Global ​ ​DevOps ​ ​Tech ​ ​Test ​ ​1

I decided to use terraform and packer as tools for building the required infrastructure.

As I decided to create only one module, which will deploy the whole infrastructure at once due the simplicity of the task and to avoid the use of already developed modules. Another reason to follow this approach is, that I don’t see use case in which my code will be reusable for another use cases.

For most of the resources I followed well known best practices for developing terraform code, but for some I had to implement them in more simple way, due the language limitations (for example the web instances resources could be implemented with count argument, not one by one as I did it, but when I had to add the instances to alb target group, there is issue with this resource and terraform language itself.aws\_alb\_target\_group\_attachment resource can only provision a single target, where as one would expect to be able to define multiple targets. There are few workarounds, but I choose more simple way for this use case).

For complex projects, better approach is for each component to be developed different module, which will provide more flexibility, reusability, versioning of the code. For such projects good practices are unit and integration test. They could be created with Inspec and ServerSpec frameworks.

**Part** ​ ​**1:** ​ ​**Standard** ​ ​**Requirements. Details about the implementation**

* + 1. Fronted – This part consists of one Application Load Balancer, which forward requests to two Nginx web servers, which are deployed from custom prebuild AMI in public subnets. Such approach provides HA for the web tier. Auto Scale could be implemented as well.

The traffic is over HTTP, I decided to avoid use of custom SSL certification for this demo, but it could be implemented with few rows of code.

I implemented and health checks from ALB to the both instances.

* + 1. Backend – this part consists of one Postgresql Server, which is deployed from custom prebuild AMI. This instance could be accessed only from the two web servers. Its deployed in private subnet.

HA for the db tier could be implemented via different approaches ( RDS supports “Master – Slave “, storage replication and custom scripts) , but this requires more time for implementation and costs more.

Source code could be checked here:

Terraform code:

<https://github.com/pe6ev/fundingcircle>

Packer code:

<https://github.com/pe6ev/fundingcircle/tree/master/packer-nginx-ws>

<https://github.com/pe6ev/fundingcircle/tree/master/packer-postgresql-backend>

Architecture Diagram:

<https://github.com/pe6ev/fundingcircle/blob/master/FundCircle.jpg>

**Part** ​ ​**2:** ​ ​**Advanced** ​ ​**Requirements**

* + 1. Logging ​ ​Reference ​ ​Architecture
       - For currently deployed infrastructure, I would choose more simple solution and only AWS services.

S3 as storage for the logs:

* It’s cheap from cost perspective
* It’s with durable, highly scalable location to store log data
* long-term retention and archiving of log data
* support lifecycle rules

CloudWatch Logs for collection logs from the instances:

* The CloudWatch Logs agent can be easily installed and configured on Linux and Windows instances to send application and system log files to CloudWatch.
* Native integration with CloudWatch for monitoring and Auto Scale
* can subscribe to real-time CloudWatch Logs event feeds which they can either process themselves with Amazon Kinesis and AWS Lambda

ALB Logs:

* ALB support out of the box writing access logs to s3 bucket

For more complex projects there are a lot of different tools and approaches for logging and monitoring. Depends on budget, frameworks, the used programing languages, data size, team capacity, etc. …

ELK stack is one of the most used, its good choice for a lot of different reasons. Some of them are:

Elasticsearch handles storage and provides a RESTful search and analytics endpoint.

Kibana lets you visualize your Elasticsearch data and navigate the Elastic Stack.

There is open source version. Big community and designed with scalability in mind.

Here is link to the reference architecture and brief description:

<https://github.com/pe6ev/fundingcircle/blob/master/FundingCircleLoggingArch.jpg>

AWS Services from the reference architecture:

- ELB (Elastic Load Balancer)

- S3 bucket (log events source)

- One more S3 bucket (logging destination, could be the same bucket as source)

- CloudFront distribution

- Lambda (Python / boto3 library)

- RDS DB instance

- SQS

For some services as ELB(ALB), Cloud Front, S3, RDS -logs supports logging out of the box. They could be configured to store logs in S3 bucket.

RDS sends log files by default. For logging configuration switch the variables in instance parameter group. RDS is different. Log files in your database already single and prepared for analysis. All you need – download and store them locally connecting to RDS API vi Lambda function, not S3.

For EC2 logs could be sent to Logstash with Filebeat.

For CloudFront logging we need Lambda function, because the service sent logs in gzip format. The lambda will download them, unzip, write them in single file and sent them for analyze.

CloudTrial events could be delivered directly to S3.

SQS – its Highly Available, managed by AWS queue service, Lambda function that will read SQS queue once a minute, log records and put them in another SQS queue for later processing by consumer.

**Additional ​ ​Security ​ ​Considerations:**

Below are some of the possible security Considerations, which could be implemented in AWS.

- SSL certificates could be managed with KMS service

- Key rotation policy

- IAM roles/groups/profiles

- WAF service

- AWS Inspector for vulnerability scanning

- AWS System Manager/Patch Manager for patching instances

- Multi-Factor Authentication

- CloudTrial for API calls monitoring

- Rotate Credentials Regularly

**Disaster ​ ​Recovery ​ ​Reference ​ ​Architecture:**

Packer and Terraform are great tools for implementing DR strategy.

They are developed with the idea for immutable infrastructure.

All you need for quick and safe DR is to keep them in version control system.

In case of disaster, the whole infrastructure could build in few minutes again.

For data like RDS/ EC2 storage, snapshots with correct retention policy and S3 as storage works perfectly. As here the main factors for implementing correct DR strategy are the RTO and RPO for the services.

RTO – recovery time objective

RPO – recovery point objective

**Monitoring:**

I created a dashboard with basic monitoring metrics for the purpose of the test.

There is no data for the metrics, because the infrastructure is not used.

As Cloud Watch also could be developed with Terraform.   
Here is link to the Cloud Watch dashboard:

[https://console.aws.amazon.com/cloudwatch/home?region=us-east-1#dashboards:name=FundingCircle](https://console.aws.amazon.com/cloudwatch/home?region=us-east-1%23dashboards:name=FundingCircle)

For larger projects, monitoring should be much more detailed and complex.