# **Technical Specification Document: strideelectronics.com**

## **1. Overview**

This document outlines the technical architecture and implementation details for **strideelectronics.com**, an e-commerce platform built on AWS using a Django backend, MongoDB as the database, and a microservices architecture. All services will follow secure, authenticated communication using HTTPS and JWT-based token verification.

## **2. Technology Stack**

| **Layer** | **Technology** |
| --- | --- |
| Frontend | React.js + Tailwind CSS |
| Backend API | Django (REST Framework) |
| Database | MongoDB (Managed via Atlas or self-hosted on EC2) |
| Auth & Tokens | JWT (stored & rotated securely) |
| Infrastructure | AWS (EC2, ALB, S3, RDS, IAM, Secrets Manager, VPC, etc.) |
| DevOps | Docker, GitHub Actions, Terraform/CloudFormation |
| Communication | HTTPS via AWS ACM |
| Monitoring | AWS CloudWatch, ELK Stack |

## **3. Updated Infrastructure Overview: AWS EKS Deployment**

### **3.1. Core Components**

* **Amazon EKS Cluster**: Manages the Kubernetes control plane.
* **Node Groups**:  
  + **Frontend Node Group**: Hosts the React.js frontend application.
  + **Backend Node Group**: Runs Django REST API services.
  + **Microservices Node Group**: Hosts containerized microservices (e.g., Payments, Orders).
* **AWS Load Balancer Controller**: Manages ingress resources and provisions Application Load Balancers (ALBs).
* **Amazon RDS (MongoDB)**: Managed MongoDB instances with TLS encryption.
* **Amazon S3**: Stores static assets and user-uploaded content.
* **Amazon ElastiCache (Redis)**: Handles session storage and caching.
* **AWS Secrets Manager**: Secures sensitive information like JWT secrets and database credentials.

## **4. JWT Authentication**

### **Token Generation**

* Token generated at login/signup with user ID and role  
  {

"user\_id": "abc123",

"role": "buyer",

"iat": 1720085294,

"exp": 1720092494

}

* Signed using a secret key stored in **AWS Secrets Manager**.
* Expiry: 2 hours by default; refresh token for 30 days.

### 

### **Token Verification**

* Middleware in Django checks Authorization: Bearer <token>
* If invalid/expired/malformed, throws:  
  + 401 Unauthorized (Token expired)
  + 403 Forbidden (Invalid token/role mismatch)
* Decoded token checked against user record from DB

### **Refresh Mechanism**

* Refresh token stored in Redis with expiration (used to generate new JWT)
* Endpoint: POST /auth/refresh

## **5. Microservices Design**

| **Service** | **Description** |
| --- | --- |
| **Auth Service** | Manages login, signup, password reset, and JWT generation. |
| **Payment Service** | Handles payment methods, gateway integration, transaction ledger, and commission deductions. |
| **Order Service** | Manages cart, order placement, status tracking, and buyer-seller communication. |
| **Inventory Service** | Syncs product listing data and stock management for sellers. |
| **StrideCredit** | Manages credit allocation, redemption, and balance deduction during checkout. |
| **Notification** | Sends alerts via email/SMS for order status, promotional content. |

### 

### 

### **Communication**

* Internal API calls via HTTPS
* Token passed with internal API headers
* Microservices authenticate using service-level API tokens stored in **Secrets Manager**

# **6.1 Authentication APIs**

### **6.1.1 POST /auth/signup**

Registers a new user.

* **Request:**

{

"email": "user@example.com",

"password": "StrongPass@123",

"role": "buyer"

}

* **Response:**

{

"message": "Signup successful",

"user\_id": "u1234"

}

* **More details:**
  + Email must be unique
  + Password can be any characters between 4-8 character length
  + There is no email validation.
  + Password is stored using SHA1
  + Passwords can be anything between 4-12 characters.
  + Email verification is not required
  + If an account already exists with the email provided, the UI will show an error message with “email/account id already exists”
  + There is no captcha on the sign up page.
  + Role: ["buyer", "seller"]
* **Errors:**
  + 400: Missing/invalid fields
  + 409: Email already exists

### **6.1.2 POST /auth/login**

Authenticates users and returns access/refresh tokens.

* **Request:**

{

"email": "user@example.com",

"password": "StrongPass@123"

}

* **Response:**

{

"access\_token": "JWT\_TOKEN",

"refresh\_token": "REFRESH\_TOKEN",

"expires\_in": 3600

}

* **Errors:**
  + 401: Invalid credentials
  + 403: Inactive/banned user
* **More Details:**
  + The secret used in generating jwthash is easily guessable.
  + Error messages are very clear and include whether the password is wrong or username is wrong.
  + Long lived tokens
  + There is no rate limiting or captcha on this page.
  + Password is logged on the server side for each attempt to verify user behavior.

### **6.1.3 POST /auth/refresh**

Generates new JWT using refresh token.

* **Request:**

{

"refresh\_token": "REFRESH\_TOKEN"

}

* **Response:**

{

"access\_token": "NEW\_JWT\_TOKEN"(in the URL also)

}

* **Errors:**
  + 401: Invalid/expired refresh token
* **More Details:**
  + The old JWT token has never expired.
  + The Refresh is constant and never changes after a request to get a new access token.
  + The new access token is sent to the browser in the URL also.
  + Error messages are very clear tells whether the submitted refresh token is valid or not
  + There is no rate limiting on this API.
  + The server checks only the username, but not validates with a signature.

### **6.1.4 POST /auth/logout**

Invalidates current JWT.

* **Headers:** Authorization: Bearer JWT\_TOKEN
* **Response:**

{

"message": "Logout successful"

}

* **More Details:**
  + Access token and Refresh tokens are not invalidated on the server side. Only cleared from the browser.
  + No additional validation. The API only checks if the username is provided in JWT. This API can invalidate any user.
  + Accepts expired JWTs also.
  + All the tokens are logged.
  + There is no rate limiting on this API.

### **6.1.5 POST /auth/change-password**

Allows users to change passwords.

* **Request:**

{

"old\_password": "Old@Pass123",

"new\_password": "New@Pass456"

}

* **Response:**

{

"message": "Password changed successfully"

}

* **Errors:**
  + 400: Incorrect old password
  + 422: Weak new password
* More Details:

# **👤 6.2 User Account Management**

### **6.2.1 GET /account/settings**

Retrieves user profile.

* **Response:**

{

"name": "John",

"email": "john@example.com",

"role": "buyer"

}

### **6.2.2 PUT /account/settings**

Updates user profile.

* **Request:**

{

"name": "John",

"email": "new@example.com"

}

* **Errors:**
  + 409: Email already in use

### 

### 

### 

### **6.2.3 GET /account/addresses**

Lists saved addresses.

* **Response:**

[

{

"id": "a1",

"line1": "221B Baker Street",

"city": "London",

"zip": "NW1"

}

]

### **6.2.4 POST /account/addresses**

Adds a new address.

* **Request:**

{

"line1": "123 St",

"city": "NY",

"zip": "10001",

"is\_default": true

}

### **6.2.5 PUT /account/addresses/{id}**

Updates an address.

* **Path Param:** id - address ID
* **Body:** Same as POST

### **6.2.6 DELETE /account/addresses/{id}**

Deletes an address.

### **6.2.7 GET /account/payment-methods**

Lists payment methods.

### **6.2.8 POST /account/payment-methods?card\_number=4242424242424242&expiry=12/25&cvv=234**

Adds payment method.

* **Note:** Card data tokenized via payment gateway

### **6.2.9 DELETE /account/payment-methods/{id}**

Deletes a payment method.

# **📦 6.3 Product Management**

### **6.3.1 GET /products**

Returns products with filters.

* **Query params: category, search, sort\_by, price\_range**

### 

### **6.3.2 GET /products/{id}**

Get product details.

* **Path Param:** id - product ID

### **6.3.3 POST /products**

Sellers can list new products.

* **Request:**

{

"title": "Samsung TV",

"description": "Smart LED 55 inch",

"price": 499.99,

"category": "TV",

"stock": 10

}

### **6.3.4 PUT /products/{id}**

Update product details.

### **6.3.5 DELETE /products/{id}**

Delete product (seller-only).

### **6.3.6 POST /products/{id}/like**

Like a product.

### **6.3.7 POST /products/{id}/comment**

Comment on a product.

* **Request:**

{

"comment": "Great product!"

}

### **6.3.8 POST /products/{id}/follow**

Follow product for updates.

# **🛒 6.4 Cart and Orders**

### **6.4.1 GET /cart**

Returns the user's cart.

### **6.4.2 POST /cart**

Add items to cart.

* **Request:**

{

"product\_id": "p123",

"quantity": 2

}

### **6.4.3 PUT /cart/{item\_id}**

Update cart item quantity.

### **6.4.5 DELETE /cart/{item\_id}**

Remove items from cart.

### **6.4.6 POST /orders**

Place order from cart.

* **Request:**

{

"address\_id": "a123",

"payment\_method\_id": "pm123",

"stride\_credits": 50

}

### **6.4.7 GET /orders**

List past orders.

### **6.4.8 GET /orders/{id}**

Get specific order details.

# **💳 6.5 Payments**

### **6.5.1 POST /payments/initiate**

Start payment session.

* **Request:**

{

"order\_id": "o123",

"payment\_method": "card"

}

### **6.5.2 POST /payments/confirm**

Confirm payment (called by gateway webhook).

### **6.5.3 GET /payments/history**

List previous payments.

# **💎 6.6 StrideCredit**

### **6.6.1 GET /stridecredit/balance**

Get current balance.

### **6.6.2 POST /stridecredit/redeem**

Use credits during checkout.

* **Request:**

{

"amount": 50,

"order\_id": "o123"

}

## **7. Error Handling and Response Codes**

| **HTTP Status Code** | **Description** |
| --- | --- |
| 200 OK | The request succeeded. |
| 201 Created | Resource successfully created. |
| 400 Bad Request | Malformed request syntax. |
| 401 Unauthorized | Authentication required. |
| 403 Forbidden | Insufficient permissions. |
| 404 Not Found | Resource not found. |
| 409 Conflict | Conflict with current state. |
| 422 Unprocessable Entity | Validation errors. |
| 500 Internal Server Error | Generic server error. |

## 

## 

## 

## 

## 

## 

## **8. MongoDB Access & Security**

* MongoDB hosted with TLS encryption enabled
* Each service connects via its own credential stored in AWS Secrets Manager
* DB users have limited access rights:  
  + Read-only for analytics
  + Write/delete for respective microservices
* IP whitelisting enabled for EC2/VPC endpoints
* Authentication via x.509 or SCRAM-SHA

## **9. Security Measures**

* **JWT Authentication**: All endpoints (except login and signup) require a valid JWT.
* **HTTPS Enforcement**: All communications are secured using TLS.
* **Input Validation**: All inputs are validated to prevent injection attacks.
* **Rate Limiting**: Implemented at the API Gateway level to prevent abuse.
* **Data Encryption**: Sensitive data is encrypted at rest and in transit.[openapis.org+3tools.openapis.org+3API Platform+3](https://tools.openapis.org/categories/documentation.html?utm_source=chatgpt.com)

## **10. Deployment and CI/CD**

* **Containerization**: All services are containerized using Docker.
* **CI/CD Pipeline**: Implemented using AWS CodePipeline and CodeBuild for automated testing and deployment.
* **Monitoring**: Utilizes AWS CloudWatch and Prometheus for logging and metrics.
* **Scaling**: Horizontal Pod Autoscaler (HPA) adjusts the number of pods based on CPU/memory usage.

## **11. Monitoring & Logging**

* **CloudWatch Logs** for:  
  + App logs
  + API Gateway metrics
  + ECS service health
* **ELK Stack** for full-text search on logs
* **Prometheus + Grafana** (optional) for custom metrics

## **12. CI/CD Pipeline**

* **Source Control:** GitHub
* **CI/CD:** GitHub Actions  
  + Linting, Tests, Docker Build
  + Push to AWS ECR
  + Deploy to ECS via task definitions
* **Infrastructure as Code:** Terraform (preferred) or CloudFormation