TODO application

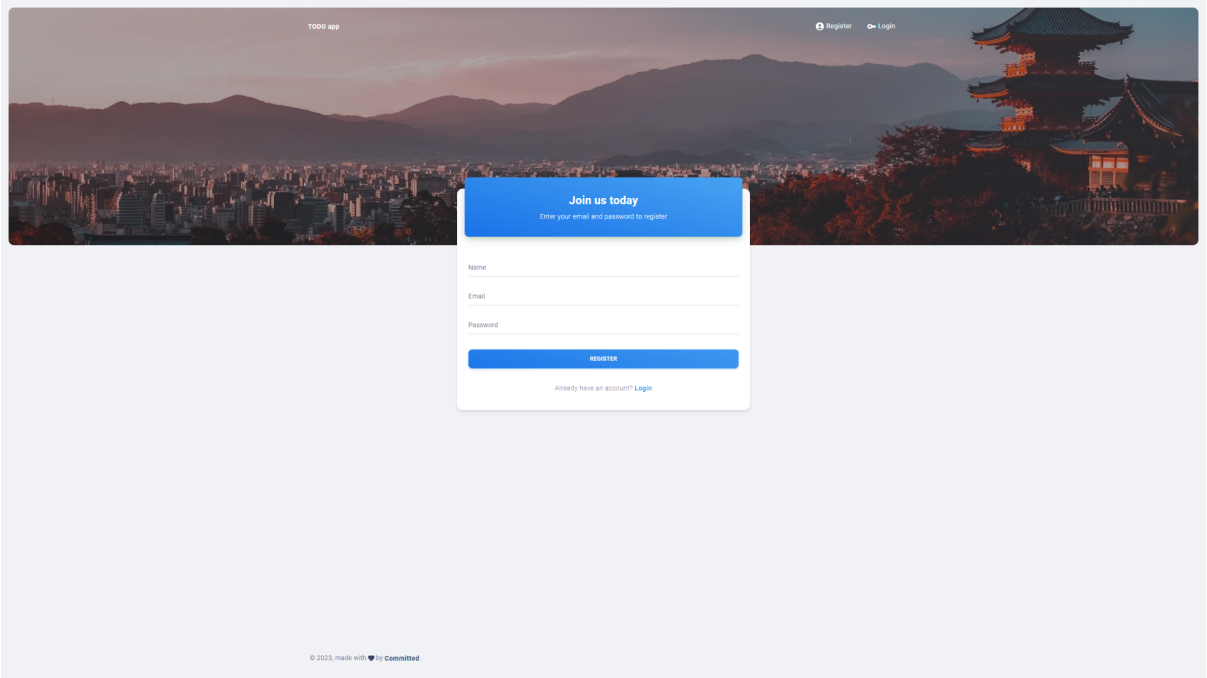
made by Committed

Original documentation can be found here

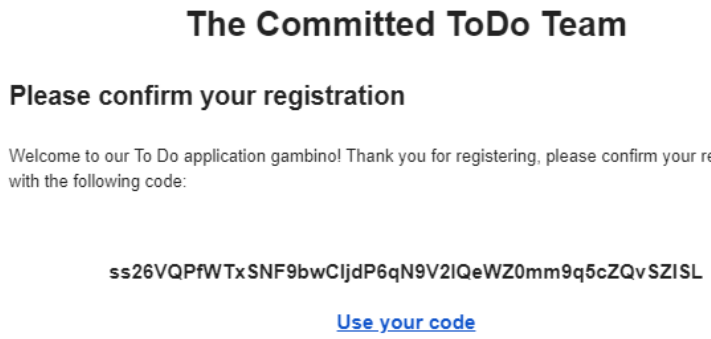
About the app

Register page (*/auth/register*)

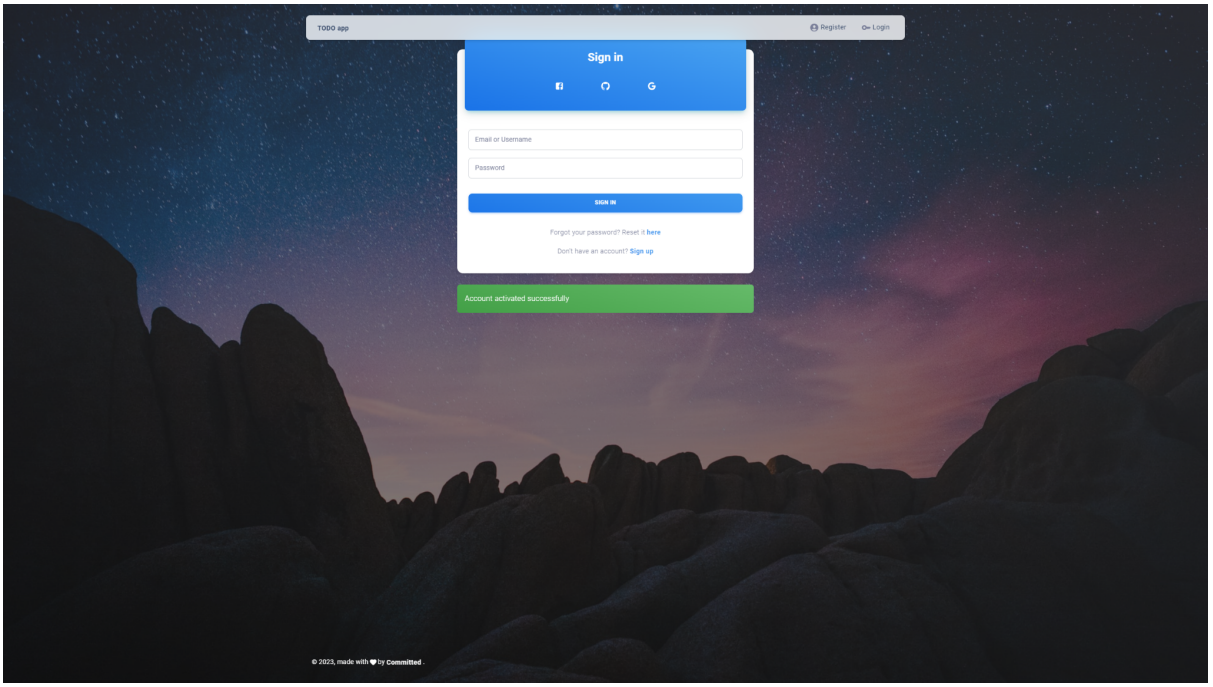
You can register here, you will need a unique username and valid email address and also a password, which should be at least 8 characters long and should contain at least 1 number, 1 character, 1 uppercase character and also 1 special character (.?,.-_§’”+!%/=()).



After successful registration you will receive an email for the activation.



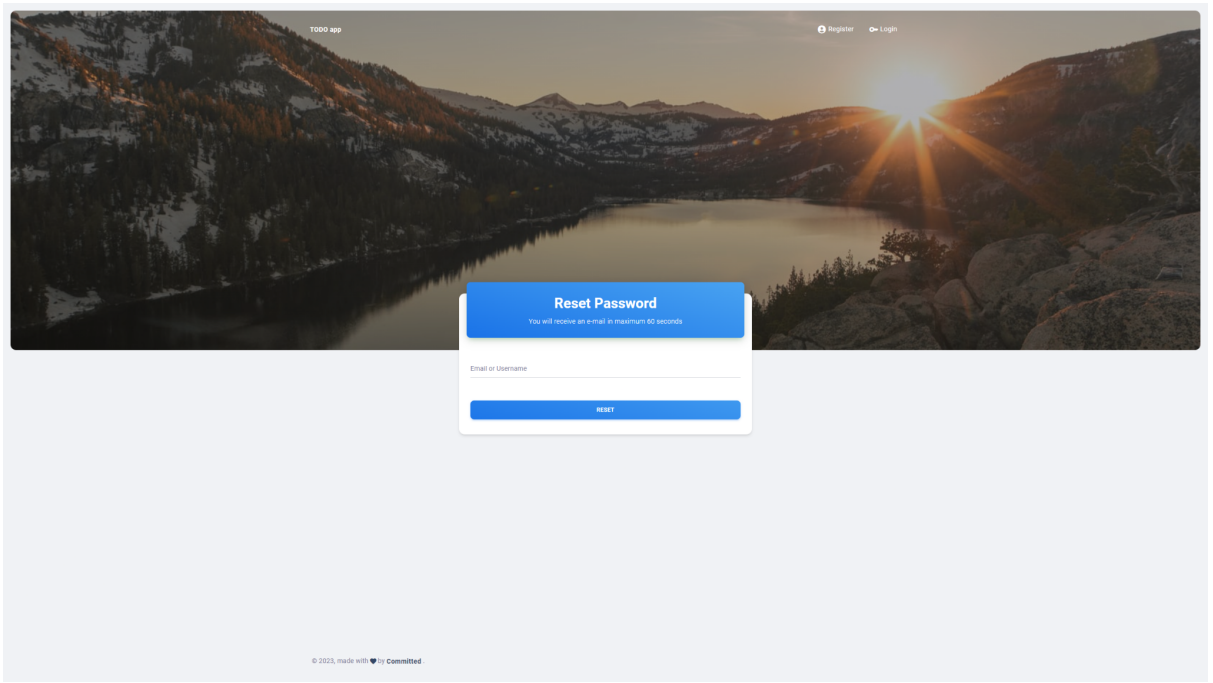
The email contains a code and also a link (Use you code). You can click the link and you should see the Login page with the “Account activated successfully” message.



Alternatively you can use the code in the following format: /auth/login/<code>. Example: /auth/login/ss26VQPfWTxSNF9bwCIjdP6qN9V2lQeWZ0mm9q5cZQvSZISL

Forgot password page (*/auth/forgot-password*)

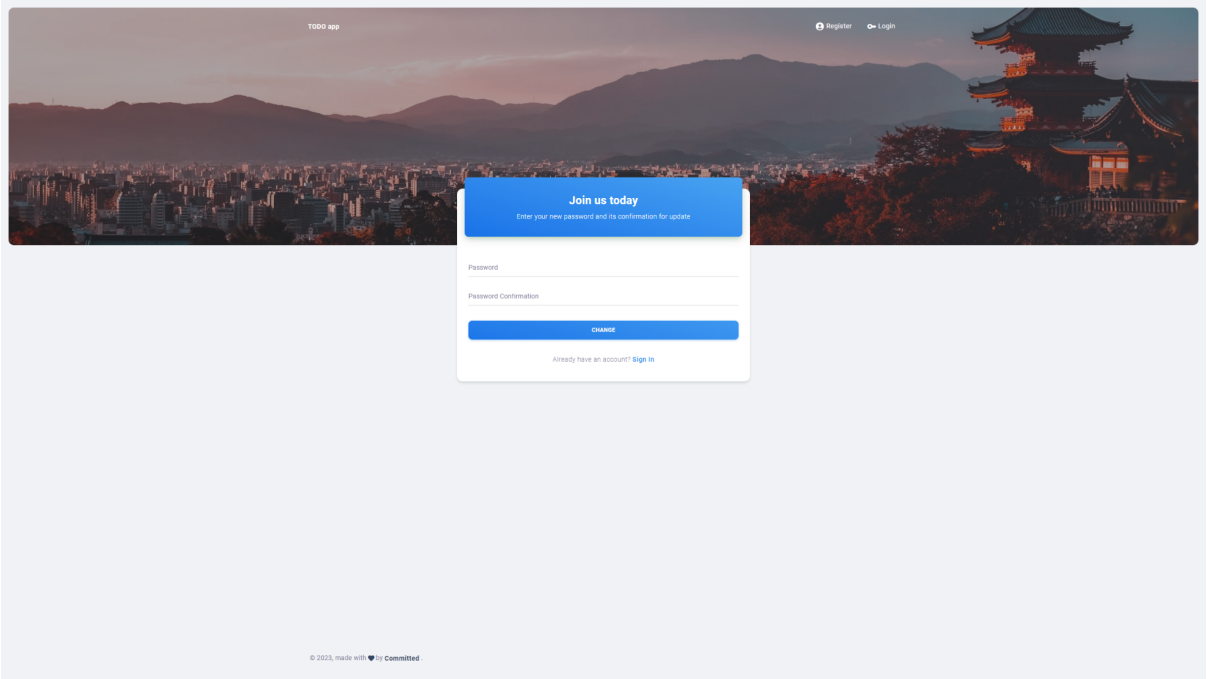
You only need to give your username or email address and you will get an email with the code and link that will send you to the Reset password page.



The code can be used in the same way as in the activation process in the Register page. Example: /auth/reset-password/RGnQPtUxTaW1yQKDpiGOdN9rouchEU2UNfuKDh5j0SSj9V9V

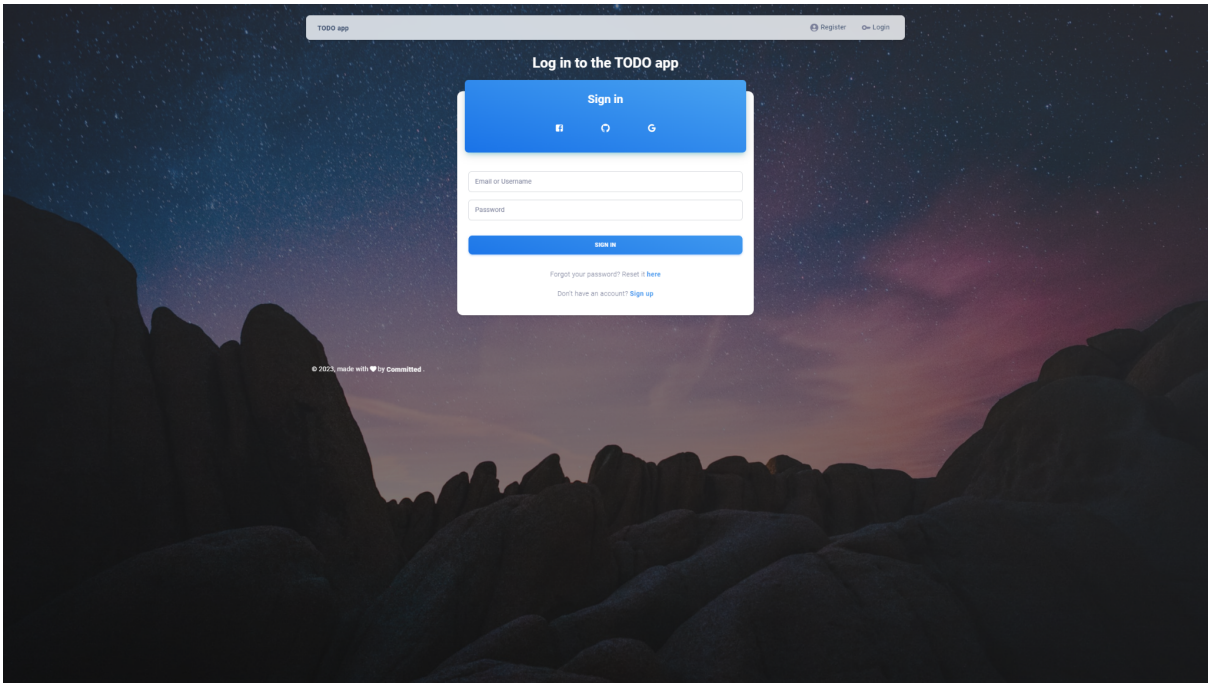
Reset password page (*/auth/reset-password*)

Here you can give the new password.

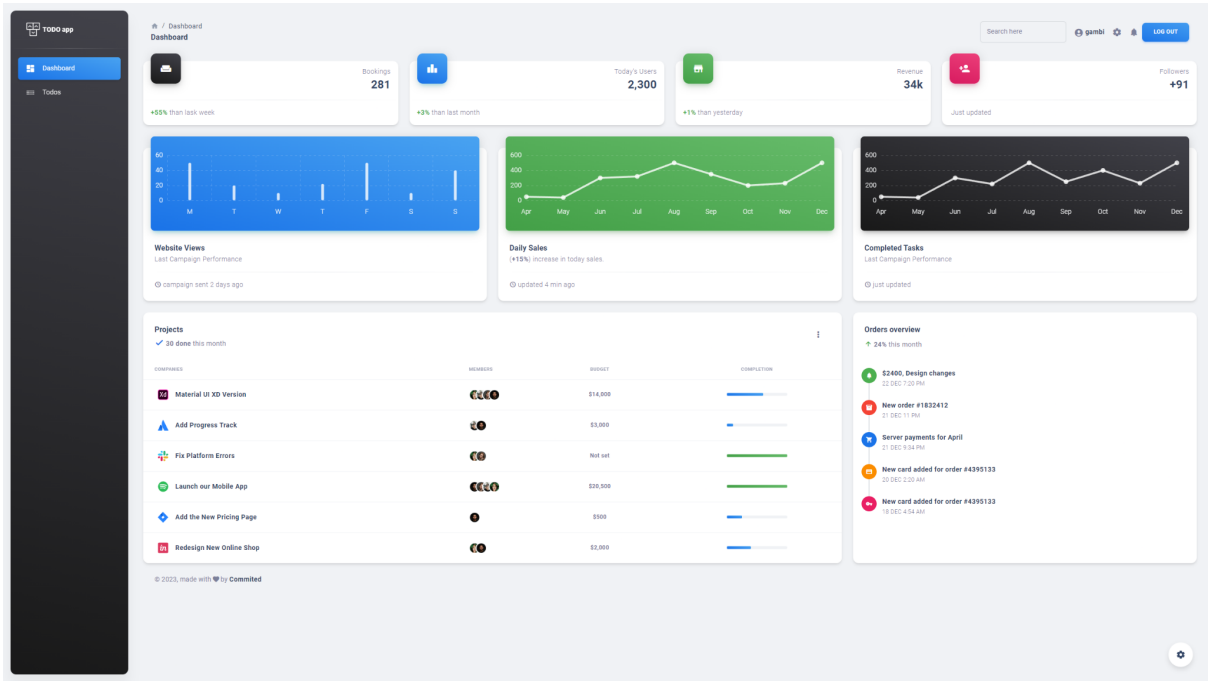


Login page (*/auth/login*)

You can log in here using username or email address and a password, if the password doesn’t match the same requirements as in registration, the login process won’t even send data to the server. After successful login you will be redirected to the Dashboard page.

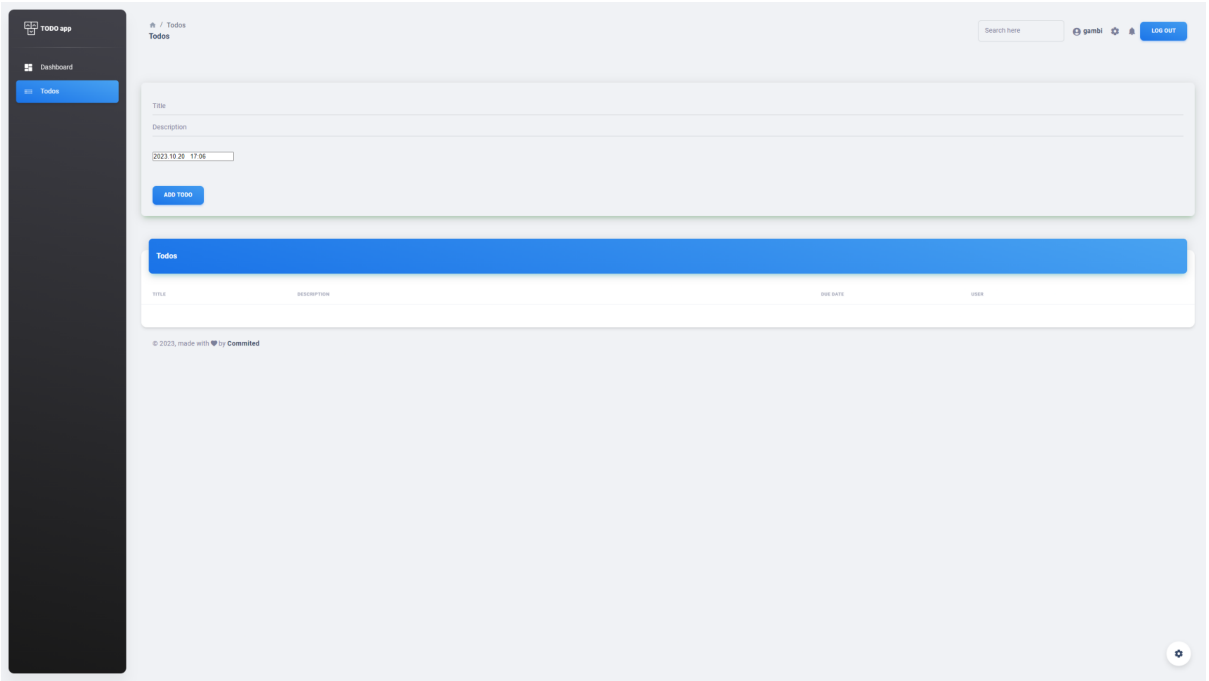


Dashboard page (*/dashboard*)



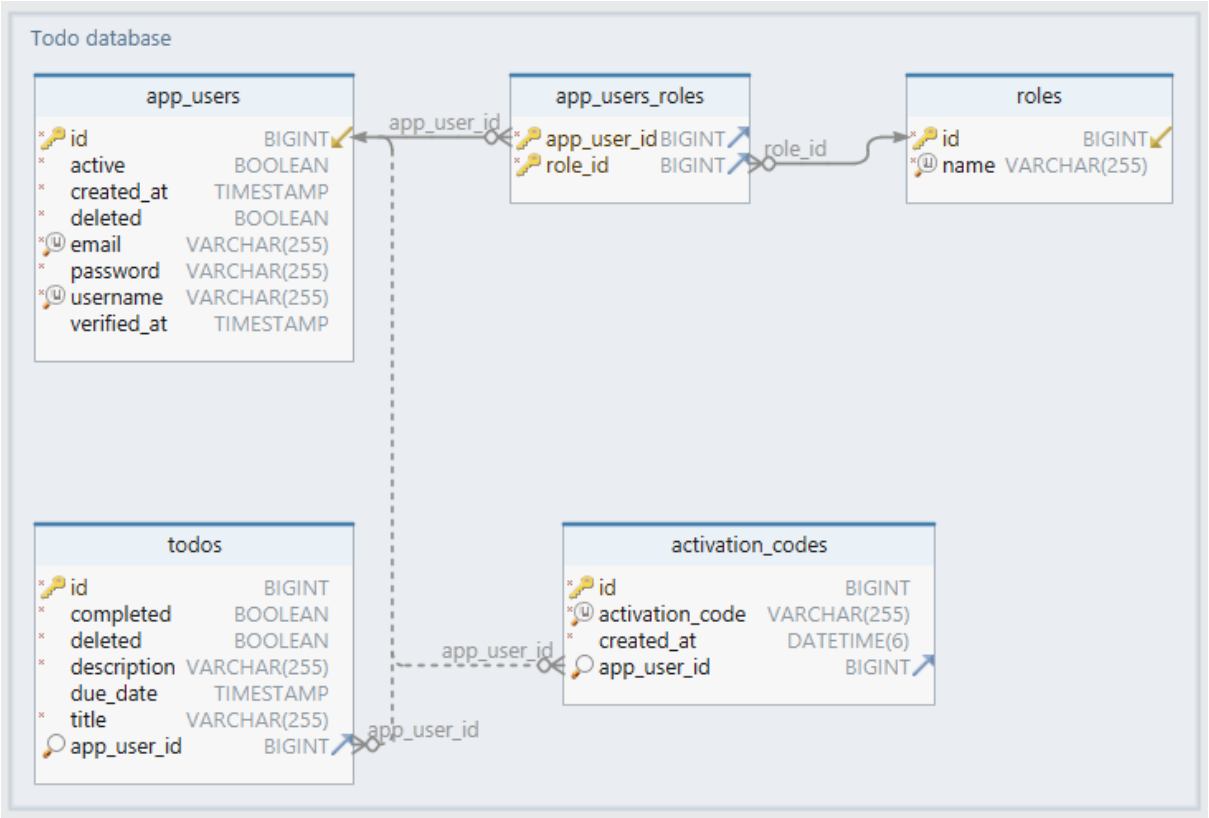
Todos page (*/todos*)

Here you can add new todos, view them in a table or delete them.



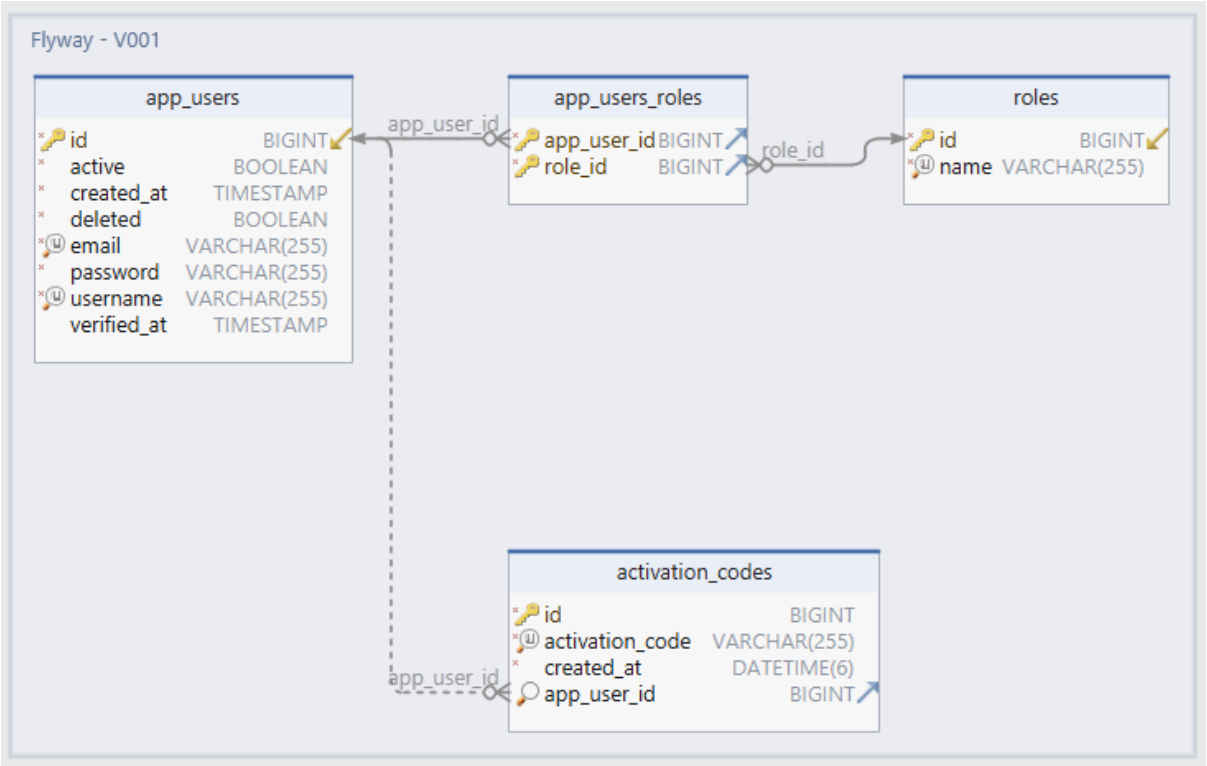
Data persistence

Database tables and their relationships:

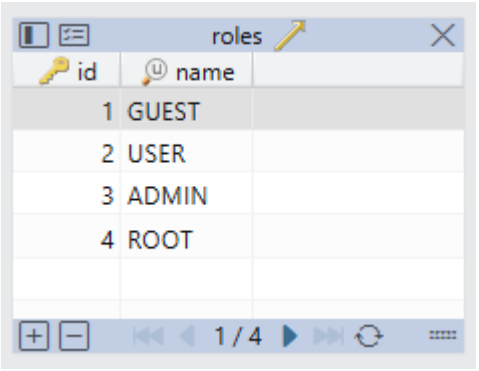


Flyway

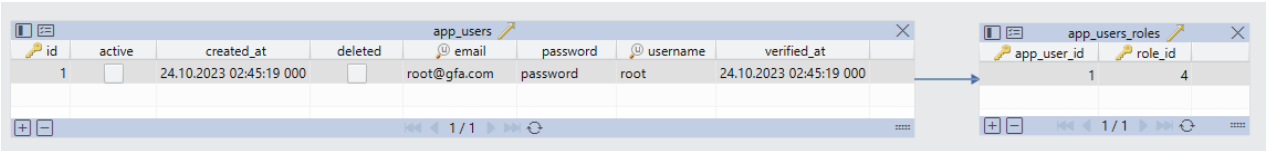
Migration V001: Creating tables and relationships



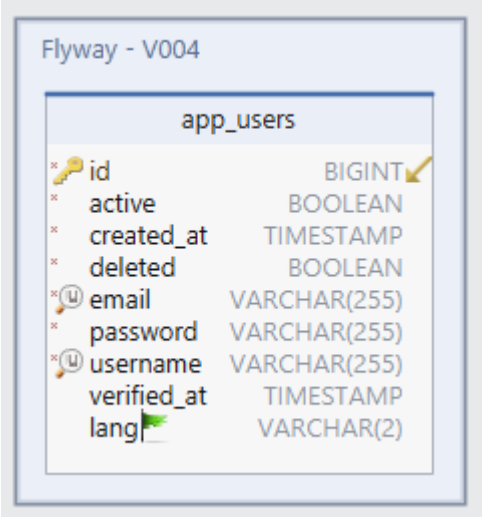
Migration V002: Add data to the roles table



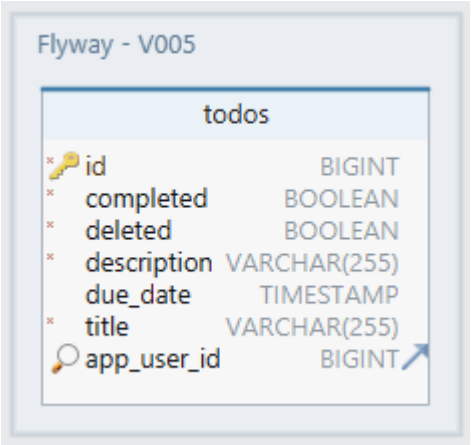
Migration V003: Add root user and assign role “ROOT”



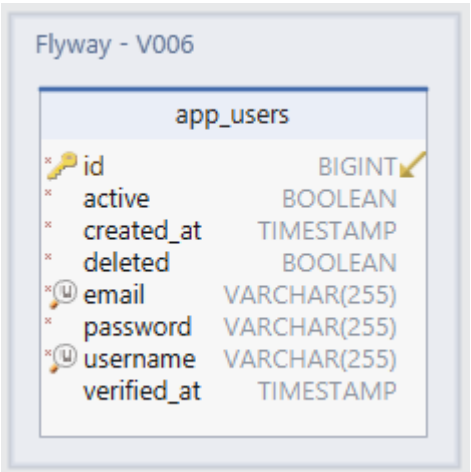
Migration V004: Add “lang” column to “app_user” table



Migration V005: Add “todo” table



Migration V006: Dropped column “lang” from “app_users”



REST API

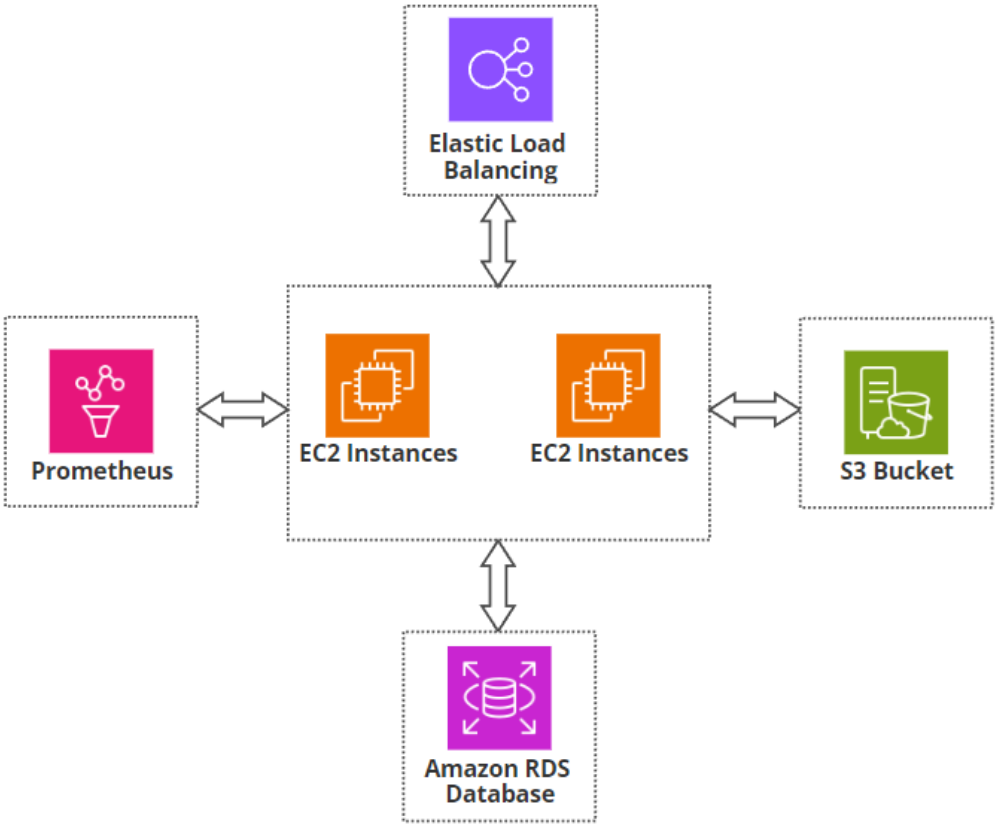
Containerization

Infrastructure

The Logical Model

Our infrastructure is cloud-based, orchestrated primarily using Terraform and utilizing Terraform's powerful IaC (Infrastructure as code) technology.

The architecture of our infrastructure



Below are the key components that make up the Todo application's architecture:

Components:

1. *S3 Buckets for Terraform State files*

We use a dedicated S3 bucket to store the Terraform state files.
This bucket is encrypted and version-controlled for added security.

2. *Virtual Private Cloud (VPC)*

Our backbone for the cloud setup, providing the network layer for our resources. Configured with public and private subnets to segregate resources that should be publicly accessible from those that need to remain internal.

3. *Elastic Compute Cloud (EC2) Instances*

Hosts the application and web servers.
Every EC2 instance has an EIP (elastic IP address) to ensure the IP doesn't change after any intentional or unexpected instance reboots. Upon creation, an extensive *user_data* script is installed on every EC2 instance. This script handles tasks ranging from setting up environment variables for the application, installing necessary dependencies, to configuring a *systemd* service that ensures our application is always running. For a detailed walkthrough of this script, see the Automation section.

4. *Relational Database Service (RDS)*

Provides a managed database solution on AWS, hosting our MySQL databases. Situated within a private subnet to ensure it's not publicly accessible, but only for our application.

5. *Load Balancer*

Resides in the public subnet and routes incoming traffic to our EC2 instances.
A DNS address is automatically assigned to the load balancer which directs users to the frontend of our application.

6. *Environment-Specific S3 Buckets*

We maintain separate S3 buckets for *dev*, *staging*, and *production* environments. These buckets contain environment-specific **.env** files and **JAR** files required for our application to run. Pulling these files to the instances are automated with the *user_data* script and *systemd* service scripts during the instance creation.

7. *Security Groups*

These define what can or cannot communicate with our resources.
Separate security groups are defined for *EC2 instances*, our *RDS databases*, and for the *Load Balancers* to enforce the principle of least privilege.

8. *Network Interfaces*

Virtual network cards that facilitate the EC2 instances network communication. These resources are manually defined in our terraform script to ensure they are using the dedicated security groups and subnets and avoid unexpected errors upon auto creation.

9. *Route Tables*

Define how network traffic is directed within the VPC and ensure proper data flow between subnets and the outside world. Our terraform script defines a public route table for incoming and outgoing traffic and separates a private route table for the database using the private security group. This way we make sure our database is not accessible through the internet, but only for our application through port definition.

10. *Outputs*

Our terraform script collects necessary data after creating the infrastructure and outputs them to the terminal. These variables are exported to the resources folder into a .txt file automatically by the terraforms script. The data can also be saved manually into a JSON file and shared between team members. It includes IP addresses, DNS addresses and database credentials and the current work environment.

11. *Modules*

Every component is separated logically into unique .tf files for readability and clarity during usage. If changes are necessary the user can find the specific component without searching through hundreds of lines of code.

12. *Variables*

Variables are introduced throughout the whole script and only have to be defined upon environment creation for unique variables. Common variables have default values which are used automatically by the script. This ensures a dynamic creation process for the script and makes value changes accessible.

13. *Terraform workspaces*

The terraform script leverages the convenient solution from Terraform, “Workspaces”. For our three different environments (development, staging, production) we use three different workspaces and with a single variable used throughout the whole script we annotate the workspaces name to each and every resource. This ensures for the user to know everytime which workspace/environment is being observed or modified.

Relationships:

S3 Buckets to Terraform: The S3 bucket storing Terraform state files is used as the backend for Terraform, allowing for state to be shared and locked to prevent conflicts.

VPC & Subnets: All other components are part of the VPC specific to the environment created by terraform, residing in either the public or private subnets depending on their roles.

Security Groups to Resources: Security groups are attached to EC2 instances, RDS databases, and Load Balancers to regulate inbound and outbound traffic.

Network Interfaces to EC2: Network interfaces are part and parcel of EC2 instances, enabling their networking capabilities.

Route Tables to Subnets: Route tables are associated with subnets to control the traffic routing.

Load Balancer to EC2: The Load Balancer directs traffic to EC2 instances based on the load balancing algorithm. Health checks are performed to ensure traffic only goes to healthy instances.

EC2 to RDS: EC2 instances communicate with the RDS database for CRUD operations. The communication happens within the private network for enhanced security.

EC2 to Environment-Specific S3 Buckets: EC2 instances fetch the relevant .env and JAR files from the corresponding S3 bucket based on the environment they are in. This setup allows for easy configuration management across environments.

Data Flow:

1. External traffic first hits the Load Balancer.
2. The Load Balancer then forwards the requests to one of the healthy EC2 instances.
3. EC2 instances interact with the RDS database as needed to serve the request.
4. EC2 instances may also fetch or update .env or JAR files from the environment-specific S3 buckets as needed.
5. The response travels back through the EC2 and Load Balancer to be served to the user.

The Physical Model

We operate within AWS's Free Tier to ensure cost-effectiveness while maintaining functionality. Below are the specifics of the resources allocated:

EC2 Instances
Instance Type: t2.micro
Operating System: Ubuntu
vCPU: 1
Memory: 1 GB
Storage: EBS Only (Elastic Block Storage)

RDS Database
Instance Type: t3.micro
Engine: MySQL
vCPU: 2
Memory: 1 GB
Storage: 10GB SSD

S3 Buckets
Tier: Standard (Free tier eligible)
Region: Specified based on latency requirements
Versioning: Enabled
Encryption: Enabled

Virtual Private Cloud (VPC)
CIDR Block: Specified in var.vpc_cidr_block
DNS Hostnames: Enabled

Internet Gateway
VPC Association: Associated with the VPC defined above

Public Subnets
Count: Defined in var.subnet_count.public
CIDR Blocks: Specified in var.public_subnet_cidr_blocks

Availability Zones: Auto-selected from available zones
Public IP on Launch: Enabled

Private Subnets
Count: Defined in var.subnet_count.private
CIDR Blocks: Specified in var.private_subnet_cidr_blocks
Availability Zones: Auto-selected from available zones

Route Tables
Public Route Table:
Routes: All public traffic (0.0.0.0/0) is routed through the Internet Gateway
Private Route Table:
Routes: No explicit routes defined, adjustable based on need

Network Interface Cards (NICs)
Count: Corresponds to the number of public subnets
Security Groups: Defined in aws_security_group.instance_security.id

Elastic IPs
Count: Defined in var.instance_count
Association: Associated with EC2 instances

Tags and Naming Conventions
All resources are tagged with an environment-specific name to facilitate management and identification. This is achieved using the Name tag and ${var.environment} variable.

Environments

Our infrastructure is designed to be environment-agnostic, meaning the same Terraform script can be used to set up multiple environments. We currently support three types of environments (specified by workspaces): Dev, Staging, and Production. Below are the unique characteristics for each environment .

Dev Environment

- Purpose: For individual developers to test their code.
- Instances: One t2.micro EC2 instance running Ubuntu.
- Database: One t3.micro RDS instance.
- Load Balancing: Not applicable.
- Monitoring: Not applicable.
- Tag Naming: All resources are tagged with dev in their names for easy identification.
- State File Storage: S3 bucket specifically for Dev Terraform state files.

Staging Environment

- Purpose: For integrated testing and pre-release verification.
- Instances: One t2.micro EC2 instance running Ubuntu.
- Database: One t3.micro RDS instance.
- Load Balancing: Not applicable.
- Monitoring: Not applicable.
- Tag Naming: All resources are tagged with staging in their names.
- State File Storage: S3 bucket specifically for Staging Terraform state files.

Production Environment

- Purpose: For running the live application, serving real users.
- Instances: Two t2.micro EC2 instances running Ubuntu.
- Database: One t3.micro RDS instance.
- Load Balancing: Load balancer present to route incoming traffic to multiple EC2 instances.
- Monitoring: Prometheus for monitoring system performance and alerts.
- Tag Naming: All resources are tagged with production in their names.
- State File Storage: S3 bucket specifically for Production Terraform state files.

To deploy a specific environment the user has to change between workspaces. Based on the current selected workspace a local variable called environment_name is initialized which is passed on to the module and to every component from the module.

```
locals {
  environment_name = terraform.workspace
}
```

The following commands should be used for navigating Terraform workspaces:

- terraform workspace new “name” - creating a new workspace
- terraform workspace select “name” - switching to an existing workspace
- terraform workspace delete “name” - deleting an existing workspace
- terraform workspace list - list all of the available workspaces

After selecting the workspace the following section can be modified in the terraform script:

```
module "todo_app" {
  source = "../todo-app-modules"
  environment      = local.environment_name
  instance_type    = "t2.micro"
  ami              = "ami-05b5a865c3579bbc4"
  instance_count   = 2
  database_name    = "todo_app_db_${local.environment_name}"
  database_port    = "3306"
  database_user    = "root"
  database_pass    = "password"
  db_exists        = false # should only be false at FIRST run!
  bucket_name      = "committed-todo-app-${local.environment_name}"
}

module "prometheus" {
  source             = "../prometheus-modules"
  count              = local.environment_name =="production" ? 1: 0
  vpc_id             = module.todo_app.vpc_id
  environment        = local.environment_name
  instance_type      = "t2.micro"
  ami                = "ami-05b5a865c3579bbc4"
  depends_on         = [module.todo_app]
}
```

The module todo_app is responsible for creating infrastructure for our environment and these variable values can be modified for a desired outcome. The prometheus module thanks to the count parameter only runs if we operate in the production environment, and creates a prometheus instance with necessary security groups and ports that allow for monitoring. The prometheus module creates the instance within the same VPC as the production instances, on a public subnet to allow connection to the production instance.

Automation with Terraform

Prerequisites:

- Terraform
- Visual Studio Code (Recommended)
- AWS CLI
- AWS account and credentials
- AWS key pair

Getting started:

1. Installing Terraform and prepping the environment

1./a Install Terraform to your computer (Windows)

1. Download the terraform.exe using this link https://developer.hashicorp.com/terraform/downloads?product_intent=terraform

2. Create a directory on your Local Drive called Terraform and copy the terraform.exe into the folder
(e.g.; C:\Program Files\Terraform) <- copy your location with Ctrl + C

3. On windows open the Edit the system environment variables tab. (Search for "env" in your start search bar and it will show up) Select the Environment Variables.
   * Under the System Variables look for the system variable called *Path*.
   * Select the *Edit* button and add the location of your terraform.exe by selecting the *New* button and pasting the path with Ctrl + V.

To make sure you have installed terraform correctly, open up your CLI or CMD and type in *terraform -v*.
If you configured the environment variable correctly you will see the version of terraform displayed in the CLI or CMD.
If you see anything else there was an issue and you should restart the process.

Now terraform is installed and is ready to use.

1./b Install Visual Code to your computer with Terraform extension (Optional but recommended)

1. Download and install Visual Code (Free software)
https://code.visualstudio.com

2. Open up VS code and navigate to the *Extensions* bar. Search for terraform and install the extension provided by HashiCrop (The developers of Terraform)

It provides Auto-completion, Syntax highlighting and other QOL improvements.

2. Initialization of Terraform

2./a Basic commands

Before we dive in to setting up the infrastructure a couple of important commands:

terraform init

This command configures the backend for our terraform project. It tells terraform

- the location of the state files
- it downloads the necessary providers based on the .tf file
- initializes settings and modules necessary to run the script
- downloads dependencies if necessary
-

If you are running init for the first time in a folder it will generate a terraform.tfstate file, which is our state file.

terraform plan

1. Analyze - This command checks our local configuration and compares to the existing deployed state either locally or remotely, depending on where we deployed our app.
2. Check - After checking, it finds out what actions should it take (add, change, delete) to align our real-world infrastructure with our config.
3. Plan - Then it prints out an execution plan to show what changes will be made, based on the second step.
4. Resolve - It sorts dependencies between resources ensuring they get created or destroyed in the correct order.

terraform plan -out=folder_structure/file_name
Use this command to save the plan into your desired folder with the desired name and run the apply command using the plan file.

terraform apply

This command will start constructing - or modifying - the actual infrastructure based on our script. First it will show the execution plan (similar to terraform plan) and ask for an approval.
The only accepted response in the terminal for proceeding is "yes".

After hitting yes, it will create, update or destroy the resources as necessary in the correct order.
When the changes have been done it will update the terraform state file, which serves as a snapshot of our resources.

To run this command without approval check you should add the *-auto-approve* tag to the end

terraform destroy

Similar to the terraform apply command this will show us an execution plan that indicates which resources will be destroyed and then ask for a confirmation.
The only accepted response in the terminal for proceeding is "yes".

After confirmation terraform will destroy all the resources in a reverse order to how they were created, thus completing a full and clean tear down process of our resources. It will update the terraform state file to show which resources have been destroyed.

Be cautious when running this on a live project that you are currently working on.

2./b Setting up the infrastructure

Open the terraform project folder in VS code and open a CLI.
In the CLI type a*ws configure* and log in with the AWS accounts credentials.

- AWS Access key
- AWS Secret key
- Region: the region where you will operate
- Press enter for default json formats

Navigate to the aws-backend folder in CLI and run the terraform init command. This will download files needed for terraform to operate and initialise the folder as a working directory. Then run the command terraform apply.
This will initialise the backend for terraform state files in an S3 bucket that is created with the script. All of the terraform state files related to our project will be stored online in this bucket. Make sure to specify the region at the provider section.

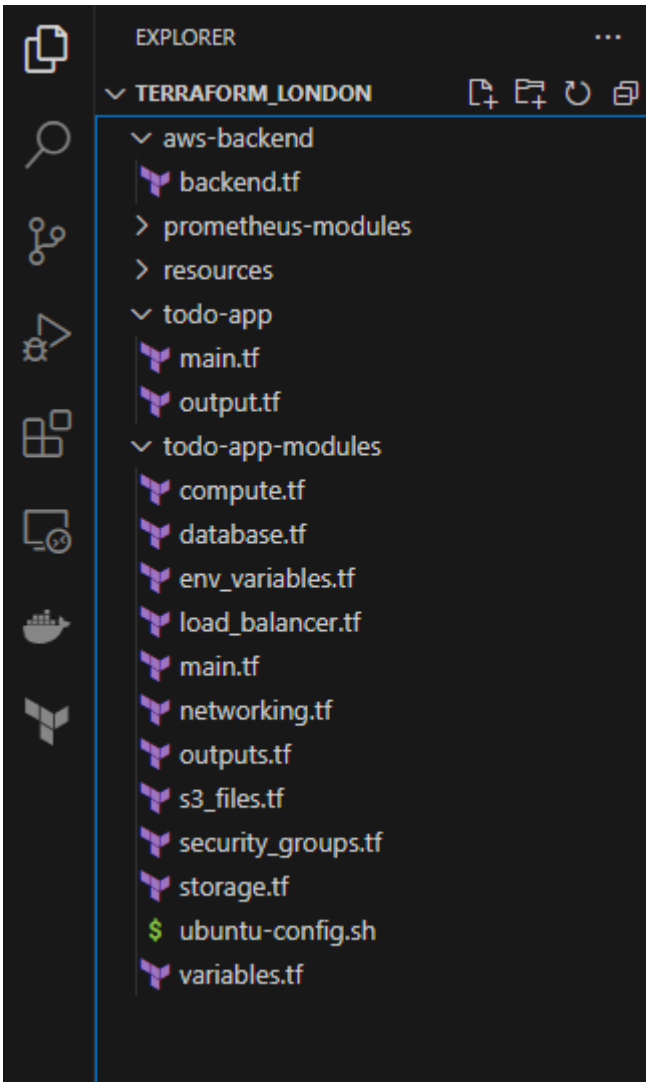
```
provider "aws" {
  region = "eu-west-2"
}

resource "aws_s3_bucket" "terraform-state" {
    bucket        = "committed-todo-app-tf-states-london"
    force_destroy = true
}
```

Now the backend has been created, navigate to the todo-app folder with CLI and run terraform init. This will download terraform to this folder again and set it as a working directory where we can run terraform commands.

1. Specify a workspace by creating a new one
2. Run terraform plan to create a plan and save the plan
3. Run terraform apply with the specified plan to create the infrastructure
4. Repeat steps 1-3 for every environment changing the instance count to specify the number of instances created to given environment

Code Structure



The infrastructure code is organized into directories and files that help in modularizing and managing various components efficiently.

Root Directory:
Contains overarching Terraform configuration and scripts for the entire infrastructure.

prometheus-modules:
Contains the Terraform configurations and resources necessary for setting up Prometheus for monitoring.

resources:
This is a placeholder for future resource configurations or a space for shared resources. Also the output variables are automatically saved in this folder after infrastructure creation.

todo-app:
Contains main configurations and output values for the todo-app.

main.tf: The main configuration file.

output.tf: Contains the output values from the configurations which can be used in other Terraform files or displayed after terraform apply.

todo-app-modules:
Houses modularized configurations for different components of the todo-app.

compute.tf: Contains configurations related to EC2 instances.

database.tf: Configurations related to the database setup for the todo-app.

env_variables.tf: Environment variables or configurations that the application uses and are passed onto the shell script.

load_balancer.tf: Contains configurations for load balancing the traffic for the todo-app.

main.tf: The main configuration file for the module.

networking.tf: Houses the networking-related configurations like VPC, subnets, route tables, network interfaces, elastic ips, etc.

outputs.tf: Specifies the outputs after deploying this module.

s3_files.tf: If files need to be uploaded to the s3 bucket from the script, it can be done here.

security_groups.tf: Contains security group configurations for the todo-app, the database and load balancer..

storage.tf: Includes the code for creating the environment specific S3 bucket.

variables.tf: Defines the input variables that the module uses.

Scripts:

ubuntu-config.sh: A shell script likely used to configure or bootstrap Ubuntu instances upon launch.

The Ubuntu config shell script

Script Functionality Overview
This script performs several critical actions that set up and configure an Ubuntu server environment tailored for running the Todo Java application. Its main functions include:

Logging:
All standard outputs are redirected to /home/log.log.
Error logs are directed to /home/error.log.

Environment Variables Configuration:
Sets up various environment variables like database configurations, SMTP details, JWT settings, and other application-specific values in /etc/environment.

Package Installation & System Updates:
Updates the Ubuntu package lists.
Installs necessary packages: AWS CLI, JQ, netcat, net-tools, tldr, and nginx.
Upgrades the system to the latest packages.

SSH Configuration:
Adds a new SSH user named developer and configures SSH to allow password authentication.
Grants the developer user sudo privileges without a password.

JDK Installation:
Downloads and installs JDK 17 from Oracle's official website.

Application Syncing from S3:
Syncs the Todo Java application's .jar file from an S3 bucket.
Also syncs an environment-specific .env file from S3.

Service Configuration:
Creates a shell script that finds and runs the .jar application.
Sets up a systemd service to ensure the Todo Java application starts on boot and automatically restarts if it fails.

Service Initialization:
Reloads the systemd daemon to recognize the newly created service.
Enables and starts the Java application service.
Enables the nginx web server.
Lastly, the script initiates a system reboot.

CI/CD pipeline

Monitoring

Initial setup