Technology Practices Manual

Open Source Enterprise Solution

STARTER ARCHITECTURE

Development Environment
Approach: Containerization
Technologies: Docker Compose
Syntax: YAML
Language: Go

Back-end API
Approach: S.O.L.I.D Principles
Technologies: Laravel
Syntax: Object-Oriented design (OOD)
Language: PHP

Front-end SPA/SSR
Approach: Composition API
Technologies: VueJs/Nuxt
Syntax: Typescript
Language: Javascript

Server Provisioning
Approach: Configuration Management
Technologies: Ansible
Syntax: YAML
Language: Python

Release Engineering
Approach: Continuous Integration and Continuous Deployment (CI/CD)
Technologies: Jenkins, Docker
Syntax: Jenkinsfile (declarative or scripted pipeline syntax)
Language: Groovy (used in Jenkinsfile)

Infrastructure as Code
Approach: Cold Start
Technologies: Terraform
Syntax: HashiCorp Configuration Language (HCL)
Language: N/A (HCL is a domain-specific language for Terraform)

1. INTRODUCTION

This section sets the stage by defining the purpose, scope, and intended audience of the document.

Purpose and Scope

[PENDING]

Audience

[PENDING]

2. ARCHITECTURE OVERVIEW

This section dives into the core of the Starter Architecture, providing insights into the technologies used, the project's folder structure, and how these components are integrated and work together.

Technologies Used

Here are short descriptions for each of the technologies used in the Starter Architecture. These technologies play integral roles in the Starter Architecture, collectively enabling the development, deployment, and management of robust web applications.

Terraform

Terraform is an Infrastructure as Code (IaC) tool that enables the provisioning and management of cloud resources and infrastructure. It uses HashiCorp Configuration Language (HCL) to define infrastructure components and ensures their consistent deployment and configuration.

Ansible

Ansible is a powerful automation tool that simplifies configuration management, application deployment, and task automation. It uses YAML syntax and requires minimal setup, making it ideal for managing server provisioning and application deployment.

Docker

Docker is a containerization platform that allows applications and their dependencies to be packaged into lightweight containers. These containers can be easily deployed and run consistently across various environments, improving application portability.

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It simplifies the management of complex applications with multiple services by specifying them in a single YAML file.

Jenkins

Jenkins is an open-source automation server that facilitates Continuous Integration and Continuous Deployment (CI/CD) pipelines. It uses Jenkinsfiles, written in Groovy, to define and automate build, test, and deployment processes.

Laravel

Laravel is a widely-used PHP web application framework known for its elegant syntax and developer-friendly features. It follows the principles of Object-Oriented Design (OOD) and promotes the use of SOLID principles for building robust back-end APIs.

Vue.js

Vue.js is a progressive JavaScript framework for building user interfaces. It emphasizes simplicity and flexibility, allowing developers to create interactive and dynamic front-end applications with ease.

Nuxt

Nuxt is a framework for server-side rendering (SSR) Vue.js applications. It simplifies the development of universal or isomorphic web applications, providing SEO benefits and improved performance.

Folder Structure

This is the link to the Git repository containing all of the code, all the separate segments have their respective updated README.md files that contain more specific information: https://github.com/esofstarter/starter-architecture

The folder structure of the components is the following:

infrastructure
dev_env (Docker Compose)
ci_cd (Jenkins Pipeline, Docker Compose Build Environment, Ansible Deployment Playbook)
host (Ansible Provisioning Playbook)
jenkins (Ansible Provisioning Playbook)
terraform (Terraform Cold Start)

server
api (Laravel API)
client
admin (Vue SPA Admin Panel)
public (Nuxt SSR Public Web)

Integration and Interoperability

Our architecture seamlessly integrates two distinct frontend applications backed by a single backend API to provide a comprehensive user experience. This integrated approach allows us to maintain separation between administrative tasks and public-facing content delivery, ensuring a responsive and SEO-friendly web application for all users.

Admin Panel (Vue.js Single Page Application)

The integration of Laravel and Vue.js begins with Laravel serving as the foundation. Laravel bootstraps the Vue.js Single Page Application (SPA) by providing the essential web routes, including user authentication and authorization mechanisms. Admin users benefit from a feature-rich Admin Panel built using Vue.js. This panel, hosted within the same domain as the Laravel API, offers an intuitive interface for managing and overseeing various aspects of the application.

Public Content (Nuxt Server Side Rendering)

Recognizing the importance of a distinct, public-facing interface, we've provisioned a separate front-end built on Nuxt, leveraging server-side rendering (SSR). Nuxt, powered by Node.js and managed by pm2, optimizes content delivery for public users. This architecture not only provides swift and efficient content rendering but also excels in search engine optimization (SEO). The Nuxt SSR application utilizes the Laravel application as an API, ensuring data consistency and efficient communication between the frontend and backend.

Infrastructure (Docker, Terraform, Ansible, Jenkins)

Our infrastructure seamlessly integrates with the application layers. For local development, Docker Compose is employed to create a Dockerized environment consisting of Apache/PHP, MySQL, Node.js, and Redis containers. This environment closely mirrors our production setup, ensuring consistency during development.

On live servers, Terraform orchestrates the creation and configuration of servers and DNS records. Ansible Playbooks are then used for provisioning these servers. Two servers are created: one hosts the application, while the other, a Jenkins server, manages our CI/CD processes. The Jenkins server is tightly integrated with GitHub, automating the creation of multipipeline jobs. Each project defines its pipelines, including a Dockerized build environment via Docker Compose. These pipelines manage the build stage and incorporate Ansible Playbooks for zero downtime deployments and rollback mechanisms in case of failures.

3. BEST PRACTICES AND STANDARDS

This section details the coding standards, security practices, and documentation guidelines that should be followed when working with the architecture.

Coding Standards

Frontend Coding Standards

Our Frontend development adheres to a set of coding standards and patterns to ensure consistency and maintainability:

1. TypeScript Utilization: We harness TypeScript to take advantage of static typing. This enhances development consistency and helps identify errors early in the development process.

2. Composition API: We embrace the Composition API, which provides standalone functions representing Vue's core capabilities. This allows us to extract and reuse shared logic between components efficiently.

3. Vue Patterns:

Components: We structure our Vue applications around components, which encapsulate markup (HTML), logic (JS), and styles (CSS).

Composables: Composable functions within the Composition API help break down the application's context into smaller, reusable pieces, promoting modularity and code separation.

State Management: We employ Pinia, a state management pattern and library for Vue.js, to handle application-level state in a structured and scalable manner.

Provide/Inject: The provide() and inject() functions in Vue enable seamless data propagation throughout the component tree, eliminating the need for manual prop-drilling.

4. Global SASS Structure

In our project, we follow the well-established 7-1 Pattern for organizing SASS codebase. All relevant details and guidelines can be found in the respective subfolders' README.MD files within the 'assets/sass' directory.

The 7-1 Pattern is a widely recognized and proven structure for SASS projects, providing a scalable and comprehensible approach to styling. It's rooted in dividing the project into seven distinct folders along with a main file, simplifying the development process by categorizing styles into logical sections. This division makes navigation, debugging, and collaboration more straightforward. From fundamental base styles to specific component designs, the 7-1 arrangement ensures that every aspect of styling has its dedicated space. By adopting this methodology, we aim to enhance code clarity, promote best practices, and maintain efficient and organized stylesheets, regardless of the project's size or complexity.

Backend Coding Standards

Our Backend development adheres to coding standards aligned with the principles of Domain-Driven Design and SOLID principles:

1. Domain-Driven Design (DDD): Our back-end architecture is structured around DDD principles, separating the Business Logic Layer (Services) responsible for core business rules from the Data Access Layer (Repositories) responsible for data persistence.

2. Key Architectural Elements:

Dependency Injection: We actively employ Dependency Injection, which promotes modularity and simplifies the process of testing components.

Repository Pattern: A conduit between Services and Repositories ensures decoupling of business logic from data management, enhancing maintainability.

S.O.L.I.D Principles: S.O.L.I.D principles are rigorously followed throughout the development process, strengthening code clarity, adaptability, and maintainability.

3. Microservices-Ready: Each entity within our Laravel backend resides in a separate folder with its Provider, Routes, Requests, Controllers, Services, Repositories, and Models. This meticulous organization ensures that our application can seamlessly scale into multiple microservices based on the scalability needs arising from increasing user demands.

Infrastructure Coding Standards

Our Infrastructure as Code (IaC) and Dockerization standards ensure consistent and reproducible environments:

1. Infrastructure as Code (IaC): We employ Terraform for IaC, which uses HashiCorp Configuration Language (HCL) to define infrastructure. This approach allows us to create, modify, and version control infrastructure in a consistent manner.

2. Dockerization: Docker and Docker Compose are integral parts of our infrastructure. Docker provides containerization for our application components, ensuring that they run consistently across different environments. Docker Compose allows us to define and manage multi-container applications.

3. Folder Structure: Our infrastructure code and Docker configurations are organized into separate folders, promoting clarity and maintainability. The structure ensures that code for different components remains isolated and manageable.

4. Reusability: We encourage the development of reusable infrastructure modules and Docker images, simplifying the provisioning and deployment of our application components.

By adhering to these standards, we ensure the reliability, scalability, and maintainability of our infrastructure, enabling seamless deployments and operations.

Security Practices

[PENDING]

Documentation Guidelines

[PENDING]

4. IMPLEMENTATION WORKFLOW

This section walks through the step-by-step process of developing, testing, deploying, and monitoring applications within the architecture. This section provides valuable guidance on how to use the architecture effectively.

Development Flow

This section outlines step by step instructions for starting the project locally.

Prerequisites

1. Install Docker Compose and Git locally

Build Development Environment

Step 1.

Clone Starter Architecture Git repository

git clone git@github.com:esofstarter/starter-architecture.git

Step 2.

Create environment variable files in the **infrastructure/dev_env** and **server/api** folders (use sample files as reference)

Step 3.

In folder **infrastructure/dev_env** run:

docker compose build

docker compose up -d

Step 4.

Build the application using the respective Docker containers

For the Laravel API start the app container by running:

docker exec -it app /bin/bash

Then in folder within the app container **server/api** run the following commands:

composer install && php artisan config:clear && php artisan view:clear && php artisan route:clear && composer dump-autoload && php artisan cache:clear && php artisan config:cache && php artisan route:cache

Run these commands to migrate and populate the database:

php artisan migrate:fresh && php artisan db:seed

For the Vuejs Admin Panel SPA start the app container by running:

docker exec -it node /bin/bash

Then in folder within the node container **server/client/admin** run the following commands:

npm install && npm run dev

For the Nuxt Public Content SSR start the app container by running:

docker exec -it node /bin/bash

Then in folder within the node container server/client/public run the following commands:

npm install && npm run dev

Testing Process

[PENDING]

Deployment Process

This section outlines step by step instructions for creating the hosted infrastructure.

Prerequisites

1. Install Terraform and Ansible locally

Cold Start Infrastructure as Code

Step 1.

Create DigitalOcean Personal Access Token

https://cloud.digitalocean.com/account/api/tokens

Step 2.

Clone Starter Architecture Git repository

git clone git@github.com:esofstarter/starter-architecture.git

Step 3.

Create sensitive information files.

For Terraform project create file **terraform.tfvars** in **infrastructure/terraform** with the DigitalOcean access token and the path to your ssh keys

For Host server create folder sensitive in **infrastructure/host** with the following files containing a single string:

.env (This is the .env file for the hosted Laravel backend)

database_password

domain_name

supervisor_password

For Jenkins server create folder sensitive in **infrastructure/jenkins** with the following files containing a single string:

admin_password

database_password

domain_name

github_access_token

Step 4.

In folder **infrastructure/terraform** run Terraform commands:

terraform init

terraform plan

terraform apply

Step5. Go to **jenkins.thestarter.net** and login with admin user

admin

(admin_password)

Monitoring and Logging

[PENDING]

5. TROUBLESHOOTING AND ISSUE RESOLUTION

This section equips readers with the knowledge needed to handle common issues and resolve problems. It also introduces debugging techniques to assist in issue resolution.

Common Issues

[PENDING]

Debugging Techniques

[PENDING]

SPECIFICS

Bootstrap VueJs SPA using Laravel

Firstly we create a single Blade View:
Inside resources/views, we create app.blade.php. This file will serve as the main template for our SPA. Ensure we have ** **. This is where Vue will mount our SPA.

Secondly we define the web routes and direct them to a single controller method:
Route::get('/{any}', [HomeController::class,'index'])->where('any', '.*');

Asset Bundling

Vite is a build tool that aims to provide a faster and leaner development experience for modern web projects. Unlike traditional tools like Webpack, Vite serves your source files over native ES modules during development, resulting in significantly faster update and startup times.

In order to bootstrap the Vite integration it is necessary to add it in the blade template:
@vite('src/app.ts')

SPA Authentication

Sanctum also exists to provide a simple method of authenticating single page applications (SPAs) that need to communicate with a Laravel powered API. These SPAs might exist in the same repository as your Laravel application or might be an entirely separate repository.

For this feature, Sanctum does not use tokens of any kind. Instead, Sanctum uses Laravel's built-in cookie based session authentication services. This approach to authentication provides

the benefits of CSRF protection, session authentication, as well as protects against leakage of the authentication credentials via XSS.

Routing and Components

In the router we define the layout components that wrap the page components and also define if the user needs to be authenticated to access them.

The folder structure of the components adheres to the below described rules:

layouts: This folder will contain all layout components, which are essentially wrappers or parent components that define the structure but not the main content of the page. They usually include headers, footers, sidebars, and a slot or router-view for child components. So, your AdminLayout.vue, AdminHeader.vue, and AdminSidebar.vue would be in this folder.

pages: This folder will contain components that represent entire views or pages in your application. Each component in this folder typically corresponds to a route in your application. They are the main content that gets displayed within the layout components.