Journey of adopting Elixir

@gzzengwei

Before we start

- About Me

- About this talk

- Disclaimer

Background

COMPANY:

- Small ruby shop
- Multi tenancy
- Heavy 3rd-party integrations & data exchange

TECH STACK:

- Rails for web
- Customized background jobs(ruby/node)
- Just finished transition to kubernetes

First blood

- introduce by ex CTO
- prototype of background job dashboard
- phoenix (1.2) + elm (0.18)
- workers fetching job status from multi tenants under supervisor
- channels push update to frontend

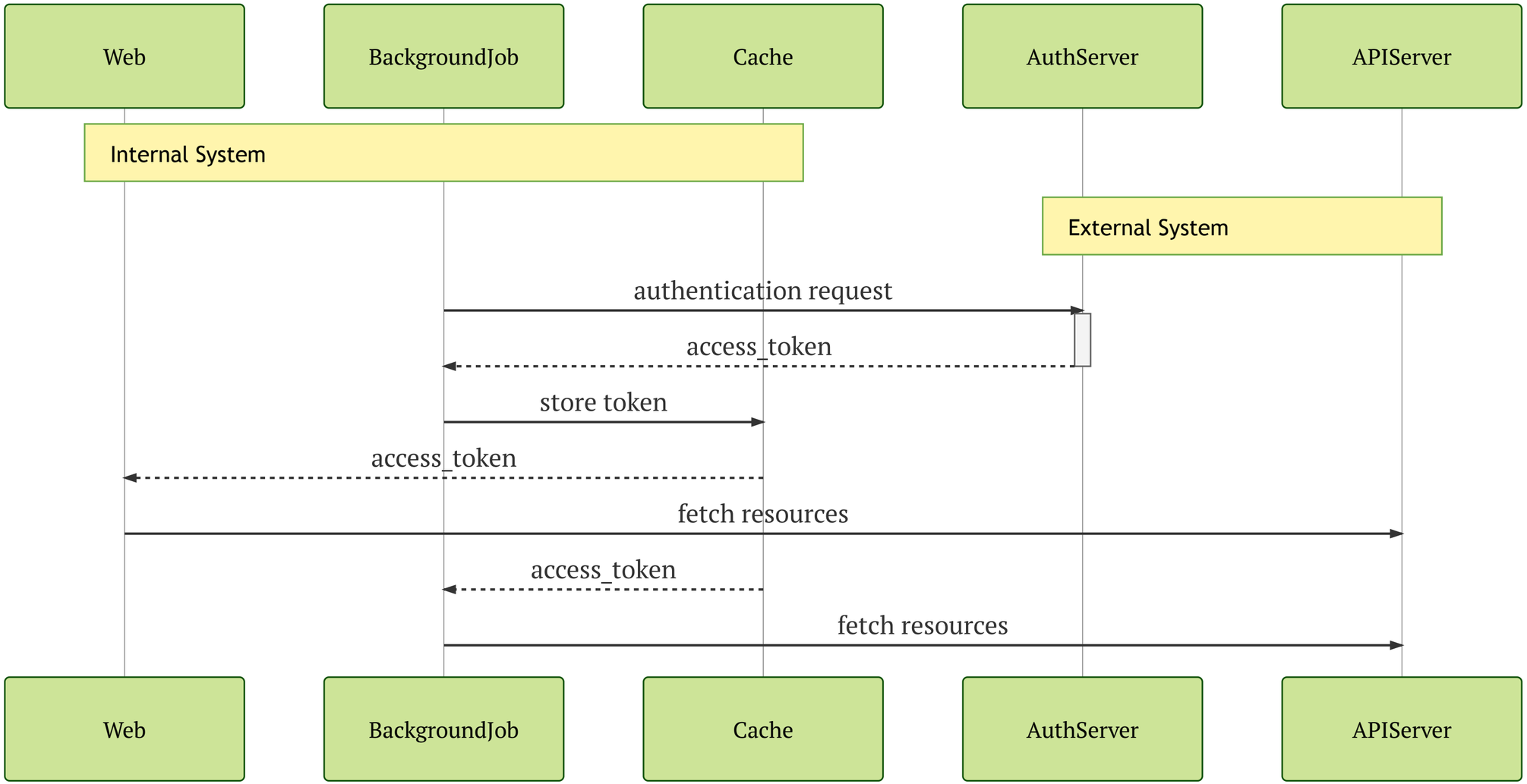
Moving forward

NEW OAUTH2 INTEGRATION

- access_token to be refreshed over half of its life-cycle
- both web and background-job servers need to connect to API

Stack consideration 1

WITHIN RAILS CODE BASE DIRECTLY



WITHIN RAILS CODE BASE DIRECTLY

Pros:

- straight forward
- single change will works for both web/background-job serivces

Cons:

- need cache/db to share token between web/job servers
- additional job to fetch token
- tightly coupled, lack of abiltiy to extend other services

Extracting service #1

AUTHENTICATION PROXY

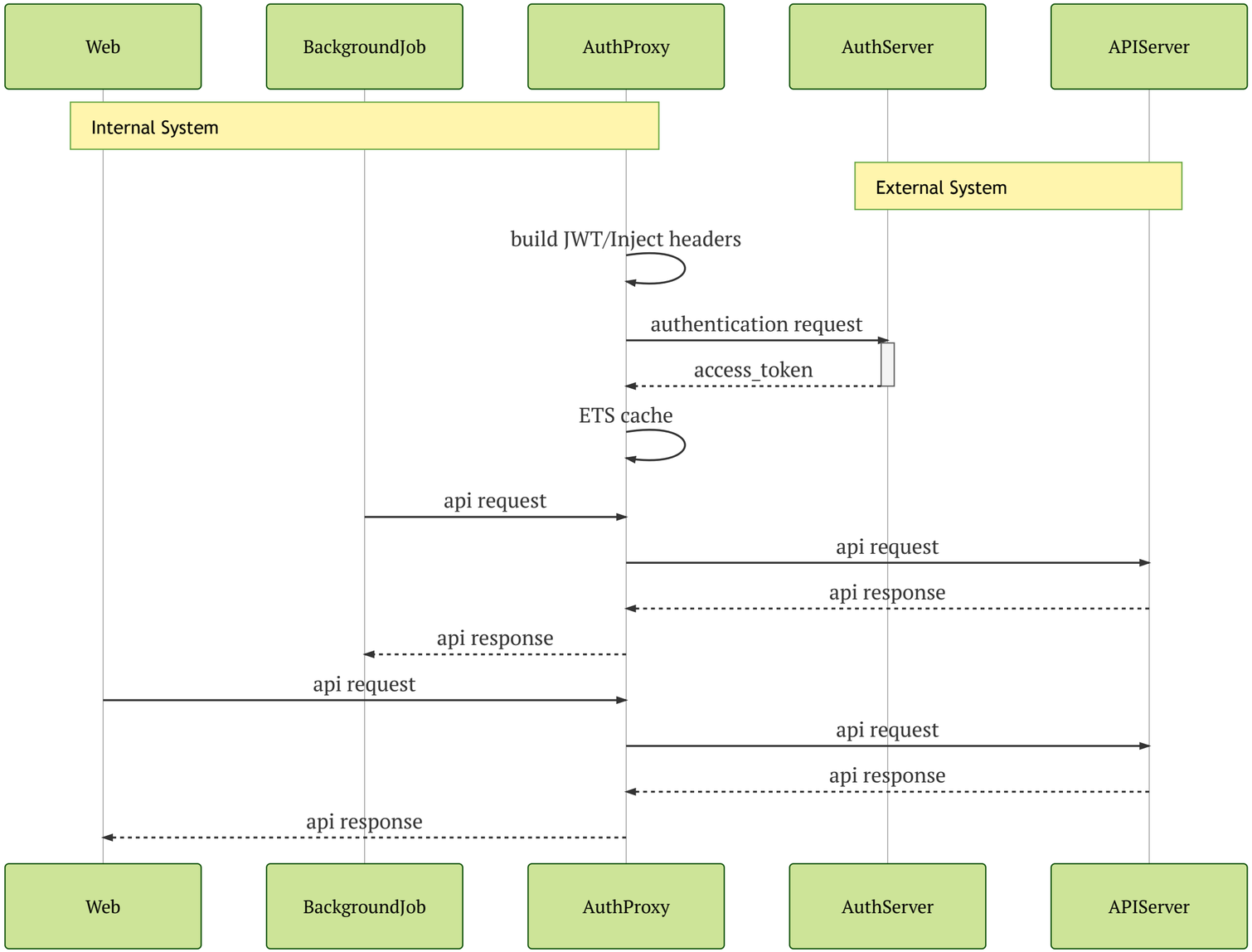
- handle the authentication
- within cluster network, no external ingress needed
- serve web/background-job and potential more services

DIRECT ACCESS FOR CALLERS(WEB/BACKGROUND-JOB)

```
## from direct access with `access token` in headers
https://api.third-party.com/resources/:resource_id


## to local k8s cluster access without token
http://auth-proxy.my-namespace.svc.cluster.local/resources/:resource_id
```

Extracting service #2



Extracting service #3

FETCH TOKEN WITH INTERVAL

```
defmodule MyApp.Scheduler.Worker do
  use GenServer

  @default_interval_seconds 1000 * 60 * 50

  def handle_info(:work, state) do
    MyApp.Auth.fetch_token()
    schedule_work()
    {:noreply, state}
  end

  defp schedule_work() do
    Process.send_after(self(), :work, @default_interval_seconds)
  end
end
```

TOKEN SAVED IN CACHE(ETS) WITH EXPIRY

```
  Cachex.put(:token, client_id(), token, ttl: :timer.seconds(cache_ttl_mins()))
```

Extracting service #4

- it just works! and fast
- can be mantained and deploy separately
- itself is extendable, like caching most frequently/large calls
- low resources consumption
- extendable by other comsumers

Goging futher #1

- one of the endpoint provides raw meta data
- BI for data warehouse is planned
- And Importer/AdminGUI service is needed

NEW IMPORTER SERVICE

- Talks to the AuthProxy endpoint to fetch data
- process up to 50+ types of CSV daily across tenants
- the stack we choose

Goging futher #2

- FP feels nature while processing data flow

```
def call(%{module_name: module_name, data: data}) do
  data
  |> Base.decode64!()
  |> unzip_content
  |> import_csv(module_name)
  |> delete_temp_file()
end

def import_csv(file_name, module_name) do
  file_name
  |> load_csv(module_name)
  |> Stream.chunk_every(@chunk_size)
  |> Enum.each(&process_rows(&1, module_name))

  file_name
end

def load_csv(file_name, module_name) do
  file_name
  |> Path.expand(File.cwd!())
  |> File.stream!()
  |> CSV.decode(headers: true)
end
```

Goging futher #3

- `Quantium` cronjob to trigger import
- GenSever worker handle trigger
- move to `Task` to avoid blocking
- use `Torch` for admin GUI

Goging futher #4

NATIVE CSV LIB

- Stream func by default
- Process per line and lazy evaluation
- Low memory usage

TIPS/TRICKS #1

- Stream with `chunk_every`
- 2 mins with `Stream.map` + `Ecto.Repo.insert`
- less than 30 seconds with `chunk_every` and `Ecto.Repo.import_all`
- few testing to decide what `chunk_every` value for the case

Kubernetes Basics

- Summary: Automating deployment, scaling, and management of containerized applications.
- Pod: Smallest deployable units of computing that you can create and manage.
- Deployment: A Deployment provides declarative updates for Pods and ReplicaSets.
- Jobs: A Job creates one or more Pods and will continue to retry execution of the Pods until successfully terminate
- Liveness/Readiness Probe: Indicates whether the container is running/ready

Tips/Tricks #2

HEALTH CHECK ISSUE

- k8s liveness/readness probe to check service is alive/ready
- query every x seconds

```
livenessProbe:
  httpGet:
    path: /health_check
    port: http
  initialDelaySeconds: 30
  timeoutSeconds: 10
  periodSeconds: 30
```

Tips/Tricks #2

HEALTH CHECK ISSUE

- define in controller

```
defmodule MyAppWeb.HealthCheckController do
  use MyAppWeb, :controller

  def index(conn, _params) do
    conn
    |> send_resp(200, "ok")
    |> halt()
  end
end
```

- it will generate lots of noise in the logs

```
2021-07-20 09:35:20.778 request_id=FpN2FHzsg-UyTt0APj2B [info] GET /health_check
2021-07-20 09:35:20.935 request_id=FpN2FHjEwLMoo8oAPj0B [info] Sent 200 in 227ms
```

and `log` option in route not working

```
get "/health_check", HealthCheckController, :index, log: false
```

Tips/Tricks #2

BYPASS REQUEST LOGGING

- define endpoint

```
defmodule MyAppWeb.Plug.HealthCheck do
  import Plug.Conn
  def init(opts), do: opts
  def call(%Plug.Conn{request_path: "/health_check"} = conn, _opts) do
    conn
    |> send_resp(200, "ok")
    |> halt()
  end
  def call(conn, _opts), do: conn
end
```

and inject it in Endpoint

```
defmodule MyAppWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :my_app

  plug(MyAppWeb.Plug.HealthCheck)
```

Tips/Tricks #3

MIGRATIONS

First attempt

```
# Dockerfile
CMD [mix ecto.migrate && mix phx.server]
```

Moves from deployment to k8s Job

```
apiVersion: batch/v1
kind: Job
spec:
  containers:
    command:
      - "/app/bin/my_app"
    args:
      - "eval"
      - "MyApp.Release.migrate"
```

Tips/Tricks #3

MIGRATIONS TO K8S JOB

- separate pod(process) in term of deploy life-cycle

  different log / metric/ exception aggregator

- different image:

  different system libs added

- different resource spec:

  less resouce for migration while more for production web server

Mix Release

- starts with just mix ecto.migrate/phx.server
- taking too long to compile during production deployment
- move to mix release, including migration to mix task
- pod service up time from 30+sec -> 5sec

Multistage Dockerfile

- single stage Dockerfile, slow CI pipelines
- split stages to elixir, assets, release
- CI build time 2+mins -> 20-30sec
- Docker image size: 60MB -> less than 20MB

performance & resources

with low resources spec

```
resources:
  limits:
    cpu: 1000m
    memory: 1Gi
  requests:
    cpu: 250m
    memory: 512Mi
```

process still working even over the CPU limit thanks to BEAM scheduler’s extreamly fast CPU context switching

PERFORMANCE & RESOURCES DIAGRAM

Table 1.1 Comparison of technologies used in two real-life web servers

Technical requirement	Server A	Server B
HTTP server	Nginx and Phusion Passenger	Erlang
Request processing	Ruby on Rails	Erlang
Long-running requests	Go	Erlang
Server-wide state	Redis	Erlang
Persistable data	Redis and MongoDB	Erlang
Background jobs	Cron, Bash scripts, and Ruby	Erlang
Service crash recovery	Upstart	Erlang

- from `Elixir In Action` by *Sasa Juric*

Overall

- reasonable learing curve (for ruby dev)
- fast develop with framework(phoenix)
- High performance
- Low resource consumption
- Confidence with BEAM OTP

Thank You

Questions ?