FusionOMS Infrastructure: Unified Terraform Deployment Guide

This guide explains how to use the unified Terraform configuration to deploy the FusionOMS infrastructure across multiple environments (LocalStack, staging, production and any future environments) using a single codebase. It also covers one-time bootstrap setup, environment-specific configuration files, and how to automate deployments with GitHub Actions. At the end of the guide, you should be able to provision a complete AWS or LocalStack environment, understand where to place configuration files, and know which actions are manual versus automated.

Overview

The FusionOMS infrastructure is defined in Terraform and supports three main deployment targets:

1. **LocalStack** – a local AWS emulator for development and testing. Resources are created locally via Docker. You control LocalStack manually; it is not deployed through GitHub Actions.
2. **Staging** – a real AWS environment used for pre-production testing. Infrastructure is created and updated via Terraform and optionally automated through GitHub Actions.
3. **Production** – a real AWS environment for live workloads. Infrastructure is created and updated with the same Terraform code, but using different variables. Deployments to production can also be automated via GitHub Actions.

All environments share the same `.tf` files. Behaviour is controlled by **variables** and **backend configuration files**. New environments (e.g. development, UAT) are created simply by adding new variable files and state keys—no new Terraform code is required.

Directory Structure !

```
/
├─ bootstrap/                      # One-time bootstrap: creates the S3 bucket
and DynamoDB table for state
│    └─ bootstrap.tf
│
├─ backend-config.hcl              # Backend configuration for real AWS
environments (staging, production, etc.)
├─ backend-config-localstack.hcl   # Backend configuration for LocalStack
│
├─ localstack.example.tfvars       # Example variables for LocalStack
├─ staging.example.tfvars          # Example variables for staging
├─ production.example.tfvars       # Example variables for production
│
├─ variables.tf                    # Global variables (environment name, VPC
CIDRs, DB settings, images bucket flags)
```

```
├─ locals.tf                       # Derived values (name prefixes, bucket
names)
├─ vpc.tf                          # VPC, subnets, route tables, NAT, Internet
Gateway
├─ security-groups.tf              # Security groups (ALB, ECS, DB, default)
├─ rds.tf                          # RDS instance, subnet groups, parameter
group, encryption key
├─ alb.tf                          # Application Load Balancer, listeners,
target groups
├─ ecs.tf                          # ECS cluster, task definitions, services,
autoscaling
├─ ecr.tf                          # ECR repositories
├─ iam.tf                          # IAM roles and policies
├─ ssm.tf                          # SSM parameters for configuration secrets
├─ images.tf                       # S3 bucket for user images and CloudFront
distribution
├─ outputs.tf                      # Aggregated outputs (VPC IDs, ALB DNS, ECR
URIs, images bucket, etc.)
│
└─ .github/
    └─ workflows/
        ├─ terraform-staging.yml  # GitHub Actions workflow for staging
environment
        ├─ terraform-production.yml # GitHub Actions workflow for production
environment
        ├─ web-deploy.yml          # Build, test, and deploy web container
        ├─ api-deploy.yml          # Build, test, and deploy API container
        └─ mobile-build.yml        # Build mobile app (no deployment)
```

One-time Bootstrap Folder

The bootstrap/ folder contains a small Terraform module that provisions the S3 bucket and DynamoDB table used by Terraform to store state and coordinate locking. You run this once per AWS account. After the bootstrap is complete, you do not run these files again unless you move to a new AWS account.

Prerequisites

To use these scripts you need:

- An AWS account with permissions to create VPCs, ECS clusters, RDS instances, ECR repositories, IAM roles, S3 buckets, CloudFront distributions and SSM parameters. Access keys should be stored securely (see the section on GitHub Secrets).
- Docker installed if you plan to use LocalStack.
- Git installed if you plan to clone the repository and push updates.
- Terraform installed locally for manual runs (version > 1.5 recommended).
- A GitHub account if you plan to automate via GitHub Actions.

Step 1 – Bootstrap the State Storage (AWS Only)

Before using the unified configuration on AWS, you must create a **state bucket** and **state lock table**. These are used by Terraform to store the state file and to lock resources during concurrent runs.

1. Clone the infrastructure repository or copy the `bootstrap/` folder to your local machine.

2. Navigate into the `bootstrap/` folder:

   ```
   cd bootstrap
   ```

3. Initialise Terraform and run the bootstrap: (you have to change the bucket name for each new account, because S3 buckets must have unique global names, so if you have already created a bucket named random243rand1234, no other AWS user in the world should be able to create a bucket that has the same name)

   ```
   terraform init
   terraform apply
   ```

   Terraform will ask for confirmation and then create:

   - An S3 bucket (e.g. `fusionoms-terraform-state-1234567890`)
   - A DynamoDB table (e.g. `fusionoms-terraform-locks`)

   Take note of the **bucket name** and **table name**; you will reference them in the backend configuration files.

   ⚠️ You only need to run the bootstrap once per AWS account. Keep the generated state bucket and table; do not recreate them for each environment.

Step 2 – Prepare Backend Configuration Files

Terraform uses a **backend** to store state. Instead of passing many flags on the command line, you use HCL files to specify these settings. There are two backend files included:

`backend-config.hcl` (for AWS)

Example content (adjust values to match your bootstrap outputs):

```
bucket         = "fusionoms-terraform-state-1234567890"
dynamodb_table = "fusionoms-terraform-locks"
region         = "us-east-1"
key            = "staging/terraform.tfstate"   # change 'staging' to
'production' or other env
encrypt        = true
```

When you run `terraform init` for a real AWS environment, Terraform reads this file to know where to store the state and which lock table to use.

backend-config-localstack.hcl (for LocalStack)

Example content:

```
bucket         = "fusionoms-localstack-state"
region         = "us-east-1"
key            = "localstack/terraform.tfstate"
endpoint       = "http://localhost:4566"
skip_credentials_validation = true
skip_metadata_api_check     = true
force_path_style            = true
```

When using LocalStack, state is stored in a local S3 emulation. The `endpoint`, `skip_credentials_validation`, `skip_metadata_api_check`, and `force_path_style` directives ensure Terraform points to the LocalStack endpoints and does not attempt to contact AWS.

Step 3 – Create or Edit Variable Files (`.tfvars`)

Variables allow you to customise each environment without changing the Terraform source. You supply variable values in a `.tfvars` file. There are example files provided for each environment:

- `localstack.example.tfvars` – base values for LocalStack
- `staging.example.tfvars` – base values for staging
- `production.example.tfvars` – base values for production

You must copy an example file to a new file named for your environment (e.g. `staging.tfvars`) and fill in the missing secrets:

```
cp staging.example.tfvars staging.tfvars
```

Important variables to set:

Variable	Description	Example
`environment`	Name of the environment; used to prefix resource names	`staging`
`project_name`	Short name for the project	`fusionoms`
`region`	AWS region	`us-east-1`
`vpc_cidr`	CIDR block for the VPC; must **not** overlap between environments	`10.15.0.0/16`
`db_username` / `db_password`	RDS master credentials	`fusion_admin` / `changeme`
`admin_cidr_for_rds`	IP CIDR allowed to connect to RDS (e.g. your office IP)	`203.0.113.0/32`

Variable	Description	Example
`acm_certificate_arn`	ARN of an ACM certificate for ALB HTTPS listeners (leave empty to use HTTP)	`arn:aws:acm:us-east-1:123456789012:certificate/abc123`
`service_subdomains`	Hostnames used by the ALB for each service (frontend, auth, backend, projects)	see example tfvars
Images variables	`enable_images_bucket`, `enable_images_cloudfront`, `images_bucket_name` – enable and configure the user images S3 bucket and CloudFront (see below)	

Images Bucket and CloudFront Variables !

If you need a bucket to store user images and an optional CloudFront distribution in front of it, set the following in your tfvars file: (already set in the version I am sending, but if you decide not to go with this approach you can simply put false or remove or comment them)

```
enable_images_bucket     = true
enable_images_cloudfront = true   # set to true if you want CloudFront, false if only S3
images_bucket_name       = "fusionoms-staging-user-images"
```

- `images_bucket_name` is optional. If left blank, Terraform derives a name from `project_name` and `environment` (e.g. `fusionoms-staging-user-images`).
- `enable_images_bucket` controls whether the bucket and permissions are created.
- `enable_images_cloudfront` controls whether a CloudFront distribution is deployed in front of the bucket. If true, Terraform creates an origin access control (OAC) and denies direct public access to the bucket.

After deployment, Terraform outputs the **images bucket name** and the **CloudFront domain**, which your application can use to build URLs.

Step 4 – Running Terraform Manually

You can run Terraform directly from your terminal. These commands apply to **any environment** by changing the backend config and tfvars file.

LocalStack Deployment

1. **Start LocalStack** using Docker Compose (from the directory containing your `docker-compose.yml`):

   ```
   docker-compose up -d
   ```

The provided `docker-compose.yml` starts LocalStack on port 4566 and mounts a local directory for persistent data. You can also specify `DATA_DIR` and `PERSISTENCE=1` to persist state between runs.

2. **Initialise** Terraform with the LocalStack backend:

   ```
   terraform init -backend-config="backend-config-localstack.hcl"
   ```

3. **Plan and Apply** using your localstack tfvars file:

   ```
   terraform plan  -var-file="localstack.tfvars"
   terraform apply -var-file="localstack.tfvars"
   ```

Terraform will create all resources in LocalStack. Outputs include the VPC ID, subnet IDs, ALB DNS name, ECR URIs, SSM prefix, and optional images bucket and CloudFront domain.

Staging / Production Deployment

1. **Initialise** using the AWS backend configuration file:

   ```
   terraform init -backend-config="backend-config.hcl"
   ```

   Ensure `key` inside `backend-config.hcl` matches the environment you are deploying (e.g. `staging/terraform.tfstate`, `production/terraform.tfstate`). The same S3 bucket and DynamoDB table from the bootstrap can be shared across environments as long as the `key` is unique.

2. **Plan** your changes:

   ```
   terraform plan  -var-file="staging.tfvars"
   ```

   Review the plan to see which resources will be created or updated. Terraform is incremental; only changes to your environment are applied.

3. **Apply** the changes:

   ```
   terraform apply -var-file="staging.tfvars"
   ```

   Terraform will ask for confirmation. Enter `yes` to proceed. After completion, note the outputs for the ALB DNS name, RDS endpoint, ECR URIs, SSM prefix, and (if enabled) the images bucket and CloudFront domain. Use these values in your application code or environment variables.

Repeat the same procedure for production using `production.tfvars` and updating the `key` in `backend-config.hcl` to `production/terraform.tfstate`.

Step 5 – Integrating with GitHub Actions (To be delivered)

Step 6 – Manual Versus Automated Actions

The table below summarises which tasks you need to perform manually and which are automated by GitHub Actions or Terraform:

Task	Manual action?	Automated?	Notes
Bootstrap S3 + DynamoDB	✅	❌	Run once per AWS account; creates state storage.
Modify tfvars files	✅	❌	Create or update `staging.tfvars`, `production.tfvars`, etc., with your environment values.
Run `terraform init -backend-config=<config>`	✅	✅ via GitHub Actions	You can run locally; CI runs it automatically in workflows.
Run `terraform plan`	✅	✅	Run locally to preview changes; CI runs on pull requests.
Run `terraform apply`	✅	✅	Run manually or let the workflow apply on merge to `main`.
Build/push application containers	❌	✅	GitHub Actions workflows handle build/test/deploy on merge.
Update secrets in GitHub	✅	❌	You must add AWS and ECR credentials as repository secrets.
Create new environment (e.g. dev)	✅	❌	Create a new tfvars, set environment and CIDR, update backend `key`, run `init`, `plan`, `apply`.

Step 7 – Adding a New Environment (Dev/UAT)

1. Copy the staging tfvars as a base:

   ```
   cp staging.tfvars dev.tfvars
   ```

2. Edit `dev.tfvars`:

   - `environment = "dev"`
   - `vpc_cidr = "10.4.0.0/16"` or another non-overlapping range
   - Set `enable_images_bucket` and `enable_images_cloudfront` as needed
   - Update any secrets or hostnames for the dev environment

3. In `backend-config.hcl`, change `key` to `dev/terraform.tfstate`.

4. Run:

```
terraform init -backend-config="backend-config.hcl"
terraform plan  -var-file=dev.tfvars
terraform apply -var-file=dev.tfvars
```

You will now have an isolated dev environment. Add a new GitHub Actions workflow if you want automatic deployments for dev as well.

Step 8 – Final Notes

- **State management:** Do not share the same state key between environments; each environment must have its own `key` in the backend configuration file.
- **Network isolation:** Choose unique `vpc_cidr` ranges so VPCs do not overlap. Overlapping CIDRs can cause routing issues if you connect VPCs or run them in the same account.
- **Environment names:** The `environment` variable is used throughout to tag resources. Changing this value updates resource names accordingly.
- **IAM Permissions:** Ensure the AWS credentials used in GitHub Secrets have permission to create and update the required resources. Use least privilege where possible.
- **Cleanup:** To destroy an environment, run `terraform destroy -var-file=<env>.tfvars` with the same backend configuration. Be careful to confirm before destruction; this will delete VPCs, RDS instances and data.

With this guide, you should be able to set up, deploy, and manage multiple environments using a single Terraform codebase. Adjust the variable files to suit your staging and production requirements, run the bootstrap once, and rely on GitHub Actions to handle ongoing deployments and updates.