Design document

Contents

1. Introduction 2
2. Requirements 3
3. Design 4
    - 3.1. Architecture 4
    - 3.2. S3 bucket 5
    - 3.3. Route 53, CloudFront and ACM certificates 7
    - 3.4. HTTP API Gateway 9
    - 3.5. Lambda 11
    - 3.6. DynamoDB 13
    - 3.7. Firebase Authentication 15
    - 3.8. Terraform Configuration 17
    - 3.9. Frontend design 18
    - 3.10. GitHub Actions (wip) 20
4. Cost Calculations (wip) 21

1. Introduction

This document explains the design for a serverless football merchandise store, focusing on how it uses AWS services, Firebase Authentication, CloudFront, and Terraform. The platform is designed to be secure, scalable, and cost-effective, combining a modern frontend with a serverless backend to create a smooth e-commerce experience.

The key features include browsing products stored in DynamoDB, retrieving images from S3 using API Gateway, and handling user authentication through Firebase Authentication. CloudFront speeds up the website by caching static assets, making load times faster and improving the overall user experience. Terraform is used to define the architecture, allowing for high availability, easy scalability, and quick deployments.

This project is intended as a portfolio piece to demonstrate my skills in cloud architecture, serverless solutions, and modern web development. The target audience includes individuals interested in the project who seek a deeper understanding of its design and functionality. It showcases how various AWS and cloud tools can be integrated to build a functional, efficient, and professional-grade e-commerce platform. The ultimate goal is to highlight my expertise while delivering a fast, reliable, and user-friendly experience.

2. Requirements

The goal is to deliver a serverless football merchandise store that meets my predefined objectives for the project. The solution must align with the following requirements:

- **Cost-Effective:** Keep operational costs as low as possible by utilizing serverless and pay-as-you-go AWS services. Costs during the development need to be below 10$ and costs during production less than 30$ for a smaller user base. The free tier must be utilised as much as possible.
- **Efficient and Well-Designed:** Build the architecture using AWS Well-Architected Framework principles, ensuring it is modular, maintainable, and easy to extend for future improvements. Infrastructure as code coverage must be at 100% level and the project must be fully documented until the of the semester.
- **AWS-Centric:** Take full advantage of AWS services for storage, APIs, and serverless computing to showcase my cloud skills effectively. A maximum of 1 external service is allowed, like Firebase Authentication, to ensure high AWS coverage.
- **Secure and Reliable:** Implement strong authentication, data protection, and fault-tolerance measures to ensure a robust and dependable platform. Implement AWS IAM for role-based access control, API Gateway authorization, and HTTPS with TLS for all API endpoints.
- **High Performance:** Optimize the platform for fast response times and smooth user experience while managing costs efficiently. Optimize API Gateway and Lambda functions to achieve response times under 500ms for 95% of requests under expected load.
- **Scalable:** Ensure the platform is capable of handling future traffic growth and feature expansions without requiring major redesigns. Design the architecture to handle a 5x increase in traffic and data with no more than a 20% increase in operational costs.

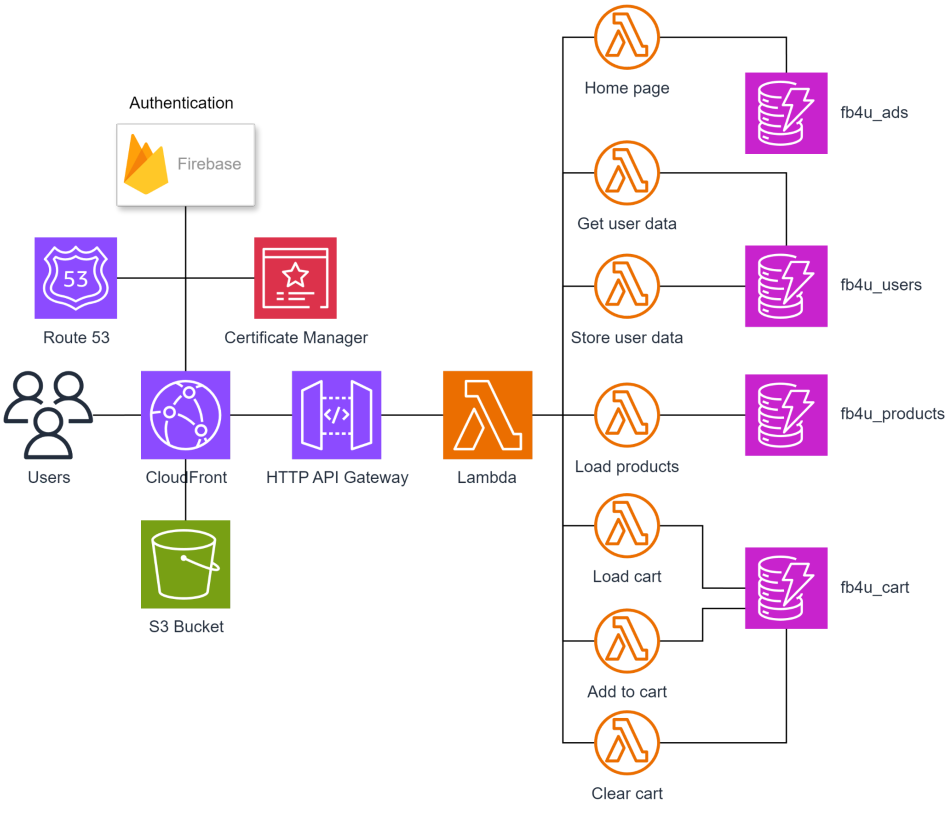
3. Design

3.1. Architecture

The architecture of the football merchandise store is built on a serverless model, ensuring scalability, cost-efficiency, and ease of management. Key components include AWS DynamoDB for storing product and user information, S3 for hosting static assets and product images, and API Gateway for routing HTTP requests to backend services powered by AWS Lambda functions written in Python. User authentication is managed through Firebase Authentication, simplifying secure login and session handling. AWS services are primarily deployed in eu-central-1 (Frankfurt), while global services like CloudFront and TLS certificates are deployed in us-east-1 (North Virginia), AWS's primary region for such features.

To enhance performance, security, and availability, the platform integrates AWS CloudFront as a Content Delivery Network (CDN), paired with a TLS certificate to deliver content securely over HTTPS. CloudFront leverages its globally distributed edge locations to cache static assets for faster delivery to users worldwide. The website is accessible through a custom domain managed by Amazon Route 53. Terraform is used to define and deploy the infrastructure as code, ensuring consistency and easy reproducibility.

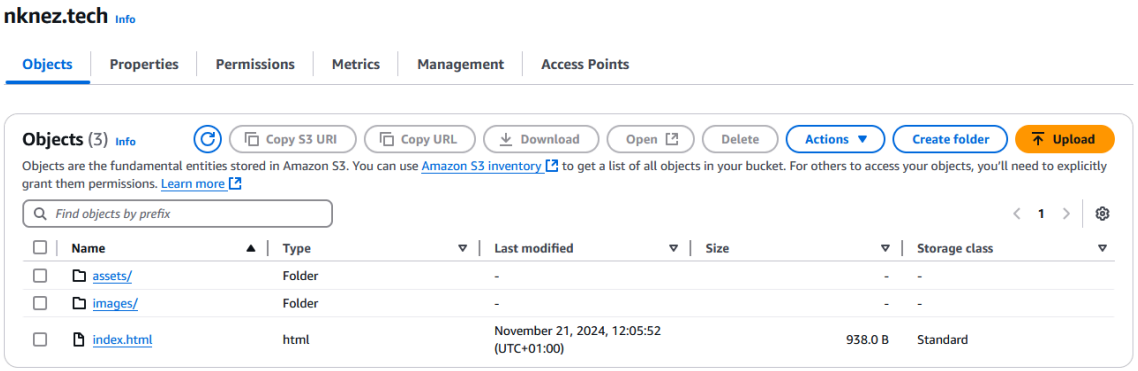
The diagram below visually represents the architecture. Users access the website through CloudFront, configured with Route 53 for a custom domain and secured with TLS certificates from Certificate Manager. CloudFront retrieves uncached static assets from the S3 Bucket and serves them globally. Backend services, including user management, product data, and cart operations, are handled by dedicated Lambda functions. DynamoDB serves as the primary database engine for efficiently storing product details, user profiles, and cart information.



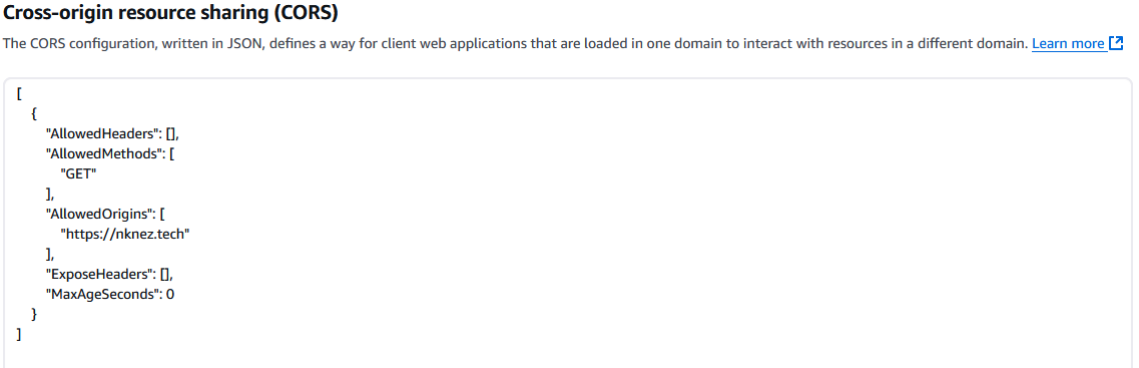
3.2. S3 bucket

The S3 bucket for the football merchandise store is configured to serve static website files efficiently and securely. It is tailored to host and deliver the website's frontend assets, including HTML, CSS, JavaScript, and images. Below are the key aspects of its configuration:

- **Bucket name**: The bucket is named after the project's domain, **nknez.tech**. By associating the bucket with this custom domain, the website can be accessed through a professional URL. Combined with CloudFront and Route 53, this setup ensures fast and secure delivery of content over HTTPS with a TLS certificate managed by AWS Certificate Manager.
- **CORS**: The bucket allows GET requests exclusively from the project's domain through its Cross-Origin Resource Sharing (CORS) settings. This ensures that only approved origins can request resources, enhancing security.
- **Public access and policies**: Public access is enabled to serve static website files publicly. The bucket policy grants s3:GetObject permissions to all users, ensuring that files are accessible without permission issues, while security is managed with strict origin controls.
- **Static website hosting**: The bucket is configured to operate as a static website, with index.html set as the main entry point and error.html as the fallback page for handling errors or invalid URLs.



S3 bucket with uploaded website files.



CORS settings inside the S3 bucket.

.

Block public access (bucket settings)

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that p
apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, er
access to your buckets or objects within, you can customize the individual settings below to suit your specific storage use cases. Learn more

Block *all* public access

⚠ Off

▼ **Individual Block Public Access settings for this bucket**

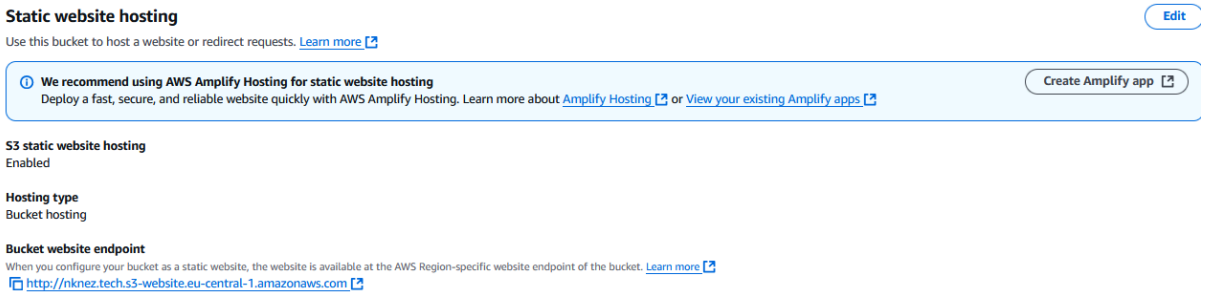
- **Block public access to buckets and objects granted through *new* access control lists (ACLs)**
  S3 will block public access permissions applied to newly added buckets or objects, and prevent the creation of new public access ACLs for existing resources using ACLs.
- **Block public access to buckets and objects granted through *any* access control lists (ACLs)**
  S3 will ignore all ACLs that grant public access to buckets and objects.
- **Block public access to buckets and objects granted through *new* public bucket or access point policies**
  S3 will block new bucket and access point policies that grant public access to buckets and objects. This setting doesn't change any existing policie
- **Block public and cross-account access to buckets and objects through *any* public bucket or access point policies**
  S3 will ignore public and cross-account access for buckets or access points with policies that grant public access to buckets and objects.

Bucket policy

The bucket policy, written in JSON, provides access to the objects stored in the bucket. Bucket policies don't apply to objects owne

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::nknez.tech/*"
    }
  ]
}
```

Block public access feature disabled and bucket policy set up to ensure the website is accessible.

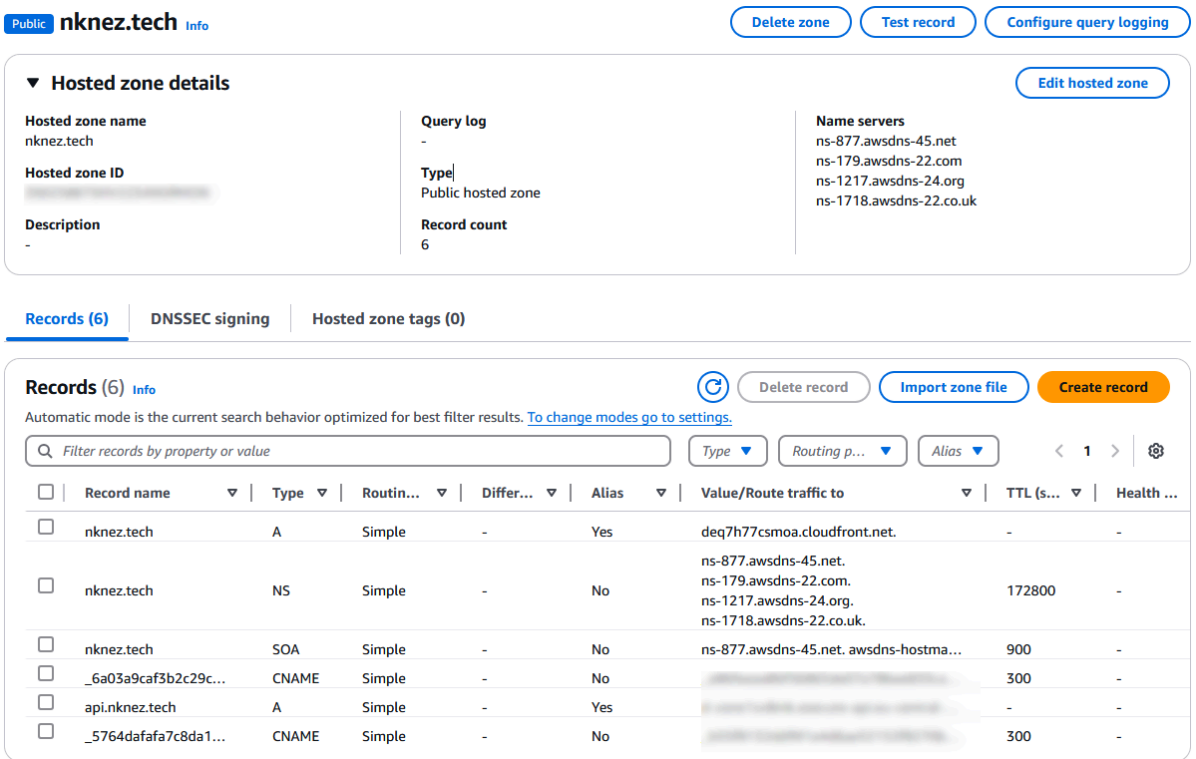


Static website hosting feature enabled.

3.3. Route 53, CloudFront and ACM certificates

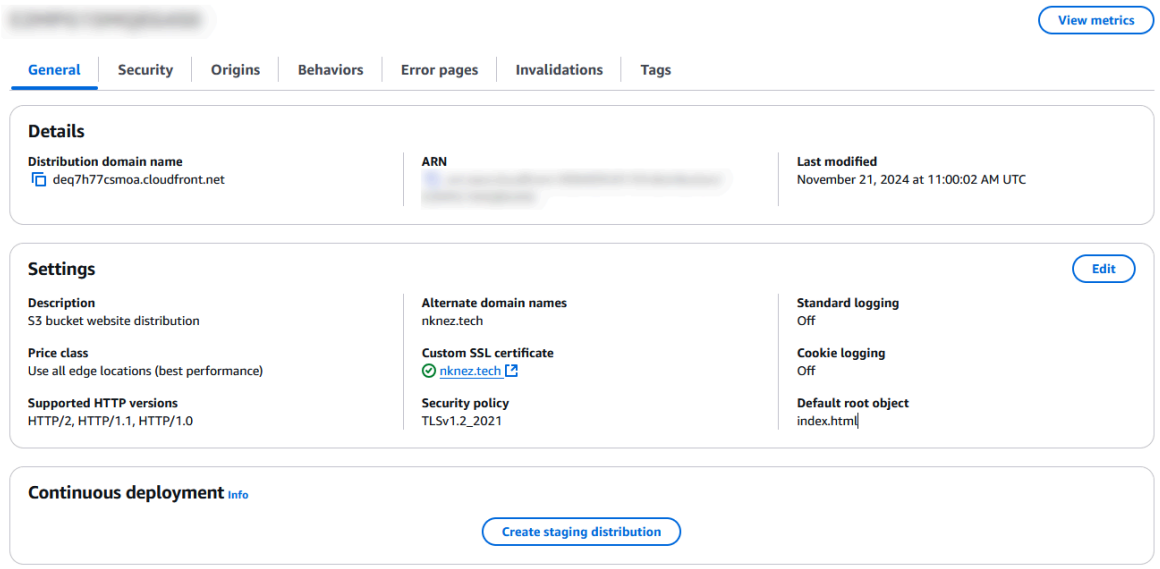
The football merchandise store utilizes Route 53, CloudFront, and AWS Certificate Manager (ACM) to ensure secure and efficient delivery of the website over the project's custom domain. These services work together to enhance the platform's accessibility, security, and performance. Below are the key aspects of their configuration:

- **Route 53 for domain management**: It manages the DNS settings for the **nknez.tech** domain, enabling seamless redirection to the CloudFront distribution. An A record is created to alias the domain to the CloudFront distribution, ensuring the website is accessible via a professional and user-friendly URL. Additionally, an A record for **api.nknez.tech** simplifies the API Gateway URL, facilitating communication between the frontend and backend. While the hosted zone is currently configured manually, automating this process in the future could enhance deployment consistency and reduce manual effort.

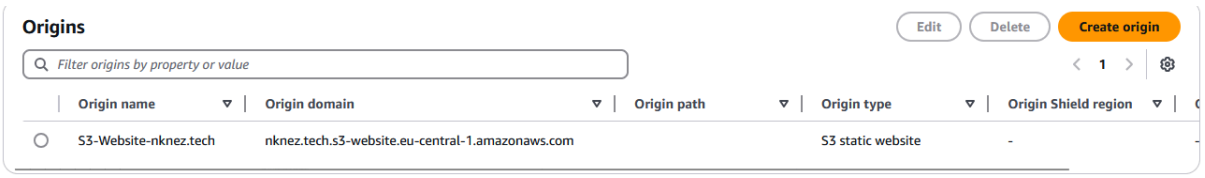


The hosted zone configuration in the AWS Console displays A records for both the CloudFront distribution and the API Gateway custom domain, along with CNAME records used for ACM certificate validation. Certain details have been blurred for security purposes.

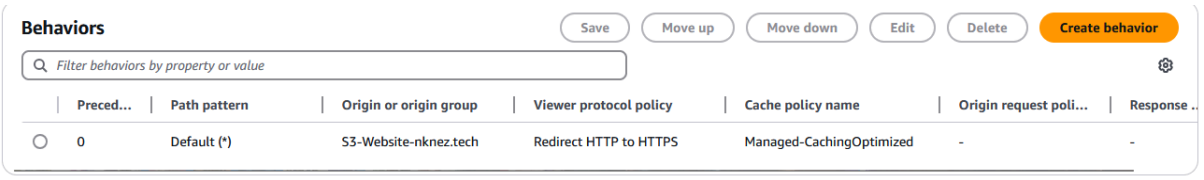
- **CloudFront as a CDN**: It is used to cache static assets like HTML, CSS, and images at edge locations to deliver the website globally. This reduces latency and ensures fast load times for users worldwide. The distribution is configured to fetch content from the S3 bucket's website endpoint over HTTP, with an automatic redirect to HTTPS for secure connections. Key features include:
    - **Viewer Protocol Policy**: Enforces HTTPS connections by redirecting HTTP requests.
    - **Default Cache Behavior**: Allows only GET and HEAD requests, with caching policies optimized for static content.
    - **TLS Configuration**: Supports TLSv1.2_2021 for enhanced security.



CloudFront distribution. It has set up an SSL certificate with an alternate domain, price class to all edge locations, and supported HTTP versions.

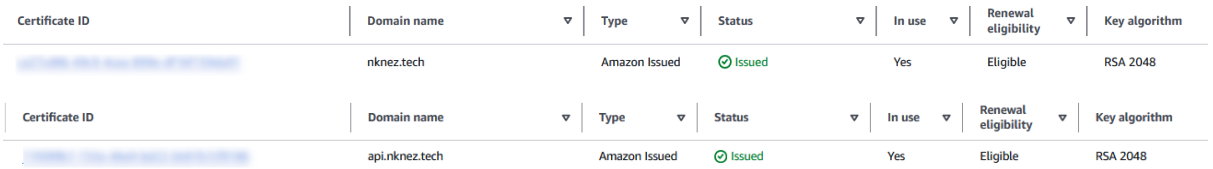


Origin set up for the S3 website endpoint. Necessary to cashe the website from the S3 bucket.



The default behaviour for the cashing. It redirects HTTP to HTTPS and uses the Managed-CashingOptimized policy.

- **Certificate Manager for SSL/TLS certificates**: An ACM certificate is provisioned in the us-east-1 region to secure the CloudFront distribution with HTTPS, as required for global services. This certificate ensures that the website is delivered securely over the nknez.tech domain. Additionally, another ACM certificate is deployed in the eu-central-1 region to secure the API Gateway custom domain (api.nknez.tech). This certificate uses a regional endpoint type and supports TLS 1.2, ensuring secure communication between the frontend and backend. The certificates are validated using DNS, with plans to automate the validation process in the future for streamlined deployment.



Certificate ID	Domain name	Type	Status	In use	Renewal eligibility	Key algorithm
[redacted]	nknez.tech	Amazon Issued	Issued	Yes	Eligible	RSA 2048

Certificate ID	Domain name	Type	Status	In use	Renewal eligibility	Key algorithm
[redacted]	api.nknez.tech	Amazon Issued	Issued	Yes	Eligible	RSA 2048

Two TSL certificates for website and API.

3.4. HTTP API Gateway

The HTTP API Gateway is a critical component of the football merchandise store, facilitating seamless and secure communication between the frontend and backend services. It routes HTTP requests to AWS Lambda functions, which handle key application functionalities such as user data management, product loading, and shopping cart operations. Below are the key features of the HTTP API Gateway configuration:

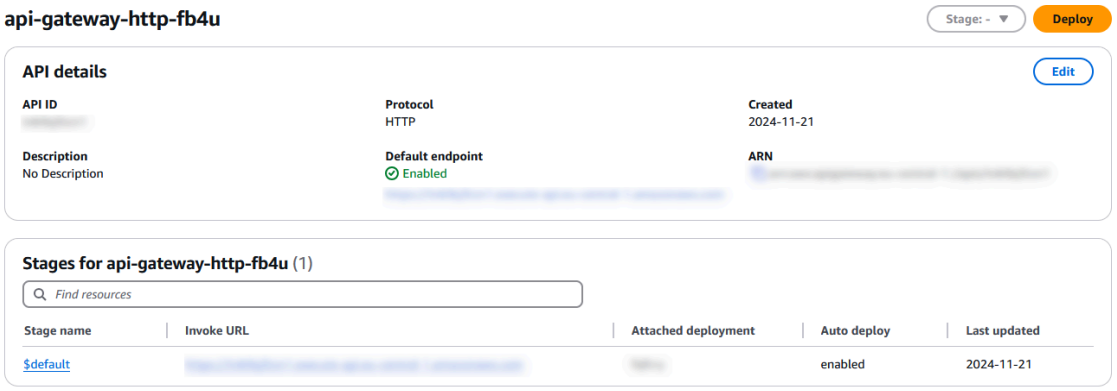
- **Protocol and CORS Configuration**: Configured with the HTTP protocol for cost-efficiency and flexibility, the API Gateway supports GET, POST, and OPTIONS methods. A permissive CORS policy is implemented, allowing requests from any origin for ease of testing, though this can be restricted in the future.

- **Custom Domain and Security**: The API Gateway is associated with the custom domain (api.nknez.tech) for a professional user experience. It is secured using an ACM certificate in the eu-central-1 region, with TLS 1.2 encryption ensuring all communication is secure.

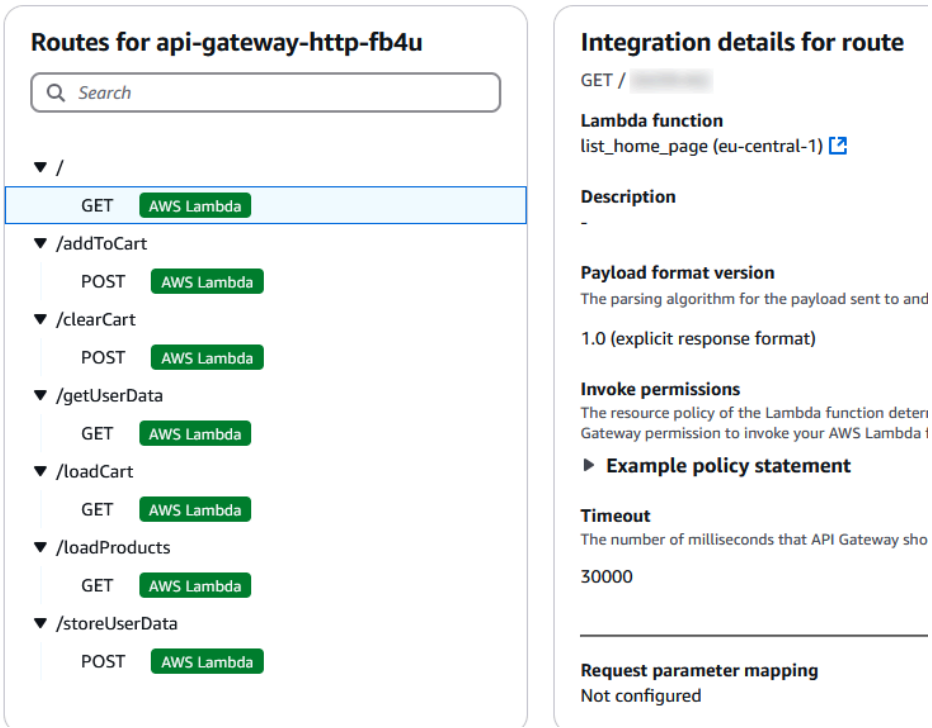
- **Deployment:** The API Gateway uses the $default stage with auto-deployment enabled, ensuring that any configuration changes are applied immediately without manual intervention.

- **Lambda Permissions**: IAM permissions are configured to allow the API Gateway to invoke specific Lambda functions securely, ensuring restricted access to backend services.

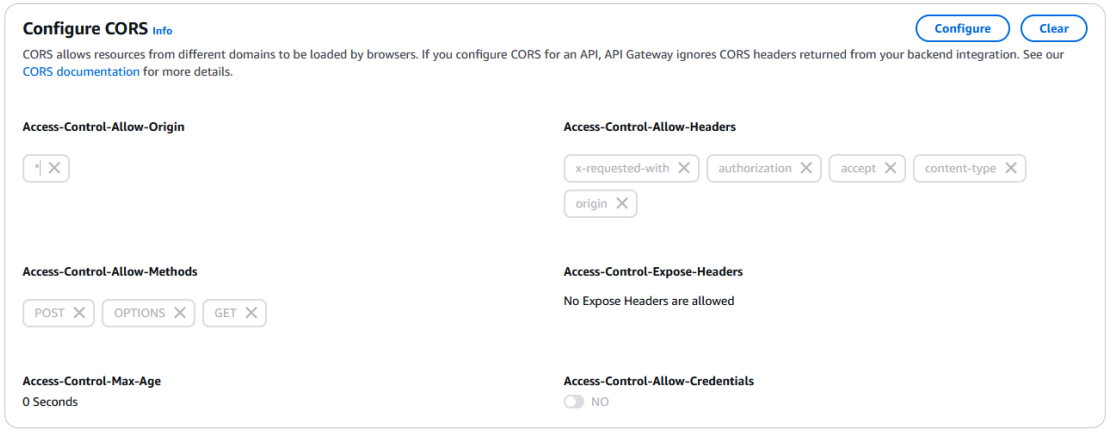
- **Defined Routes and Integrations**: Routes are defined for each functionality, with each route linked to a specific AWS Lambda function to handle the business logic. Key routes include:
    - GET /: Fetch ads to display on the homepage.
    - GET /loadProducts: Fetch products to list them on a page..
    - POST /storeUserData: Save custom user data in the database.
    - GET /getUserData: Fetch user details to display them on the user page.
    - POST /addToCart: Add items to the cart.
    - GET /loadCart: Retrieve cart contents to display them to the user.
    - POST /clearCart: Fully clear the shopping cart.



API Gateway set up. HTTP protocol is used and the default stage is deployed.



Exposed routes. Each route is integrated with a specific lambda for the backend functionality.



Cors settings. Only POST OPTIONS and GET are allowed. All origins and headers are allowed.

3.5. Lambda

AWS Lambda functions form the backbone of the backend for the football merchandise store, providing serverless and scalable execution of business logic in response to API Gateway requests. These functions interact with DynamoDB for data management and are configured for optimal performance and security. Below are the key details of their setup:

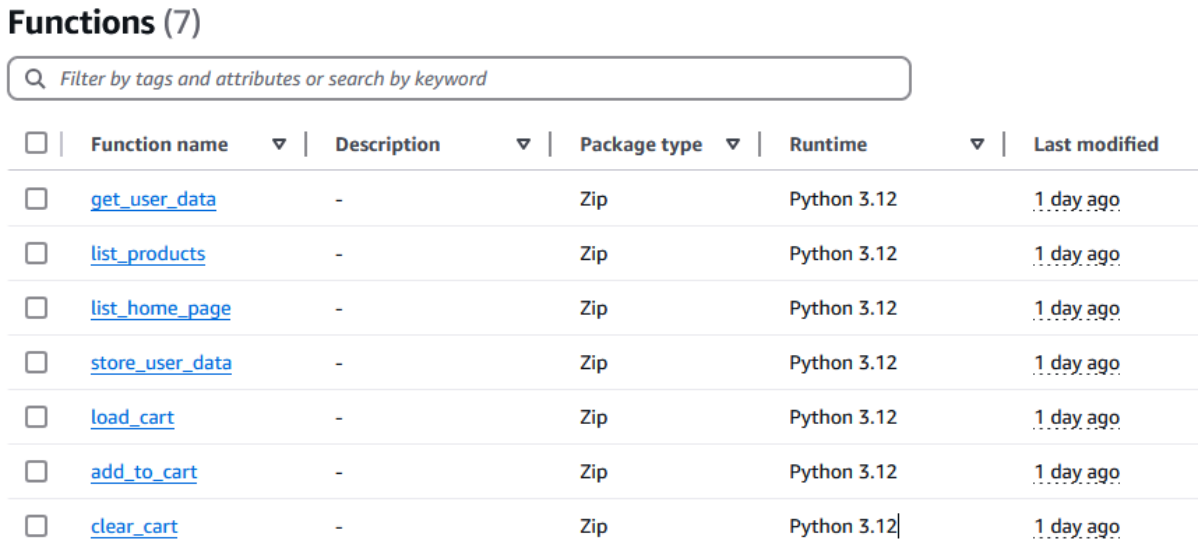
- **Runtime and Configuration**: The Lambda functions are built using Python 3.12, ensuring compatibility with modern libraries and tools. Each function is deployed with standardized settings for memory size (1GB), execution timeout (20 seconds), and runtime, defined through Terraform variables for consistency.

- **Roles and Permissions:** A dedicated IAM role is assigned to all Lambda functions, following the principle of least privilege. The role grants:
    - Read/write access to DynamoDB tables (fb4u_products, fb4u_ads, fb4u_users, fb4u_cart) for managing application data.
    - Write permissions to CloudWatch for creating log groups, streams, and events, enabling detailed monitoring and debugging.

- **Logging and Debugging**: All functions log activity and errors to CloudWatch, enabling real-time monitoring and troubleshooting.

- **Functionality:** Each Lambda function handles a specific operation and is integrated with the corresponding API Gateway route:
    - list_products Lambda - Route: GET /loadProducts
    - list_home_page Lambda – Route: GET /
    - store_user_data Lambda - Route: POST /storeUserData
    - get_user_data Lambda - Route: GET /getUserData
    - add_to_cart Lambda - Route: POST /addToCart
    - load_cart Lambda- Route: GET /loadCart
    - clear_cart Lambda - Route: POST /clearCart

**Each lambda’s functionality is briefly explained in the API Gateway paragraph and more in detail in the Python documentation pdf file.



Each Lambda function is added through zip files. Lambdas are using Python 3.12 runtime.

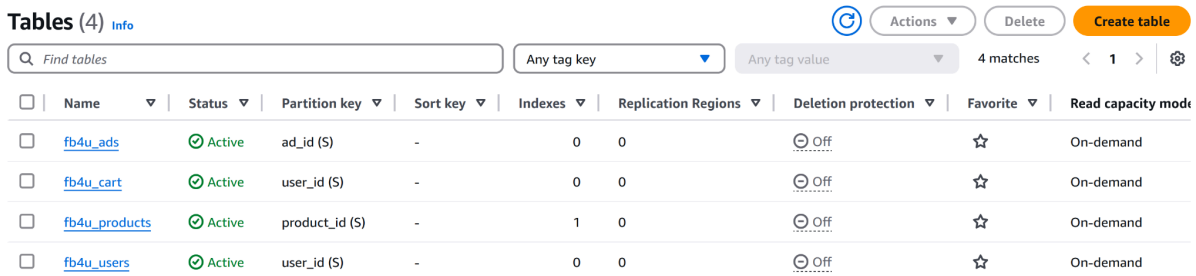
3.6. DynamoDB

DynamoDB tables serve as the primary data storage layer for the football merchandise store, enabling fast, scalable, and cost-efficient data management. Each table is tailored for specific application functionalities, with a PAY_PER_REQUEST billing mode to optimize costs based on usage. Below are described tables in terraform:

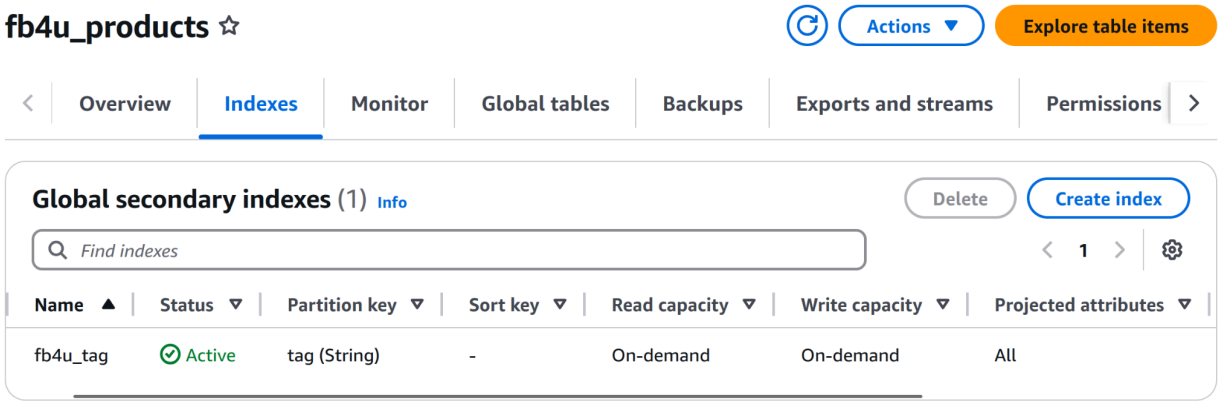
- **fb4u_products**
  - The structure of the table is: product_id (primary key), team_name, side, season, price, tag and description.
  - Stores product information, enabling the application to query product details efficiently.
  - An additional Global Secondary Index, fb4u_tag, is used for querying products by tags or categories, enhancing flexibility in product searches.
- **fb4u_ads**
  - The structure of the table is ad_id (primary key), ad_label and product_id.
  - Stores advertisements and featured products are displayed on the homepage.
- **fb4u_users**
  - The structure of the table is user_id (primary key), email, first_name and last_name. There are optional keys like street_name, post_code, city, country, and phone_number, which are dependent if the user fills in data on the website or not.
  - Maintains user personal data for the application.
- **fb4u_cart**
  - The structure of the table is user_id (primary key) and items that contain product_id and size in array format.
  - Stores shopping cart details for each user, linking the cart directly to their unique user_id.

Key features:

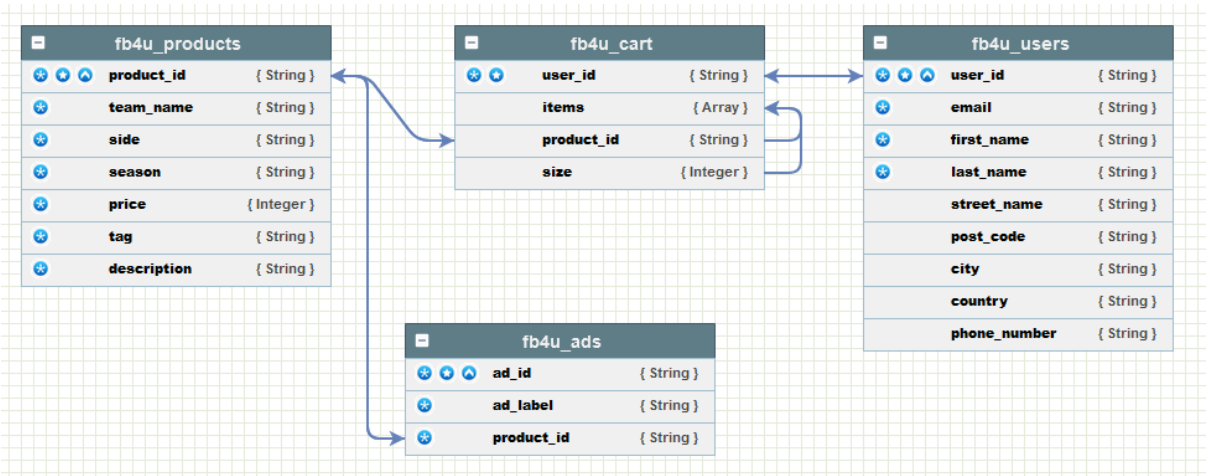
- **Cost Efficiency**: DynamoDB's PAY_PER_REQUEST billing mode ensures cost-effectiveness by charging only for actual reads and writes.
- **Performance and Scalability**: DynamoDB automatically scales to handle peak loads, ensuring consistent performance.
- **Securit**y: IAM roles restrict table access to authorized Lambda functions, ensuring secure interactions.
- **Data Consistency**: Strongly consistent reads ensure up-to-date data for critical operations.



Tables created shown in AWS console. You can see the names, partition keys, read capacity mode switch to on-demand and other useful information.



Global secondary index created in fb4u_products for querying purposes.

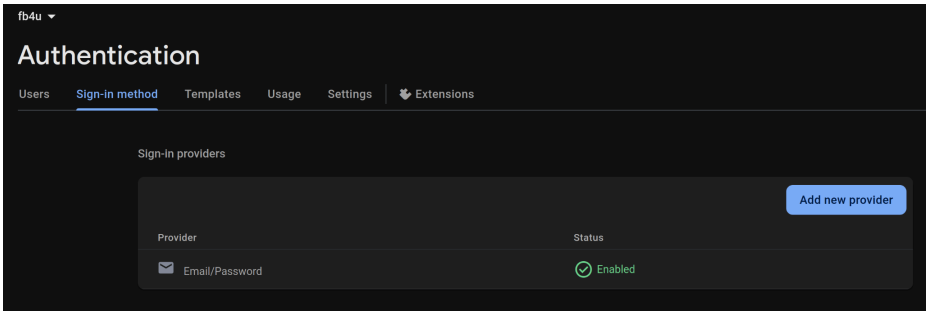


Current NOSQL database model. Variables without a star are optional.

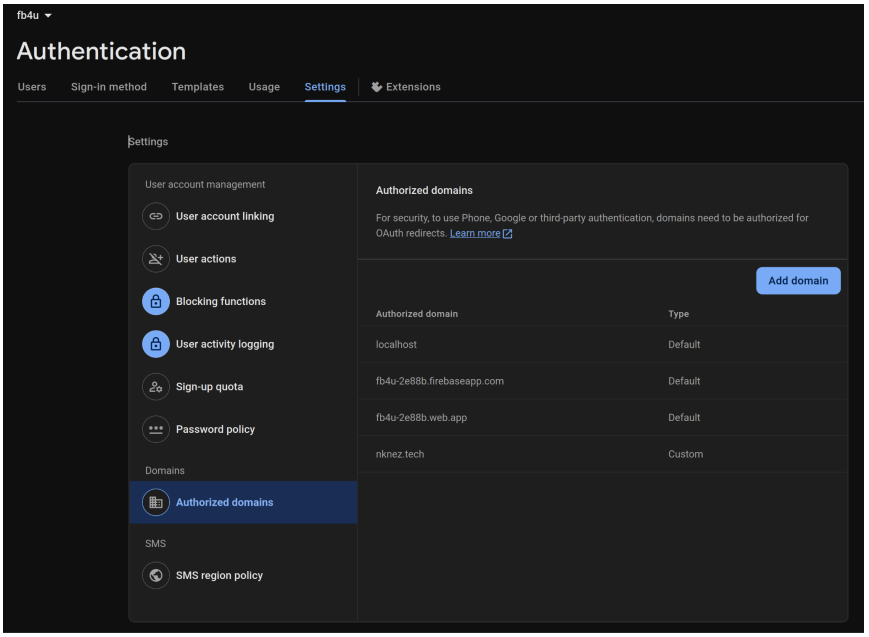
3.7. Firebase Authentication

Firebase Authentication is implemented in the football merchandise store to allow users to create their own accounts, enabling features such as viewing order history and saving the current shopping cart. Currently, the authentication method supports email/password login, with the potential to add an anonymous login in the future. Password policy is setup to require a special, uppercase, lowercase and a number and a password needs to be a minimum length of 8 characters. Firebase Authentication is directly integrated into the Vue.js frontend, providing a seamless and user-friendly experience. Other user details that are defined in registration or account settings are stored in DynamoDB.

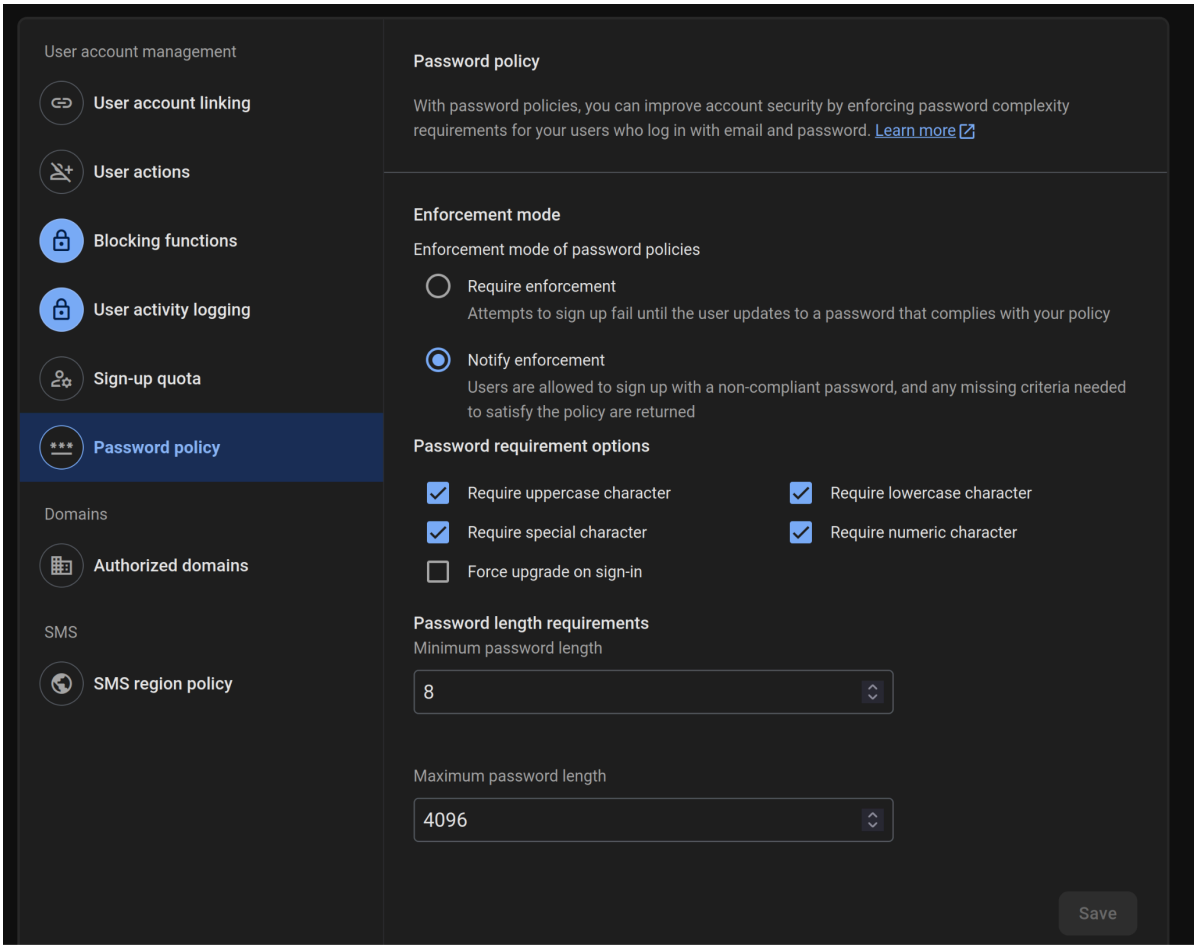
While Firebase Authentication is primarily used for account creation and login, it also supports secure token validation and integrates with Vue.js for real-time authentication updates. Although it serves the current requirements, the authentication system may be transitioned to AWS Cognito in the future to align more closely with the AWS ecosystem and enable advanced features like multi-factor authentication (MFA) or deeper integration with AWS services.



Authentication setup in Firebase with email/password sign in provider.



nknez.tech domain authorized to use the api keys of the firebase.



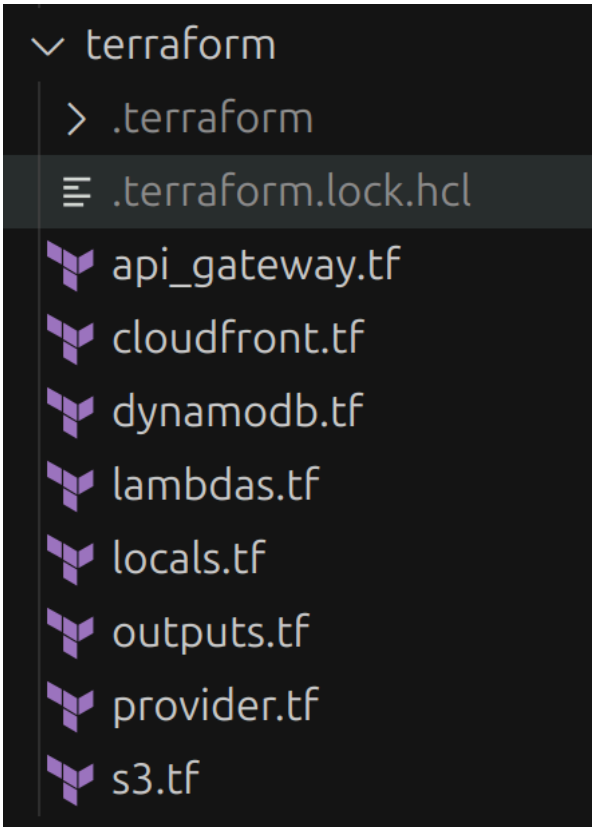
Password policy setup to password to be a minimum length of 8 characters and require lowercase, uppercase, special character and a number.

3.8. Terraform Configuration

The football merchandise store's infrastructure is fully defined and deployed using Terraform, enabling a consistent, repeatable, and scalable setup across environments. Terraform acts as the foundation for managing all AWS resources, ensuring infrastructure as code (IaC) principles are strictly followed. Below are the highlights of the Terraform implementation:

- **Infrastructure Definition**: Each AWS service mentioned earlier is configured using dedicated Terraform modules. This modular approach avoids redundancy and enhances maintainability.
- **Automation and Scalability:** The use of Terraform allows automated deployment of resources, with various variables ensuring flexibility and scalability without the need for hardcoding.
- **Integrated Configuration:** All components, from Route 53 DNS records to CloudFront distributions and ACM certificates, are integrated seamlessly. The configuration also employs resource dependencies to ensure proper deployment order.
- **Outputs for Easy Management:** Outputs are defined for key resources, such as the S3 website URL and API Gateway endpoints, simplifying access and management during development and testing.

**Each terraform file is explained in more detail in the Terraform Documentation pdf.



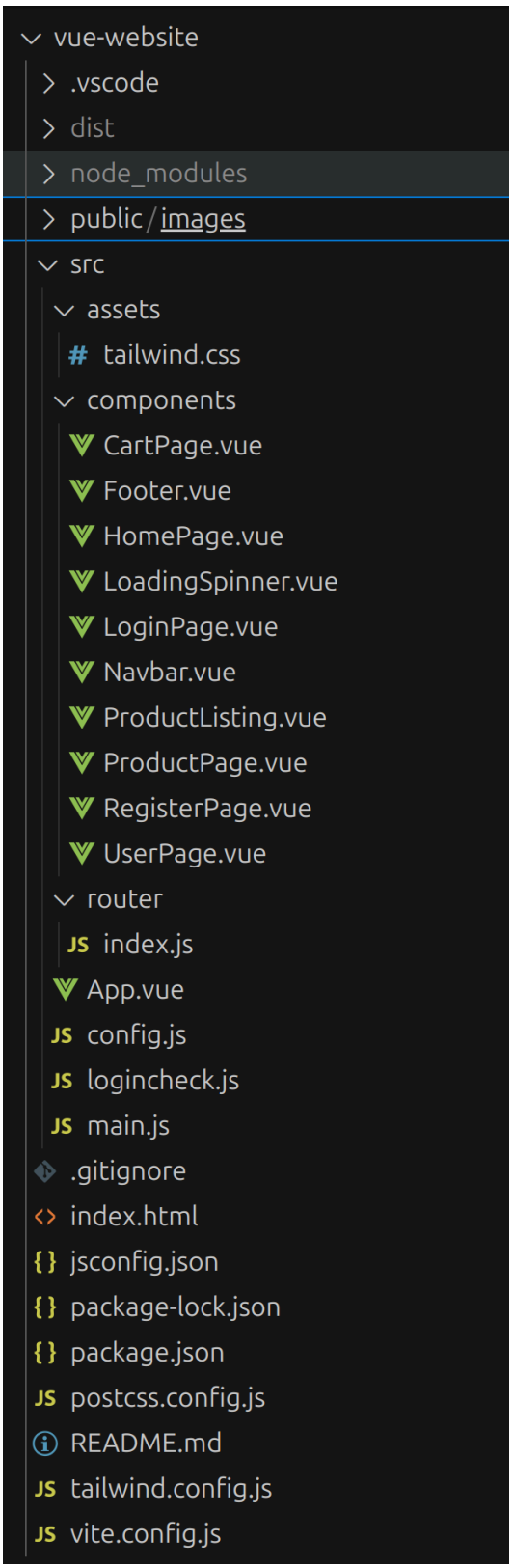
Current file structure of Terraform environment.

3.9. Frontend design

The frontend for this project is built using Vue.js, styled with Tailwind CSS, and utilizes Vite as a bundler to streamline development and optimize performance.

- **Framework:** Vue.js provides a reactive and modular approach to building the application, enabling efficient component-based development. The use of .vue files allows for a clean separation of HTML, JavaScript, and CSS for better maintainability. Core functionalities, such as user authentication and dynamic UI rendering, are implemented using Vue's reactivity system.
- **Styling:** Tailwind CSS simplifies the styling process by providing utility-first classes. This approach ensures a consistent design throughout the application without the overhead of writing custom CSS. The flexibility of Tailwind enables rapid prototyping and supports responsive design with its built-in breakpoints and utilities.
- **Bundler**: Vite serves as the project's build tool, offering fast Hot Module Replacement (HMR) during development and optimized builds for production. It efficiently handles the project's JavaScript, CSS, and other assets, ensuring quick load times and minimal bundle sizes.
- **Integrations**:
    - The HTML structure is defined in index.html, which links the required fonts, styles, and scripts. It utilizes Google Fonts for typography and serves as the entry point for Vue components.
    - The application's main logic resides in the main.js file, where the Vue instance is initialized and the App.vue root component is rendered.
    - Additional functionality, such as login checks and configuration, is modularized into separate JavaScript files (config.js, logincheck.js) to maintain code organization.
    - The functionality of the web application is split into vue components to enhance manageability and scalability.

By leveraging Vue.js for componentized architecture, Tailwind CSS for rapid styling, and Vite for efficient development, the front end is optimized for scalability, maintainability, and user experience. This structure ensures that the project can be extended and improved upon in the future with minimal effort.
** Frontend code is more detailed in the Javascript Documentation pdf.

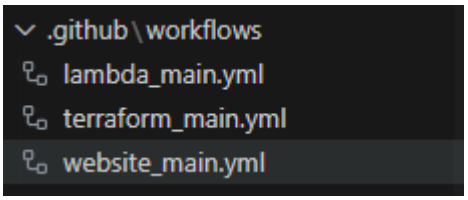


Current website file environment.

3.10. GitHub Actions (wip)

The project employs GitHub Actions for Continuous Integration and Continuous Deployment (CI/CD) to streamline development and deployment processes. Each component of the stack is automated with dedicated workflows, ensuring efficient updates and consistency across environments.

- **lambda_main.yml** - Automates the packaging and deployment of backend Lambda functions. Triggers on changes to the “lambdas/**” directory in the main branch.
- **terraform_main.yml** - Handles infrastructure updates and data initialization in DynamoDB. Triggers on changes to “terraform/**” and “scripts/**” in the main branch.
- **website_main.yml** - Automates the build and deployment of the Vue.js frontend to an S3 bucket. Triggers on changes to “vue-website/**” in the main branch.



CI/CD file environment

**More detailed descriptions of CI/CD pipelines are in the GitHub Actions documentation pdf.

4. Cost Calculations (wip)

This section provides a detailed analysis of the estimated costs for the football merchandise store across three different usage scenarios, considering the serverless stack and architecture outlined in this document. Each scenario includes a breakdown of service-specific costs to account for differing traffic levels and operational requirements.

Scenarios:

- **Development Scenario**: Minimal usage during development and testing.
    - Frontend: 1,000 monthly visits, minimal traffic to CloudFront and S3.
    - Backend: Light usage with 500 API Gateway requests and corresponding Lambda invocations.
    - Database: DynamoDB handling 100 reads and writes daily, all within the free tier.
    - Authentication: 50 Firebase Authentication sign-ups or logins per month.
- **Small User Base Scenario:** Modest production usage with cost-efficiency as a priority.
    - Frontend: 10,000 monthly visits with CloudFront caching most assets, reducing S3 costs.
    - Backend: 5,000 API Gateway requests and corresponding Lambda invocations.
    - Database: DynamoDB handling 1,000 reads and 500 writes daily with occasional queries on Global Secondary Index.
    - Authentication: 500 Firebase Authentication logins per month.
- **Peak Traffic Scenario:** High traffic volumes during surges to test scalability.
    - Frontend: 100,000 monthly visits during peak usage with a high percentage of CloudFront cache hits, leading to increased bandwidth usage.
    - Backend: 50,000 API Gateway requests and corresponding Lambda invocations with some spikes in response times.
    - Database: DynamoDB handling 10,000 reads and 5,000 writes daily, including queries on Global Secondary Index.
    - Authentication: 5,000 Firebase Authentication logins during the peak.

TODO:
Change requirements to be aligned with SMART framework
Enhance the GitHub Actions paragraph
Table cost comparison
Future additions?
Conclusion