????

Matthew Paletta

Docker ???

Matthew Paletta

Docker + gRPC ??

Matthew Paletta

Docker + gRPC + Redis ?

Matthew Paletta

Docker + gRPC + Redis + DevOps

Matthew Paletta

“Everything is figureoutable”

Disclaimer

Buzzwords - feel free to ask me!

Ground Zero

Ground Zero

- This is our starting spot, based on what we’ve seen in the labs
-

Ground Zero

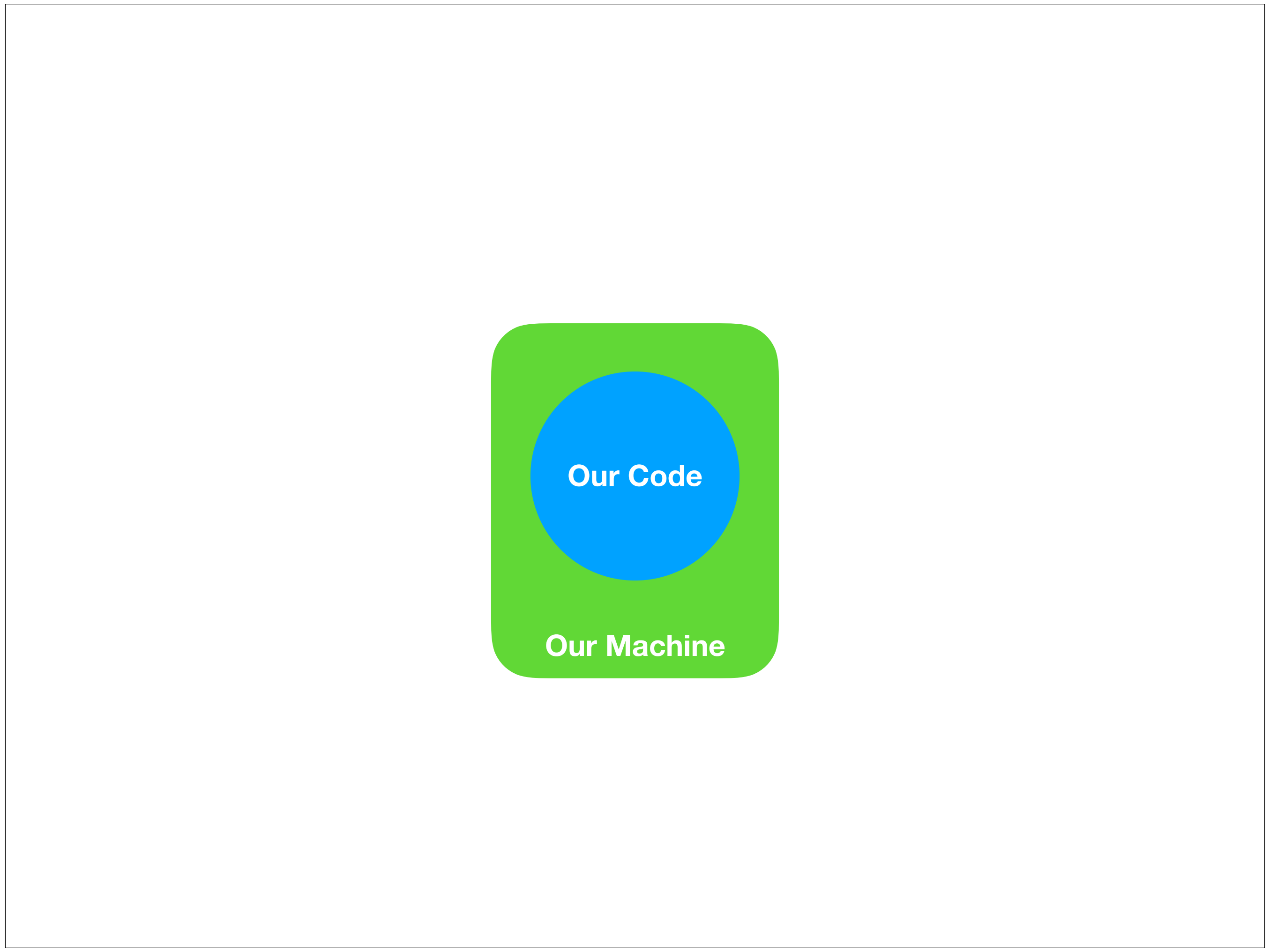
- One process

Ground Zero

- One process
- One machine

Ground Zero

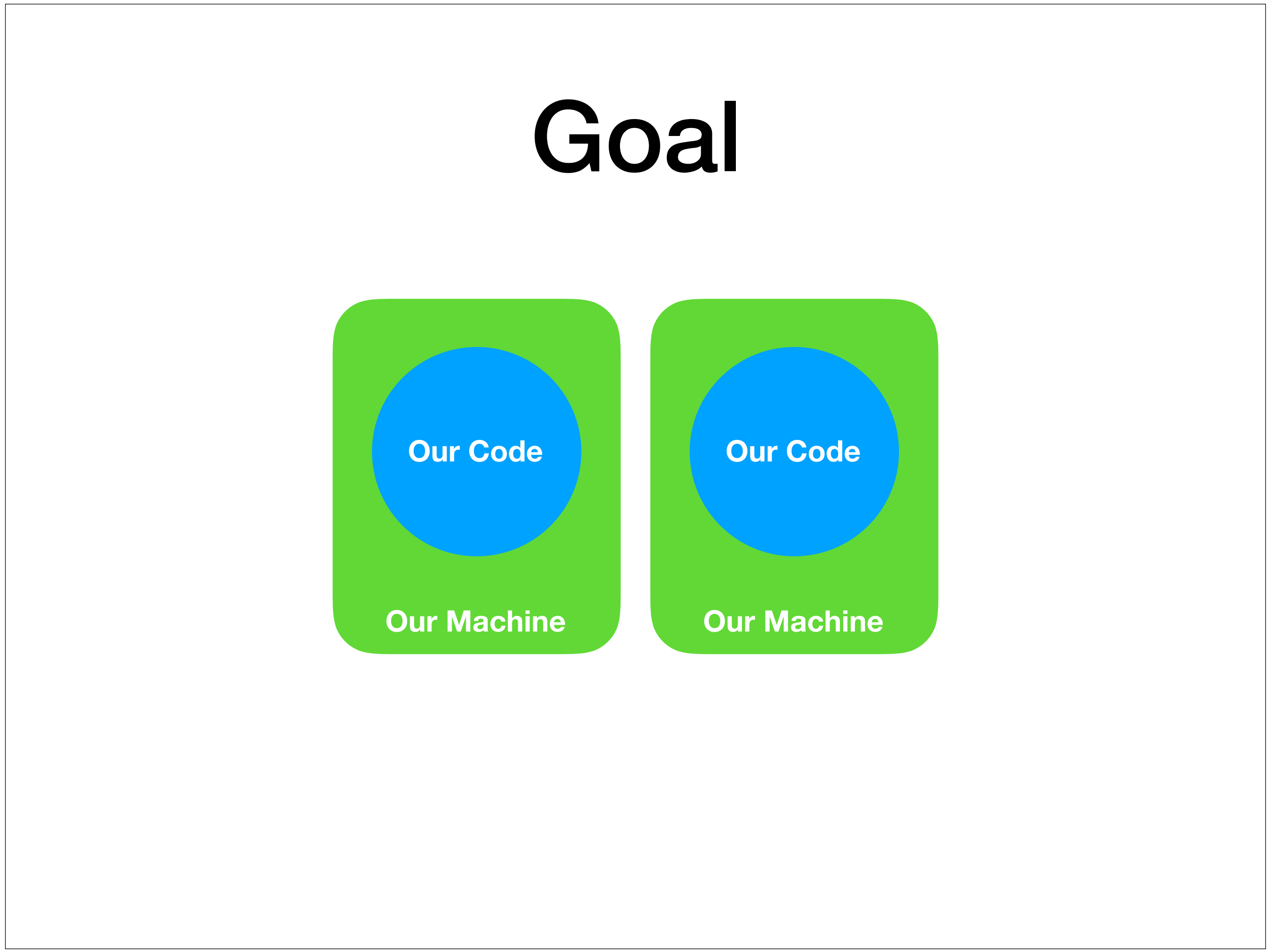
- One process
- One machine
- Multi-threaded?



Our Code

Our Machine

- Here’s our world view. We only think about our code



Goal

Our Code

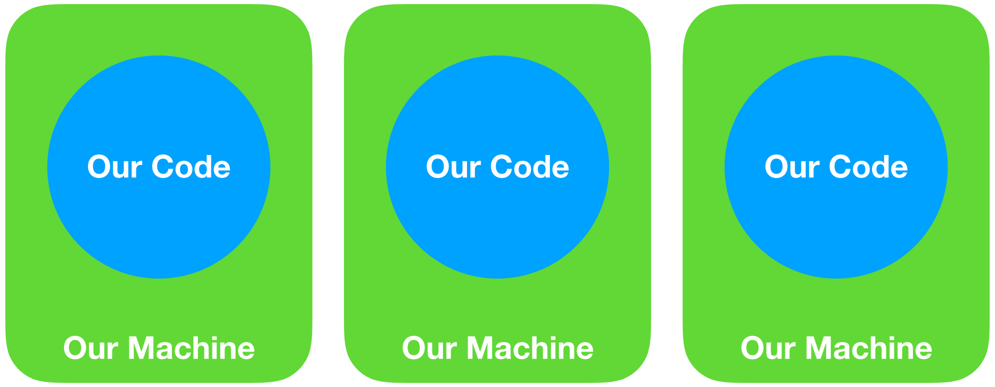
Our Machine

Our Code

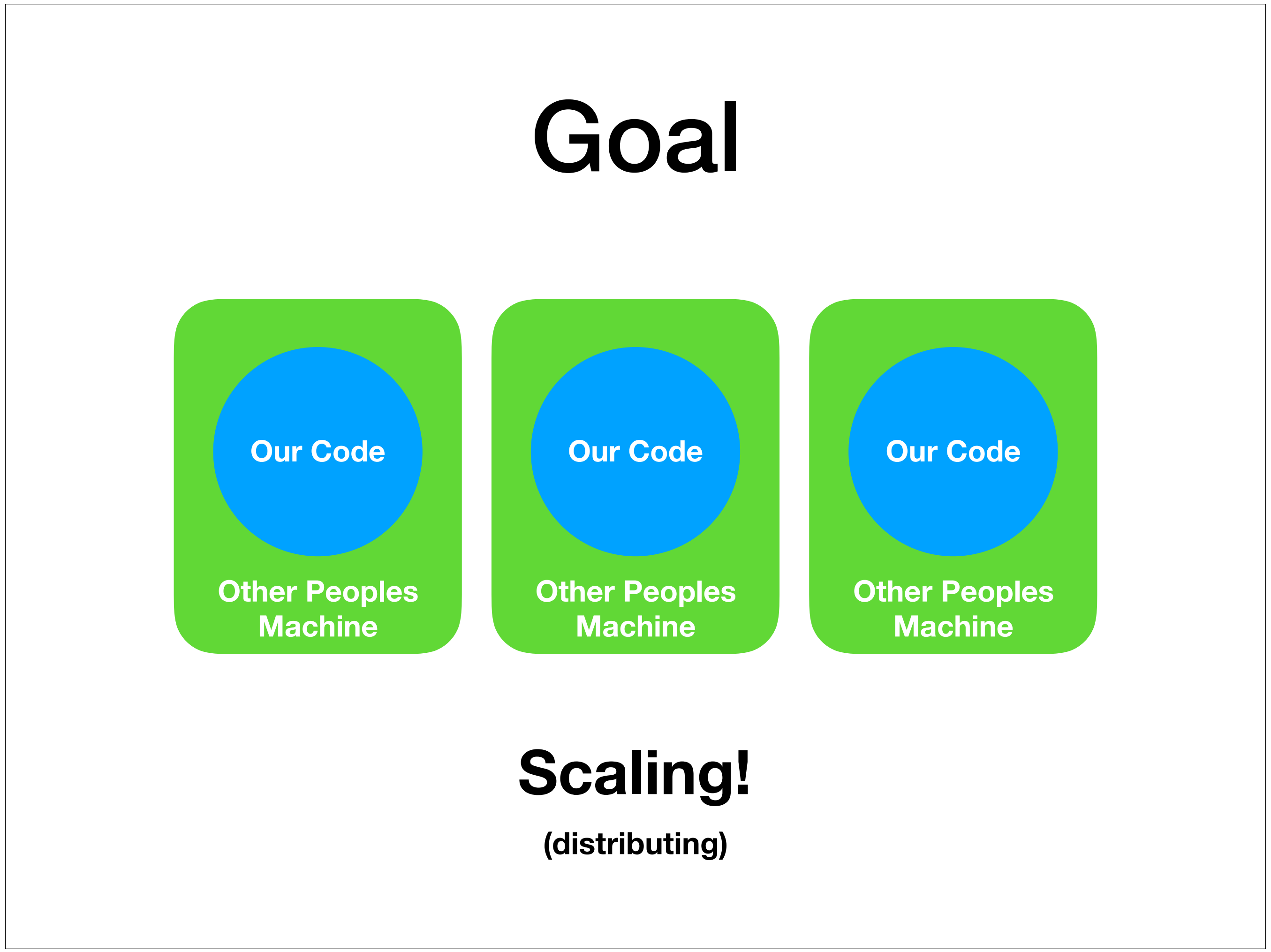
Our Machine

- We'd like to put it on 2 machines!
- Maybe this is a web server for our website, and we want to handle more requests. We want to handle more requests, or have it fault-tolerant, so our website doesn't go down.

Goal



- And once we do 2, we want 3



Goal

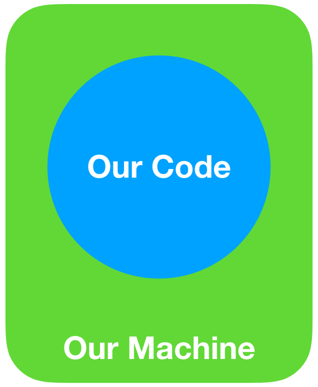
Scaling!

(distributing)

- And when we do that, we want to use other peoples machines (cloud)
- And we want to scale and distribute to increase the amount of data we can process

Containers

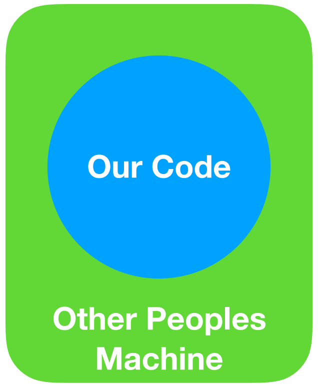
- What is this container thing -



Our Code

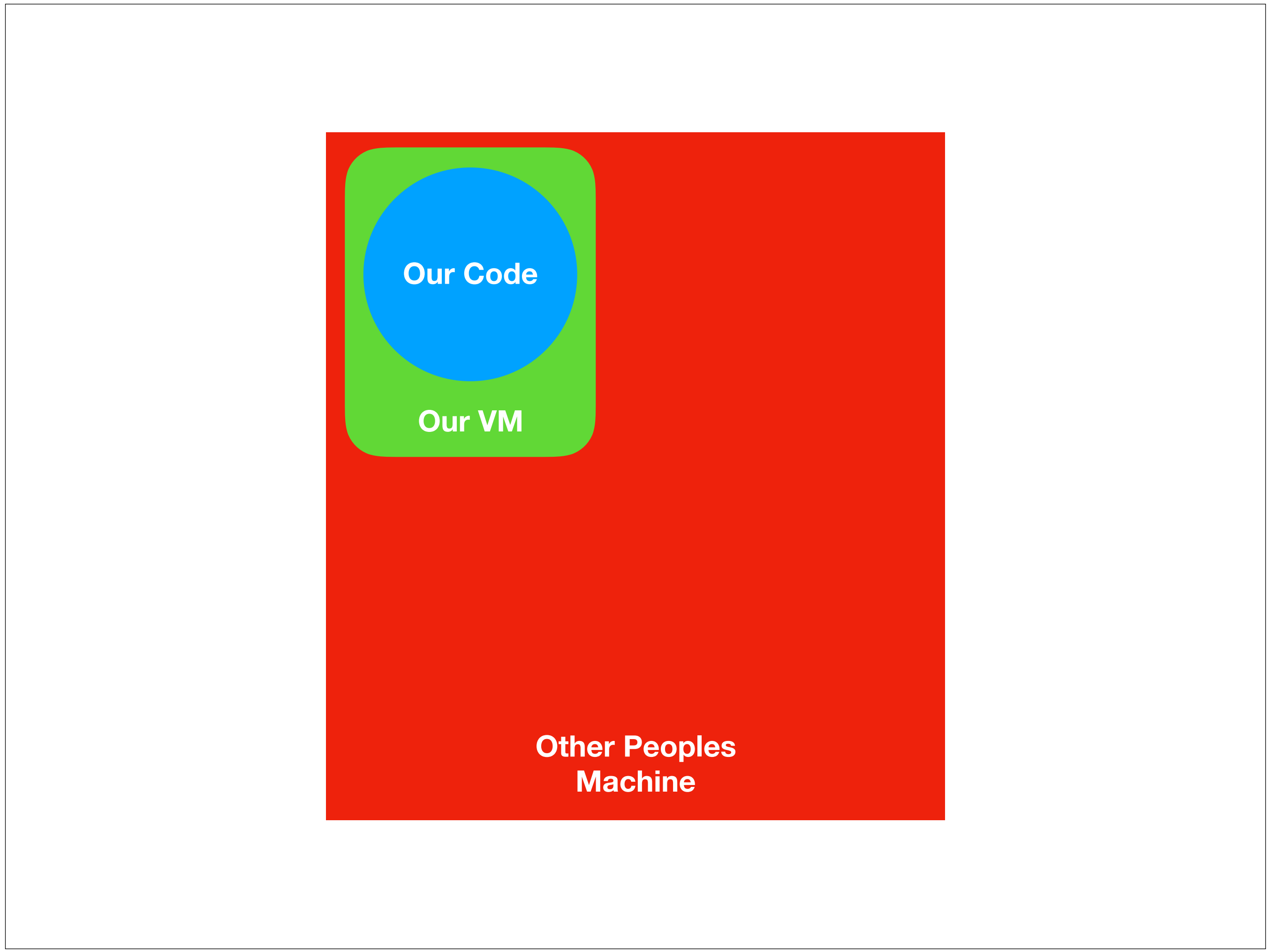
Our Machine

- Lets start again with our code running on our machine



Our Code

Other Peoples Machine

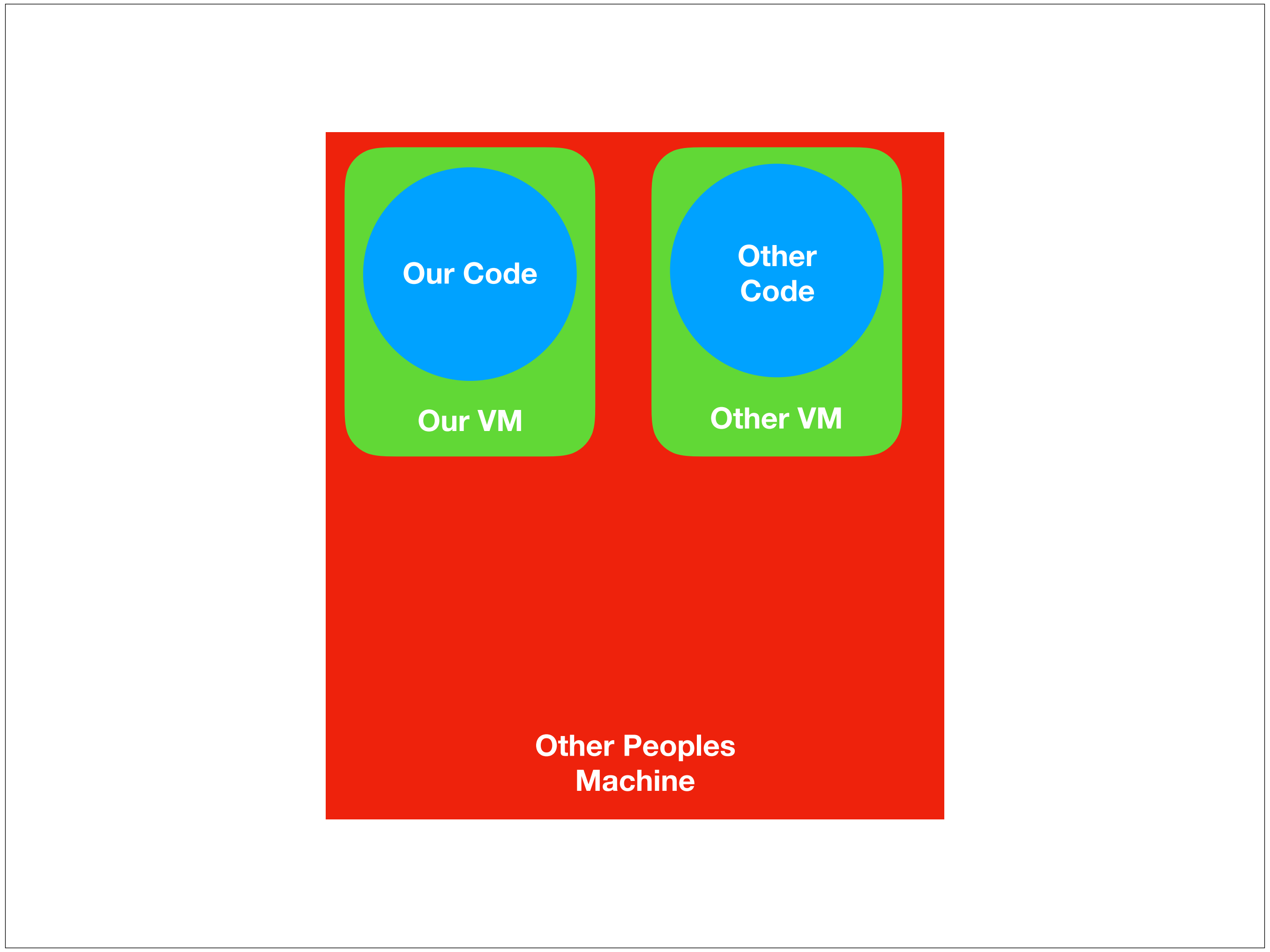


Our Code

Our VM

Other Peoples
Machine

Let’s say we share that machine with other people
This is what it actually looks like



Our Code

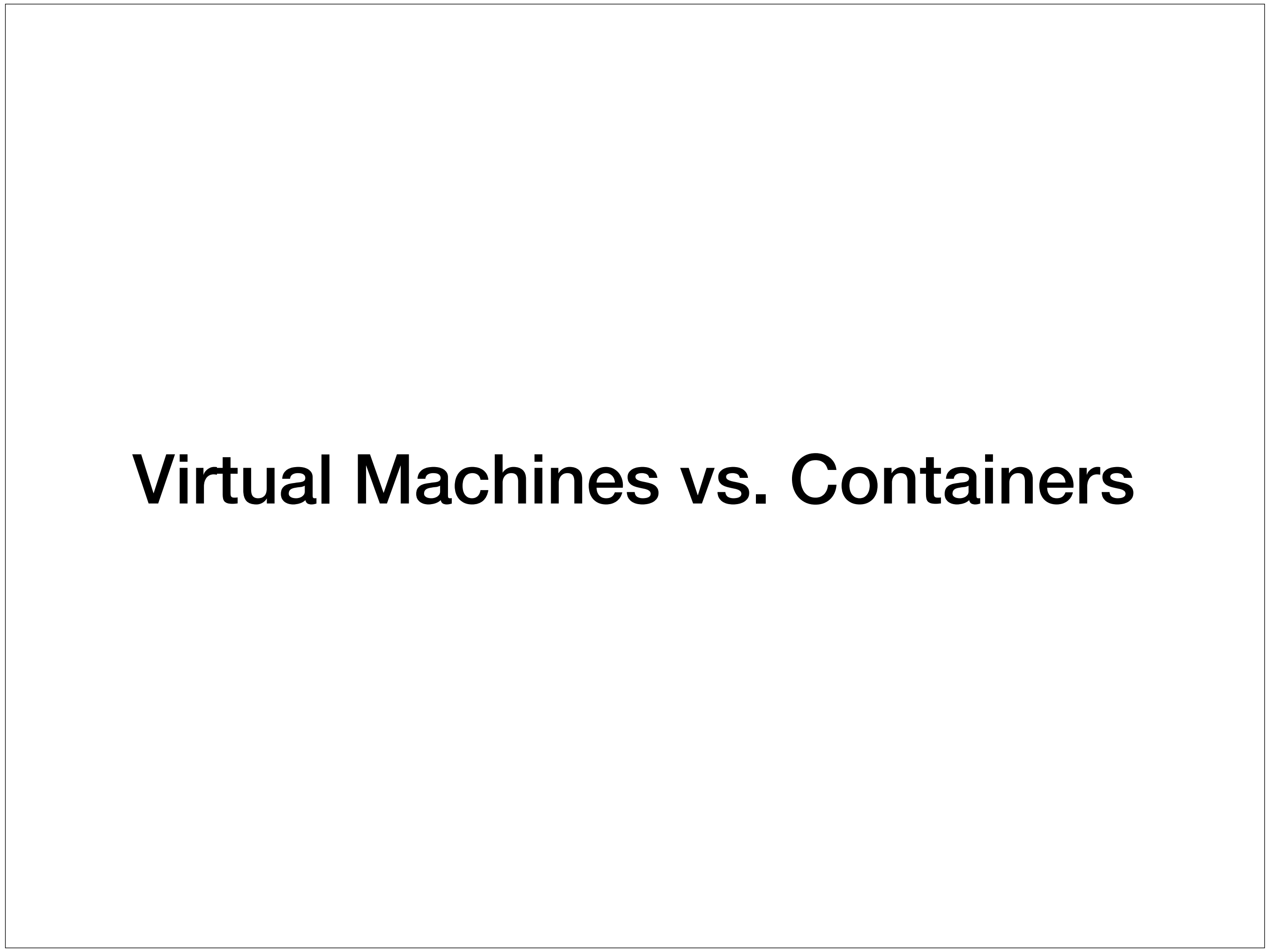
Our VM

Other Code

Other VM

Other Peoples Machine

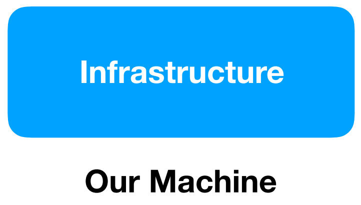
- Probably there’s other VMs running on the machine too.



Virtual Machines vs. Containers

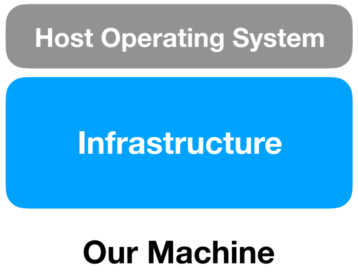
- We could look at this differently

Our Machine



Infrastructure

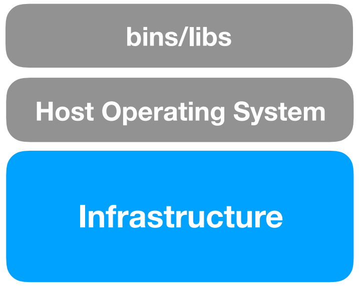
Our Machine



Host Operating System

Infrastructure

Our Machine

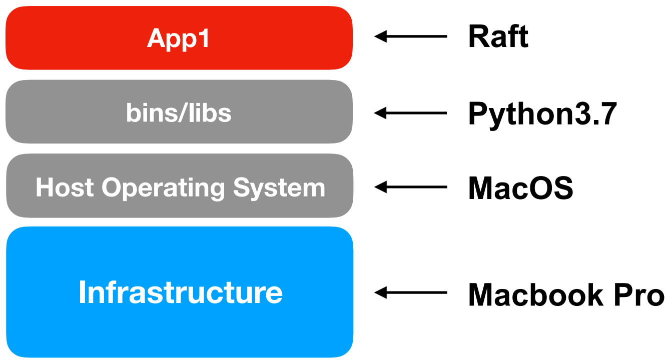


bins/libs

Host Operating System

Infrastructure

Our Machine



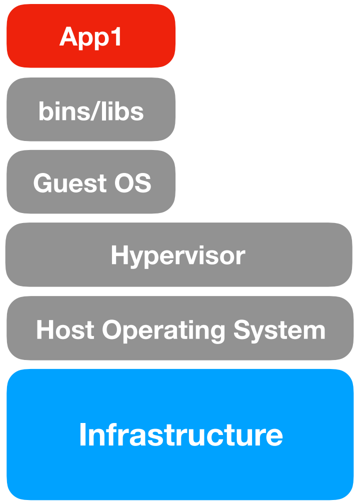
App1 ← Raft

bins/libs ← Python3.7

Host Operating System ← MacOS

Infrastructure ← Macbook Pro

Our Machine



App1

bins/libs

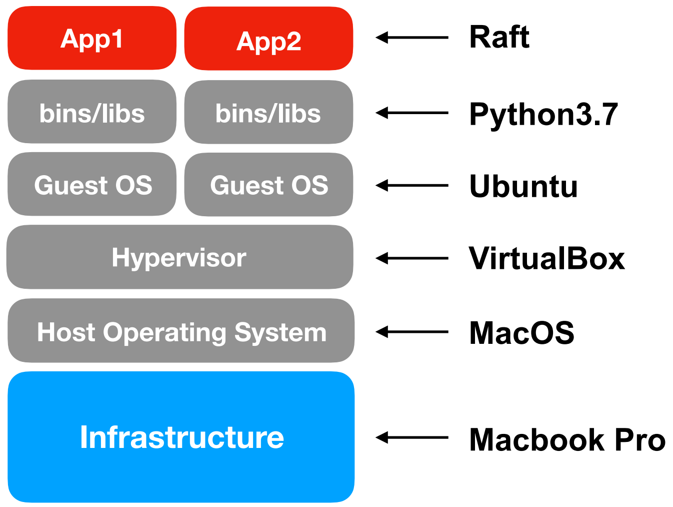
Guest OS

Hypervisor

Host Operating System

Infrastructure

Virtual Machine



App1 App2 ← Raft

bins/libs bins/libs ← Python3.7

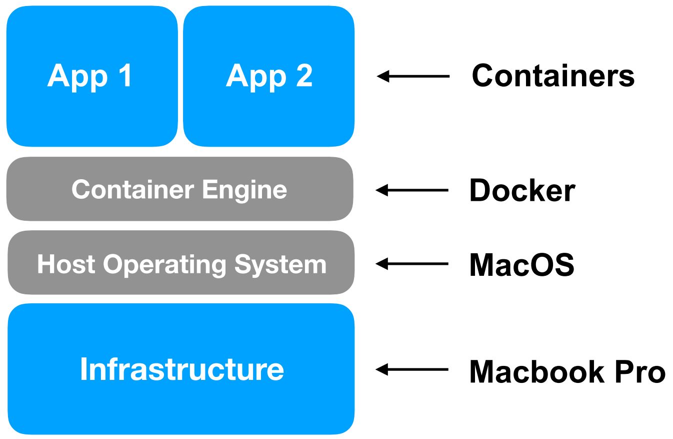
Guest OS Guest OS ← Ubuntu

Hypervisor ← VirtualBox

Host Operating System ← MacOS

Infrastructure ← Macbook Pro

Virtual Machine



App 1

App 2

Container Engine

Host Operating System

Infrastructure

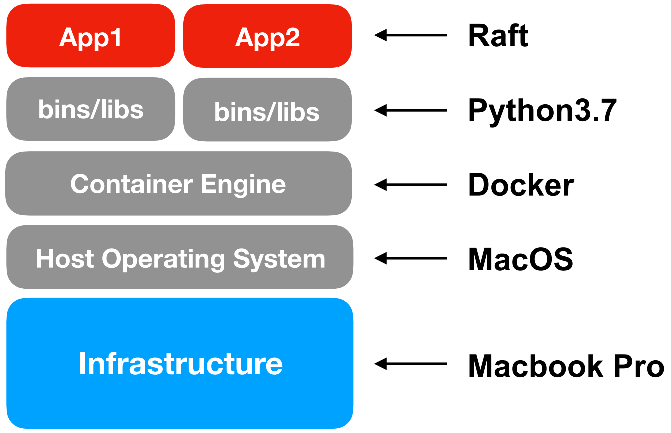
Containers

Docker

MacOS

Macbook Pro

Containers



App1 App2 ← Raft

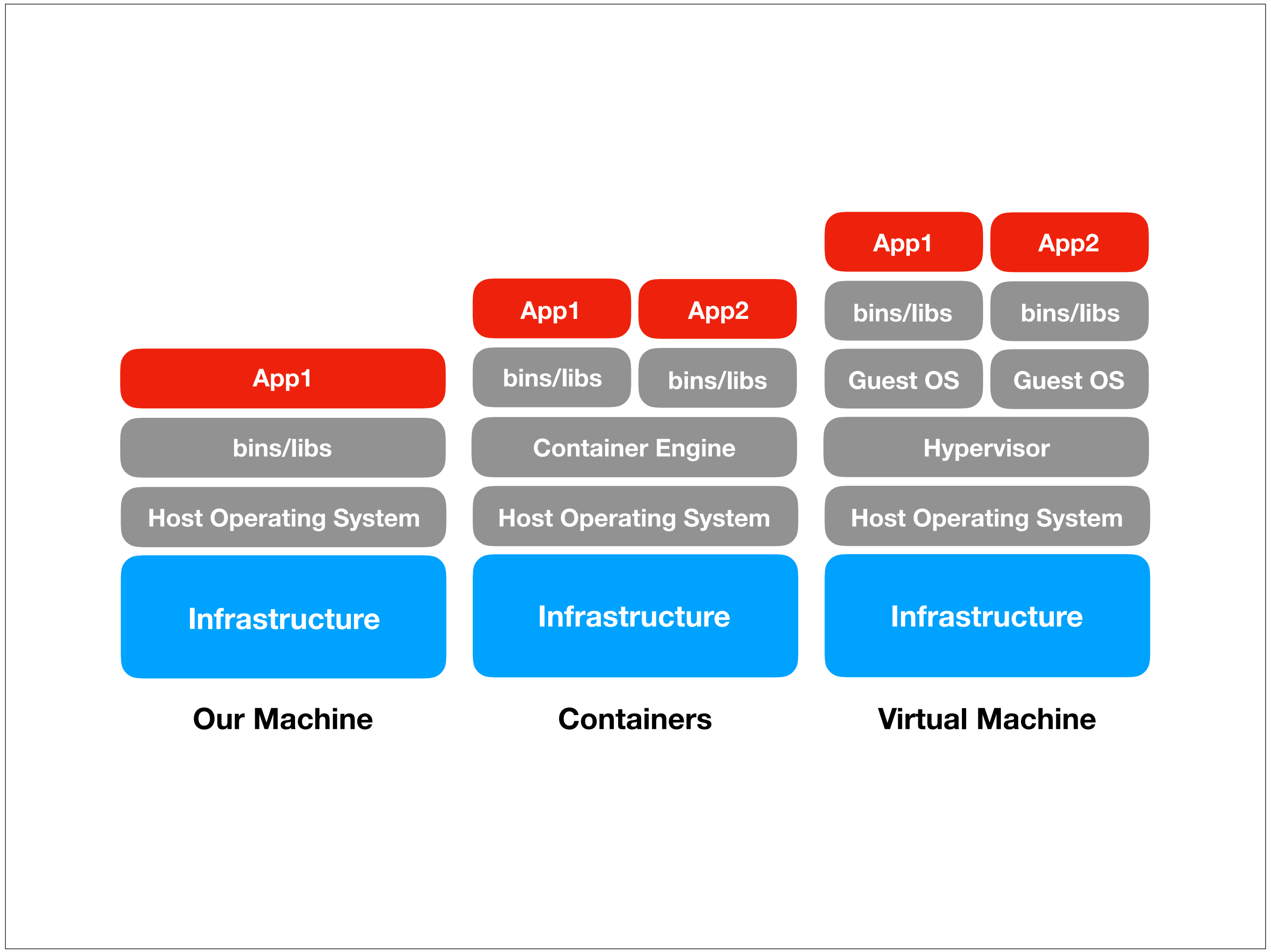
bins/libs bins/libs ← Python3.7

Container Engine ← Docker

Host Operating System ← MacOS

Infrastructure ← Macbook Pro

Containers



App1
bins/libs
Host Operating System
Infrastructure
Our Machine

App1 | App2
bins/libs | bins/libs
Container Engine
Host Operating System
Infrastructure
Containers

App1 | App2
bins/libs | bins/libs
Guest OS | Guest OS
Hypervisor
Host Operating System
Infrastructure
Virtual Machine

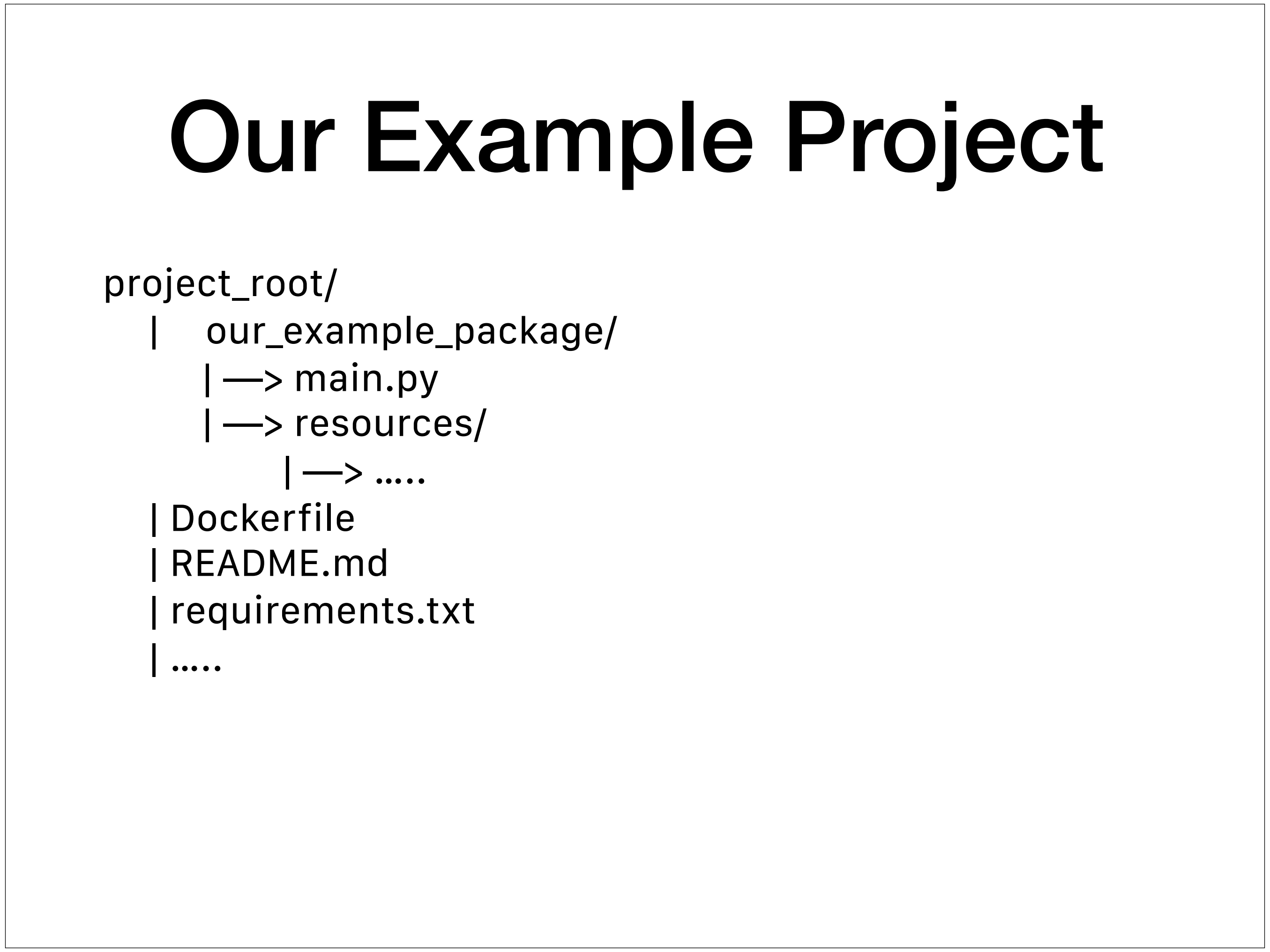
- Containers give us better performance - close to bare-metal, but each app can have each bins/libs
- Containers are distributable environment
- Ask why containers

Containers

- Better performance compared to VMs
- Only consume the resources they are actually using (disk & memory)
- Maintains per-app isolation & libraries
- Easy to distribute and scale with other developers or cloud deployments
- Faster startup time - don’t have to wait for the entire OS to boot!
- Infrastructure as code - (devops topic)

Docker

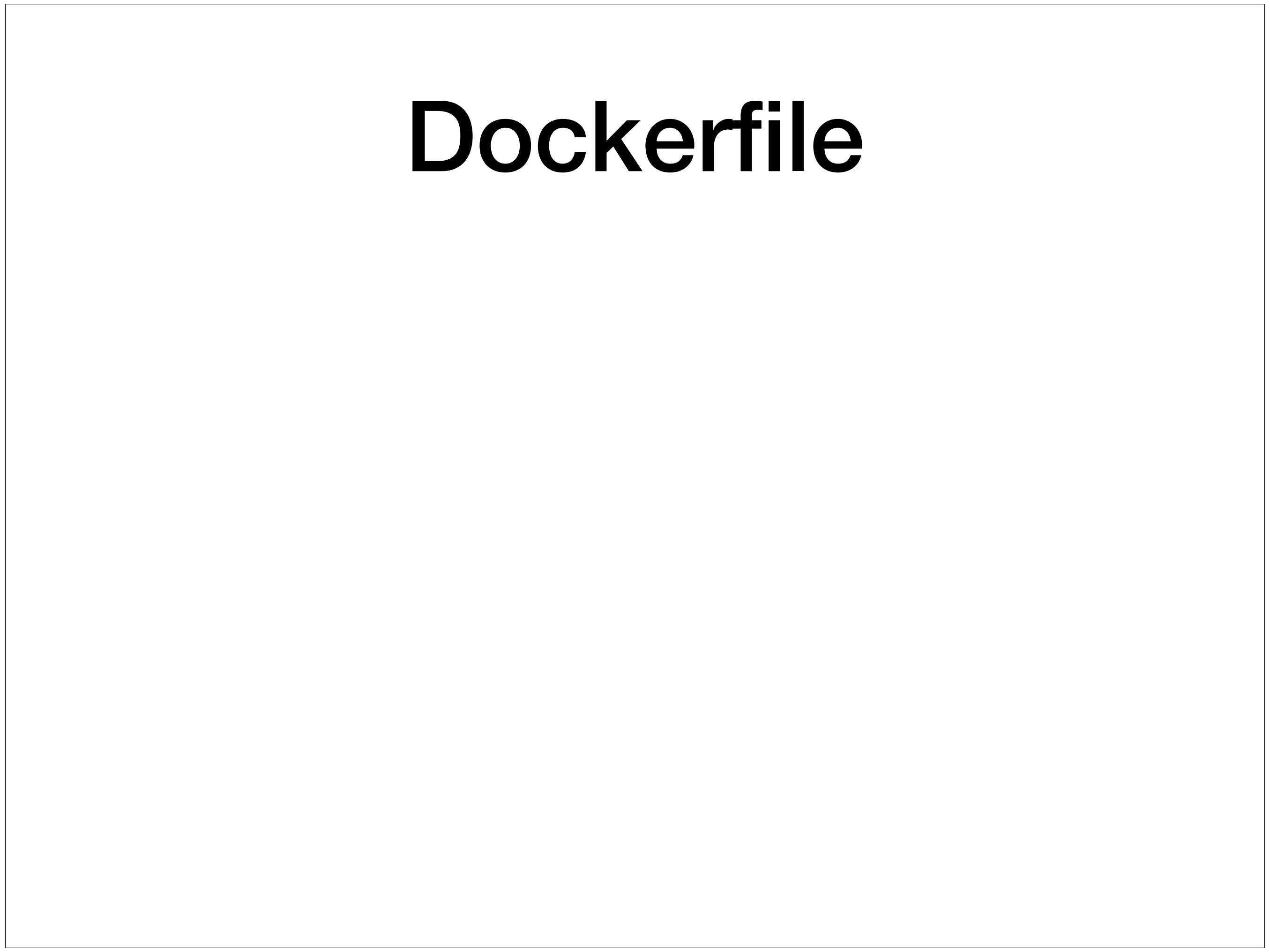
Dockerfiles



Our Example Project

```
project_root/
   |    our_example_package/
        | —> main.py
        | —> resources/
               | —> …..
   | Dockerfile
   | README.md
   | requirements.txt
   | …..
```

- This is our example project structure
- Nothing too special going on here



Dockerfile

- Here’s the Dockerfile (stepped through)
- Introduce a ‘layer’
- There are other ‘FROM’ images - we’ll come back
- Ask if there’s any questions about any of the lines

Dockerfile

```
FROM python:3.7
```

Dockerfile

```
FROM python:3.7

ADD requirements.txt /requirements.txt
```

Dockerfile

```
FROM python:3.7

ADD requirements.txt /requirements.txt

RUN pip3 install -r requirements.txt
```

Dockerfile

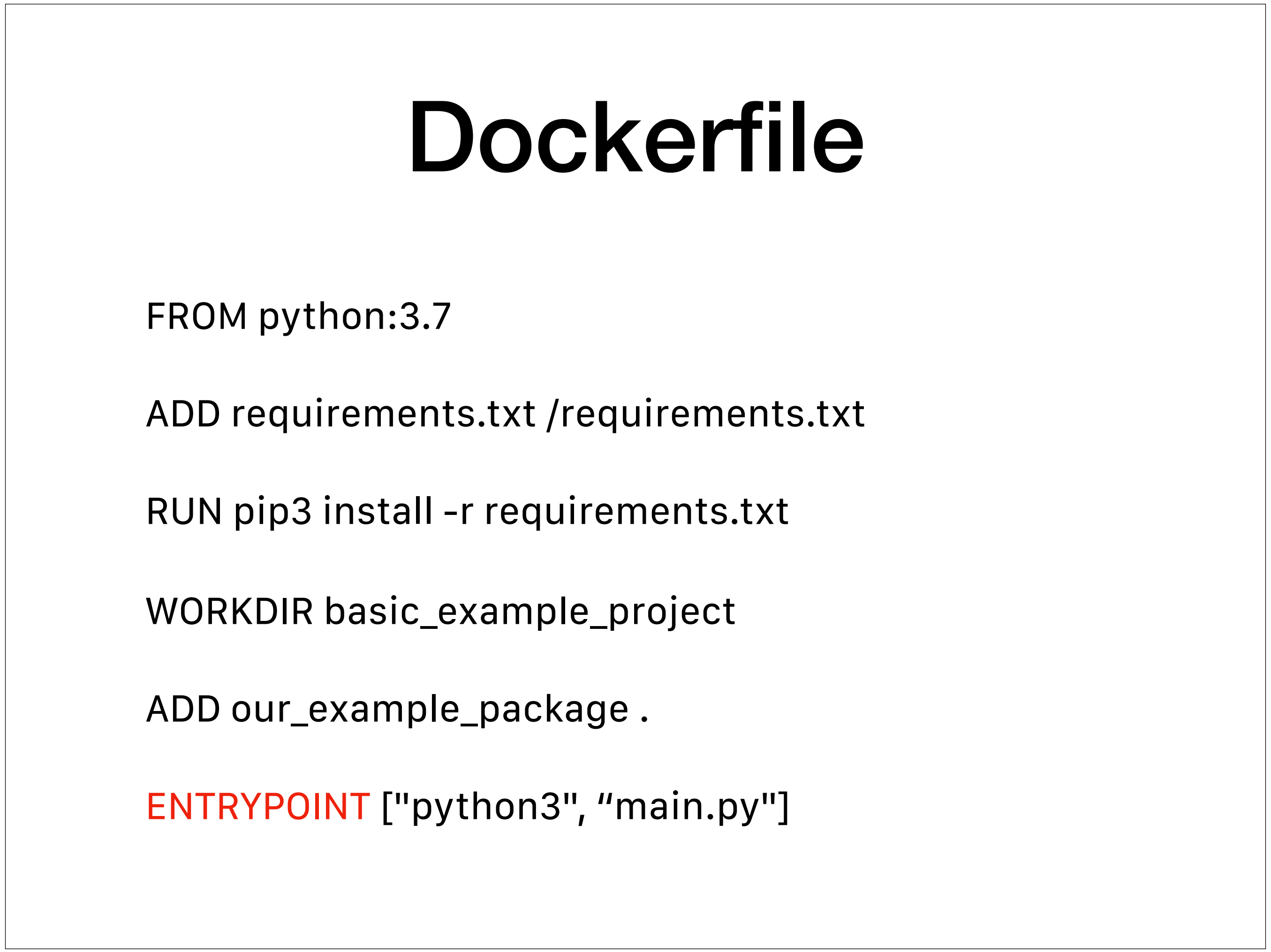
```
FROM python:3.7

ADD requirements.txt /requirements.txt

RUN pip3 install -r requirements.txt

WORKDIR basic_example_project
```

Dockerfile

```
FROM python:3.7

ADD requirements.txt /requirements.txt

RUN pip3 install -r requirements.txt

WORKDIR basic_example_project

ADD our_example_package .
```

Dockerfile

```
FROM python:3.7

ADD requirements.txt /requirements.txt

RUN pip3 install -r requirements.txt

WORKDIR basic_example_project

ADD our_example_package .

ENTRYPOINT ["python3", "main.py"]
```



Dockerfile

```
FROM python:3.7

ADD requirements.txt /requirements.txt

RUN pip3 install -r requirements.txt

WORKDIR basic_example_project

ADD our_example_package .

ENTRYPOINT ["python3", “main.py"]
```

WORKDIR - the directory in the `container!` we are running subsequent commands (until further notice)
ENTRYPOINT - command to run when starting the container
CMD - appended to entrypoint, provides arguments

ENTRYPOINT vs. CMD

our_image
(latest)

ENTRYPOINT vs. CMD

WORKDIR /
ENTRYPOINT: [“ls”]

ENTRYPOINT vs. CMD

our_image
(latest)

WORKDIR /
ENTRYPOINT: [“ls”]

```
docker run our_image:latest
```

calls ‘ls /‘

ENTRYPOINT vs. CMD

our_image
(latest)

WORKDIR /
ENTRYPOINT: [“ls”]

`docker run our_image:latest`	**calls ‘ls /‘**
`docker run our_image:latest example_dir`	**calls ‘ls /example_dir’**

ENTRYPOINT vs. CMD

our_image
(latest)

WORKDIR /
ENTRYPOINT: [“ls”]

```
docker run our_image:latest
```

calls ‘ls /‘

```
docker run our_image:latest example_dir
```

calls ‘ls /example_dir’

```
docker run --entrypoint cat our_image:latest example_file
```

calls ‘cat /example_file’

Your Turn!

Docker Hub

- Allows sharing of docker images
- Images such as `python:3.7` are hosted on Docker Hub
- https://hub.docker.com
- Signup for a free account!
- Use `docker login` on your machine

Building Images

- docker image build -t <image_name>:<image_tag> <working_dir>
- e.g. *docker image build -t run_docker:latest .*
- Can manually specify *Dockerfile* with *-f*
- All files you copy into your docker image must be in, or subdirectories of *working_dir*

Docker Labs

- https://labs.play-with-docker.com
- Free learning resources in a browser
- Can use one or more machines to experiment with Docker, Docker-Compose & Docker Swarm

github.com/mattpaletta/distributed-docker-example

Basic Task

- run an instance of the `redis:latest` image on your machine
- build and run a image/container for the code in ‘*exercises/docker/run_docker/*` on Github

Adv. Task

- search and run an image other than Python or Redis on Hub
- try running it with *-d* - what does that do?
- Look at your current containers with `docker ps`
    - What happens if you run it `—rm`
- Look at your current images with `docker images`
    - What’s the difference between `redis:latest` and `redis:alpine`

Helpful commands:

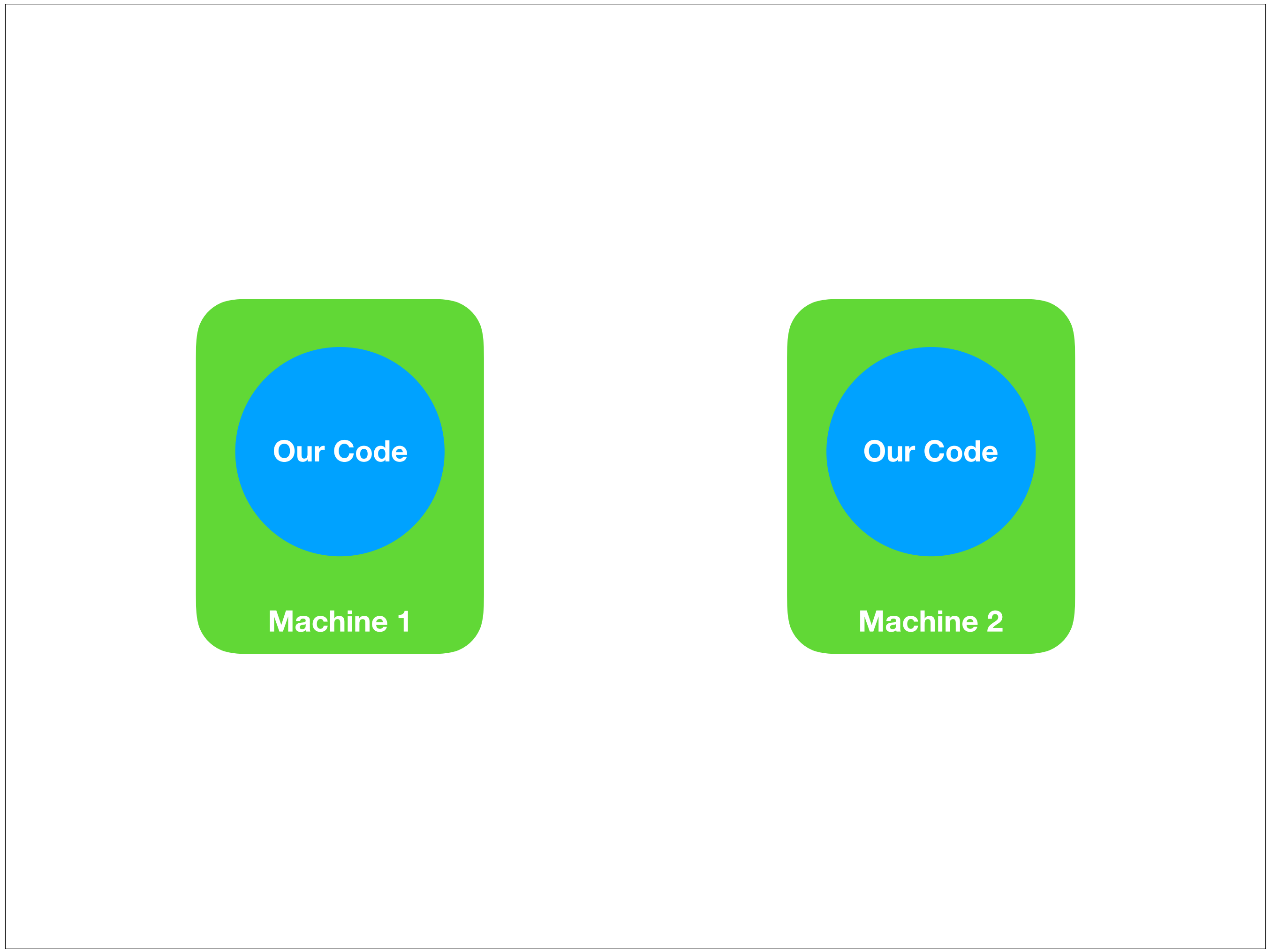
```
docker run <image>
docker image build -t <img>:<tag> <work_dir>
```

Solutions on Github

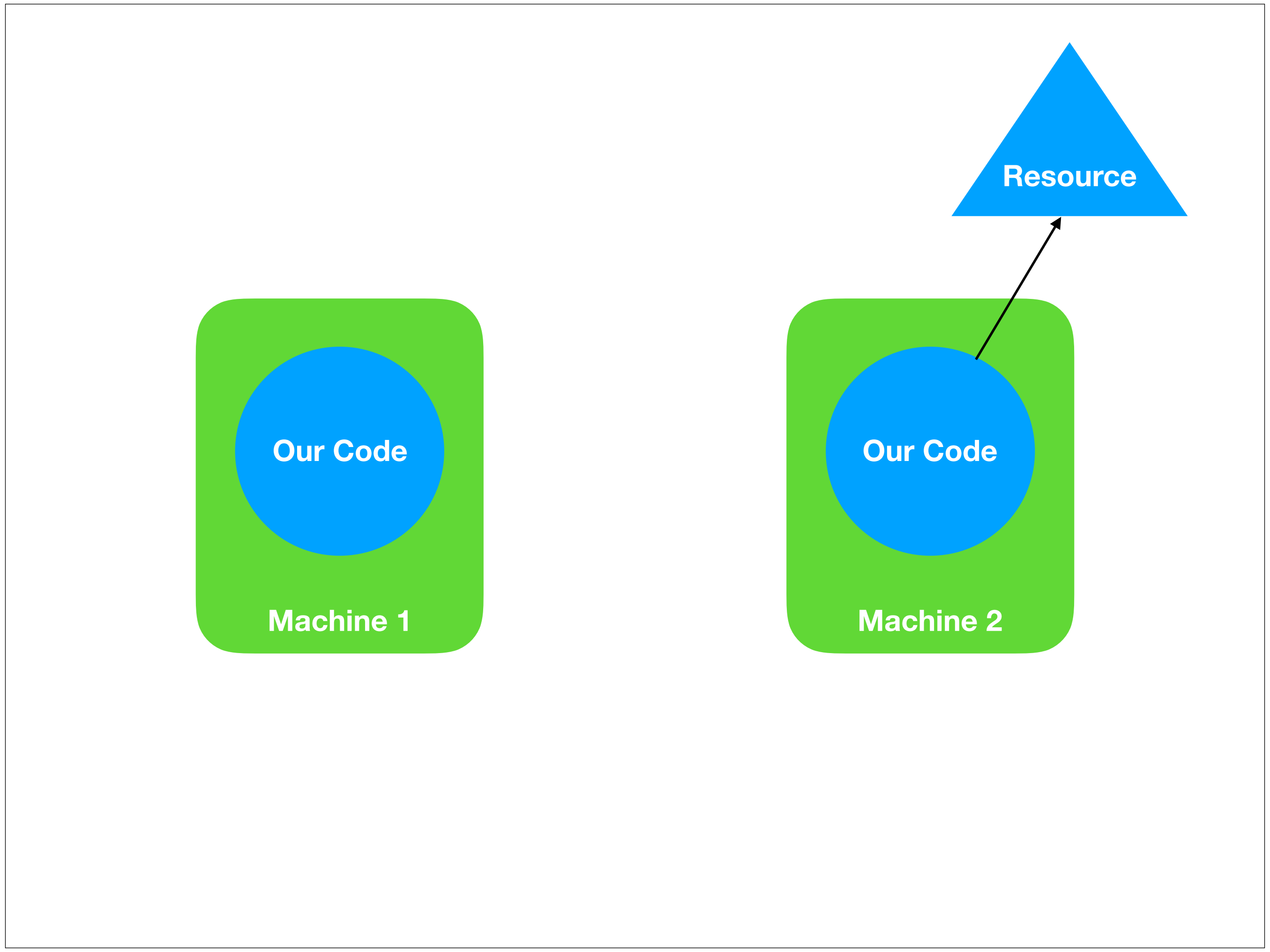
RPC

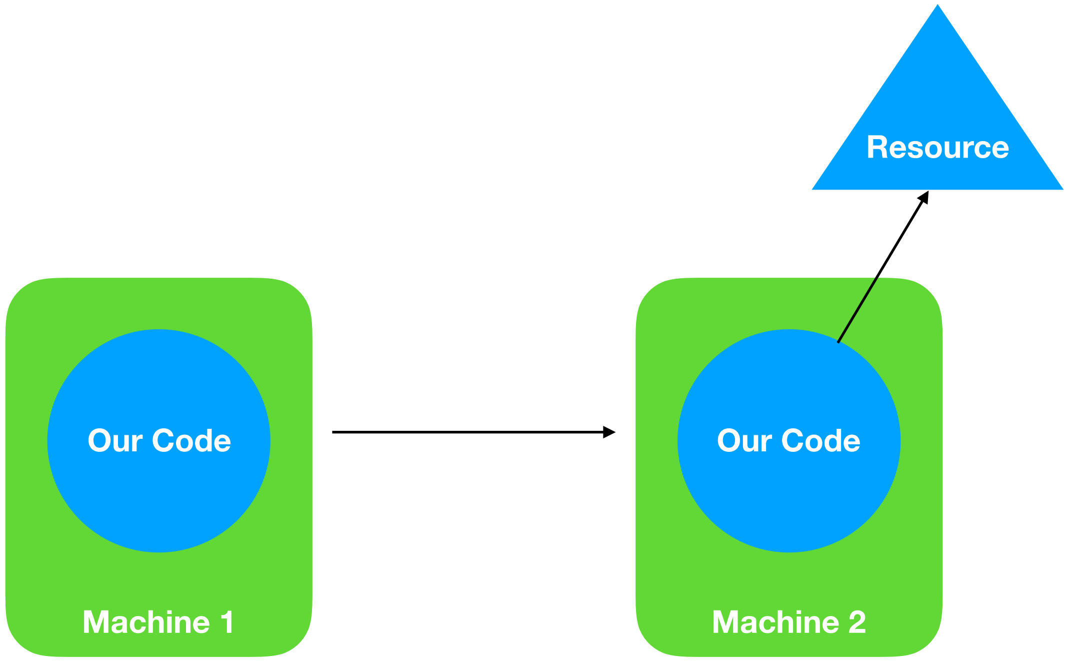
- I want to show you one more Docker thing that will let you run multiple containers and connect them, but first, RPC!

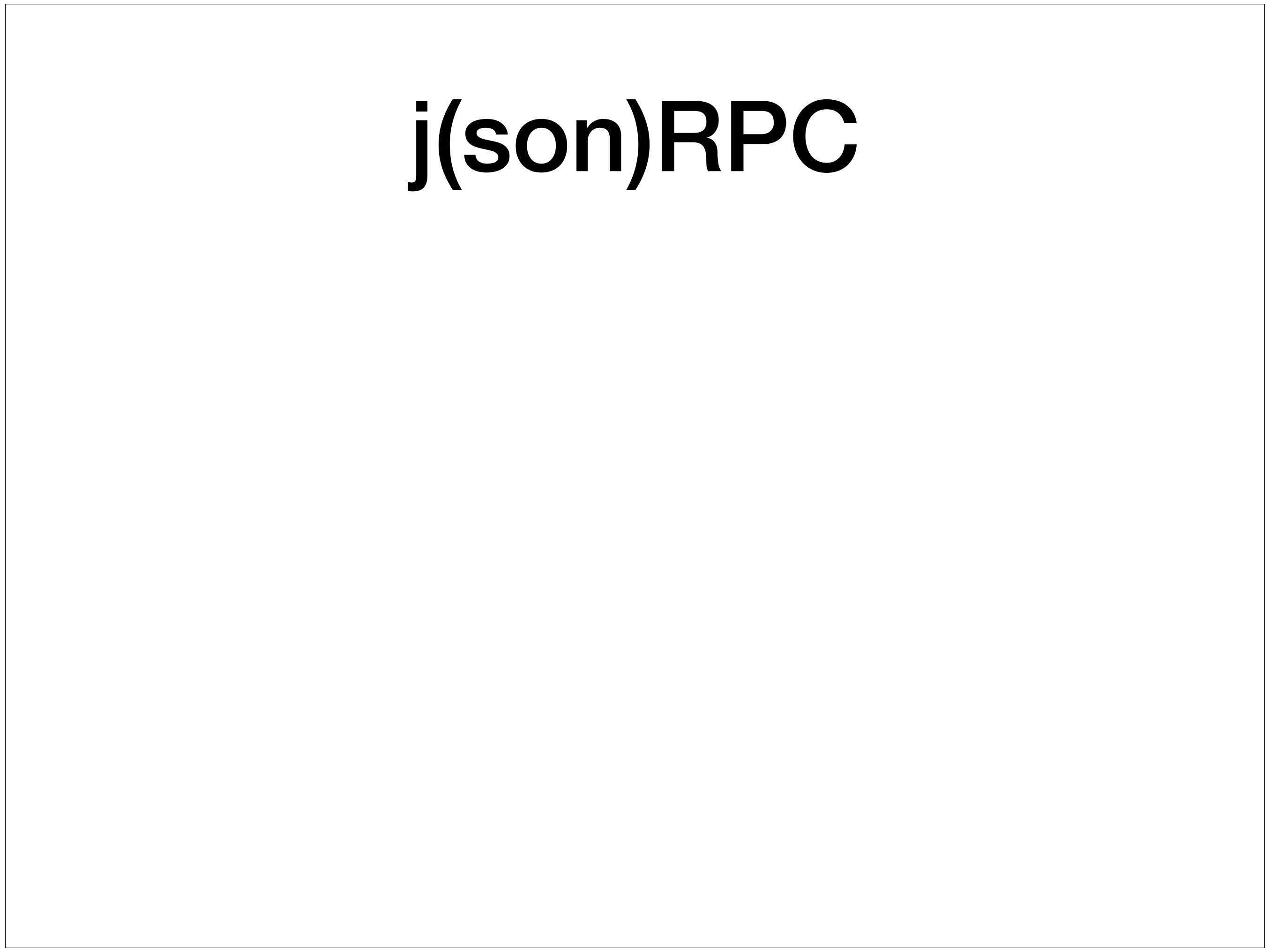
Remote Procedure Call



Goal - we have 2 systems on the network, we want to run a function from one machine to another
Maybe the second machine has access to certain data or resources we want


Resource
Our Code
Machine 1
Our Code
Machine 2


Resource
Our Code
Machine 1
Our Code
Machine 2

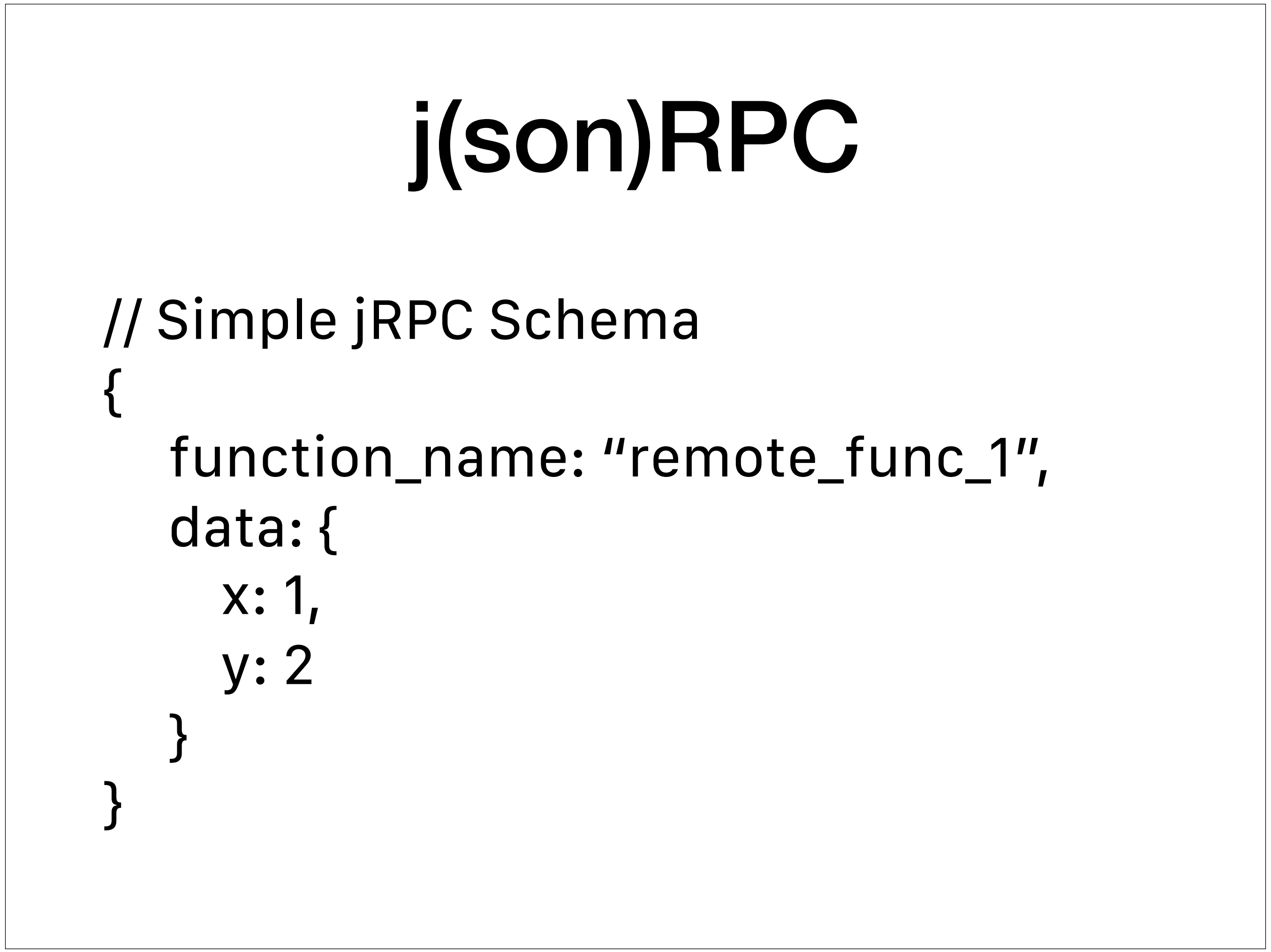


j(son)RPC

- Consider this thought experiment
- Lets implement a jRPC Protocol

j(son)RPC

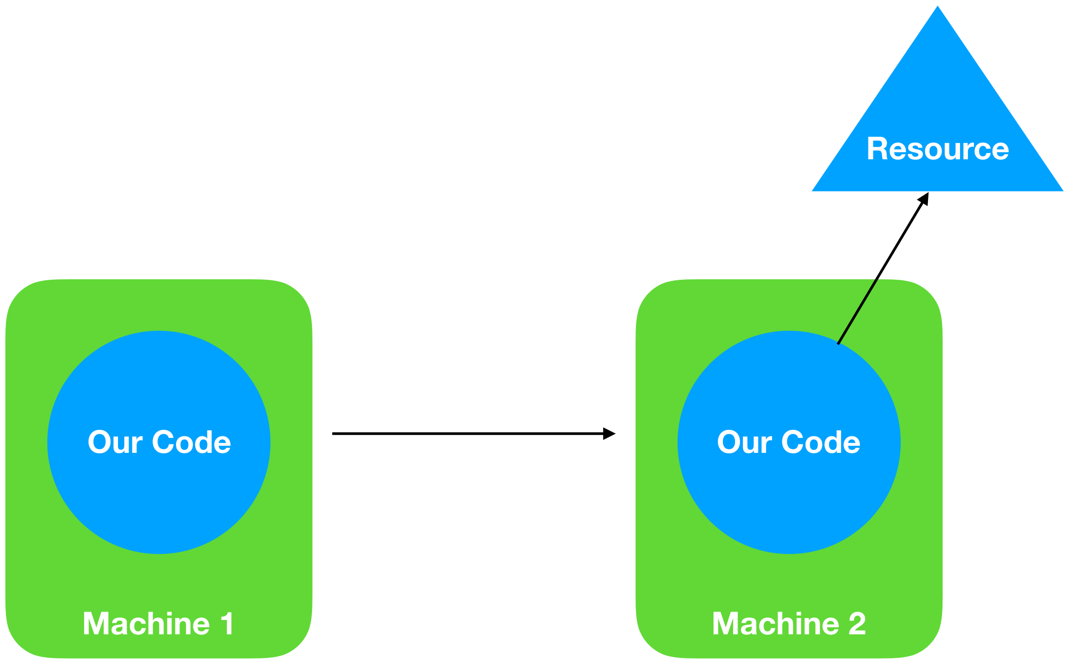
```
// Simple jRPC Schema
{
    function_name: “….”,
    data: {
        …
        …
    }
}
```



j(son)RPC

```
// Simple jRPC Schema
{
    function_name: “remote_func_1”,
    data: {
       x: 1,
       y: 2
    }
}
```

- we could fill in our code with values such as the following



Resource

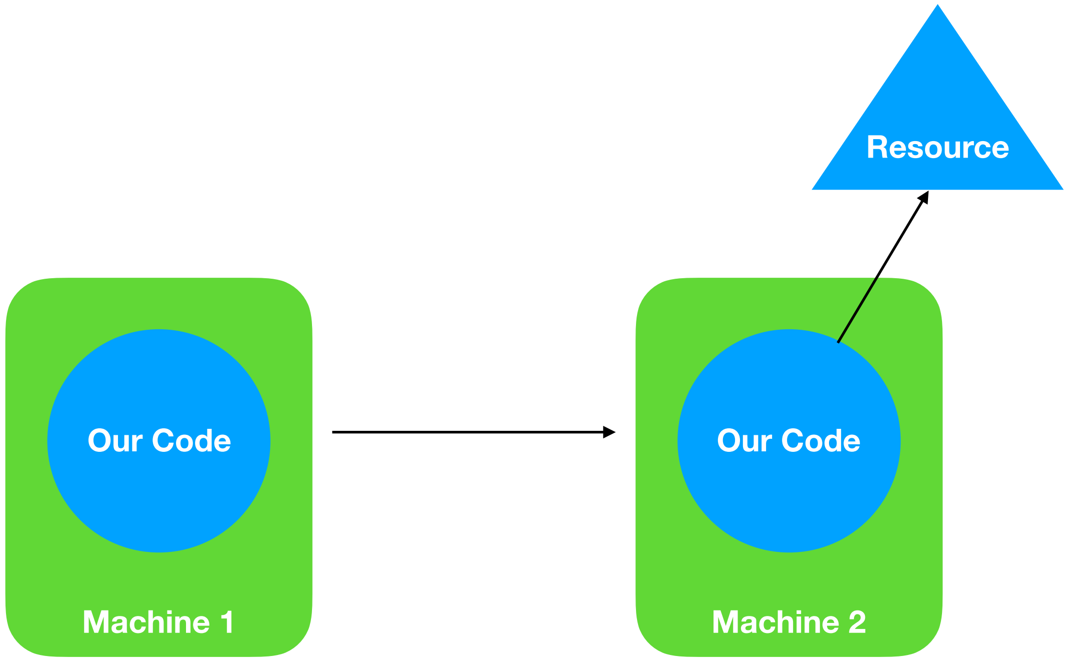
Our Code

Machine 1

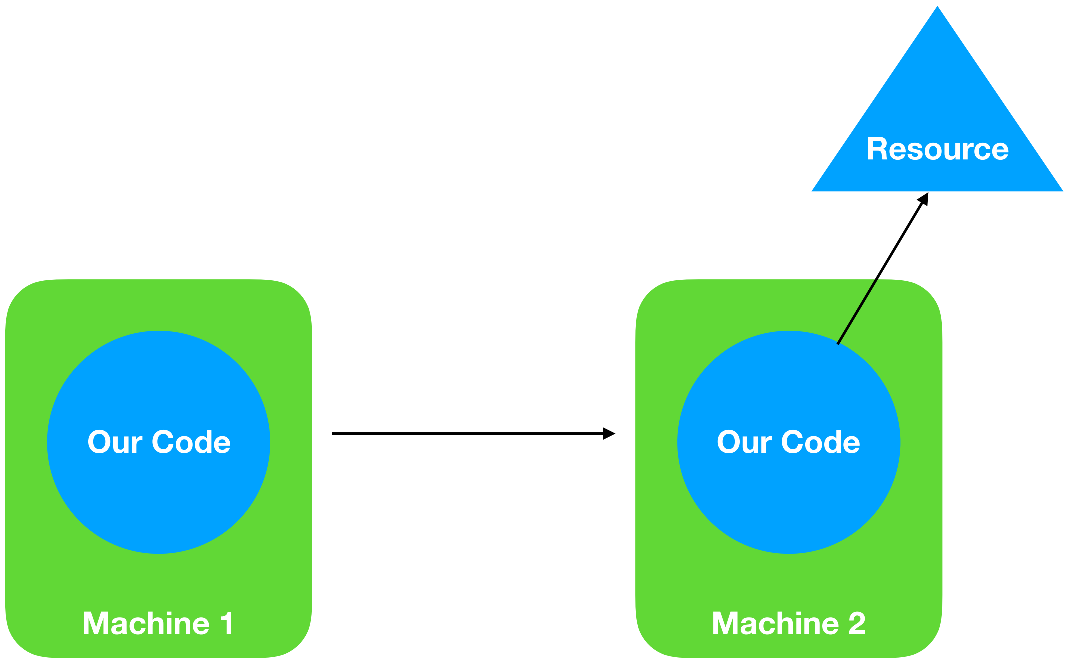
Our Code

Machine 2

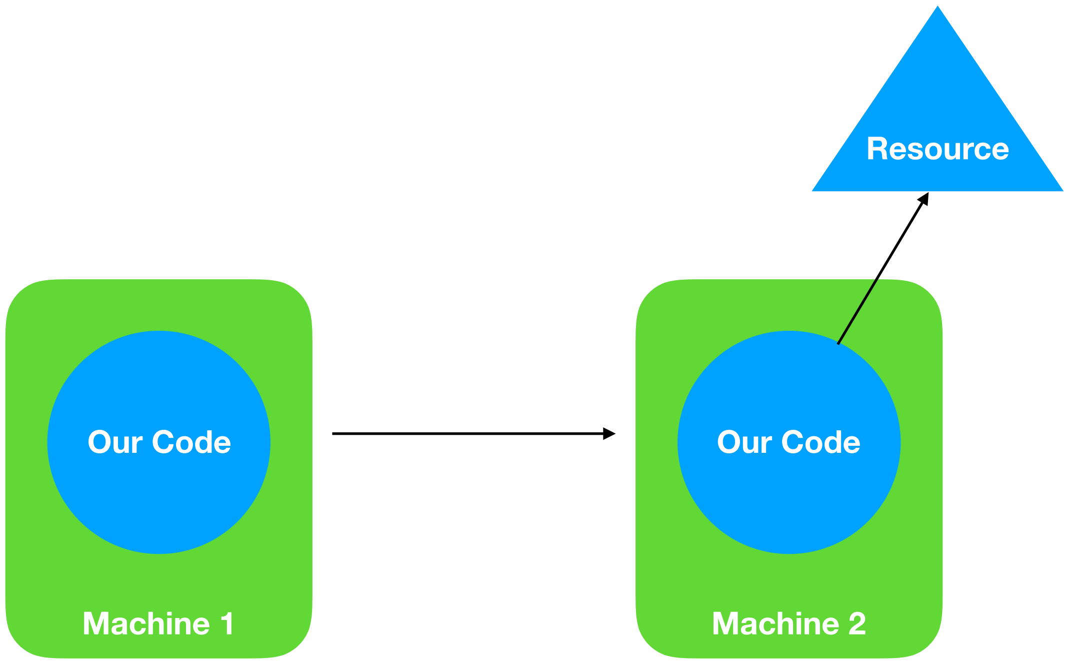
```
{
   function_name: “remote_func_1”,
   data: {
      x: 1,
      y: 2
   }
}
```



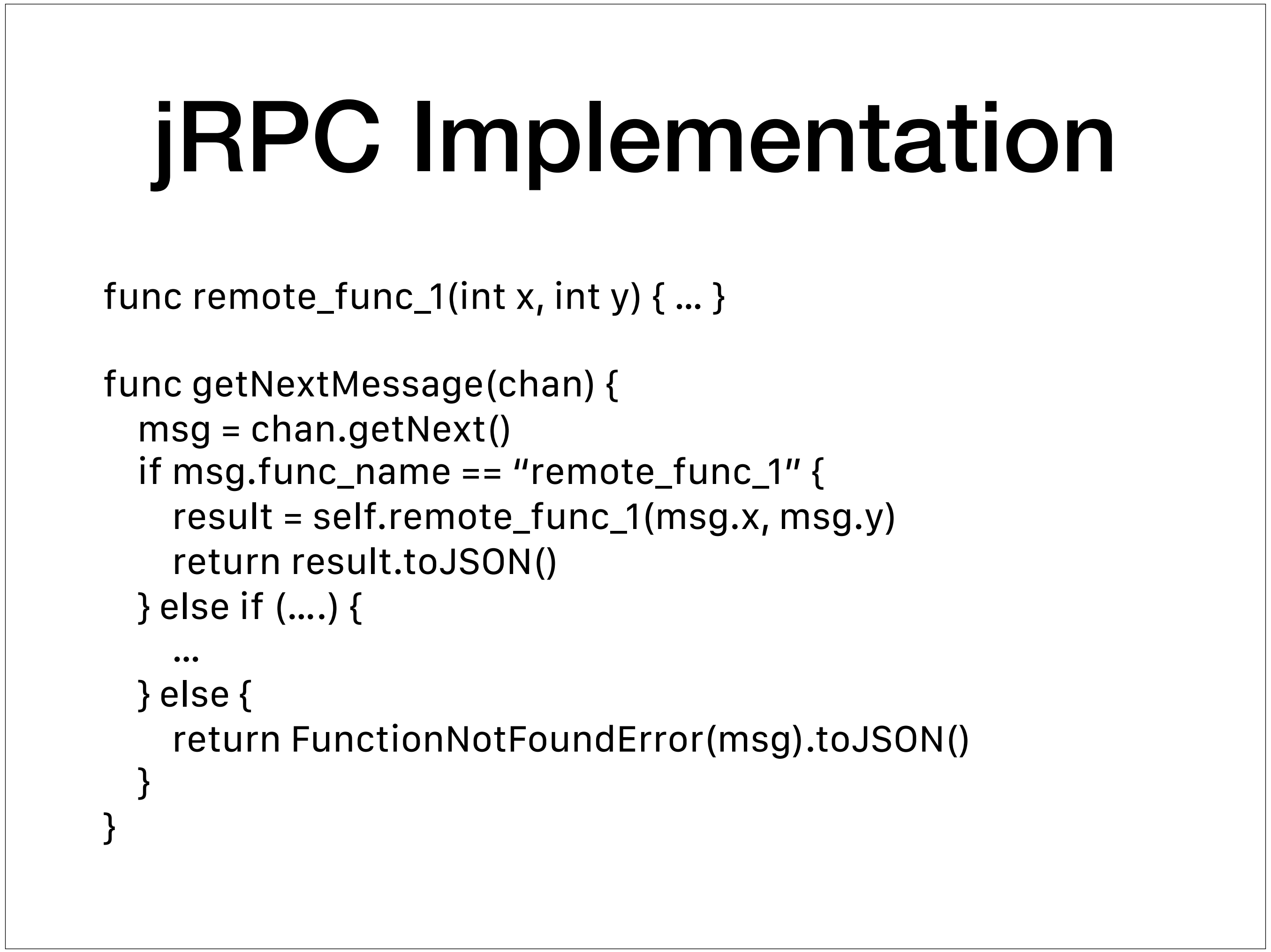
```
{
    function_name: “remote_func_1”,
    data: {
       x: 1,
       y: 2
    }
}
```



```
{
    function_name: “remote_func_1”,
    result: 3
}
```



```
{
    function_name: “remote_func_1”,
    result: 3
}
```

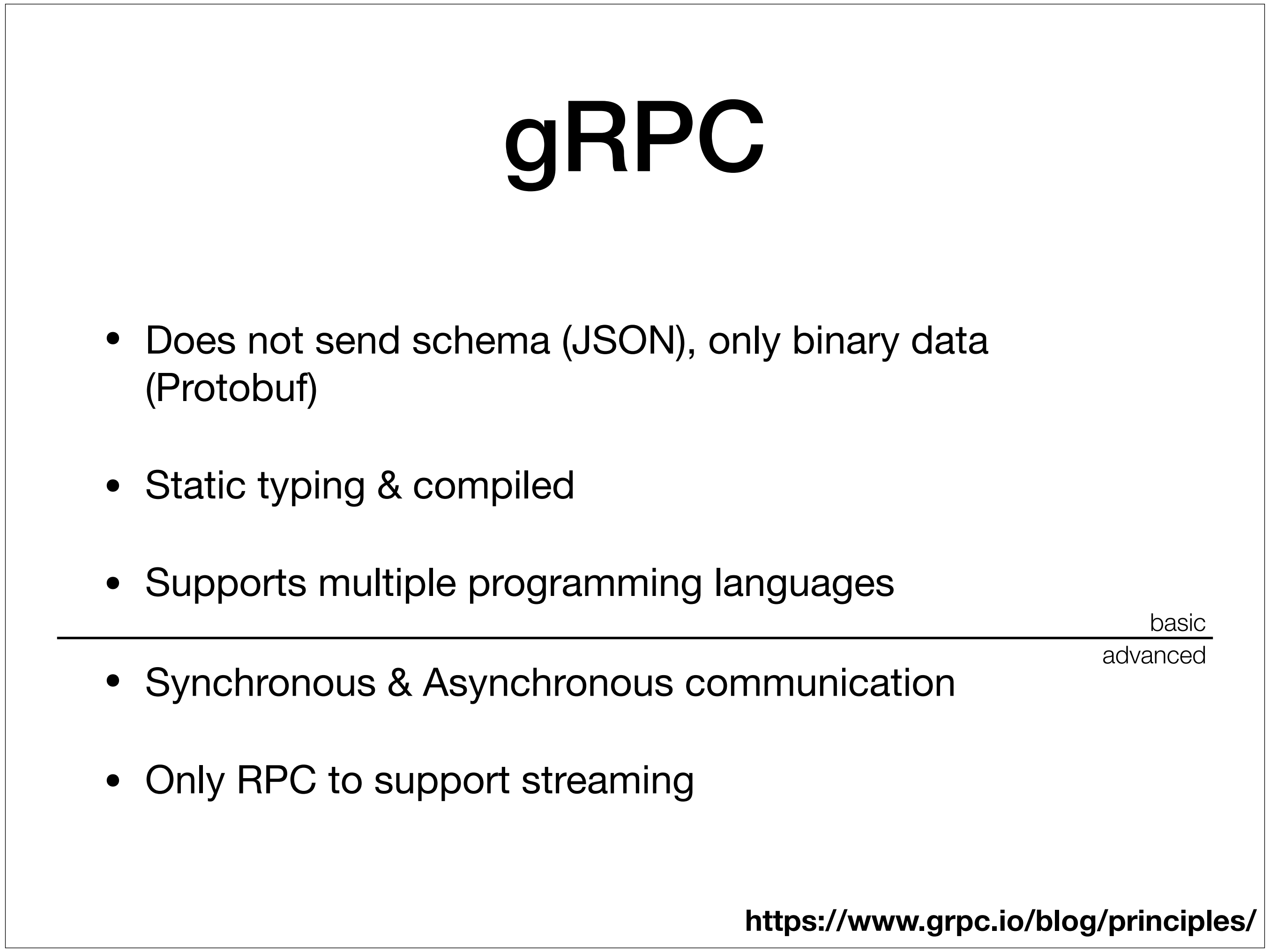


jRPC Implementation

```
func remote_func_1(int x, int y) { … }

func getNextMessage(chan) {
   msg = chan.getNext()
   if msg.func_name == “remote_func_1” {
      result = self.remote_func_1(msg.x, msg.y)
      return result.toJSON()
   } else if (….) {
      …
   } else {
      return FunctionNotFoundError(msg).toJSON()
   }
}
```

Here’s how we might implement it

gRPC



gRPC

- Does not send schema (JSON), only binary data (Protobuf)
- Static typing & compiled
- Supports multiple programming languages

basic

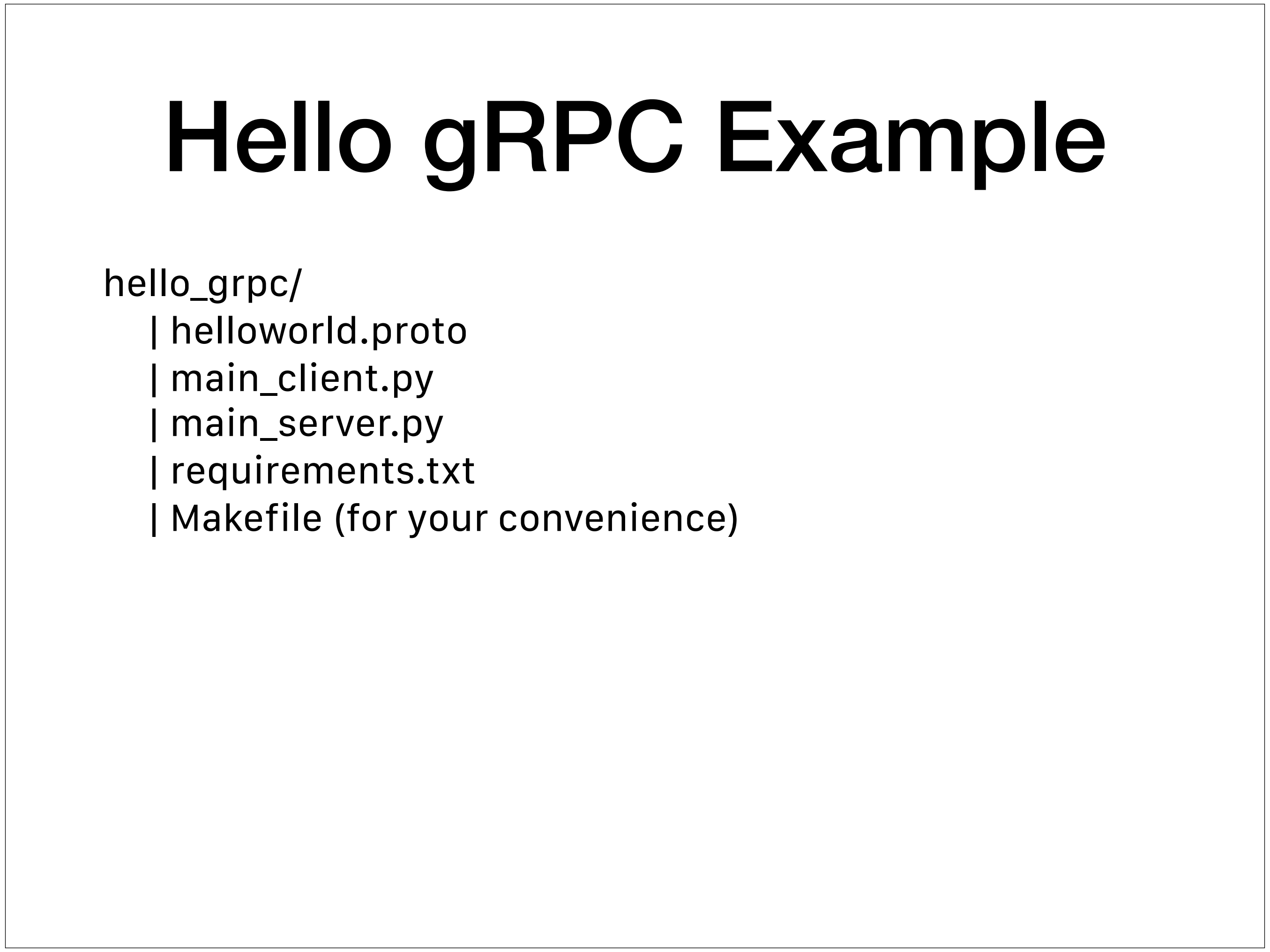
advanced

- Synchronous & Asynchronous communication
- Only RPC to support streaming

https://www.grpc.io/blog/principles/

- Don’t worry if you don’t understand the last 2!

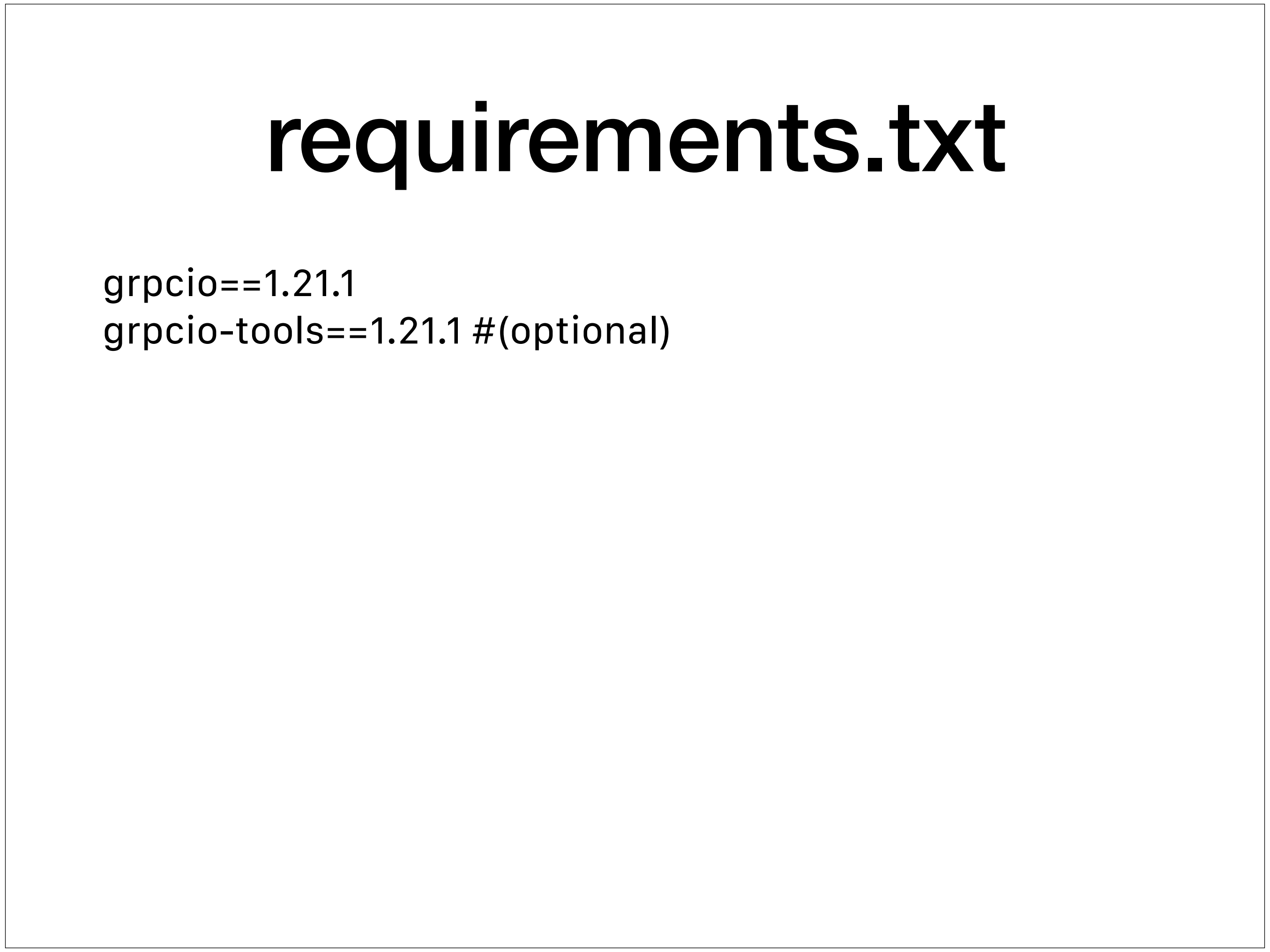
github.com/mattpaletta/distributed-docker-example



Hello gRPC Example

```
hello_grpc/
   | helloworld.proto
   | main_client.py
   | main_server.py
   | requirements.txt
   | Makefile (for your convenience)
```

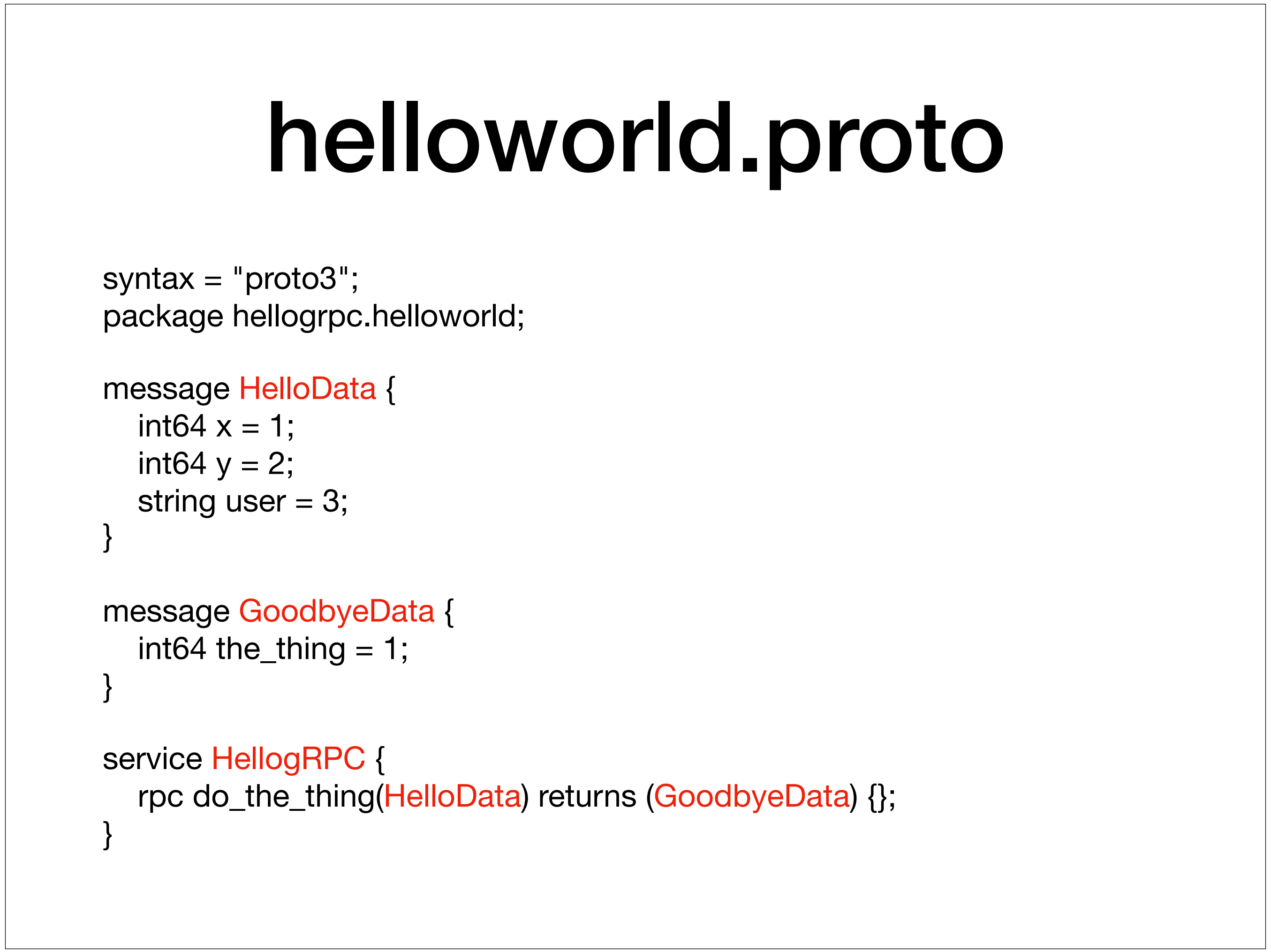
- Here’s our folder structure so you can follow along



requirements.txt

```
grpcio==1.21.1
grpcio-tools==1.21.1 #(optional)
```

- Here’s our folder structure so you can follow along



helloworld.proto

```
syntax = "proto3";
package hellogrpc.helloworld;

message HelloData {
    int64 x = 1;
    int64 y = 2;
    string user = 3;
}

message GoodbyeData {
    int64 the_thing = 1;
}

service HellogRPC {
    rpc do_the_thing(HelloData) returns (GoodbyeData) {};
}
```

- I called them different things just so you can follow along. They could be the same types! - just can’t be empty

<run>

```
python3 -m grpc_tools.protoc -I./ \
      --python_out=./ \
      --grpc_python_out=./ \
      helloworld.proto
```

Generates:

```
    helloworld_pb2.py
    helloworld_pb2_grpc.py
```

`make protos`

* if you didn’t install `grpcio-tools`, call `protoc` directly

main_server.py

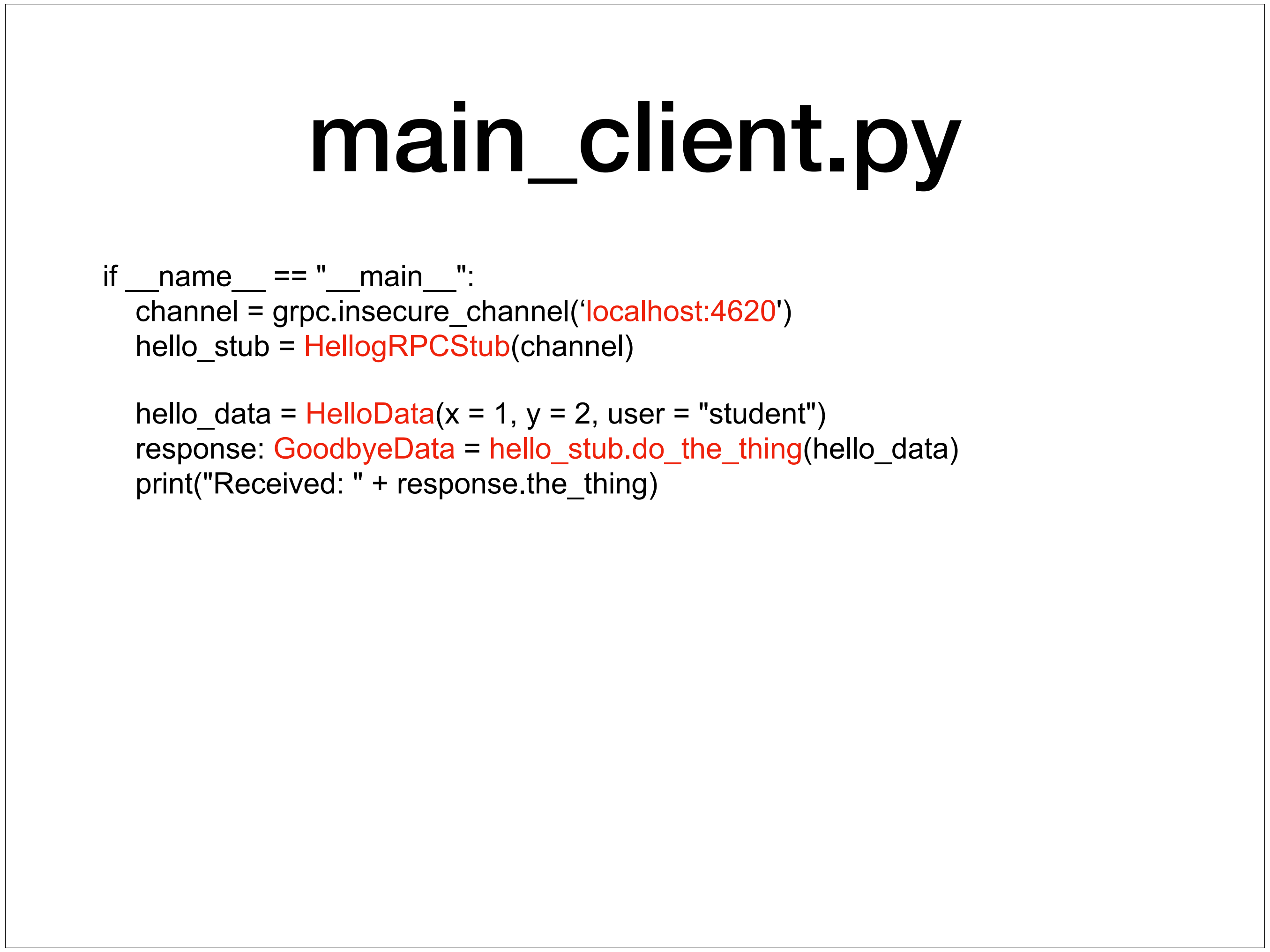
- The red things are determined by your proto file and will change per implementation
- The orange class is the class we put our implementation in
- I’ve added type annotations so its easier to follow

main_server.py

```
class HellogRPCServiceImpl(HellogRPCServicer):
    def do_the_thing(self, request: HelloData, context: grpc.RpcContext = None):
        x = request.x
        y = request.y
        user = request.user
        print("Got request from: ", user)
        return GoodbyeData(the_thing = x + y)
```

main_server.py

```
class HellogRPCServiceImpl(HellogRPCServicer):
    def do_the_thing(self, request: HelloData, context: grpc.RpcContext = None):
        x = request.x
        y = request.y
        user = request.user
        print("Got request from: ", user)
        return GoodbyeData(the_thing = x + y)


if __name__ == "__main__":
    hello_port = 4620
    server = grpc.server(futures.ThreadPoolExecutor(max_workers = 4))
    add_HellogRPCServicer_to_server(servicer = HellogRPCServiceImpl(), server = server)
    server.add_insecure_port(‘localhost:{0}’.format(hello_port))
    server.start()
    while True:
        sleep(ONE_DAY_IN_SECONDS)
```



main_client.py

```
if __name__ == "__main__":
    channel = grpc.insecure_channel(‘localhost:4620')
    hello_stub = HellogRPCStub(channel)

    hello_data = HelloData(x = 1, y = 2, user = "student")
    response: GoodbyeData = hello_stub.do_the_thing(hello_data)
    print("Received: " + response.the_thing)
```

- The red things are determined by your proto file and will change per implementation
- The orange class is the class we put our implementation in
- I’ve added type annotations so its easier to follow

One more thing

- The red things are determined by your proto file and will change per implementation
- The orange class is the class we put our implementation in
- I’ve added type annotations so its easier to follow

One more thing

Python 3.6+ has type annotations! (as you saw)

One more thing

Python 3.6+ has type annotations! (as you saw)

Package: mypy-protobuf

One more thing

Python 3.6+ has type annotations! (as you saw)

Package: mypy-protobuf

```
python3 -m grpc_tools.protoc -I./ \
    --python_out=./ \
    --grpc_python_out=./ \
    --mypy_out=./ \
    helloworld.proto
```

Generates type-annotations for your IDE!

(example in hello_grpc_typed)

Your turn!

github.com/mattpaletta/distributed-docker-example

Basic Task

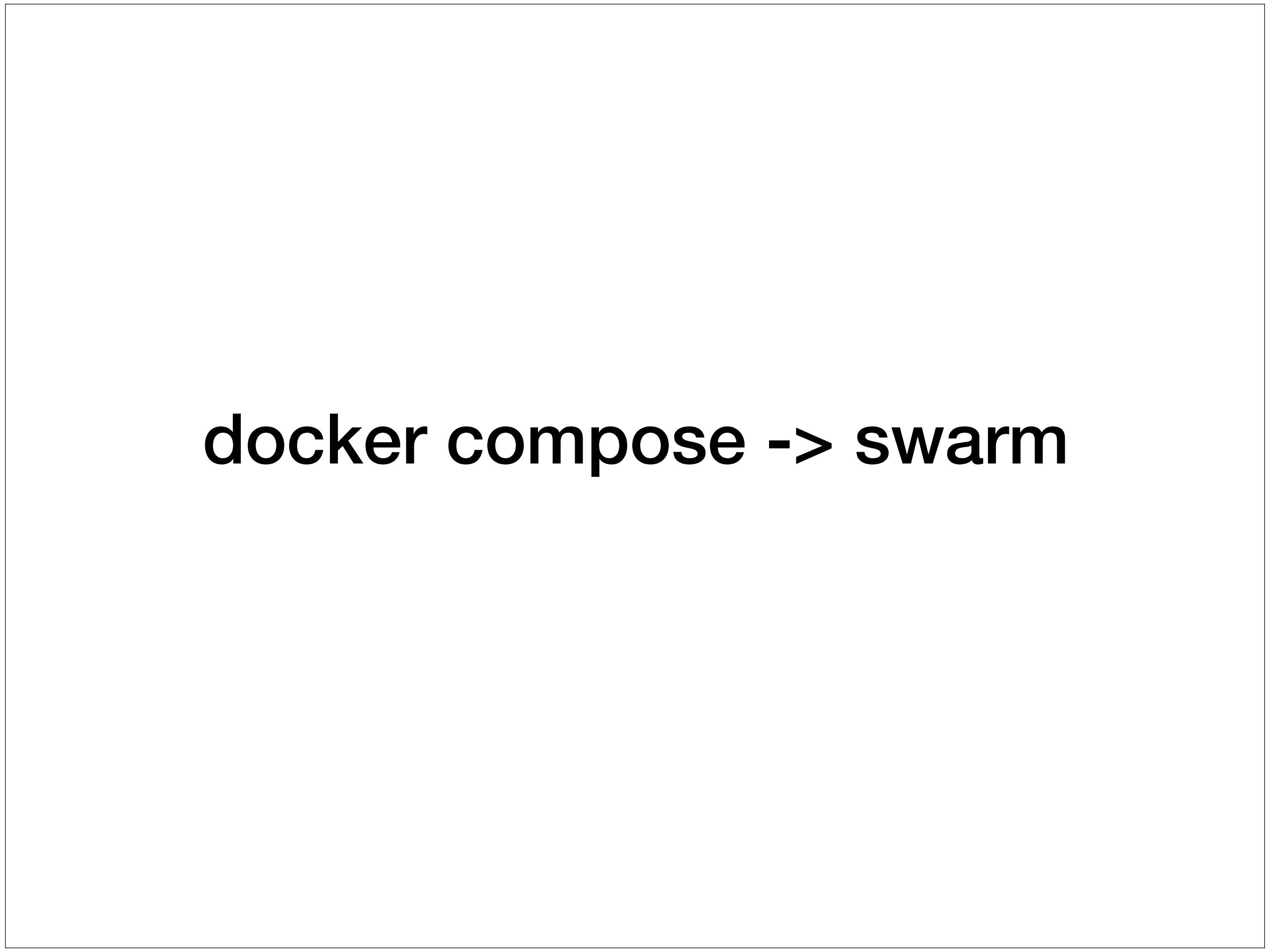
- Given a *.proto,* and a server, write the client (my_client)
- Given a *.proto*, write the server and the client (my_server)
  - You don’t have to lookup the weather! (but you can)
  - Fake the data!

Adv. Task

- Collaborate with someone, one of you write the *.proto* and the client, the other person write the server!
- Try and explore other data types in Protobufs - repeated & enums

Helpful commands:

```
python3 -m grpc_tools.protoc -I./ \
 --python_out=./ \
 --grpc_python_out=./ \
 helloworld.proto
```



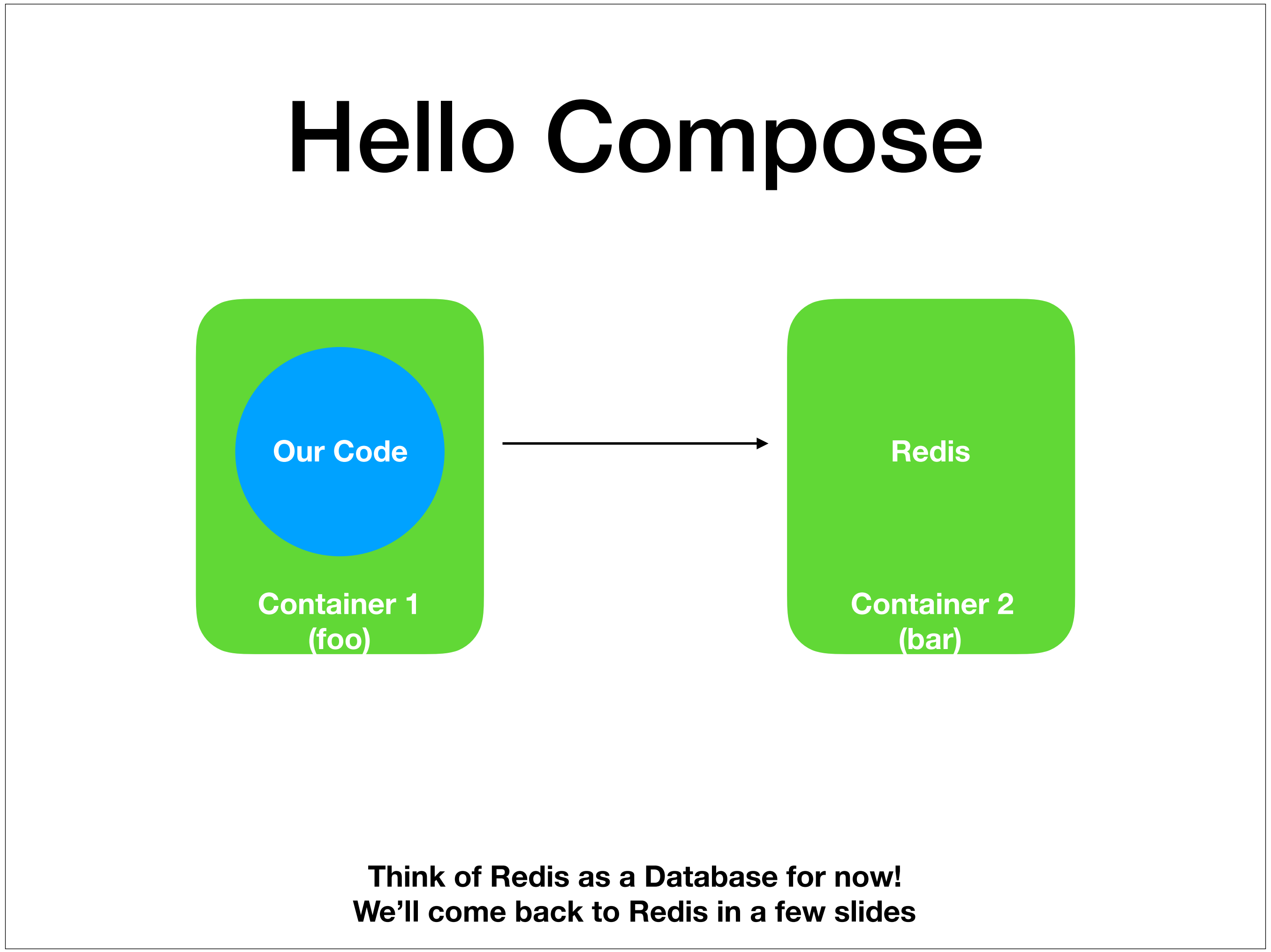
docker compose -> swarm

- We’ll start with docker compose, and just introduce swarm

YAML

- YAML Ain’t Markup Language
- Think JSON
  - With comments
  - Uses indentation instead of brackets

https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html



Hello Compose

Our Code

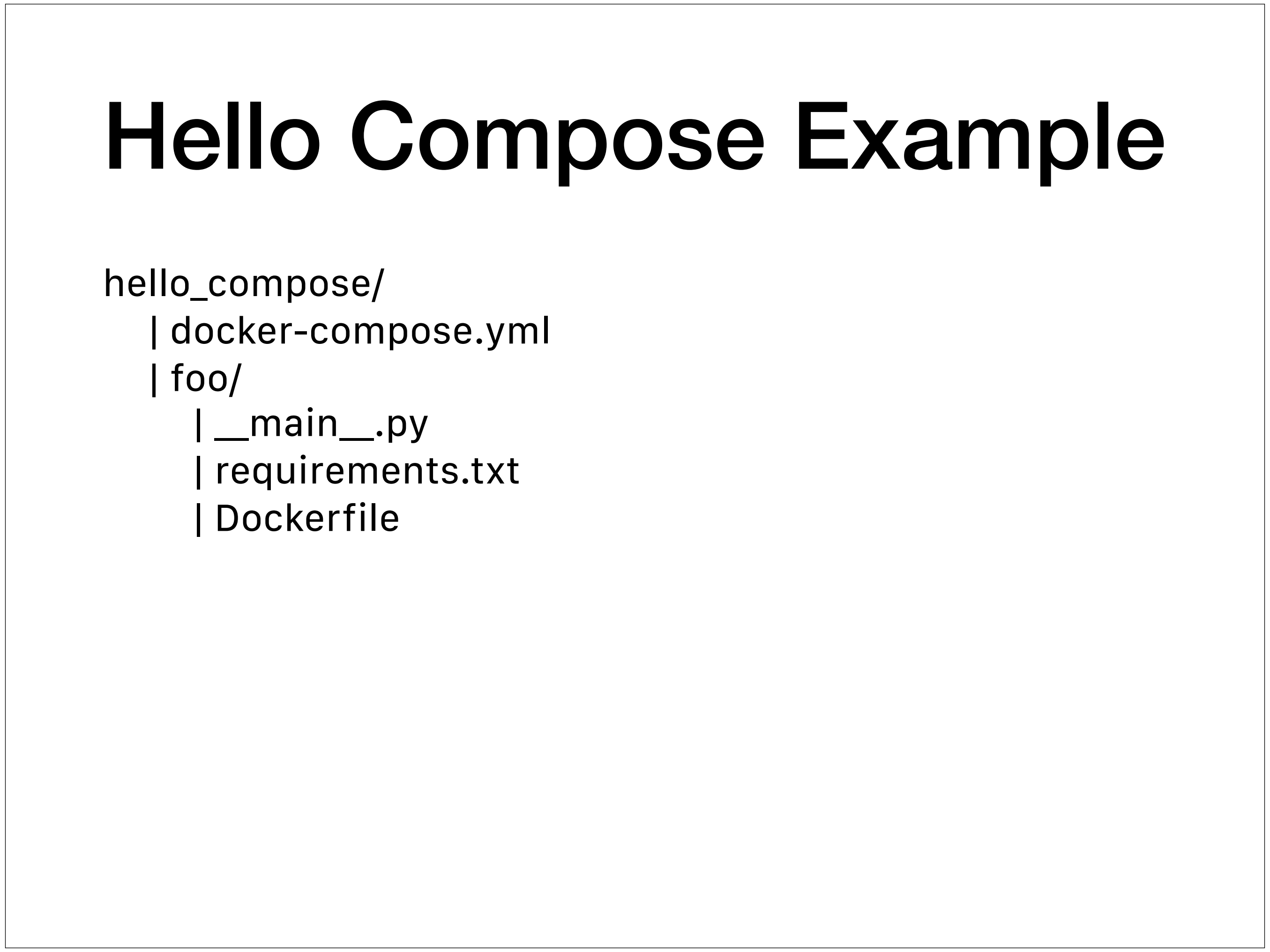
Container 1
(foo)

Redis

Container 2
(bar)

Think of Redis as a Database for now!
We’ll come back to Redis in a few slides

- we could also use MySQL or Postgres as examples, just Redis is one of the simplest



Hello Compose Example

```
hello_compose/
    | docker-compose.yml
    | foo/
        | __main__.py
        | requirements.txt
        | Dockerfile
```

- Here’s our folder structure so you can follow along

foo/requirements.txt

```
redis==3.2.1
```

foo/__main__.py

```
import redis

if __name__ == "__main__":
  redis_conn = redis.StrictRedis(host = “bar”, port = 6379)

   redis_conn.set(name = “cat”, value = 1)
   redis_conn.set(name = “dog”, value = 2)
   redis_conn.set(name = “mouse”, value = 3)

   result = str(redis_conn.get(name = “cat”))
   print(result)
```

docker-compose.yml

```
version: "3"

services:
  foo:
    build: foo
    links:
      - bar
  bar:
    image: redis:alpine
```

<run>

docker-compose up

- Most basic command
- Runs everything
- Builds and pulls images - as needed

docker-compose up --build

- Explicitly rebuilds images - with a cache

docker-compose up foo

- Run specific service(s)
- Automatically runs all dependencies

docker-compose up -d bar

- Runs service ‘bar’ in detached mode - advanced

<run> (advanced list)

```
docker-compose up -d
docker-compose down
docker-compose start
docker-compose stop
docker-compose build
docker-compose logs -f db
docker-compose scale db=4
docker-compose events
docker-compose exec db bash
```

Docker Swarm

- Compose used for testing multi-container on your machine
- Swarm used to deploy to a group of machines
  - Uses docker-compose.yml as configuration
- Alternatives include Kubernetes, and Cloud solutions
- Save for later - feel free to ask me :)

Your Turn!

github.com/mattpaletta/distributed-docker-example

Basic Task

- Take the server and client you wrote previously, create a `docker-compose.yml` file and run it!

Adv. Task

- Try adding a Redis cache as part of your compose. (next section of slides)
- Try adding a different database - Postgres, MySQL, etc.

Helpful commands:

```
docker-compose up —build
```

Solutions on Github

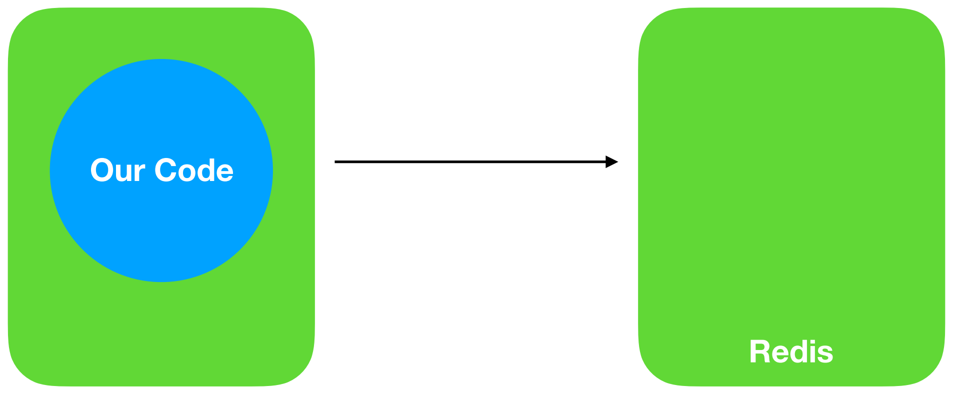
Redis

key/value store, distributed queue, etc.

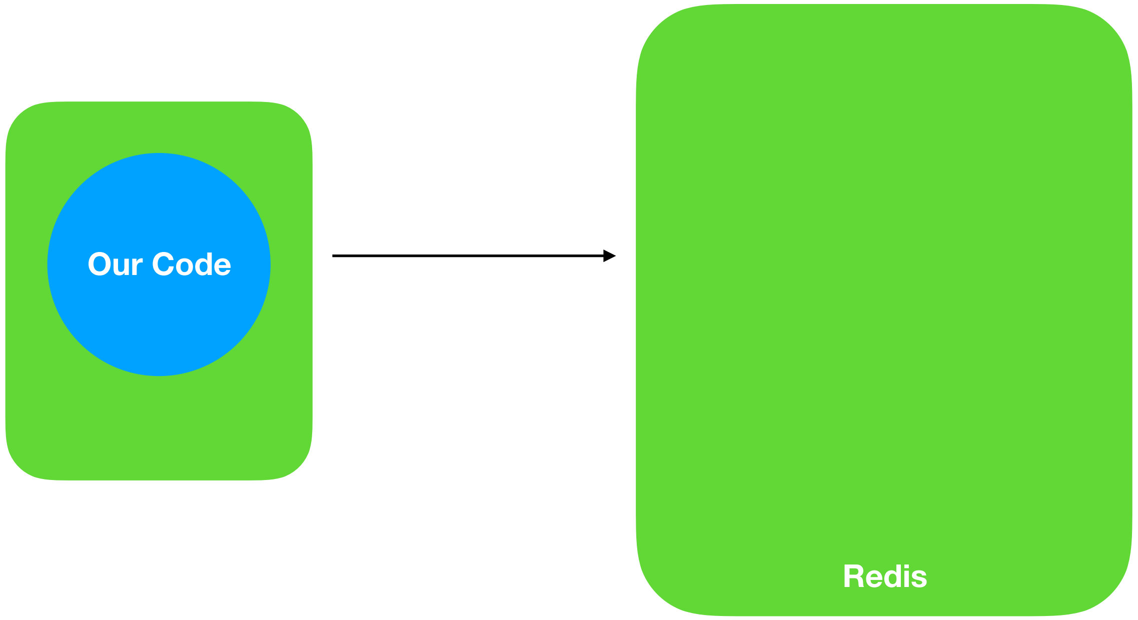
Redis

- In-Memory data structure store
- Commonly used as a cache or message queue
- Supports storing:
  - Strings
  - Hashes
  - Lists
  - Sets (sorted & unsorted)
  - Streams

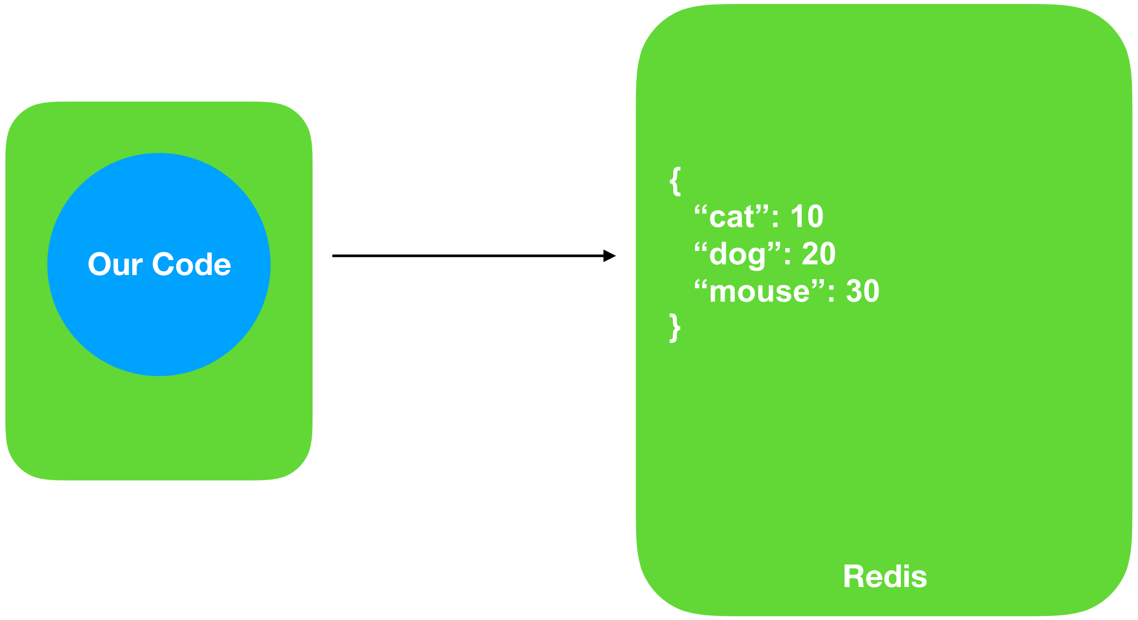
Redis (cache)



Redis (cache)



Redis (cache)

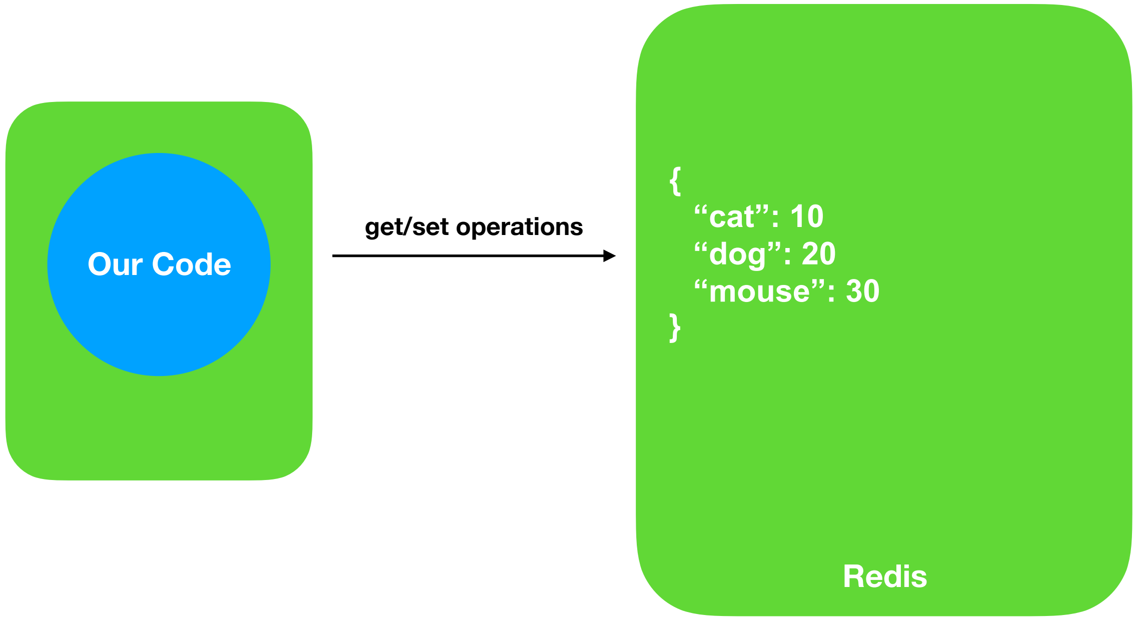


Our Code

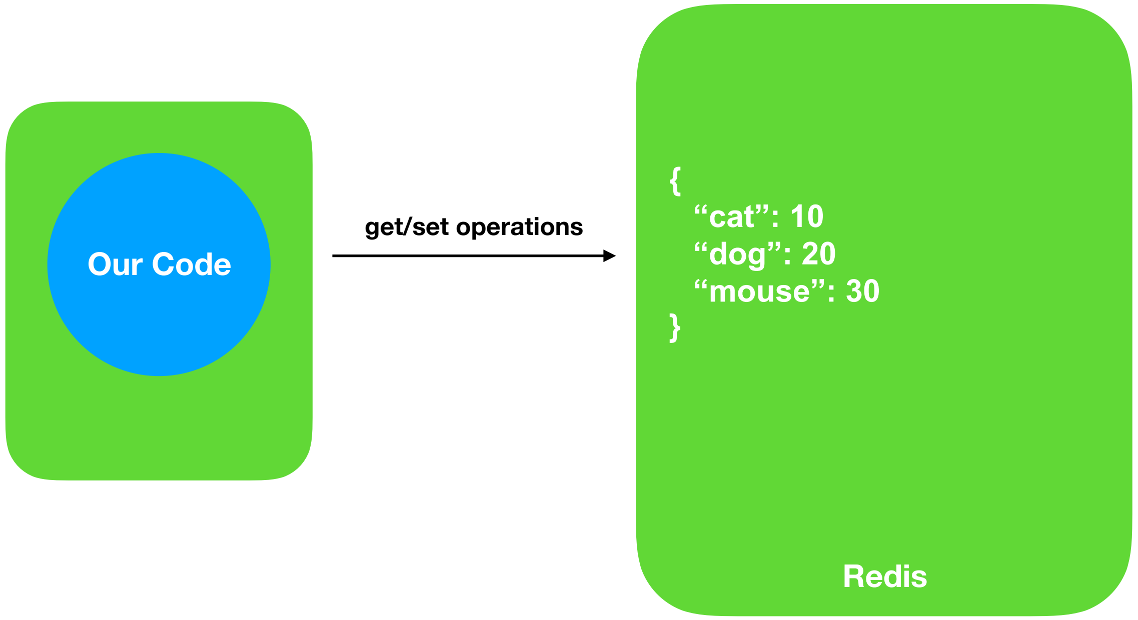
```
{
  "cat": 10
  "dog": 20
  "mouse": 30
}
```

Redis

Redis (cache)

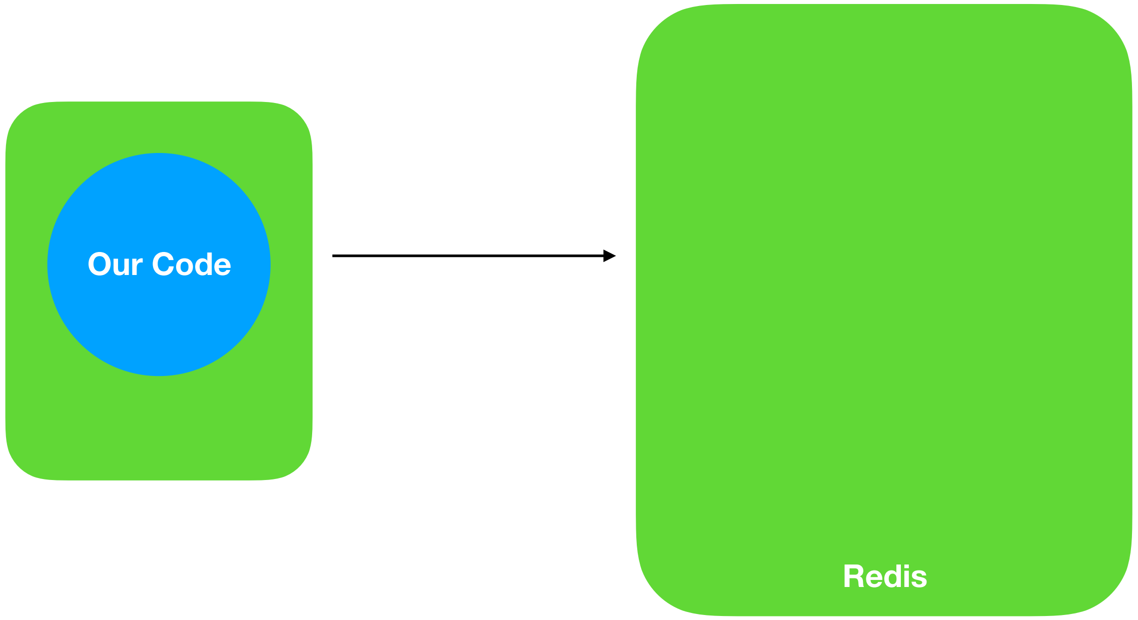


Redis (cache)

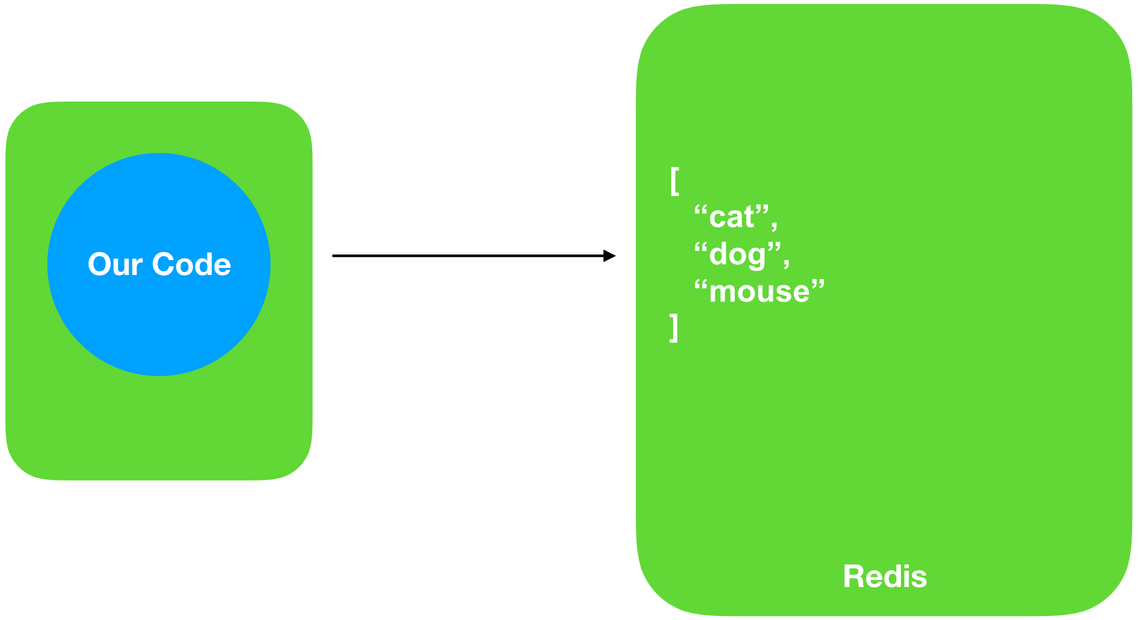


It’s just like using a dictionary on our own machine!
(but its shared and replicated)

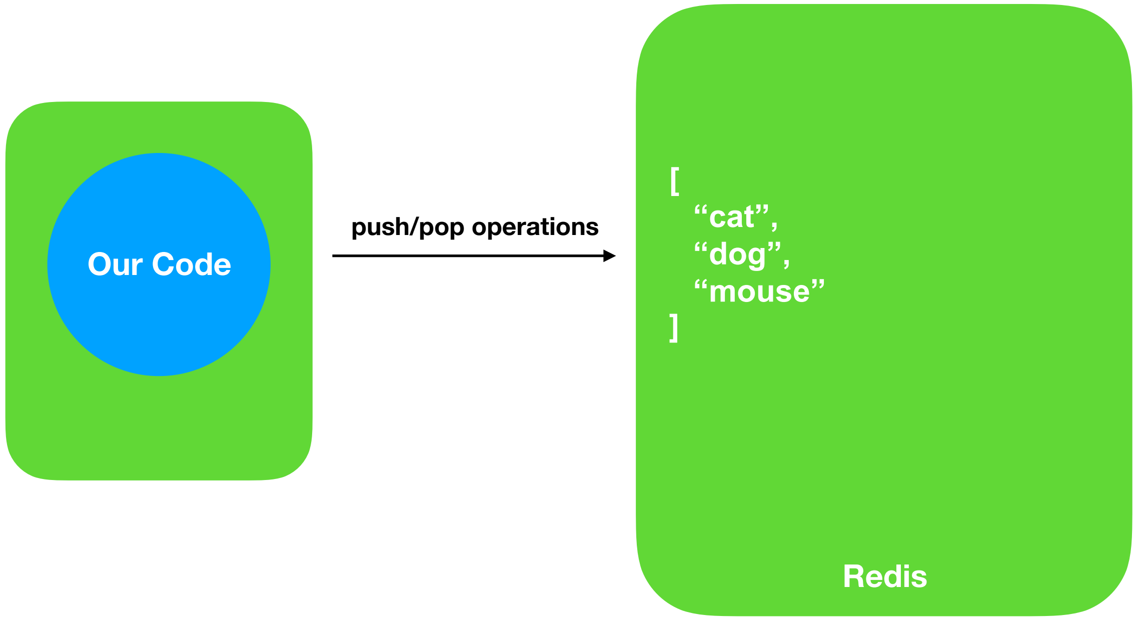
Redis (lists)



Redis (lists)



Redis (lists)



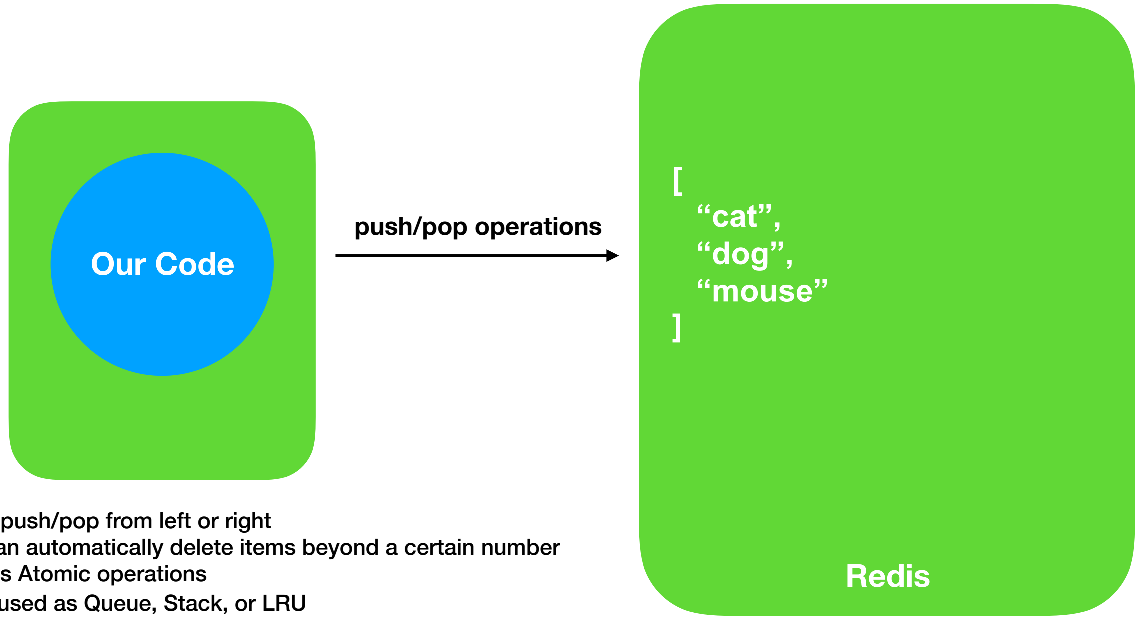
Our Code

push/pop operations

```
[
  "cat",
  "dog",
  "mouse"
]
```

Redis

Redis (lists)



Our Code

push/pop operations

```
[
  “cat”,
  “dog”,
  “mouse”
]
```

Redis

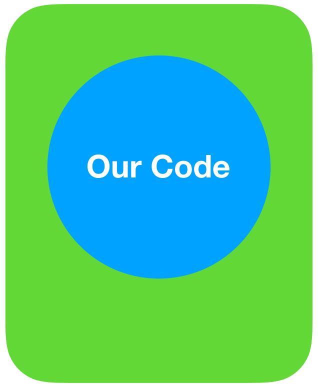
- We can push/pop from left or right
- Redis can automatically delete items beyond a certain number
- Supports Atomic operations
- Can be used as Queue, Stack, or LRU

We can use a queue to distribute work between machines

Event-Driven Architecture

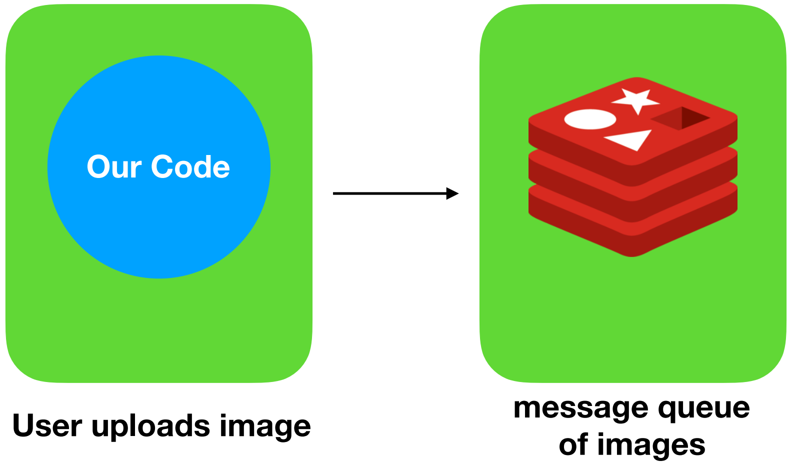
- Ask about the pros/cons of doing it this way, vs. Web server generating images

Event-Driven Architecture

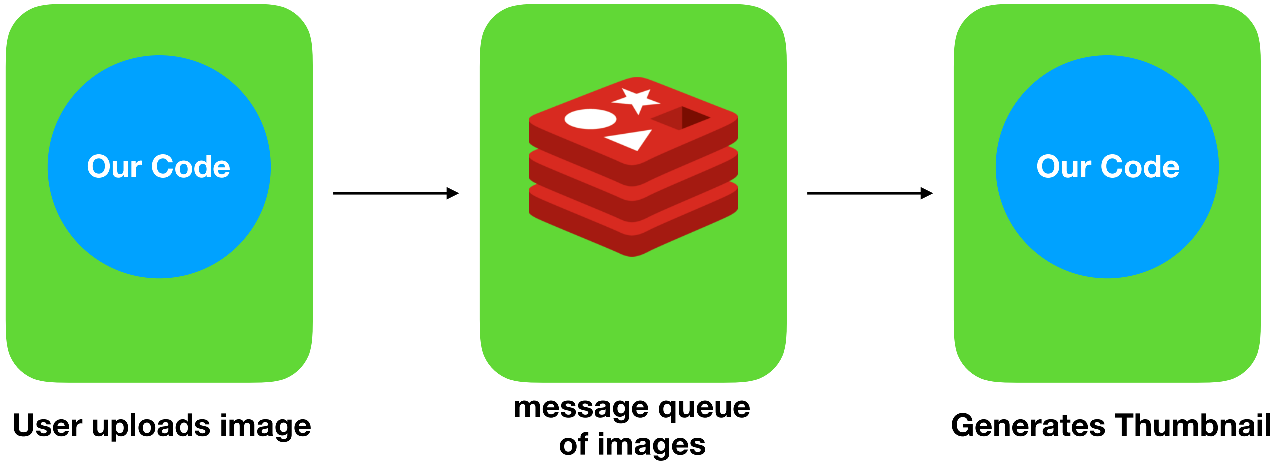


User uploads image

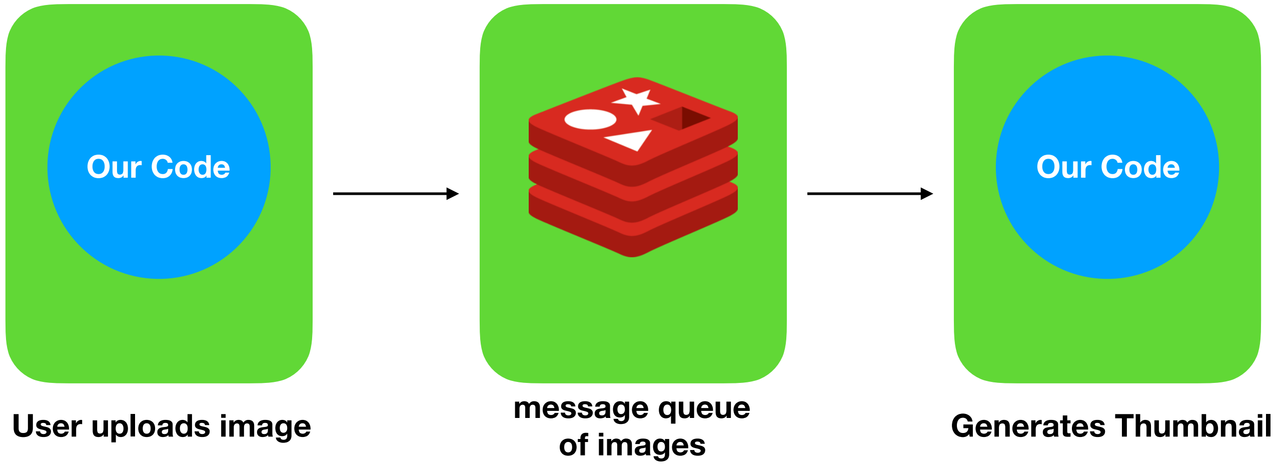
Event-Driven Architecture



Event-Driven Architecture



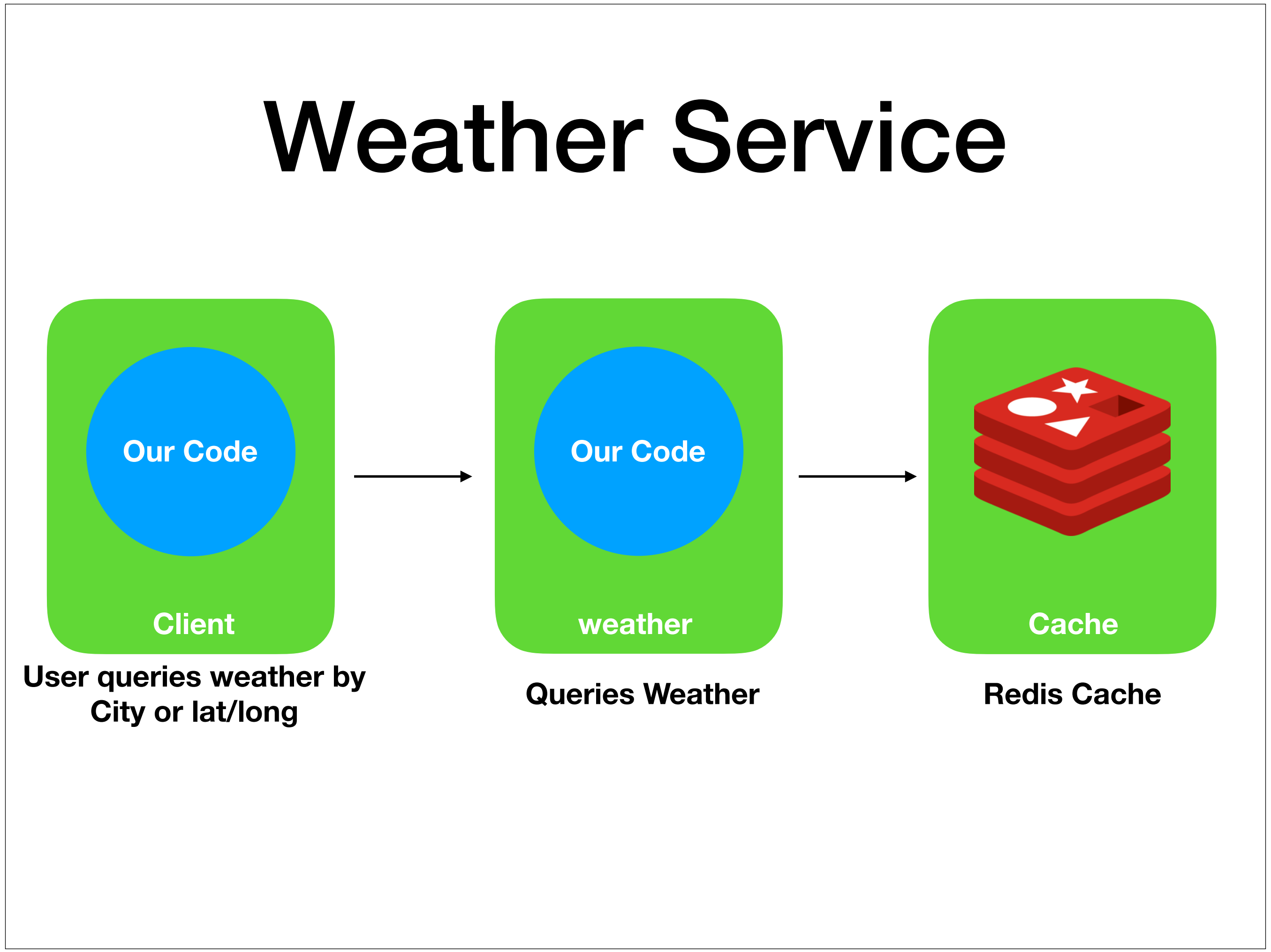
Event-Driven Architecture



- Our endpoint (uploading images) does less work
- Return to client faster
- More transactions/second for users
- Could scale the ‘generate thumbnail’ code

Umbrella Example

Weather service



Weather Service

Our Code

Client

User queries weather by City or lat/long

Our Code

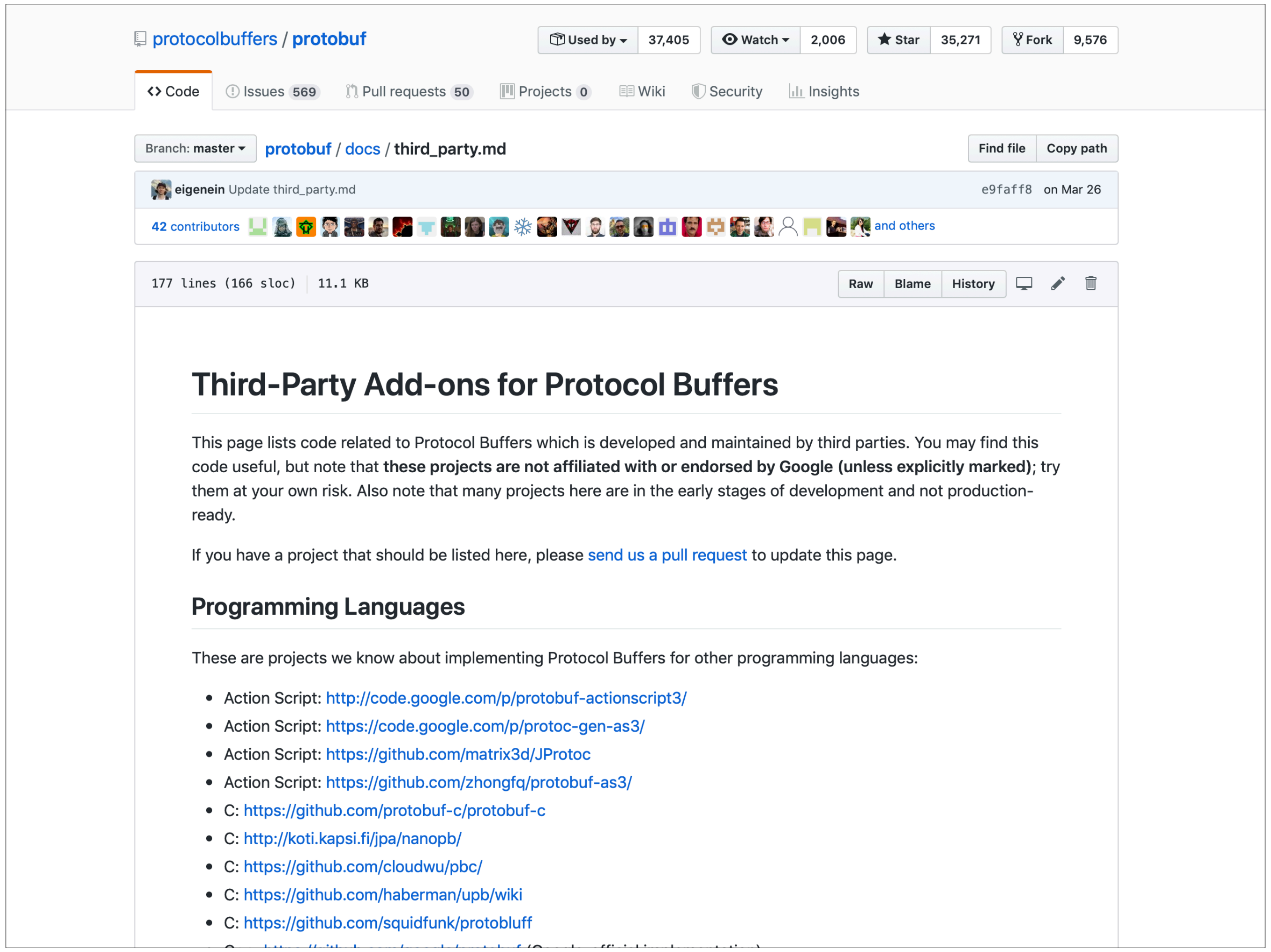
weather

Queries Weather

Cache

Redis Cache

- Do a demo! - PyCharm Presentation Mode?



Third-Party Add-ons for Protocol Buffers

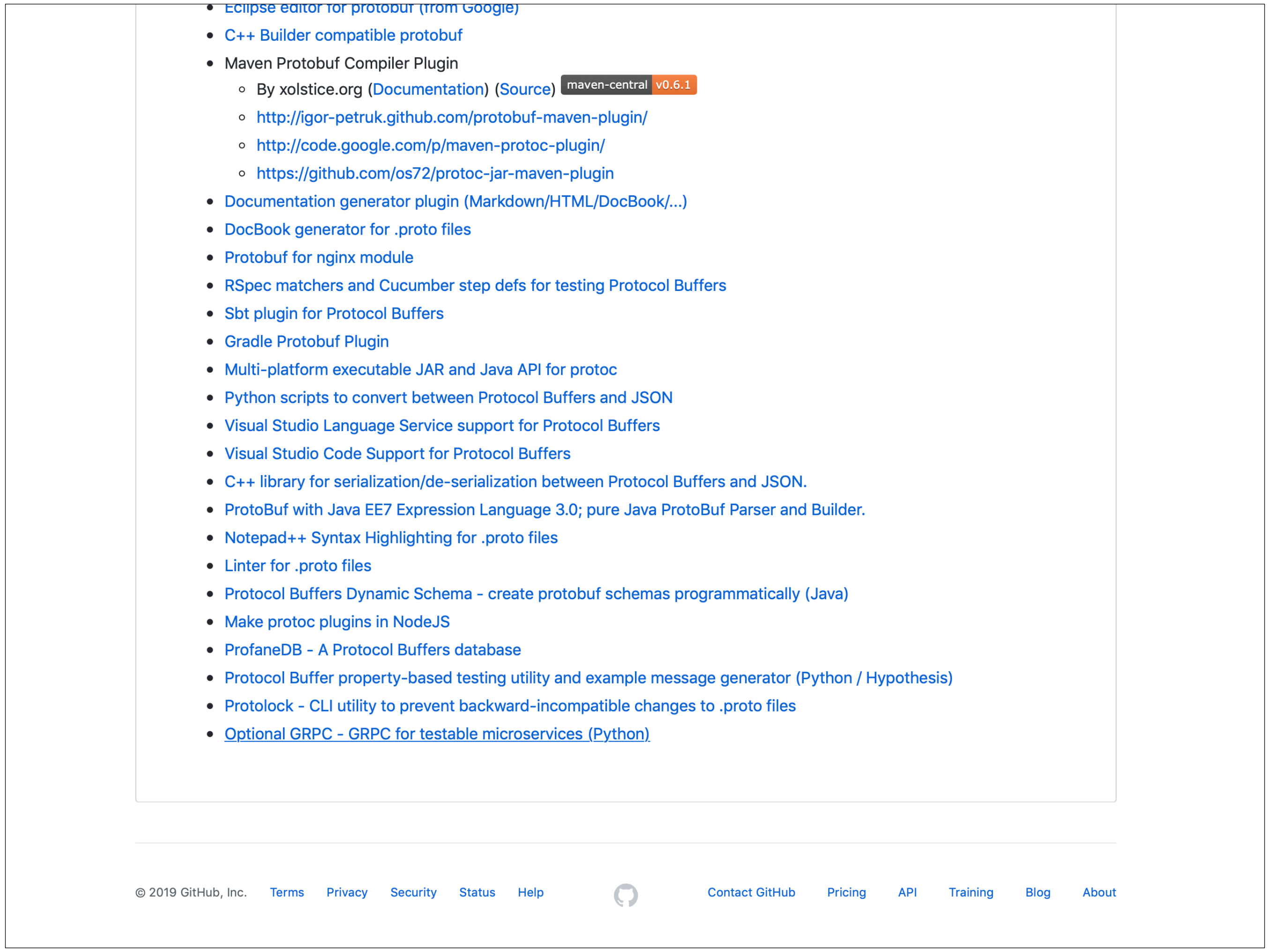
This page lists code related to Protocol Buffers which is developed and maintained by third parties. You may find this code useful, but note that **these projects are not affiliated with or endorsed by Google (unless explicitly marked)**; try them at your own risk. Also note that many projects here are in the early stages of development and not production-ready.

If you have a project that should be listed here, please send us a pull request to update this page.

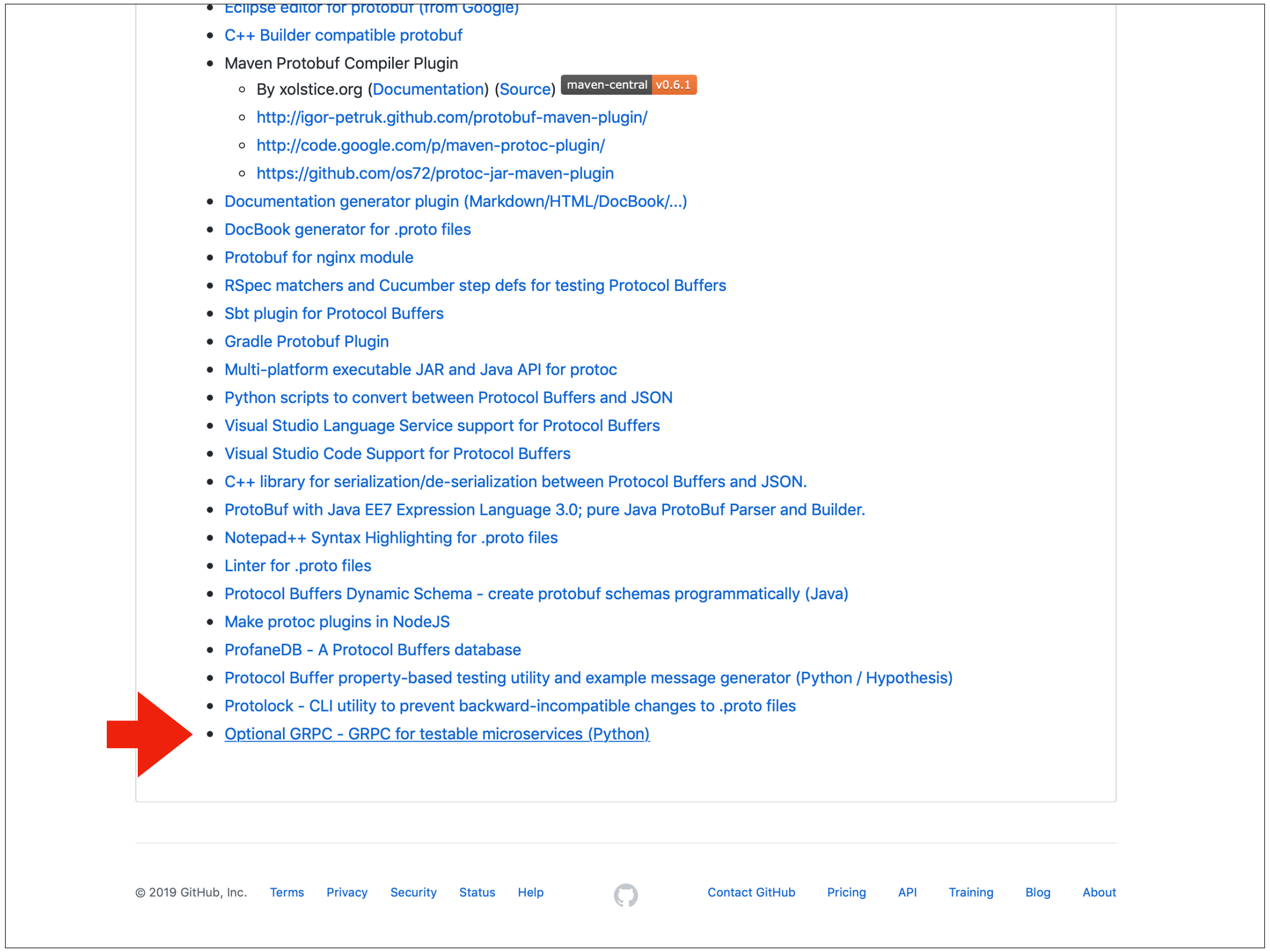
Programming Languages

These are projects we know about implementing Protocol Buffers for other programming languages:

- Action Script: http://code.google.com/p/protobuf-actionscript3/
- Action Script: https://code.google.com/p/protoc-gen-as3/
- Action Script: https://github.com/matrix3d/JProtoc
- Action Script: https://github.com/zhongfq/protobuf-as3/
- C: https://github.com/protobuf-c/protobuf-c
- C: http://koti.kapsi.fi/jpa/nanopb/
- C: https://github.com/cloudwu/pbc/
- C: https://github.com/haberman/upb/wiki
- C: https://github.com/squidfunk/protobluff
- C++: https://github.com/google/protobuf (Google-official implementation)



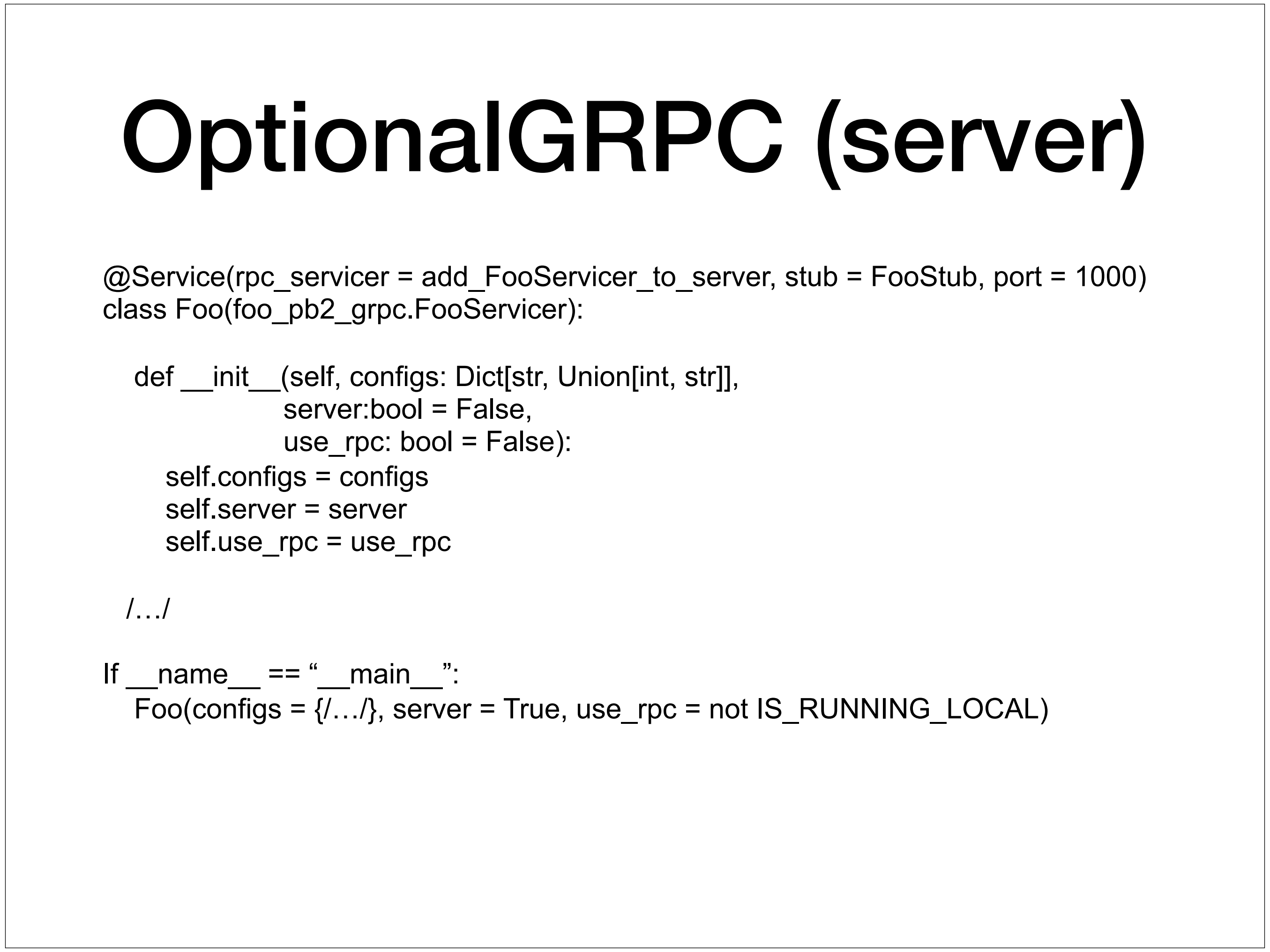
- Eclipse editor for protobuf (from Google)
- C++ Builder compatible protobuf
- Maven Protobuf Compiler Plugin
  - By xolstice.org (Documentation) (Source) maven-central v0.6.1
  - http://igor-petruk.github.com/protobuf-maven-plugin/
  - http://code.google.com/p/maven-protoc-plugin/
  - https://github.com/os72/protoc-jar-maven-plugin
- Documentation generator plugin (Markdown/HTML/DocBook/...)
- DocBook generator for .proto files
- Protobuf for nginx module
- RSpec matchers and Cucumber step defs for testing Protocol Buffers
- Sbt plugin for Protocol Buffers
- Gradle Protobuf Plugin
- Multi-platform executable JAR and Java API for protoc
- Python scripts to convert between Protocol Buffers and JSON
- Visual Studio Language Service support for Protocol Buffers
- Visual Studio Code Support for Protocol Buffers
- C++ library for serialization/de-serialization between Protocol Buffers and JSON.
- ProtoBuf with Java EE7 Expression Language 3.0; pure Java ProtoBuf Parser and Builder.
- Notepad++ Syntax Highlighting for .proto files
- Linter for .proto files
- Protocol Buffers Dynamic Schema - create protobuf schemas programmatically (Java)
- Make protoc plugins in NodeJS
- ProfaneDB - A Protocol Buffers database
- Protocol Buffer property-based testing utility and example message generator (Python / Hypothesis)
- Protolock - CLI utility to prevent backward-incompatible changes to .proto files
- Optional GRPC - GRPC for testable microservices (Python)



- Eclipse editor for protobuf (from Google)
- C++ Builder compatible protobuf
- Maven Protobuf Compiler Plugin
  - By xolstice.org (Documentation) (Source) maven-central v0.6.1
  - http://igor-petruk.github.com/protobuf-maven-plugin/
  - http://code.google.com/p/maven-protoc-plugin/
  - https://github.com/os72/protoc-jar-maven-plugin
- Documentation generator plugin (Markdown/HTML/DocBook/...)
- DocBook generator for .proto files
- Protobuf for nginx module
- RSpec matchers and Cucumber step defs for testing Protocol Buffers
- Sbt plugin for Protocol Buffers
- Gradle Protobuf Plugin
- Multi-platform executable JAR and Java API for protoc
- Python scripts to convert between Protocol Buffers and JSON
- Visual Studio Language Service support for Protocol Buffers
- Visual Studio Code Support for Protocol Buffers
- C++ library for serialization/de-serialization between Protocol Buffers and JSON.
- ProtoBuf with Java EE7 Expression Language 3.0; pure Java ProtoBuf Parser and Builder.
- Notepad++ Syntax Highlighting for .proto files
- Linter for .proto files
- Protocol Buffers Dynamic Schema - create protobuf schemas programmatically (Java)
- Make protoc plugins in NodeJS
- ProfaneDB - A Protocol Buffers database
- Protocol Buffer property-based testing utility and example message generator (Python / Hypothesis)
- Protolock - CLI utility to prevent backward-incompatible changes to .proto files
- Optional GRPC - GRPC for testable microservices (Python)

Optional-GRPC

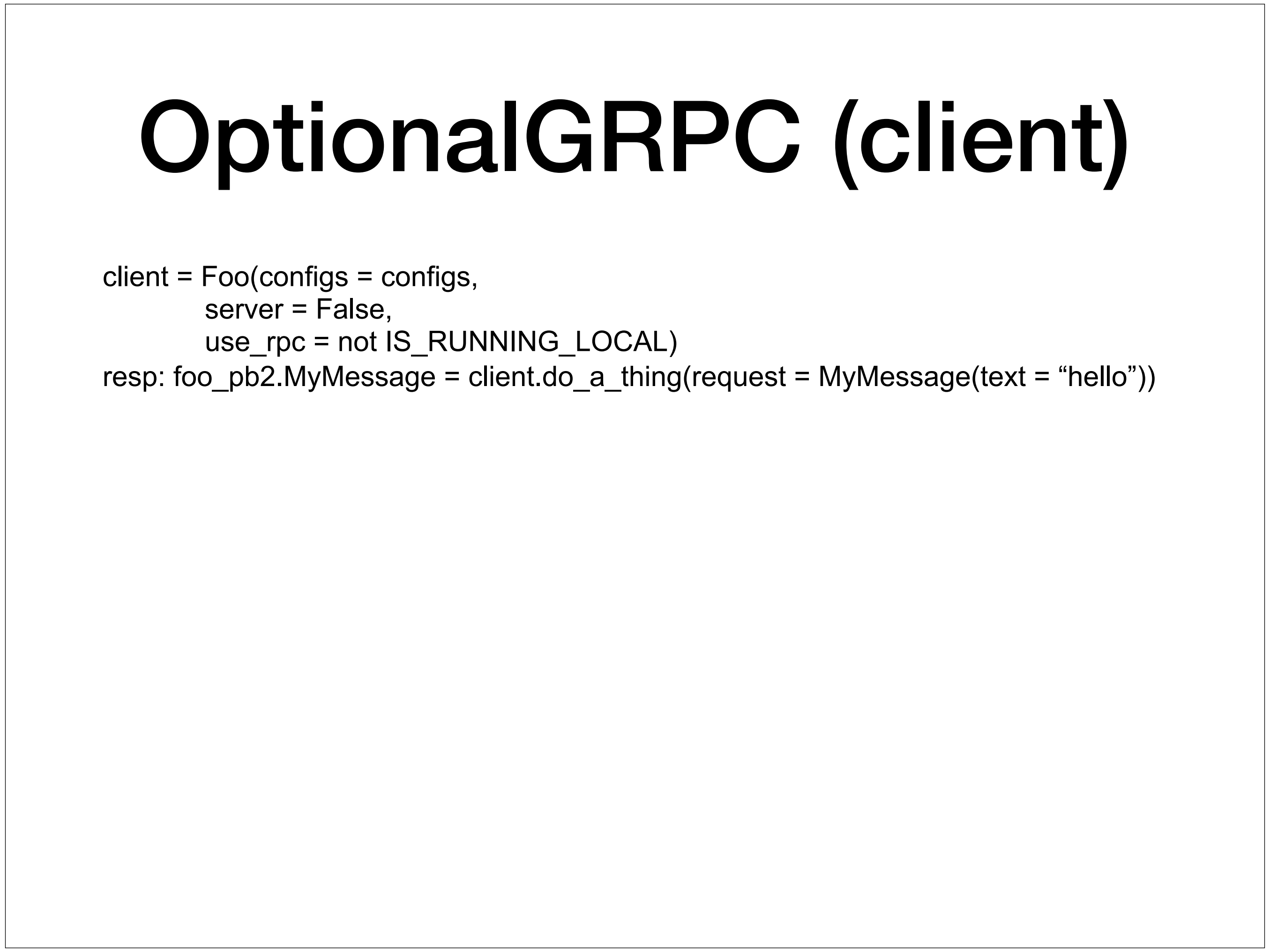
As a wrapper - if you want less code



OptionalGRPC (server)

```
@Service(rpc_servicer = add_FooServicer_to_server, stub = FooStub, port = 1000)
class Foo(foo_pb2_grpc.FooServicer):

    def __init__(self, configs: Dict[str, Union[int, str]],
                       server:bool = False,
                       use_rpc: bool = False):
        self.configs = configs
        self.server = server
        self.use_rpc = use_rpc

   /…/

If __name__ == “__main__”:
    Foo(configs = {/…/}, server = True, use_rpc = not IS_RUNNING_LOCAL)
```

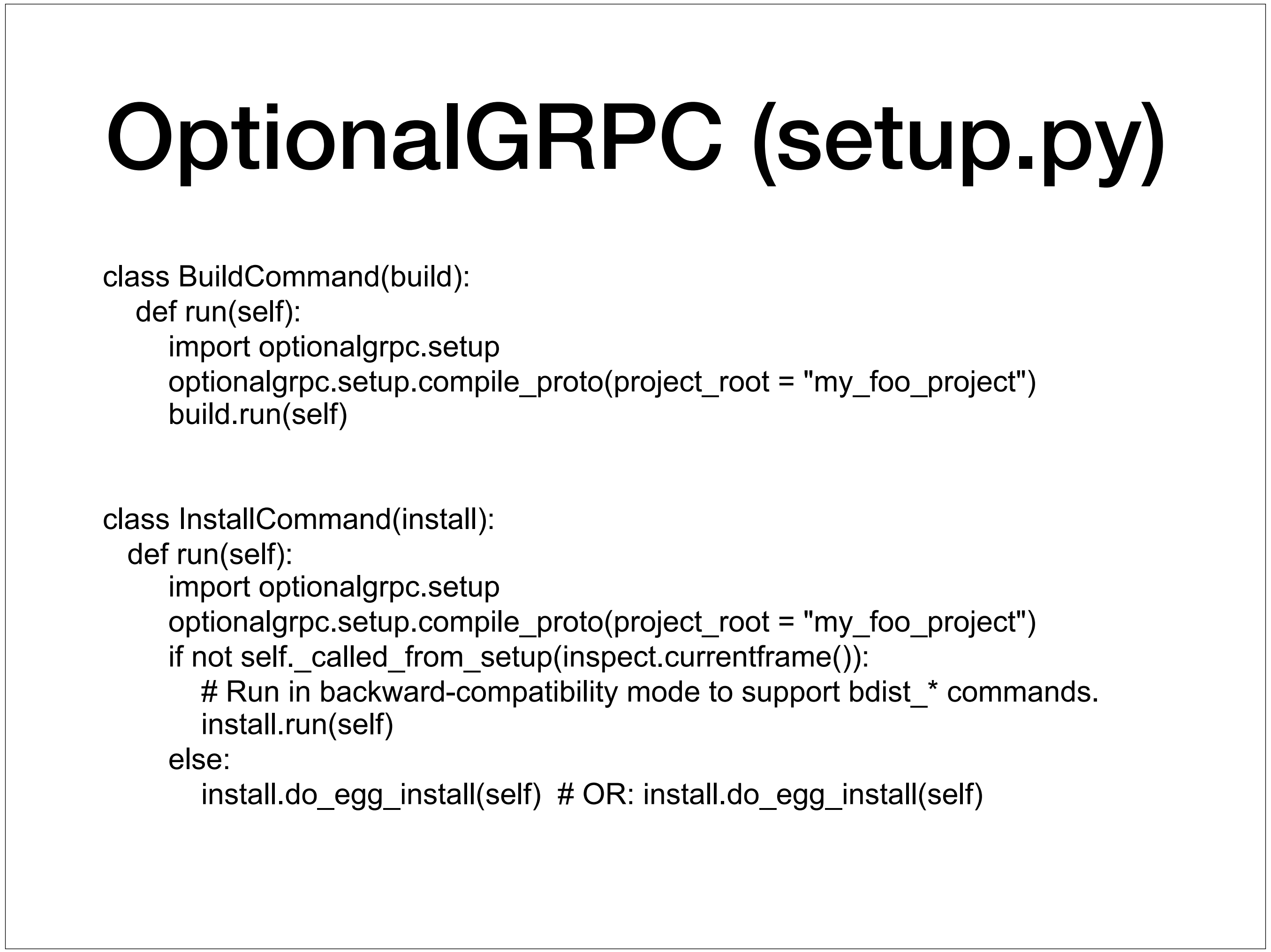
- Save yourself the boilerplate code for the server!



OptionalGRPC (client)

```
client = Foo(configs = configs,
            server = False,
            use_rpc = not IS_RUNNING_LOCAL)
resp: foo_pb2.MyMessage = client.do_a_thing(request = MyMessage(text = “hello”))
```

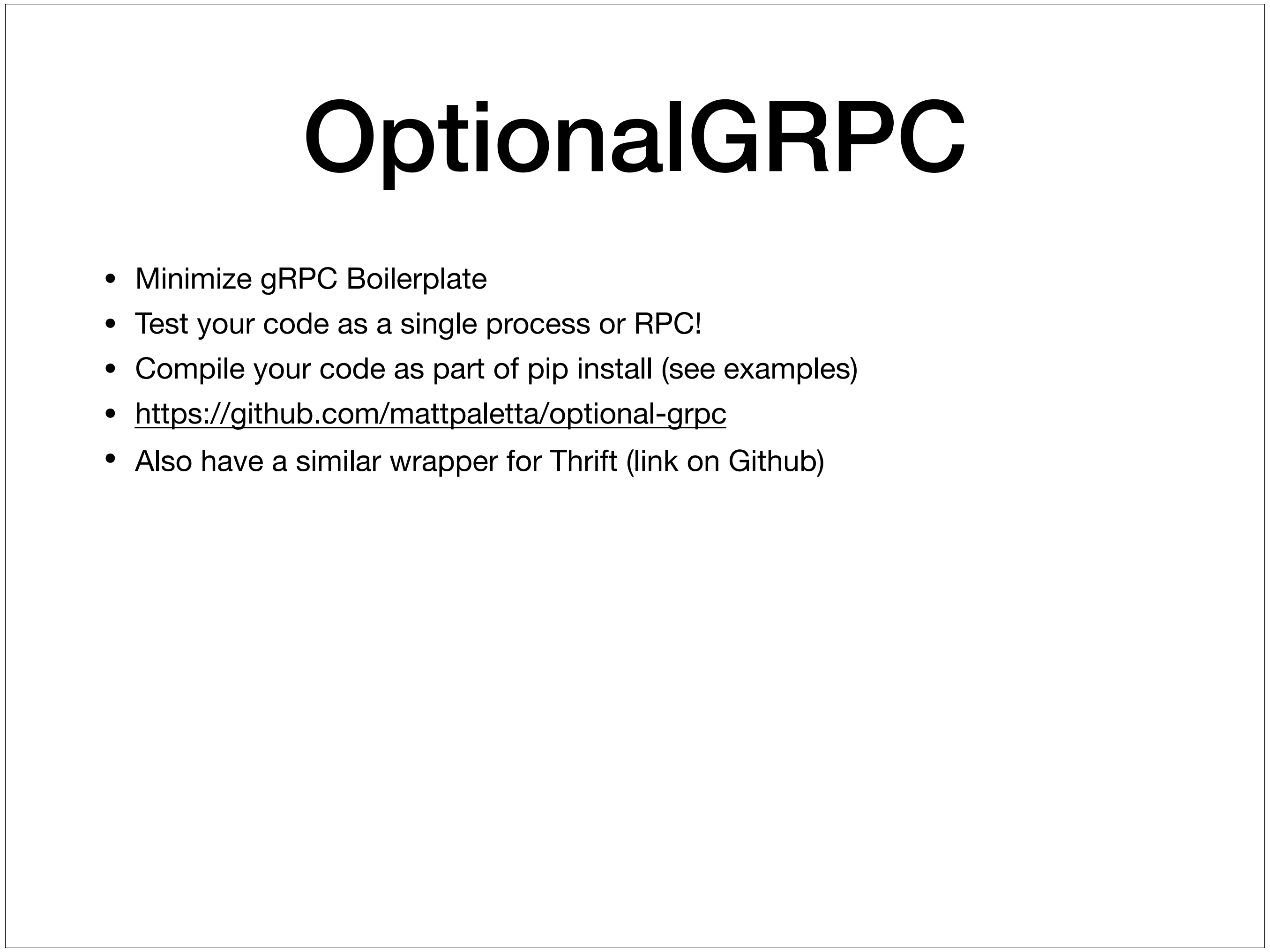
- Save yourself the boilerplate code for the server!



OptionalGRPC (setup.py)

```
class BuildCommand(build):
    def run(self):
        import optionalgrpc.setup
        optionalgrpc.setup.compile_proto(project_root = "my_foo_project")
        build.run(self)


class InstallCommand(install):
   def run(self):
        import optionalgrpc.setup
        optionalgrpc.setup.compile_proto(project_root = "my_foo_project")
        if not self._called_from_setup(inspect.currentframe()):
            # Run in backward-compatibility mode to support bdist_* commands.
            install.run(self)
        else:
            install.do_egg_install(self)  # OR: install.do_egg_install(self)
```

- Compile protobufs as part of pip install!



OptionalGRPC

- Minimize gRPC Boilerplate
- Test your code as a single process or RPC!
- Compile your code as part of pip install (see examples)
- https://github.com/mattpaletta/optional-grpc
- Also have a similar wrapper for Thrift (link on Github)

- Save yourself the boilerplate code for the server!

github.com/mattpaletta/distributed-docker-example

- https://docs.docker.com/engine/reference/builder/
- https://docs.docker.com/compose/compose-file/
- https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- https://training.play-with-docker.com/alacart/

Your Turn!

Questions, Comments, Concerns, Queries, Qwibbles?