

---

# Deploying services at Lemon

2022-05-30

## Objective

The goal of this document is to present how developers at Lemon will deploy new services in the near future. We strive to implement a flow that minimizes the effort necessary from a developer perspective to go from 0 to a service running in production. We plan to do this by a combination of a tool, provided by the infra team, and a set of conventions. In the end developers should be able to go from 0 to service with a variation of commands similar to:

```
lemi new service -name=svc-name -envs=pre,prod -domain=svc.lemon.me
git add . && git commit -m "Deploy new service"
git push
```

We understand that there are many technical constraints and concerns that have to be taken into account when deploying services and this document tackles only those that we thought were relevant given the reality Lemon is currently facing, which is:

*We are a small team that will not grow significantly in size in the next two years but will be tackling relevant projects and we believe that a services oriented architecture is the next step forward.*

For this reason, we believe it makes sense to focus on building the tools necessary to make teams productive for the short to medium term. Nevertheless, we've documented [here](#) which are those things that we are leaving pending to be solved in the future and what our action plan is for each.

## Proposal

We start by separating **Static** from **Dynamic Infrastructure**. Static infrastructure are dependencies such as databases, caches, VPN/VPCs, S3 buckets and others that require more explicit provisioning. We plan to implement terraform and a single infrastructure repository that will hold all static infrastructure. The provisioning of these resources will be done through a PR to this central repository. This will allow us to guarantee that:

- Our Infrastructure is always reproducible,
- All changes made to it are tracked through git and IaC,
- Ownership of resources will be explicit and clear since the creation of said resources will start by a developer requesting it,
- Everything created has the correct tags to facilitate observability and cost ownership.

The creation of secrets will be handled separately and it is still TBD.

With the provisioning of those dependencies out of the way what is left is to deploy the Dynamic Infrastructure, this is the service itself and everything necessary to reach it from other services and/or the internet. This includes:

- **Deployment:** Creating the Service in the ECS clusters corresponding to each environment. This service will be created with resource parameters (CPU & memory), environment variables (that can come from secrets) and auto scaling policies that will be controlled by the owners of the service.
- **Discoverability:** A service will be mapped into the DNS following a convention based on the repository+branch name. Avoiding an eventual collision issue.
- **Internal Routing:** Routing rule in the corresponding Internal Application Load Balancer that can be used within the VPC to access the service.
- **External Routing:** If the service requires access from the internet, create a routing rule in the internet facing Application Load Balancer that maps to a domain specified by the team.
- **Security Access:** A Security Group called `svcname-clients` that other resources can attach themselves to to be able to communicate with this service. Attaching the service to the necessary Security Groups to access the required dependencies (such as other services and/or databases).
- **Observability:** Datadog Dashboard dedicated to the observability of the service and all of its dependencies required to operate.

All that is mentioned above essentially lists everything developers have to provision themselves when creating a service. To remove this burden from them, we plan to provide an abstraction through a YAML specification file that abstracts away all this complexity required to put a service in production. This file will be stored within the repository in a folder called `deploy`. The specification of this file will be owned by the infra team. To generate this file developers will have to execute a CLI that the infra team will provide, this CLI will receive as input:

- **Service Name:** by convention it has to start with the name of the repository.
- **Service dependencies:** list of the names of services that this service communicates with.
- **Environments:** list of environments the service needs, only available values are those that the infra team supports (pre, prod and qa at the moment). However, this could evolve into something richer that provides brand new environments with the creation of a simple branch.
- **DB/Cache dependencies:** names of the databases (RDS) or caches (elasticache) that the service communicates with. Other services could also be supported.
- **External DNS if required:** specify whether the service requires outside access.
- **Memory and CPU requirements:** using AWS's format developers have to specify what requirements they have for these resources.
- **Auto-Scaling policies:** we will provide a wrapper for AWS's standard way of auto scaling, which for now is either based on a memory or CPU and a threshold.

Once it is executed, the CLI is in charge of:

- Generating files for dependency management (if not already present), this will be using `gradle`.
- Generating a Dockerfile for packaging (if not already present).
- Creating a PR on the central terraform repository with the changes necessary to provision all of the above.
- Generating the YAML file with the inputs that were specified in the command.
- Generating per environment configuration files to define environment variables and secrets.
- Instructing developers to fill in the YAML file with whatever else is necessary, for example: environment variables.
- Generating the `.github/workflows` file with all of the building, testing and deploying workflows already pre configured to point to the correct ECR and ECS. With the ability to perform rollbacks.

After finishing these steps and pushing the corresponding files to GitHub developers should have a service ready to be deployed.

All of the above is based on a series of **conventions** that teams will have to follow for their services. These are:

- **Communication:** services will expose an *HTTP API on port 8080* that will be the main entry point.
- **Healthcheck:** all services have to expose a `/health` endpoint on the same API that responds with a 200 when the service is healthy and any other code when it is not. This endpoint has to check all necessary dependencies for the service to function (e.g a database might be required but a cache might not).
- **Observability:** all services will implement OpenMetrics, they will expose an *API on port 9090* with an OpenMetrics endpoint that lists all relevant metrics the service needs to track.
- **On Call & Service Ownership:** each repository will have an OWNERS file with the list of GitHub users that are the owners of this service and are responsible for operating it in production.
- **Packaging:** all services will be packaged with Docker using a standard Docker Image provided by the infrastructure team.
- **Testing:** all services will use Gradle for dependency management and will have to implement three commands (the CLI will use the entry points to this commands):
  - `./gradlew test`: unit and other simple tests
  - `./gradlew integrationTest`: integration tests.
  - `./gradlew e2eTest`: end to end tests.
- **Building:** jars will be built using gradle and the service has to provide a command called `./gradlew build` (the CLI will provide the entry points to this commands)

In [this section](#) we will go into the details of how each of these steps will be implemented and what tools the infrastructure team has to provide.

## Implementation

In this section we will explain in detail how the proposal will be implemented and how each step will work both from a developer and infrastructure point of view.

We start with the provisioning of **Static Infrastructure**. This includes: Databases, Caches, VPNs / VPCs, Lambdas, S3 buckets and anything else that can be considered part of provisioning but that will not change with the deployments that teams do. This process will have a manual component for now. Once a team defines what requirements they have for static infrastructure they can go directly to our terraform repository and create a PR that provisions the requirements they have. We will provide terraform modules that will facilitate the amount of tf code devs have to write and will abstract our basic stacks (such as an app with an RDS database using mysql). This PR can eventually be generated using the CLI that was already mentioned above, but we believe it's easier to start by having them come to this repo so that we can see more directly what use cases we have to support. The naming of all these resources needs to follow a strict convention, all resources need to have the name of the service prepend. For example:

- **Databases:** bpg-aurora, bpg-rds
- **VPC/VPN:** bpg-coelsa-vpc
- **S3 buckets:** bpg-<purpose>
- **Lambdas:** bpg-<function-name>

While the process above is being taken care of, developers can start with the definition and provisioning of **Dynamic Infrastructure**. This is done with the execution of the CLI provided by the infra team. To more easily demonstrate how the process will work we will use [this service](#) that we build and maintain. First we execute the CLI specifying all the parameters mentioned above, this includes: Service Name, Service Dependencies, Database/Caches, environments, external DNS requirements:

```
lemoninfra init service -name=klausapproves -envs=pre,prod -external=true
-svc-dependencies=bpg
-datastore-dependencies=klausapproves-rds,klausapproves-cache -cpu=X
-memory=X -auto-scaling=resource=cpu,min=1,max=10
```

The command above will first create a PR to our terraform repository that will provision the routing rules for internal and external access following our conventions (such as: `svc-name.env.internal.lemon.me` and `svc-name.env.lemon.me`), create the Service in our ECS cluster for both pre and prod environments, add said service to the security group of `bpg-clients`, `klausapproves-rds-clients` and `klausapproves-elasticache-clients` and finally give all repository/service owners execution access to the containers of the service. It will then generate a YAML specification file, that leaves in a folder called `deploy`, that contains information about this service, this file will look similar to:

```
type: service
name: klausapproves
```

```

spec:
  image: klausapproves
  resources:
    cpu: X
    memory: X
  autoscaling:
    resource: cpu
    min: 1
    max: 10
  ingress:
    external: true
  dependencies:
    services:
      - bpg
    data:
      - klausapproves-rds
      - klausapproves-cache
  environment:
    - key: KLAUSAPPROVES_VARIABLE
      value: {{ .Variables["klausApproves"] }}
    - key: KLAUSAPPROVES_SECRET
      value: {{ .Variables["klausApiKey"] }}

```

In addition to this file, it will also generate two folders that will hold a configuration file to store per-env variables and secrets, example for prod:

```

variables:
  - key: klausApproves
    value: false
secrets:
  - key: klausApiKey
    value: secret_arn_prod.field_name

```

Finally the tool will generate a standard github workflow file that will implement the process of building, testing and continuously deploying the application to the different environments. For doing this the workflow will::

- Build and push a docker image tagged with the commit sha for merges done to develop.
- Build a docker image for new tags that follow the semver format will build a docker image tagged with said version.
- With the image specified, the pipeline will use our CLI and generate the corresponding `task-definition.json` that needs to be deployed. The task definition itself will almost always be identical, with the exception of the environment variables that are populated by said tool.
- Develop will be pre and semver tags will be production.

All of the above explains the workflow when creating a new service from scratch, however, it does not explain in detail how changes to this infrastructure will be made once the service is already created. We document here a summary of what should be done for each type of infrastructure:

- **Static Infrastructure:** this will always go through our terraform repository. Changes such as a modification to the VPC, upscaling a database or a redis cluster will require creating a new PR that modifies these resources.
- **Dynamic Infrastructure:** most of these changes can be controlled directly through the repository and they can be taken care of by simply modifying the YAML of the service. For example: changing the auto scaling policy or expanding the resources of the containers.

## What we are leaving for the future

Like it was mentioned previously, we are aware that there are many other technical constraints and concerns that have to be taken into account when deploying services that we have not covered in this document. This is intentional and it's due to the reality that Lemon is currently facing and the requirements it will have for the short to medium term. However, for transparency and accountability, we leave below a list of the topics that we are not covering, the reasoning for leaving out each one at this particular moment and when we plan to tackle them.

### AuthN & AuthZ

#### **Why?**

After our communications with the teams all the services that have to be launched in the short to medium term will be services used internally, meaning they will not be exposed to the internet. While having proper authentication & authorization with good role management is important, we believe we can tackle the security team once it's more clear what our requirements will be. For now we will trust on the infrastructure (private VPCs) and security groups.

#### **When?**

We don't have a specific timeline, but everything related to security is top priority to us and we plan to tackle this with the security team as soon as possible.

### Compatibility guarantees & controls

#### **Why?**

We believe this becomes problematic once the size of the organization has reached a point where Team A makes changes to Service A that Team B (owner of Service B) never hears about and problems happen in prod. For the short and mid term teams that own the services will be the same ones that implement the clients of that service, which means the possibility of this happening is reduced.

#### **When?**

We plan to observe the growth that the organization will have and act accordingly. If priorities change and Lemon grows significantly, we will re-prioritize.

### Standardized communication (REST vs gRPC)

#### **Why?**

Communication between services can be managed with HTTP for starters since we already have a few tools that facilitate this integration. If gRPC, or others, is a better approach we believe that a migration to that should be straightforward to do and could be executed in parallel. This is due to the fact that there are many tools available today that give us the possibility of having a dual stack (i.e HTTP and gRPC out of the box). This allows us to do a progressive migration to the new stack.

#### **When?**

This is the least of our priorities at the moment. We prefer to focus immediately on Observability and Developer Experience. However, [Compatibility guarantee & Controls](#) might be directly related to this so we might solve both in the same step.

### TLS Certificates and mTLS

#### **Why?**

Our approach to communication, both external and internal, will be to have two global Application Load Balancers with routing rules for each service based on the Host header and other application specific requirements. This means that TLS and mTLS will be offloaded to AWS for now.

#### **When?**

We plan to see how well offloading the responsibility to AWS works and act accordingly. If this doesn't work well, we can discuss alternatives and priorities at that time.

### Mono-repo vs multi-repo


#### **Why?**

Our infrastructure will be deployed using a mono-repo. From a developers perspective at the moment we will support the ability of creating multiple services within a single repository but they will also be able to deploy using multiple repositories.

#### **When?**

We will discuss from a tooling perspective what makes more sense in the coming months.

## Appendix

1-  Provisioning process flow