This notion is what makes managers ask for counter-intuitive things like code freeze and all-hands-on-deck deployments. This is dumb and wrong and helps no-one. Let's forever end those days!

Matt Casperson, writing for The New Stack, deftly portrays the journey that many are now making when it comes to finding a new “truth” when it comes to testing best practices:

[...] What I really wanted to do was leverage the existing microservice stack deployed to a shared environment while locally running the one microservice I was tweaking and debugging. This process would remove the need to reimplement live integrations for the sake of isolated local development, which was appealing because these live integrations would be the first things to be replaced with test doubles in any automated testing anyway. It would also create the tight feedback loop between the code I was working on and the external platforms that validated the output, which was necessary for the kind of “Oops, I used the wrong quotes, let me fix that” workflow I found myself in.

My Googling led me to [“Why We Leverage Multi-tenancy in Uber’s Microservice Architecture”](#), which provides a fascinating insight into how Uber has evolved its microservice testing strategies.

The post describes parallel testing, which involves creating a complete test environment isolated from the production environment. I suspect most development teams are familiar with test environments. However, the post goes on to highlight the limitations of a test environment, including additional hardware costs, synchronization issues, unreliable testing and inaccurate capacity testing.

The alternative is **testing in production**. The post identifies the requirements to support this kind of testing:

There are two basic requirements that emerge from testing in production, which also form the basis of multitenant architecture:



- Traffic Routing: Being able to route traffic based on the kind of traffic flowing through the stack.
- Isolation: Being able to reliably isolate resources between testing and production, thereby causing no side effects in business-critical microservices.

— Source: [Embracing Testing in Production](#)

Success

See these brilliant articles for more justification and why this is important to understand:

- [Charity Majors: I test in prod](#)
- [Cindy Sridharan: Testing in Production, the safe way](#)
- [Heidi Waterhouse: Testing in Production to Stay Safe and Sensible](#)

OK, so what can *we* do about it? In `serverless.yml` at line ~85, you'll see `type:` 
 `AllAtOnce`.

```
FakeUser:
  [...]
  deploymentSettings:
    type: AllAtOnce
    alias: Live
    alarms:
      - FakeUserCanaryCheckAlarm
```

This means that we get a classic `deploy === release` pattern. When the deployment is done, the new function version is immediately active with a clear cut-off between the previous and the current (new) version.

How we can do better

There are considerations and problems with this approach. In our AWS circumstances, running on Lambda, we won't face downtime while the instance switches over, and even a half-good PaaS solution won't create massive headaches either.

However, we should primarily concern ourselves by considering application-level issues, rather than low-level technical issues (much of which is managed in a, well, *managed service*). So when that “all at once” deployment is done, if something problematic surfaces that was not caught by testing, there is no obvious way how to proceed. Rollback? Roll forward? Hotfix, yay or nay? Sarcastic note: It's hardly unheard of that even manual testing and code freezes slip up.

Let's be honest: **Shit happens anyway.**



Figure 1.9: Words to live by, as told by Mike Tyson

Instead of being overly defensive, let's simply embrace the uncertainty, as it's already there anyway.

Using a [canary release](#) is one way to get those [unknown unknown effects](#) happening with real production traffic in a safe and controlled manner. This is where the (sometimes misunderstood) concept [testing-in-production](#) really kicks in—trying to answer questions no staging environment or typical test can address. Like a canary in the mines of old, our canary will die if something is wrong, effectively stopping our roll-out.

Example: Setting the line to type: `Canary10Percent5Minutes` makes the deployment happen through AWS CodeDeploy in a more controlled, bi-directed fashion:

- 90% of the traffic will pass to whatever function version that was already deployed and active...
- ...while the remaining 10% of traffic will be directed to the “canary” version of the function.
- The alarm configuration (defined on lines 75-83) looks for a static value of 3 or more errors on the function (I assume all versions here?) during the last 60-second window.
- After 5 minutes, given nothing has fired the alarm, then the new version takes all of the traffic.

This is `serverless.yml`:

```
FakeUser:
  [...]
  alarms:
    - name: CanaryCheck
      namespace: 'AWS/Lambda'
      metric: Errors
      threshold: 3
      statistic: Sum
      period: 60
      evaluationPeriods: 1
      comparisonOperator: GreaterThanOrEqualToThreshold
  deploymentSettings:
    type: Canary10Percent5Minutes
    alias: Live
```

```
alarms:  
  - FakeUserCanaryCheckAlarm
```

You can either manually send “error traffic” with the `erroruser@company.com` Authorization header, or use the AWS CLI to manually toggle the alarm state. See [AWS docs for how to set the alarm state](#), similar to:

```
aws cloudwatch set-alarm-state --alarm-name "myalarm" --state-value ↔  
↔ ALARM --state-reason "testing purposes"
```

Information

Use the above with the OK state value to reset the alarm when done.

This specific solution is rudimentary, but indicative enough of how a canary solution might begin to look. I highly recommend using a deployment strategy other than the primitive “all-at-once” variety.

Information

See this article at [Google Cloud Platform for more information on deployment and test strategies](#).

Observability

Finally, this last section is all about putting eyeballs (and alarms and metrics...!) on the product and making it operable as a modern solution.

- AWS baseline observability
- Structured logger
- Alerting
- Service discoverability and metadata
- Additional observability

AWS baseline observability

There's a lot written and discussed when it comes to *observability* vs *monitoring*. Let's go with Google's definition taken from [DORA](#):

Monitoring is tooling or a technical solution that allows teams to watch and understand the state of their systems. Monitoring is based on gathering pre-defined sets of metrics or logs.

Observability is tooling or a technical solution that allows teams to actively debug their system. Observability is based on exploring properties and patterns not defined in advance.

— Source: [Google: DevOps measurement: Monitoring and observability](#)

Further, let's also add that monitoring classically is said to consist of the three “pillars” [logs, metrics, and traces](#). In the cloud, these are all typically pretty easy to set up and we should aim to at least have these under control.

Information

Of the three pillars, tracing is maybe the least well-understood. Do read [Lightstep's good introductory article on tracing](#) if you are interested in that area!

Example: Our `serverless.yml` configuration enables [X-Ray](#) in the tracing object. By default, [CloudWatch](#) log groups are created. Next, we add the `serverless-plugin-aws↵↵ -alerts` plugin to give us some basic metrics (throttles, etc.).

Information

Read more about [metrics with Serverless Framework here](#).

Taken together, these give us a very good level of baseline observability right out of the box. Let's look at `serverless.yml`:

```
provider:
  [...]
  tracing:
    apiGateway: true
    lambda: true

plugins:
  - serverless-plugin-aws-alerts

custom:
  alerts:
    dashboards: true
```

Information

Refer to the [CloudWatch Logs Insights query syntax](#) if you want to set up some nice log queries. Also note that a shortcoming with CloudWatch is that they don't seem ideal for cross-service logs, but they are just fine when it comes to checking on a specific service. This is the intention behind the coming Honeycomb addition below.

You can now see dashboards (metrics) in both the Lambda function view and CloudWatch (plus the logs themselves) and get the full X-Ray tracing. It's just as easy as that! Done deal.

Structured logger

Structured logging should be an early thing you introduce into your stack.

Another best practice is to [treat logs as a source of enriched data rather than as plain, individual strings](#). To do so, we need to have a structured approach to outputting them.

In my implementation project, I'll demonstrate a handcrafted logging utility to help us do this in an easy way that nobody can fail to use correctly.

Information

Good folks like [Yan Cui](#) have written and presented on this matter many times and you can certainly also opt-in to turnkey solutions like [lambda-powertools](#).

I've provided a basic one that also uses `getUserMetadata()` to get metadata ([correlation ID](#) and user ID) that has been set in the environment at an early stage in the controller.

Example: See [src/FakeUser/frameworks/Logger.ts](#). Implementation is as simple as importing it and then:

This is `src/FakeUser/frameworks/Logger.ts`:

```
const logger = new Logger();
logger.log("My message!");
logger.warn("My warning!");
logger.error("My error!");
```

Or for that matter something more like this:

```
const logger = new Logger();
logger.log({
  accountId: "192k-d124",
  setting: "baseline",
  customization: {
    cinemaMode: false,
    highDef: true,
  },
},
```



```
    timePlayed: "412",  
  });
```

Information

As opposed to some solutions, in our case, the Logger will not replace the vanilla `console.log()` (etc) so you will need to import it everywhere you want to use it.

Using it, your logs will then all follow the format (`src/FakeUser/frameworks/Logger.ts`):

```
{  
  message: "My message!",  
  level: "INFO" | "WARN" | "ERROR" <based on the call, i.e. logger.warn↵  
    ↵ () etc.>,  
  timestamp: <timestamp>,  
  userId: <userId from metadata>,  
  correlationId: <correlationId from metadata>  
};
```

Alerting

Monitor as you will, but don't forget to set alarms to inform you of any incoming dumpster fires!

Alerts (or alarms; same thing) are usually connected to metrics. When the metric is triggered, the alert goes off. Easy peasy.

Example: We set up an alarm in `serverless.yml` on lines 73-81. This particular alarm gets attached to the `deploymentSettings` object and was (as seen previously) part of how we ensure that an alarm can cancel a canary deployment.

This is `serverless.yml`:

```
custom:
  alerts:
    dashboards: true

functions:
  FakeUser:
    [...]
    alarms:
      - name: CanaryCheck
        namespace: 'AWS/Lambda'
        metric: Errors
        threshold: 3
        statistic: Sum
        period: 60
        evaluationPeriods: 1
        comparisonOperator: GreaterThanOrEqualToThreshold
    deploymentSettings:
      [...]
      alarms:
        - FakeUserCanaryCheckAlarm
```

Information

You can extend this behavior to, for example, communicate the alert to an SNS topic which in turn can inform a pager system, Slack/Teams, or send an email to

a relevant person. A better way of doing this would probably use a shared service that can also keep a stored log of all events, rather than just relaying the alert directly to a source.

Service discoverability and metadata

[Service discovery](#) seems to be a very hot topic among the Kubernetes crowd. With serverless FaaS like we are using here (AWS Lambda), that's not really an interesting discussion. However, discoverability is not just a technical question—it's also something that absolutely relates to the social, organic, and management layers.

At some point, that *single* function will be one service, which will soon maybe become hundreds of services and then thousands of functions. **How to keep track of them?**

To some extent, it's possible to get a high-level picture inside of AWS or by being a CLI crusader. Unfortunately, if you are not, then there is no option unless you buy, build, or adapt some open-source solution.

Information

One such open-source solution is Spotify's [Backstage](#) which offers a broad set of ideas—no wonder since it's labeled as “an open platform for building developer portals”. It's a bit heavy, but some pretty big players are starting to use it. For a super-lightweight AWS-based and flexible solution, you might want to consider [catalogist](#) written by yours truly.

For a lighter-weight system, I'd argue that the reasonable data to pull in would be a subset of the metadata (and processes and output) we have generated and written previously. In a corporate context, we'd want to ensure that the source of truth of the state of our systems/applications resides with the codebase—Not in a closed tool like Sharepoint or Confluence or similar.

Example: See [manifest.json](#) for a napkin sketch of how one could work with service metadata if you had somewhere to send it and store it, during the CI stage. *This format is also very similar to the one used in [catalogist](#).*

This is `manifest.json`:

```
{
  "spec": {
    "lifecycle": "production",
```

```
    "type": "service",
    "name": "my-project",
    "team": "ThatTeam",
    "responsible": "Someguy Someguyson",
    "system": "something",
    "domain": "bigarea",
    "tags": ["typescript", "backend"],
    "securityClass": "Public",
    "dataSensitivity": "Sensitive",
    "l3ResolverGroup": "ThatTeam",
    "slo": "99.95"
  },
  "api": [
    {
      "FakeUser": "./api/schema.yml"
    }
  ],
  "metadata": {
    "annotations": {
      "sbom": "./outputs/sbom-output.txt",
      "typedoc": "./typedoc-docs/",
      "arkit": "./assets/"
    },
    "description": "The place to be, for great artists",
    "generation": 1,
    "labels": {
      "example.com/custom": "custom_label_value"
    },
    "links": [
      {
        "url": "https://admin.example-org.com",
        "title": "Admin Dashboard",
        "icon": "dashboard"
      }
    ]
  }
}
```

Additional observability

[Honeycomb](#) is a next-generation observability-focused solution. Honeycomb or a similar solution could be bolted on as an addition if you want some more bells and whistles beyond what AWS provides.

Many modern observability tools are based on the OpenTelemetry standard, making a choice basing itself on that standard a fairly future-proof decision. Using OpenTelemetry can be a bit of a slog, but I'll show you a way to move into richer observability while keeping the work quite manageable!

Information

Read more about [OpenTelemetry here](#).

Success

If you want to toy with Honeycomb, look no further than [their playground](#).

If you want to try Honeycomb with this project, it's pretty easy if we use their [Lambda extension](#):

1. [Get a free \(generous!\) Honeycomb account](#).
2. When you have the account set up and are logged in, get an API key (it's enough that it can send events and create datasets). [See the docs for more details](#).
3. Modify the provided logger as per below, [since the regular `console.log\(\)` does not seem to work correctly with Honeycomb](#).
4. Modify `serverless.yml`, uncommenting and adding your values to the environment variables `LIBHONEY_DATASET` and `LIBHONEY_API_KEY` (lines 37-38). **Note:** Ideally these are encrypted or stored in a secrets manager, but for this project, let's just do it this way and keep it simple for learning purposes.
5. Deploy the stack.
6. Add the Honeycomb Lambda integration layer by running `sh honeycomb-layer`. ↵
↵ `sh` (ensure the values correspond with your values first!).

7. Ready to go! You should see data coming into Honeycomb shortly if you start using your live endpoints.

Setting up Bunyan as a logger

You might want to use Bunyan rather than our custom logger if the logs don't quite show up structured the way we output them.

Install `bunyan` and its typings with `npm install bunyan @types/bunyan`.

Open up `src/FakeUser/frameworks/Logger.ts` and add this to the top of the file (`src/↵`
`↵ FakeUser/frameworks/Logger.ts`):

```
import bunyan from "bunyan";  
const log = bunyan.createLogger({ name: "better-apis-workshop" });
```

For the `log()`, `warn()` and `error()` methods, change the existing `console.log()`-dependent implementation lines to:

- `log.info(createdLog);`
- `log.warn(createdLog);`
- `log.error(createdLog);`

Lastly, in the `createLog()` method, go ahead and remove the `level` field, as bunyan adds that itself.

Now you can use Bunyan instead of regular `console.log()` or our own custom one!

Going even further with MikroLog, MikroTrace, and MikroMetric

If this is an area you enjoy and you have a similar preference as I do to lightweight, simple-as-in-dumb tools, then you might enjoy the Mikro family of tools:

- [MikroLog](#)

- [MikroTrace](#)
- [MikroMetric](#)

I've built these open-source three lightweight observability packages and designed them specifically to streamline your AWS serverless experience. These tools are all tiny, zero-config, and optimized for AWS Lambda environments, making them ideal for cloud-native applications. Each package is built with simplicity, minimalism, and effectiveness in mind, ensuring you get all the necessary functionality without the bloat.

MikroMetric is your go-to for seamlessly integrating with AWS CloudWatch, providing a straightforward syntax for managing metrics without the complexity of raw EMF. **MikroLog** offers a clean, structured logging solution that eliminates unnecessary fields and complexities, allowing for easy log management across multiple observability platforms. **MikroTrace** simplifies tracing by offering OpenTelemetry-like semantics with a focus on JSON logs, making it easier to integrate with AWS and Honeycomb. All three packages share the benefits of being extremely lightweight (around 2 KB gzipped), having only one dependency (aws-metadata-utils), and achieving 100% test coverage, ensuring they are both reliable and easy to use.

Happy to hear how you use them!

References and resources

Books

- [Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations](#), by Nicole Forsgren, Jez Humble, Gene Kim
- [Building Microservices: Designing Fine-Grained Systems \(2nd Edition\)](#), by Sam Newman
- [Clean Agile: Back to Basics](#), by Robert Martin
- [Clean Architecture: A Craftsman's Guide to Software Structure and Design](#), by Robert Martin
- [Clean Craftsmanship: Disciplines, Standards, and Ethics](#), by Robert Martin
- [Clean Code: A Handbook of Agile Software Craftsmanship](#), by Robert Martin
- [Design Patterns: Elements of Reusable Object-Oriented Software](#), by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- [Domain-Driven Design: Tackling Complexity in the Heart of Software](#) by Eric Evans
- [Implementing Domain-Driven Design](#), by Vaughn Vernon
- [Refactoring: Improving the Design of Existing Code \(2nd Edition\)](#), by Martin Fowler

Articles and online resources

- [Charity Majors: I test in prod](#)
- [Cindy Sridharan: Testing in Production, the safe way](#)
- [Heidi Waterhouse: Testing in Production to Stay Safe and Sensible](#)
- [Danilo Sato: Canary Release](#)
- [IBM: Continuous Testing](#)
- [DDD Resources](#)

- [Martin Fowler: Domain Driven Design](#)
- [Art Gillespie: Deploy != Release \(Part 2\)](#)
- [FeatureFlags.io](#)
- [Trunk Based Development](#)
- [Branch By Abstraction?](#)
- [REST API Design - Resource Modeling](#)
- [Clean architecture for the rest of us](#)
- [Refactoring.guru](#)
- [Google Testing Blog](#)
- [Yubl's road to Serverless — Part 2, Testing and CI/CD](#)
- [Google Cloud Architecture Center: Application deployment and testing strategies](#)
- [APIsecurity.io](#)
- [API Security Encyclopedia](#)
- [Why SOLID principles are still the foundation for modern software architecture](#)
- [Khalil Stemmler: How to Test Code Coupled to APIs and Databases; also available as video \(see below\)](#)
- [Documenting APIs: A guide for technical writers and engineers](#)
- [Google: Technical Writing One](#)
- [Principles of technical documentation](#)

Video

- [AWS re:Invent 2021 - Keynote with Dr. Werner Vogels](#)
- [Khalil Stemmler: How to Test Code Coupled to APIs and Databases](#)

Thank you for reading this book

Thank you for investing your time, energy, and money into reading this book. With every book and article I write, I strive to make it as useful as possible. Books allow us to delve deeper—or sometimes broader—into topics than we typically can at work or in short-form articles. Technical books, in particular, are unique creatures: they are both products of their time and, when well-crafted, can become timeless resources within their field. I hope this book remains relevant for (at least a few!) years to come.

I write the books I wish I had read earlier in my career and life. I've tried to be generous with references to other content, such as books and articles that have helped me improve in this subject. There are so many great authors out there and so much knowledge to keep up with.

If you found this book helpful, I would greatly appreciate it if you could rate it on the platform where you purchased it.

Please don't be a stranger! Connect with me on LinkedIn or wherever else I may be when you're reading this.

Once again, thank you, and I hope you found value in the time we spent together.