Four Pillars and a Base: The Nuts and Bolts of a Microservice Project

Gil Tayar (@giltayar)
March 2019

This presentation: http://bit.ly/microservice-nuts-bolts

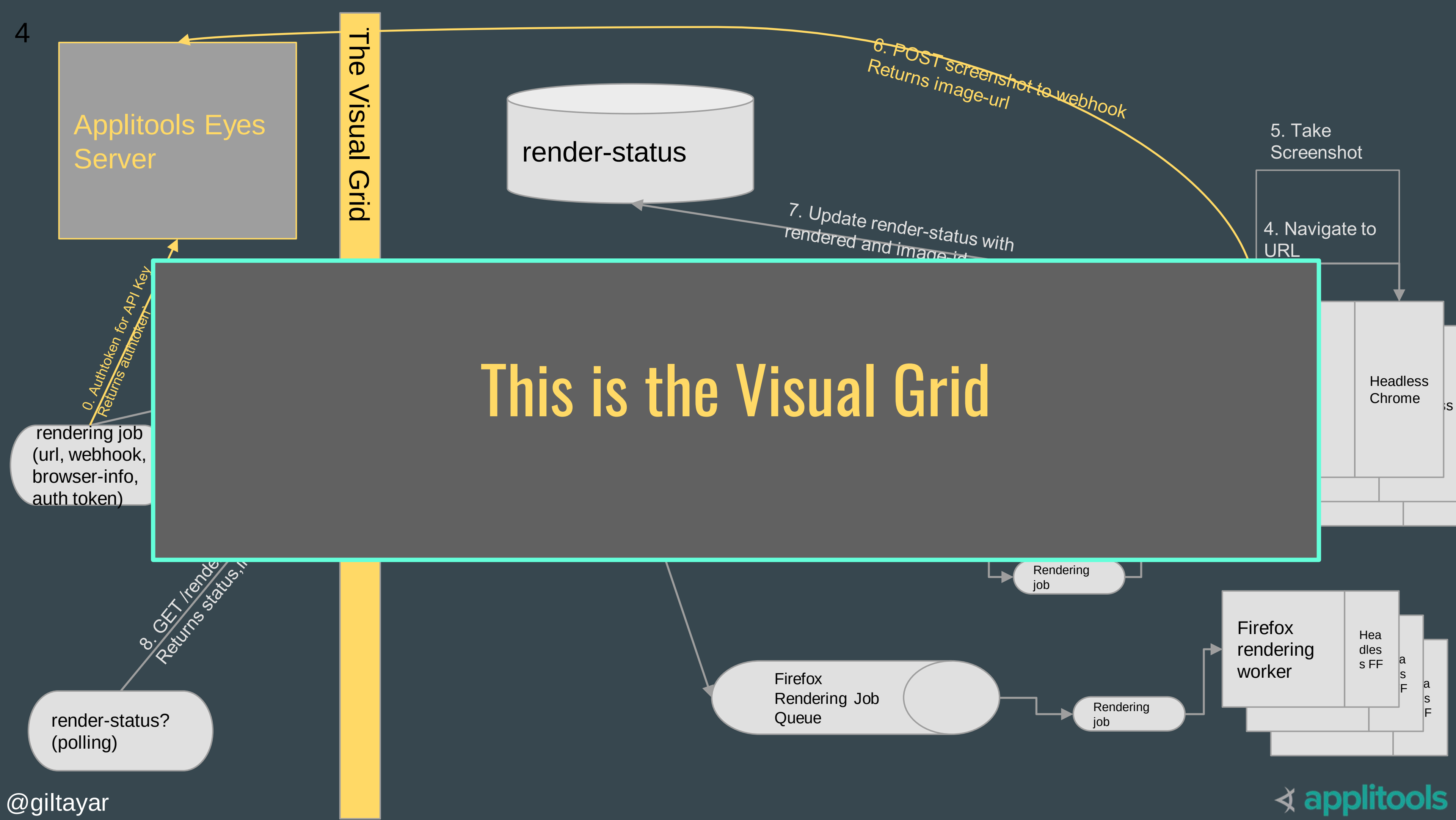
applitools

About Me

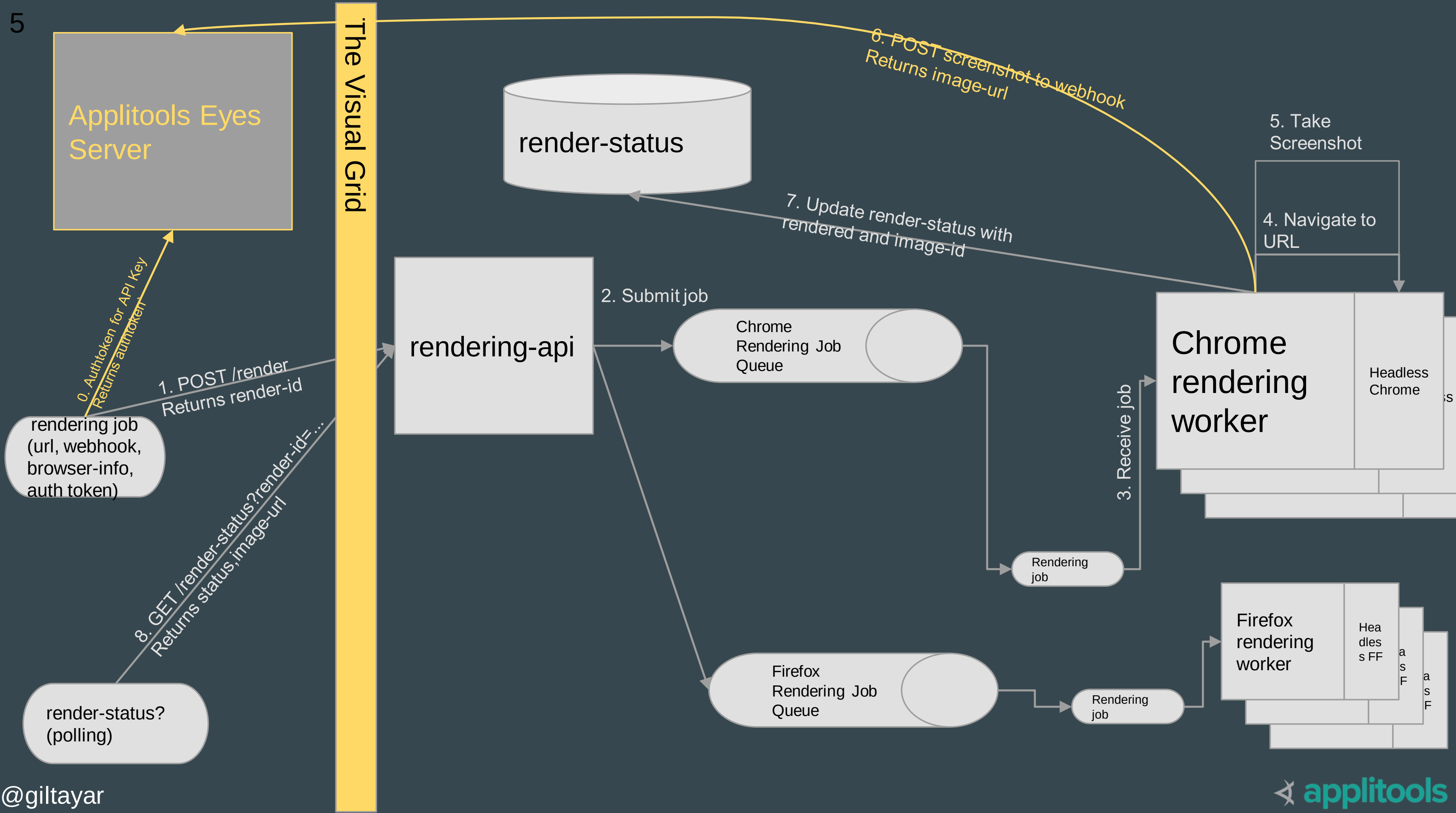
- My developer experience goes all the way back to the '80s.
- Am, was, and always will be a developer
- Testing the code I write is my passion
- Currently evangelist and architect @ Applitools
- We deliver Visual Testing tools: If you're serious about testing, checkout Applitools Eyes

- Sometimes my arms bend back
- But the gum I like is coming back in style

Four Pillars and a Base



This is the Visual Grid


Applitools Eyes Server
The Visual Grid
render-status
0. Authtoken for API Key
Returns authtoken
1. POST /render
Returns render-id
rendering job (url, webhook, browser-info, auth token)
rendering-api
2. Submit job
Chrome Rendering Job Queue
Firefox Rendering Job Queue
Rendering job
Rendering job
3. Receive job
Chrome rendering worker
Headless Chrome
4. Navigate to URL
5. Take Screenshot
6. POST screenshot to webhook
Returns image-url
7. Update render-status with rendered and image-id
8. GET /render-status?render-id=...
Returns status,image-url
render-status? (polling)
Firefox rendering worker
Headless FF

I want to talk about...

How we built the Visual Grid

But first: why we built it the way we did

Dynamic Languages vs Static Languages

And It’s War!

Between

- Java, C#, C++, Haskell, Kotlin and, yes, TypeScript

and

- Python, Ruby, Clojure, and, yes, JavaScript

Static Languages Lovers: Dynamic Languages are Scary!

- You can't trust the code
  - because no type safety
- Difficult to comprehend
  - because no type documentation

Static Languages Lovers: Dynamic Languages are Scary!

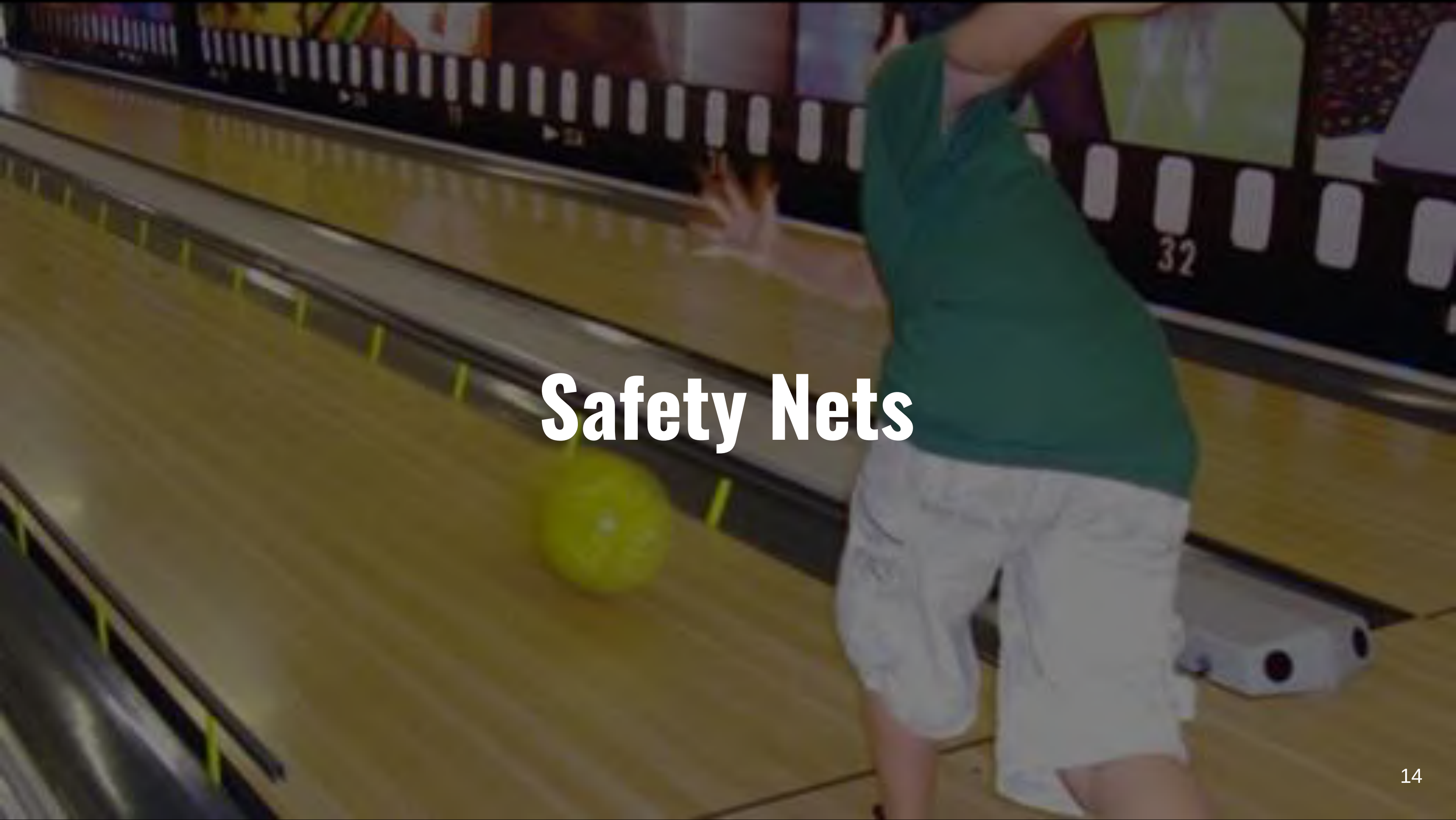
- You can’t trust the code
  - because no type safety
- Difficult to comprehend
  - because no type documentation

And they’re right!

But...

Dynamic Typing is not a weakness to overcome, but a strength to take advantage of

Dynamic Typing is not a weakness to overcome, but a strength to take advantage of



Safety Nets

Dynamic Languages *Force* You To Be Better

- You can't trust the code!
  - *Testing*
- Difficult to comprehend!
  - *Loosely-coupled small packages*

Dynamic Languages Drive Two Pillars

The Other Two Pillars (which give structure to the first two)

Testing | *Loosely-coupled small packages* | *Monorepo* | *Uniform Packages*

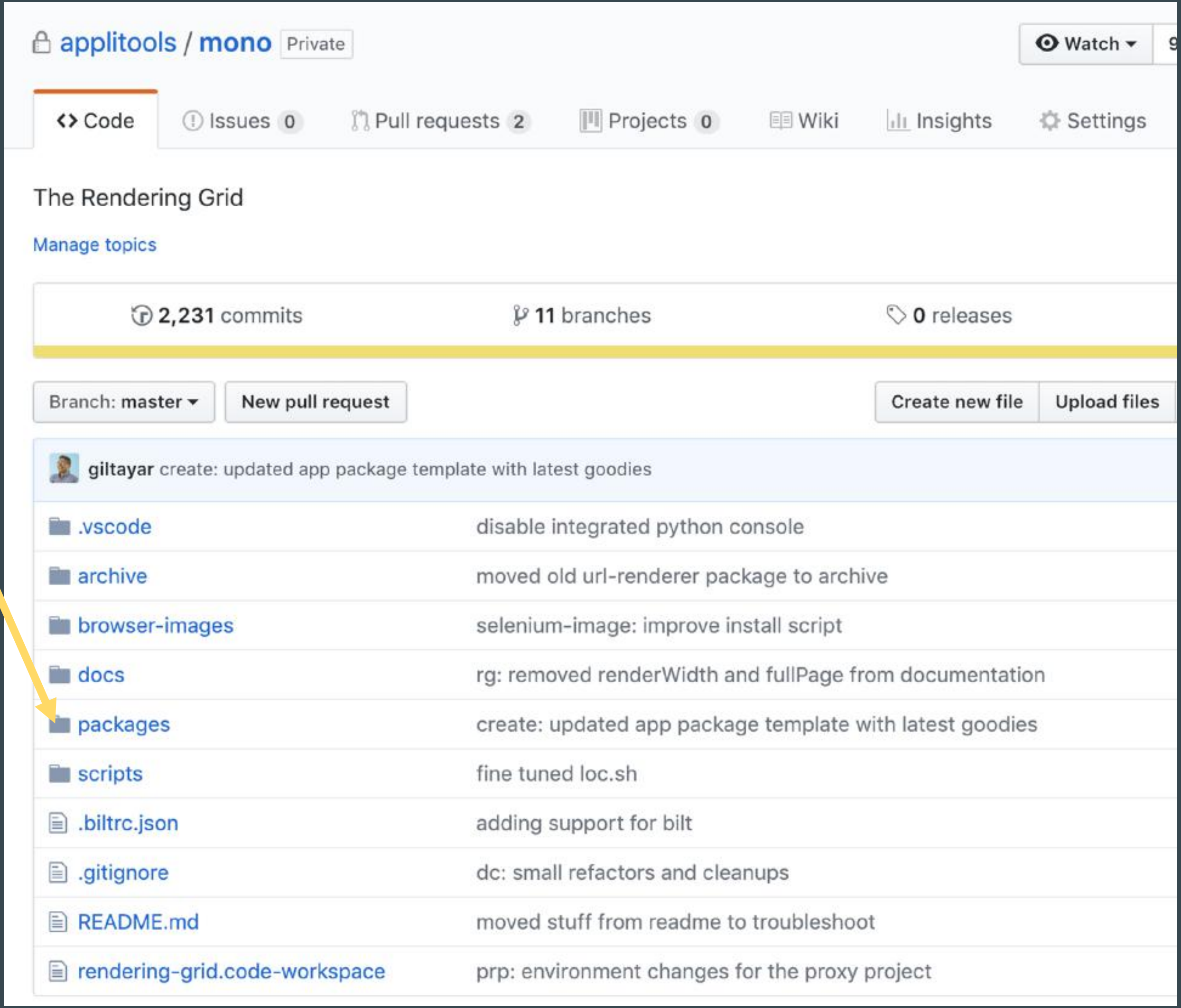
And these four pillars enable the base...

Testing
Loosely-coupled small packages
Monorepo
Uniform Packages
Frictionless Development

Testing
Loosely-coupled small packages
Monorepo
Uniform Packages
Frictionless Development

Monorepo Pillar

All Source Code is in One Git Repository



applitools / mono Private

Watch

<> Code | Issues 0 | Pull requests 2 | Projects 0 | Wiki | Insights | Settings

The Rendering Grid

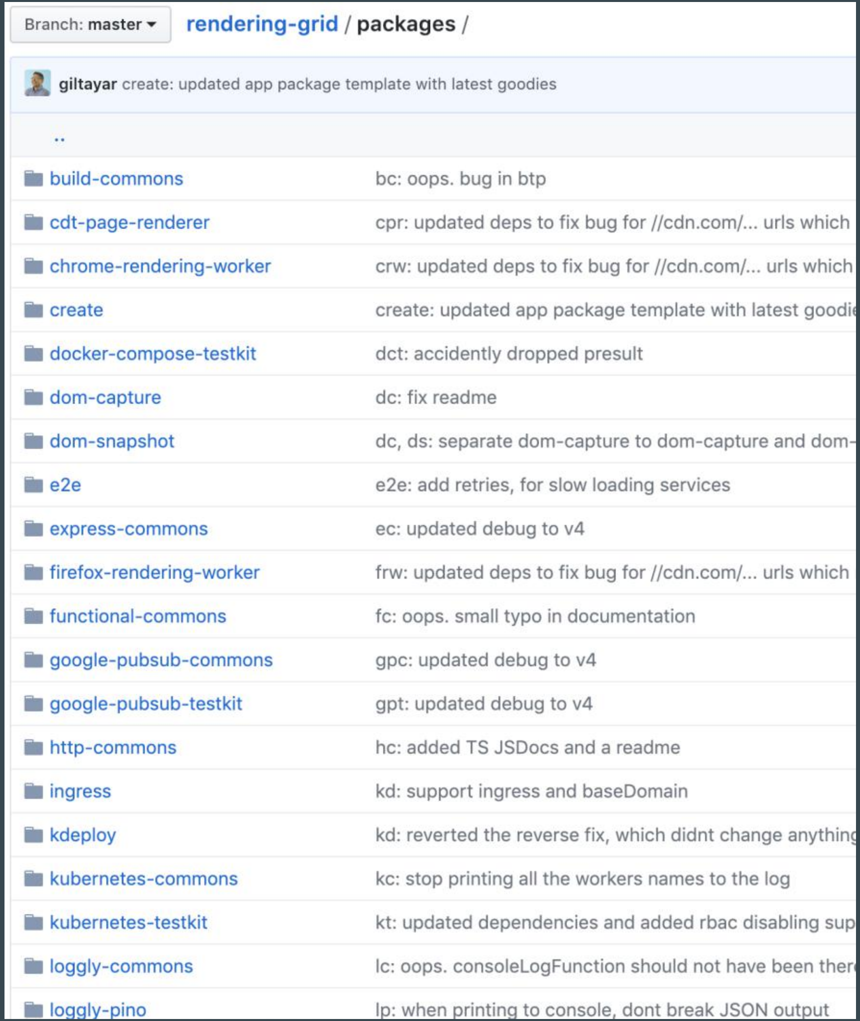
Manage topics

2,231 commits | 11 branches | 0 releases

Branch: master | New pull request | Create new file | Upload files

giltayar create: updated app package template with latest goodies

.vscode	disable integrated python console
archive	moved old url-renderer package to archive
browser-images	selenium-image: improve install script
docs	rg: removed renderWidth and fullPage from documentation
packages	create: updated app package template with latest goodies
scripts	fine tuned loc.sh
.biltrc.json	adding support for bilt
.gitignore	dc: small refactors and cleanups
README.md	moved stuff from readme to troubleshoot
rendering-grid.code-workspace	prp: environment changes for the proxy project

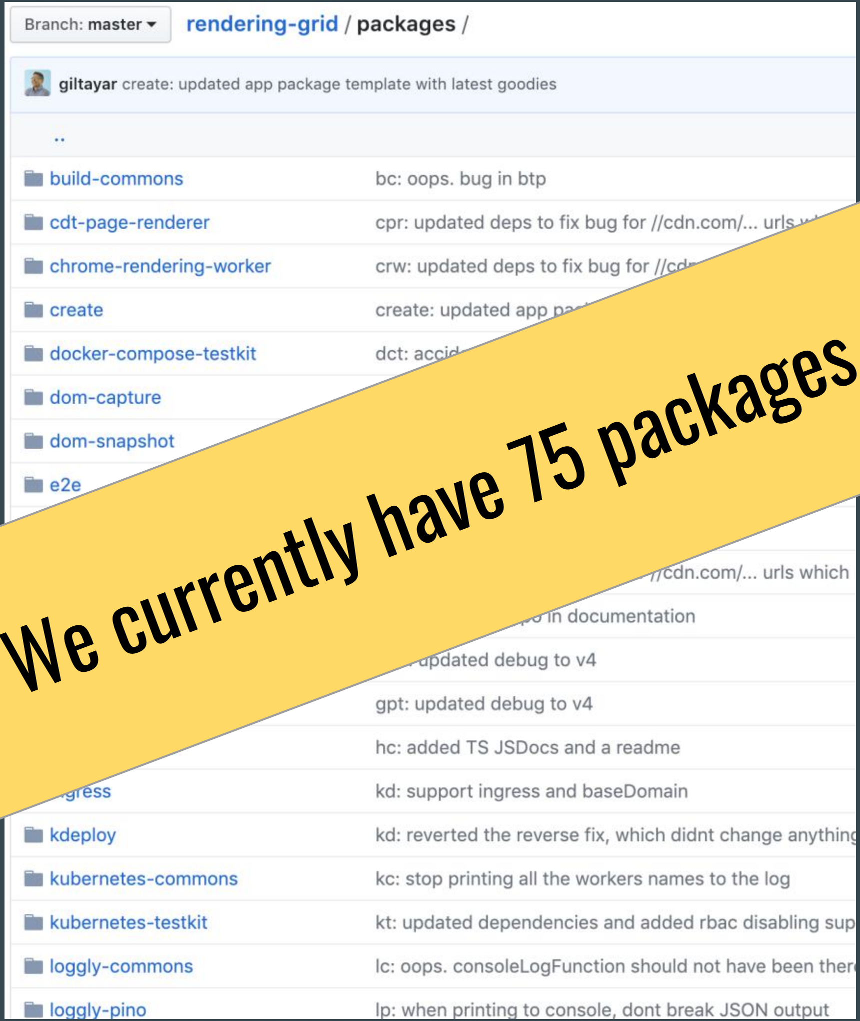


Branch: master ▾ rendering-grid / packages /

giltayar create: updated app package template with latest goodies

..

build-commons	bc: oops. bug in btp
cdt-page-renderer	cpr: updated deps to fix bug for //cdn.com/... urls which
chrome-rendering-worker	crw: updated deps to fix bug for //cdn.com/... urls which
create	create: updated app package template with latest goodie
docker-compose-testkit	dct: accidently dropped presult
dom-capture	dc: fix readme
dom-snapshot	dc, ds: separate dom-capture to dom-capture and dom-
e2e	e2e: add retries, for slow loading services
express-commons	ec: updated debug to v4
firefox-rendering-worker	frw: updated deps to fix bug for //cdn.com/... urls which
functional-commons	fc: oops. small typo in documentation
google-pubsub-commons	gpc: updated debug to v4
google-pubsub-testkit	gpt: updated debug to v4
http-commons	hc: added TS JSDocs and a readme
ingress	kd: support ingress and baseDomain
kdeploy	kd: reverted the reverse fix, which didnt change anything
kubernetes-commons	kc: stop printing all the workers names to the log
kubernetes-testkit	kt: updated dependencies and added rbac disabling sup
loggly-commons	lc: oops. consoleLogFunction should not have been there
loggly-pino	lp: when printing to console, dont break JSON output



Branch: master ▾ rendering-grid / packages /

giltayar create: updated app package template with latest goodies

..

build-commons	bc: oops. bug in btp
cdt-page-renderer	cpr: updated deps to fix bug for //cdn.com/... urls w
chrome-rendering-worker	crw: updated deps to fix bug for //cd
create	create: updated app pa
docker-compose-testkit	dct: accid
dom-capture	
dom-snapshot	
e2e	
	//cdn.com/... urls which
	o in documentation
	updated debug to v4
	gpt: updated debug to v4
	hc: added TS JSDocs and a readme
gress	kd: support ingress and baseDomain
kdeploy	kd: reverted the reverse fix, which didnt change anything
kubernetes-commons	kc: stop printing all the workers names to the log
kubernetes-testkit	kt: updated dependencies and added rbac disabling sup
loggly-commons	lc: oops. consoleLogFunction should not have been there
loggly-pino	lp: when printing to console, dont break JSON output

We currently have 75 packages

Why? Why One Repo?

- Remember many small packages?
- Remember frictionless development?

Demo: Creating a Package is Really Easy

Packages are...

- Written separately
- Tested separately
- Published Separately

Features spanning two packages

- Write, test, publish code in package B
- \``npm update`\` in package A
- Write, test in package A

Or...

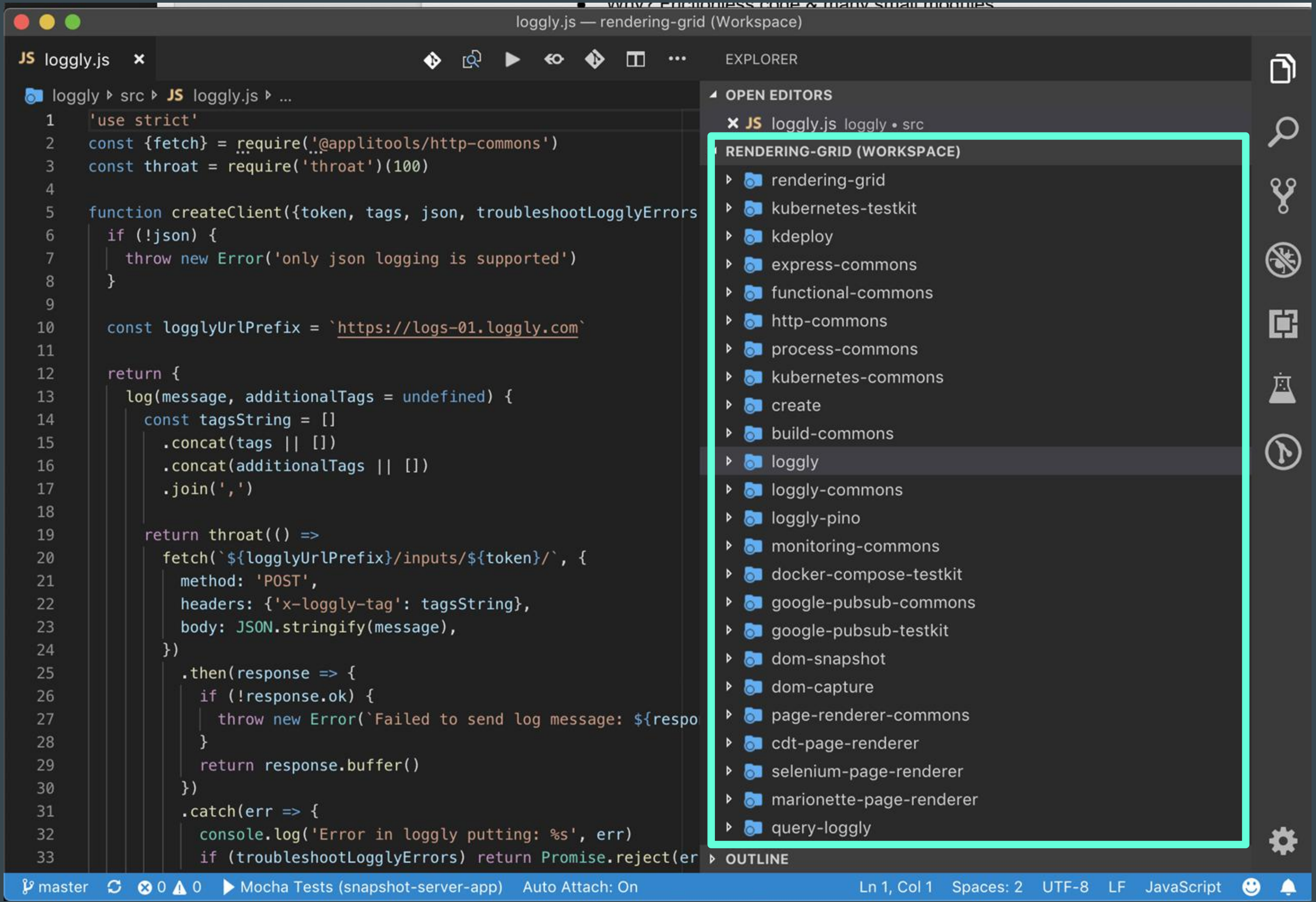
- Use \``npm link`\`
- And *then* publish both packages

NPM is our bridge between packages

- We will never EVER use `require('.../package-a')`.

- semver-major supports changes that upstream packages cannot use

Two-package development is not common



loggly.js — rendering-grid (Workspace)

JS loggly.js ×

loggly › src › JS loggly.js › ...

```
'use strict'
const {fetch} = require('@applitools/http-commons')
const throat = require('throat')(100)

function createClient({token, tags, json, troubleshootLogglyErrors
  if (!json) {
    throw new Error('only json logging is supported')
  }

  const logglyUrlPrefix = `https://logs-01.loggly.com`

  return {
    log(message, additionalTags = undefined) {
      const tagsString = []
        .concat(tags || [])
        .concat(additionalTags || [])
        .join(',')

      return throat(() =>
        fetch(`${logglyUrlPrefix}/inputs/${token}/`, {
          method: 'POST',
          headers: {'x-loggly-tag': tagsString},
          body: JSON.stringify(message),
        })
          .then(response => {
            if (!response.ok) {
              throw new Error(`Failed to send log message: ${respo
            }
            return response.buffer()
          })
          .catch(err => {
            console.log('Error in loggly putting: %s', err)
            if (troubleshootLogglyErrors) return Promise.reject(er
```

EXPLORER

OPEN EDITORS

× JS loggly.js loggly • src

RENDERING-GRID (WORKSPACE)

- rendering-grid
- kubernetes-testkit
- kdeploy
- express-commons
- functional-commons
- http-commons
- process-commons
- kubernetes-commons
- create
- build-commons
- loggly
- loggly-commons
- loggly-pino
- monitoring-commons
- docker-compose-testkit
- google-pubsub-commons
- google-pubsub-testkit
- dom-snapshot
- dom-capture
- page-renderer-commons
- cdt-page-renderer
- selenium-page-renderer
- marionette-page-renderer
- query-loggly

OUTLINE

master ⊗ 0 ⚠ 0 Mocha Tests (snapshot-server-app) Auto Attach: On Ln 1, Col 1 Spaces: 2 UTF-8 LF JavaScript

VSCode Workspaces

@giltayar

applitools

Advantages of monorepos

Simple to move between one package and another

Simple to maintain many packages

Simple to split packages

Packages tend to be small (because easy to create)

“Common” packages are easy to develop and maintain

Disadvantages

NPM link is bad

CI is a problem

Advantages

Disadvantages

Simple to move between one package and another

Simple to maintain many packages

Simple to split packages

Packages tend to be small (because easy to create)

“Common” packages are easy to develop and maintain

Frictionless Development

NPM link is bad

CI is a problem

Testing
Loosely-coupled small packages
Monorepo
Uniform Packages
Frictionless Development

Uniform Packages

All happy families are alike;
each unhappy family is unhappy in its own way

All happy families are alike;
each unhappy family is unhappy in its own way
-- Tolstoy, “Anna Karenina”

All happy **packages** are alike;
each unhappy **package** is unhappy in its own way
-- Gil Tayar, “Applitools”

All our packages are happy!

- They are uniform in **the way we build, test, and publish them.**

- They are the same in **their folder structure**
  - but that's optional

Build Uniformity

What is a Package?

(in our monorepo)

A package is source code that can generate artifacts, which are used by other packages or in production

Two Examples of Packages and their Artifacts

- ❖ Library package
  - ➢ Artifact: **an npm package** in the repository
- ❖ Microservice package
  - ➢ Artifact: **a Docker image** to be used in production
  - ➢ Artifact: **configuration values** to be used in production
  - ➢ Artifact: **an npm package** in the repository

A package in our monorepo is an npm package

- `npm install`
  - Bring in the dependent artifacts, and any artifact needed to build this one
- `npm update`
  - Update dependencies to latest (without breaking backward compatibility)
- `npm run build`
  - Build the artifact (optional for JavaScript)
- `npm test`
  - Test the artifact to ensure that it can be published
- `npm publish`
  - Publish the artifact(s)
- `npm run deploy`
  - Deploy the artifact to production (only for microservices)

Note: we can use `postpublish/postinstall` to do more non-npm stuff.

Example from a Library Package

```
"scripts": {
  "build": "#", // Yay JavaScript!
  "test": "npm run eslint && npm run test:mocha-parallel",
  "test:mocha": "mocha 'test/unit/*.test.js'
                 'test/it/*.test.js' 'test/e2e/*.test.js'",
  "test:mocha-parallel": "mocha-parallel-tests 'test/unit/*.test.js'
                 'test/it/*.test.js' 'test/e2e/*.test.js'",
  "eslint": "eslint '**/*.js'"
},
```

Example from a Microservice Package

```
"scripts": {
  "build": "npm run build:docker",
  "build:docker": "docker build -t applitools/chrome-rendering-worker
             --build-arg NPM_FILE=`cat ~/.npmrc` .",
  "test": "npm run eslint && npm run test:mocha",
  "postpublish": "npm run publish:docker",
  "publish:docker": "docker tag applitools/chrome-rendering-worker
             applitools/chrome-rendering-worker:${npm_package_version} &&
             docker push applitools/chrome-rendering-worker:${npm_package_version}
         &&
             docker push applitools/chrome-rendering-worker:latest",
  "deploy": "kdeploy deploy chrome-rendering-worker ${npm_package_version}"
}
```

Advantages

Developers can start developing (and deploying) all packages without understanding anything.

CI (when we have it) will be easier

Disadvantages

Sometimes it's like fitting a square into a circle.

Advantages

Disadvantages

Developers can start developing (and deploying) all packages without understanding anything.

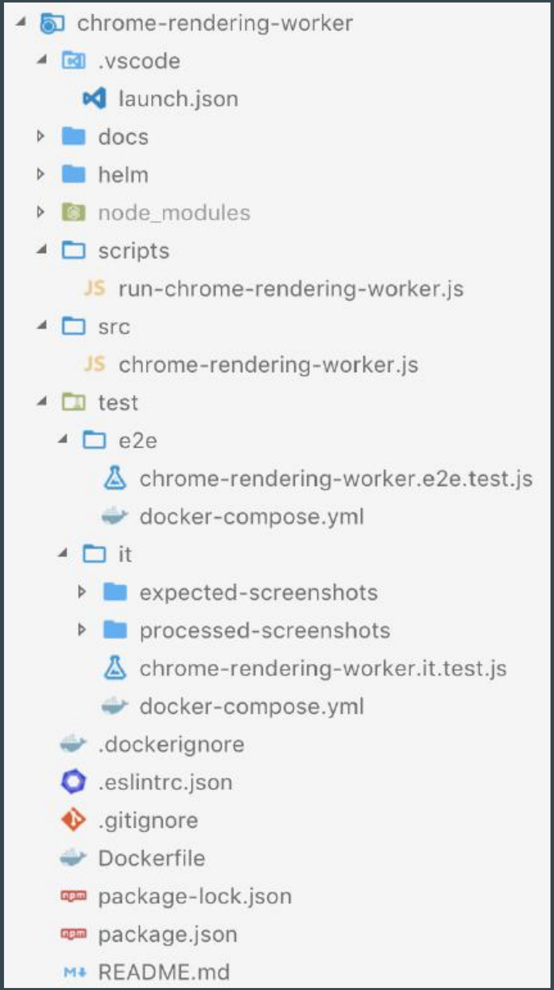
CI (when we have it) will be easier

Frictionless Development

Sometimes it's like fitting a square into a circle.

Source Code Uniformity

Folder Structure

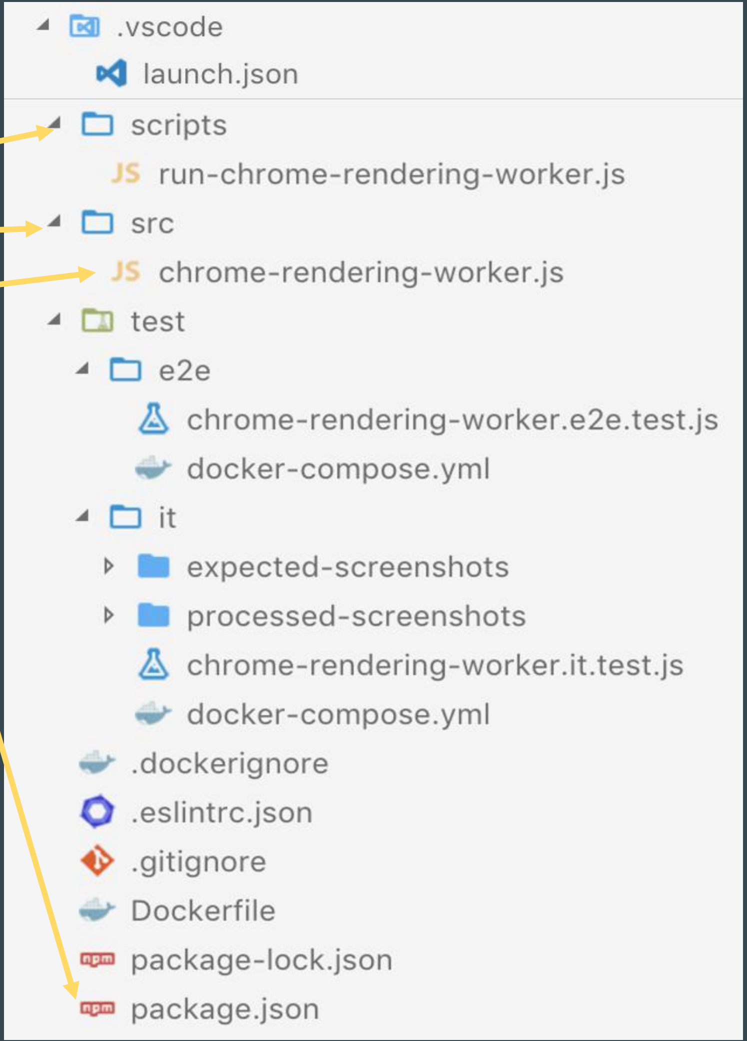


```
chrome-rendering-worker
  .vscode
    launch.json
  docs
  helm
  node_modules
  scripts
    run-chrome-rendering-worker.js
  src
    chrome-rendering-worker.js
  test
    e2e
      chrome-rendering-worker.e2e.test.js
      docker-compose.yml
    it
      expected-screenshots
      processed-screenshots
      chrome-rendering-worker.it.test.js
      docker-compose.yml
  .dockerignore
  .eslintrc.json
  .gitignore
  Dockerfile
  package-lock.json
  package.json
  README.md
```

Source Code Structure

`src` and `scripts`

- entrypoint has same name as package.
- All source is inside there.
- Whitelist the npm published files

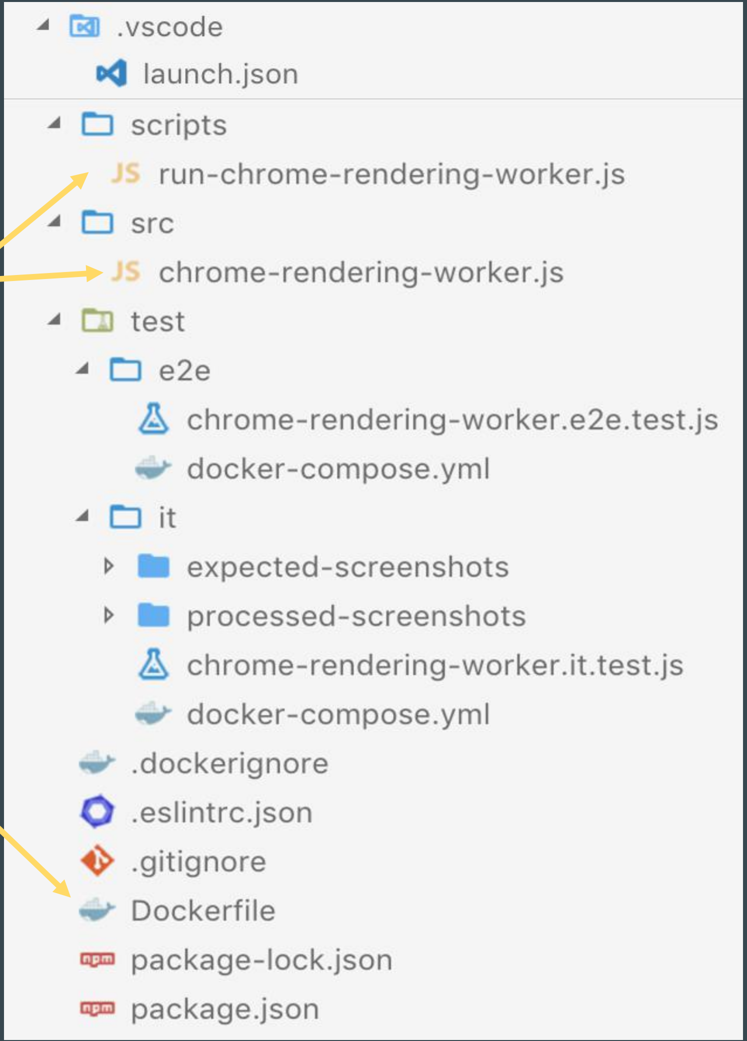


```
.vscode
    launch.json
scripts
    run-chrome-rendering-worker.js
src
    chrome-rendering-worker.js
test
    e2e
        chrome-rendering-worker.e2e.test.js
        docker-compose.yml
    it
        expected-screenshots
        processed-screenshots
        chrome-rendering-worker.it.test.js
        docker-compose.yml
.dockerignore
.eslintrc.json
.gitignore
Dockerfile
package-lock.json
package.json
```

Source Code Structure

Microservice src and script

- `src` exports a web app
- `scripts` runs the web app. This is what the `Dockerfile` runs.

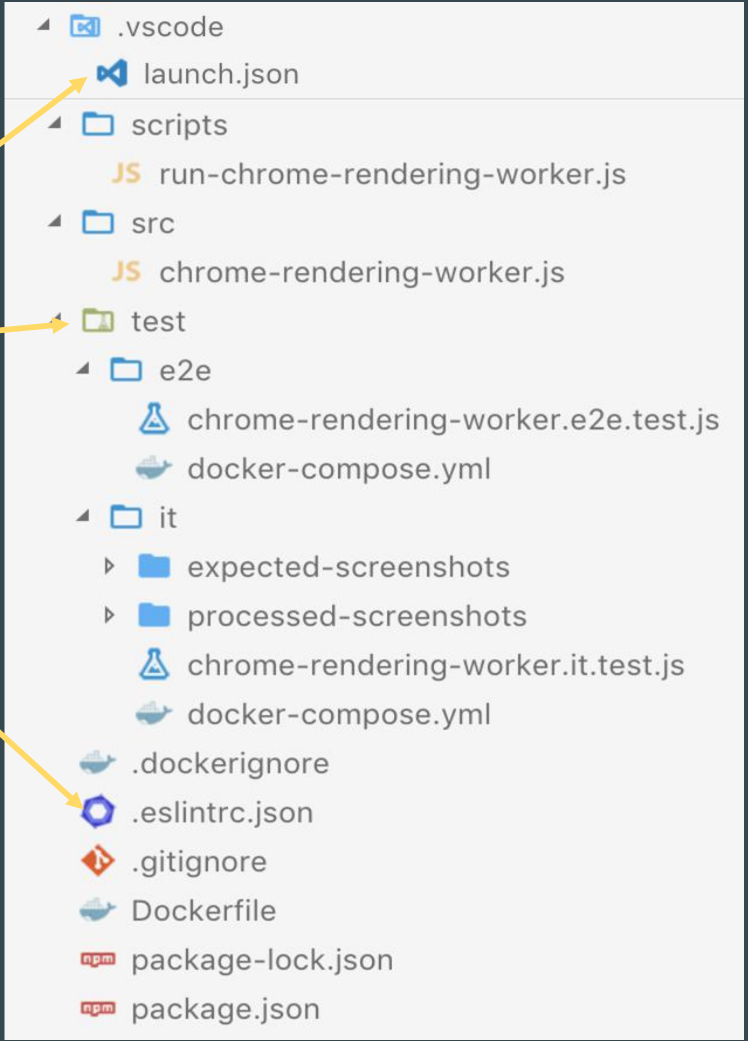


```
.vscode
    launch.json
scripts
    run-chrome-rendering-worker.js
src
    chrome-rendering-worker.js
test
    e2e
        chrome-rendering-worker.e2e.test.js
        docker-compose.yml
    it
        expected-screenshots
        processed-screenshots
        chrome-rendering-worker.it.test.js
        docker-compose.yml
.dockerignore
.eslintrc.json
.gitignore
Dockerfile
package-lock.json
package.json
```

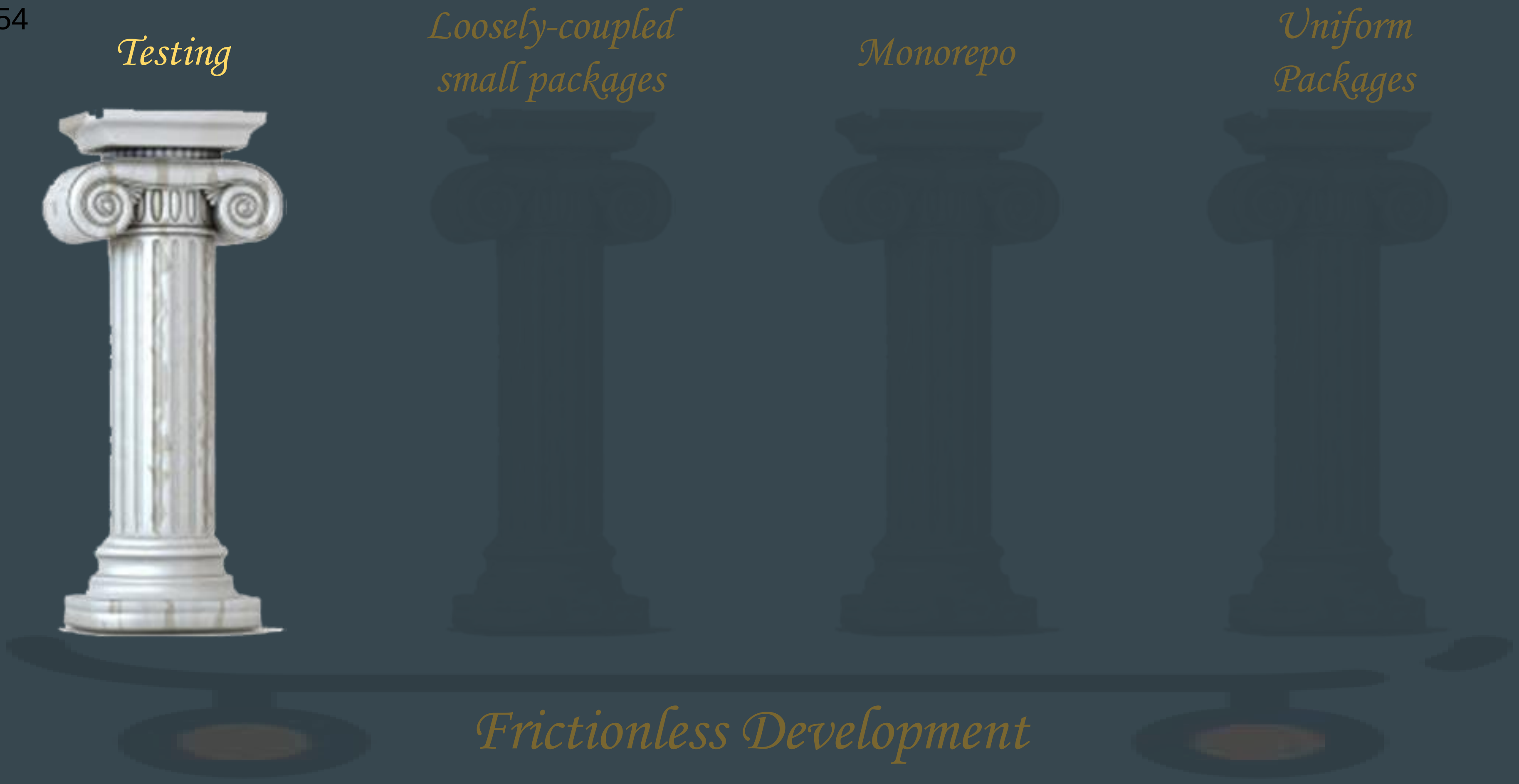
Source Code Structure

Support files

- .vscode for easy debugging
- test folder each package
- Eslint and prettier


```
.vscode
  launch.json
scripts
  run-chrome-rendering-worker.js
src
  chrome-rendering-worker.js
test
  e2e
    chrome-rendering-worker.e2e.test.js
    docker-compose.yml
  it
    expected-screenshots
    processed-screenshots
    chrome-rendering-worker.it.test.js
    docker-compose.yml
.dockerignore
.eslintrc.json
.gitignore
Dockerfile
package-lock.json
package.json
```

applitools


Testing
Loosely-coupled small packages
Monorepo
Uniform Packages
Frictionless Development

applitools

Testing

Rule #1:
Never EVER EVER run your code locally
(except in a test)

… And You Don’t Have To Test

- Want to write your feature and immediately deploy it?
  - Sure!
  - 😀
- Otherwise, write a test

Testing is the epitome of Frictionless Development

a person or thing that is a perfect example of a particular quality or type

Testing is the epitome of Frictionless Development

It is the main reason our velocity is high

It is one of the two reasons we don't *need* TypeScript

And I frankly don’t know today how to go to production without tests

But enough crap
Let’s see how we test our microservices

Unit Testing

- Test functions or modules that are mainly algorithmic, with little to no I/O.
  - We mostly don’t write classes
- Usually stateless functions
- No need for mocking
  - or the mock is so simple we don’t use a mocking library
- We don’t have a lot of these tests

Testing isBlankImage

```
const isBlankImage = require('../../src/is-blank-image')

describe('isBlankImage', function() {
 it('should return true on a blank image', async () => {
   const newImage = PNG.createImage({
     filterType: 4,
   })
   const blankImage = await p(newImage.parse.bind(newImage))(
     await p(fs.readFile)(path.join(__dirname, 'resources/blank-image.png')),
   )

   expect(await isBlankImage(blankImage)).to.be.true
 })
```

Integration Tests

- Test the whole microservice (or large parts thereof)
  - Remember: Microservices are small, so no problem
- If we need a database or the like, we use docker with docker-compose.
- Lots of these tests

```
const app = require('../..')

function setupApp(app) {
 let server

 before(async () => {
   await new Promise((resolve, reject) => {
     server = app({maxNumberOfScreenshots: 50}).
                     listen(err => (err ? reject(err) : resolve()))
   })
 })
 after(done => server.close(done))

 return {
   address: () => `localhost:${server.address().port}`,
 }
```

```
describe('screenshot-webhook-app-testkit it', function() {
 const {address} = setupApp(app)

 it('screenshot count should be correct after accepting a screenshot', async () => {
   const screenshotId = `id-${(Math.random() * 100000) | 0}`
   const countBefore = await countScreenshots({address: address()})

   const response = await fetch(`http://${address()}/accept/screenshotId`, {
     method: 'POST',
     body: Buffer.from('dummy!'),
   })
   expect(response.ok).to.be.true

   expect(await countScreenshots({address: address()})).to.equal(countBefore + 1)
 })
```

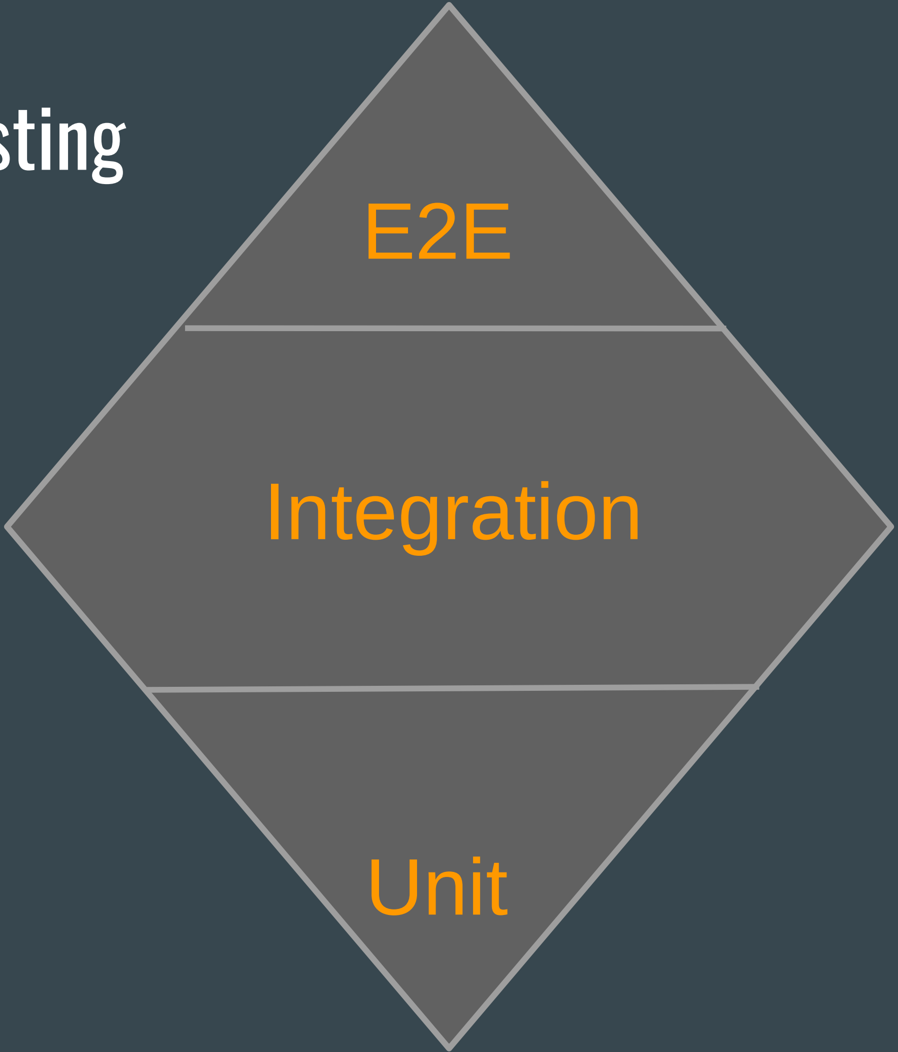
docker-compose.xml

```
services:
 redis:
   image: redis:alpine
   ports:
     - 6379
   command:
     - --requirepass
     - apassword
```

E2E Tests

- Misnamed—doesn’t test *all* the microservices.
  - These are tests for one microservice
- They do a minimal test for the *docker image*,
  - to see that it runs
  - and passes the environment variables correctly to the app.
- Looks just like the integration
  - The `docker-compose.xml` also includes a container for the microservice itself.

The Diamond Of Testing



applitools

How Do I Know I Wrote Enough Tests?

The Shakometer

The `E2E` Package

- Special package
- Deploys all microservices to minikube
- And runs tests on all the system
- Deployment uses same deployment mechanism as for production and same configuration values
- We run it only sometimes

Testing
Loosely-coupled small packages
Monorepo
Uniform Packages
Frictionless Development

applitools

Loosely-Coupled Small Packages

Why

- Each Microservice and library is easy to understand
- Which means that it is easy to test
- Which means that testing is possible
- Which is why we don’t need TypeScript
    - We’ve started exploring TypeScript JSDocs

Frictionless Development

CI/CD

Mini-CI: BTP

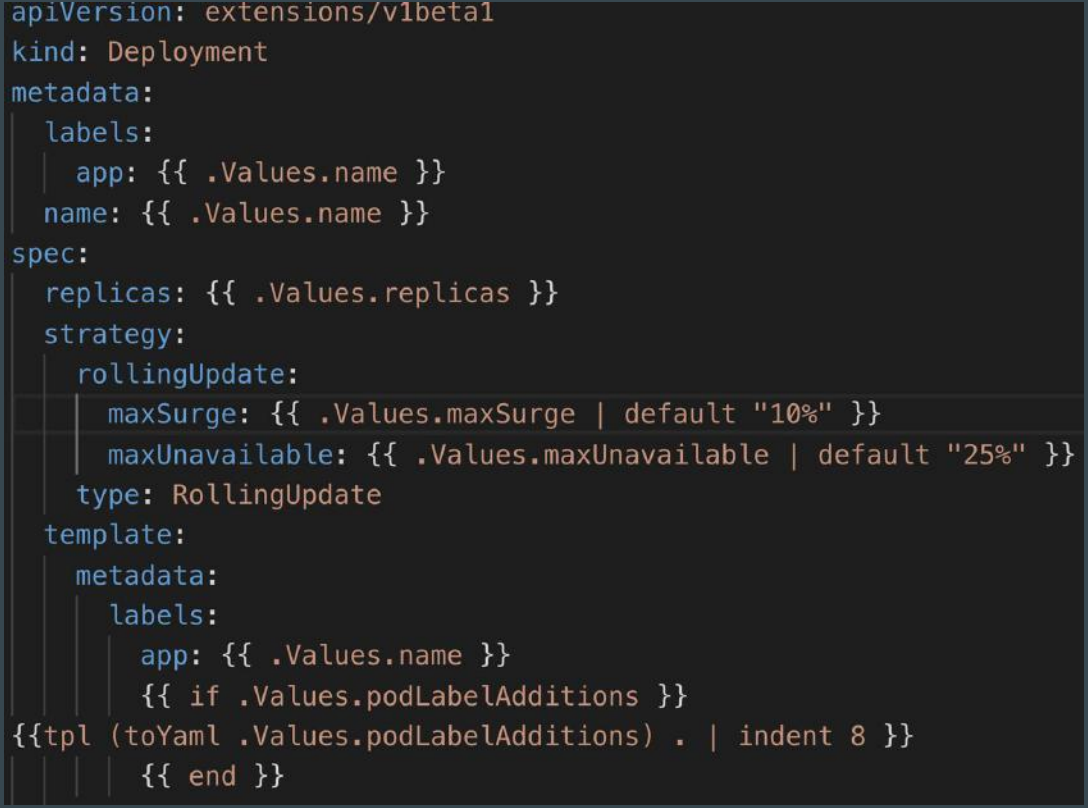
It just runs these steps...

- *Increment version*
- `npm ci`
- `npm update`
- `npm run build`
- `npm test`
- `npm publish`

CD: K8s Makes It *So* Easy

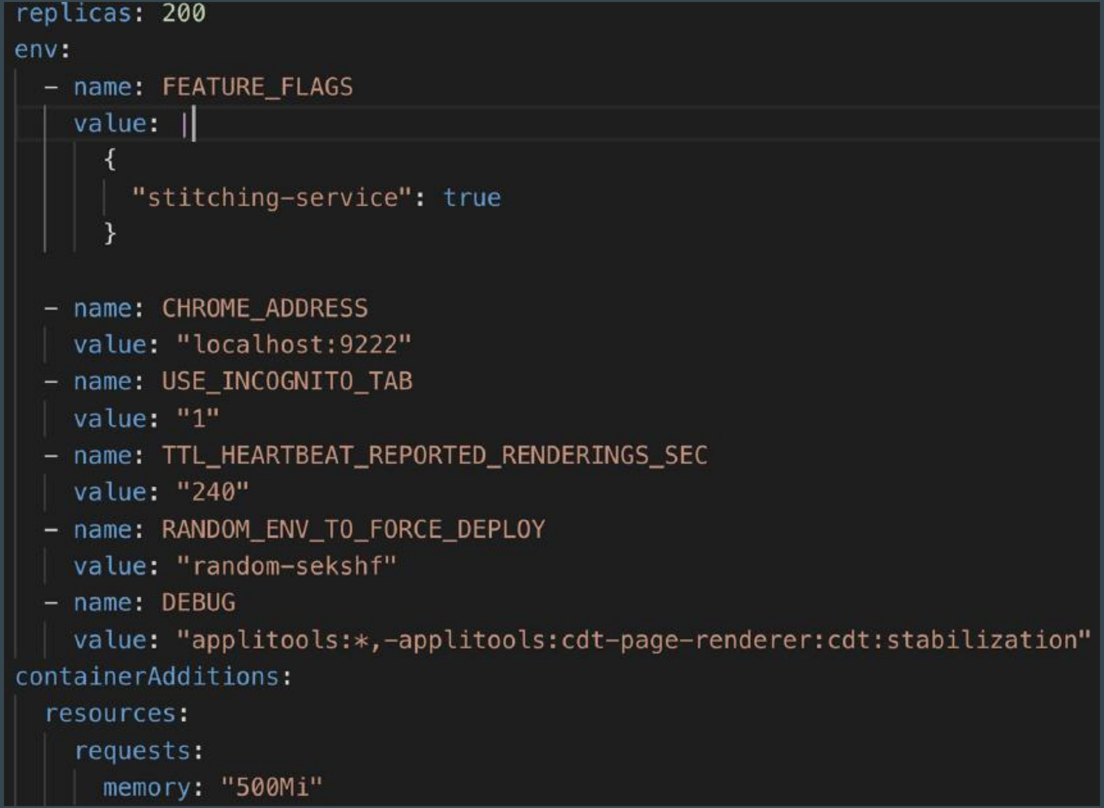
- And yet I've managed to complicate it
  - Prodigious use of over-design
  - Not to mention over-engineered
  - A simple set of yaml files, with some templating, would have sufficed
- YAGNI!
- KISS!

It’s all Kubernetes Template YAMLs (using Helm)



```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: {{ .Values.name }}
  name: {{ .Values.name }}
spec:
  replicas: {{ .Values.replicas }}
  strategy:
    rollingUpdate:
      maxSurge: {{ .Values.maxSurge | default "10%" }}
      maxUnavailable: {{ .Values.maxUnavailable | default "25%" }}
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: {{ .Values.name }}
        {{ if .Values.podLabelAdditions }}
{{tpl (toYaml .Values.podLabelAdditions) . | indent 8 }}
        {{ end }}
```

Which is customized per microservice



```yaml
replicas: 200
env:
  - name: FEATURE_FLAGS
    value: |
      {
        "stitching-service": true
      }

  - name: CHROME_ADDRESS
    value: "localhost:9222"
  - name: USE_INCOGNITO_TAB
    value: "1"
  - name: TTL_HEARTBEAT_REPORTED_RENDERINGS_SEC
    value: "240"
  - name: RANDOM_ENV_TO_FORCE_DEPLOY
    value: "random-sekshf"
  - name: DEBUG
    value: "applitools:*,-applitools:cdt-page-renderer:cdt:stabilization"
containerAdditions:
  resources:
    requests:
      memory: "500Mi"
```

```
npm run deploy
```

In Summary

Testing
Loosely-coupled small packages
Monorepo
Uniform Packages
Frictionless Development

I’ll Leave You With These Three Things:

yagni

Small is Beautiful

And above all: KISS

Testing
Loosely-coupled small packages
Monorepo
Uniform Packages
Frictionless Development

Thank You!