**Project Overview:**

Build a serverless e-commerce platform with the following features:

* **Static frontend** hosted on **S3 and distributed via CloudFront**.
* **Order processing system** utilizing **AWS Lambda, SQS, DynamoDB, and Step Functions**.
* A **backend workflow** for product search, inventory updates, and order fulfillment.

**Step 1: Setup Project Repository**

1. **Version Control**: Initialize a Git repository (use GitHub/GitLab).
2. **Structure**: Create a monorepo or microservice structure:

/frontend (React/Angular/Static files for S3)

/backend

/lambdas

- processOrder/

- updateInventory/

- notifyCustomer/

/infrastructure

**Step 2: Design the Architecture**

**1. Frontend:**

* **AWS S3**:
  + Host static assets (HTML, CSS, JS, and images).
* **AWS CloudFront**:
  + Serve assets globally with caching and SSL/TLS.
  + Setup caching policies for static files (e.g., HTML, CSS, JS).

**2. Backend:**

* **DynamoDB**:
  + Tables:
    - Products (Product details and stock levels).
    - Orders (Order status and metadata).
* **AWS Lambda**:
  + Stateless backend API to handle requests (Node.js).
* **AWS SQS**:
  + Decouple workflows by adding queues for order processing.
* **AWS Step Functions**:
  + Orchestrate multi-step workflows like order validation, payment processing, and inventory update.

**Step 3: Infrastructure as Code (IaC)**

* Use **AWS CloudFormation**, **AWS SAM**, or **AWS CDK** (Node.js).
* Example structure in CDK:

const s3 = new s3.Bucket(this, 'FrontendBucket', {

websiteIndexDocument: 'index.html',

publicReadAccess: true,

});

const dynamoOrders = new dynamodb.Table(this, 'OrdersTable', {

partitionKey: { name: 'orderId', type: dynamodb.AttributeType.STRING },

});

const dynamoProducts = new dynamodb.Table(this, 'ProductsTable', {

partitionKey: { name: 'productId', type: dynamodb.AttributeType.STRING },

});

const sqsQueue = new sqs.Queue(this, 'OrderQueue');

const lambdaProcessOrder = new lambda.Function(this, 'ProcessOrderLambda', {

runtime: lambda.Runtime.NODEJS\_18\_X,

handler: 'processOrder.handler',

code: lambda.Code.fromAsset('lambdas/processOrder'),

environment: {

QUEUE\_URL: sqsQueue.queueUrl,

},

});

sqsQueue.grantSendMessages(lambdaProcessOrder);

**Step 4: Build Frontend**

1. Develop a frontend in **React/Angular**:
   * Product listing and search.
   * Order placement page.
2. Deploy to **S3**:
   * Run npm run build and upload to S3.
   * Configure **CloudFront** for global delivery.

**Step 5: Lambda Development (Node.js)**

1. **Handler Examples:**
   * **Process Order Lambda**:

const AWS = require('aws-sdk');

const sqs = new AWS.SQS();

exports.handler = async (event) => {

const order = JSON.parse(event.body);

// Validate Order Logic

const params = {

QueueUrl: process.env.QUEUE\_URL,

MessageBody: JSON.stringify(order),

};

await sqs.sendMessage(params).promise();

return {

statusCode: 200,

body: JSON.stringify({ message: 'Order processed successfully' }),

};

};

* + **Update Inventory Lambda**:

const AWS = require('aws-sdk');

const dynamodb = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {

for (const record of event.Records) {

const order = JSON.parse(record.body);

const params = {

TableName: 'ProductsTable',

Key: { productId: order.productId },

UpdateExpression: 'SET stock = stock - :qty',

ExpressionAttributeValues: { ':qty': order.quantity },

};

await dynamodb.update(params).promise();

}

};

**Step 6: Step Functions Workflow**

1. **Design Workflow**:
   * **Steps**:
     + Order validation (Lambda).
     + Payment processing (Mock or external API integration).
     + Inventory update (Lambda).
     + Notify customer (Lambda or SNS).
   * **State Machine Example**:

{

"StartAt": "ValidateOrder",

"States": {

"ValidateOrder": {

"Type": "Task",

"Resource": "arn:aws:lambda:REGION:ACCOUNT\_ID:function:ValidateOrder",

"Next": "ProcessPayment"

},

"ProcessPayment": {

"Type": "Task",

"Resource": "arn:aws:lambda:REGION:ACCOUNT\_ID:function:ProcessPayment",

"Next": "UpdateInventory"

},

"UpdateInventory": {

"Type": "Task",

"Resource": "arn:aws:lambda:REGION:ACCOUNT\_ID:function:UpdateInventory",

"Next": "NotifyCustomer"

},

"NotifyCustomer": {

"Type": "Task",

"Resource": "arn:aws:lambda:REGION:ACCOUNT\_ID:function:NotifyCustomer",

"End": true

}

}

}

**Step 7: Testing and Deployment**

1. **Local Testing**:
   * Use **AWS SAM CLI** or **CDK synth** to test Lambda functions.
   * Mock S3, SQS, and DynamoDB using tools like **LocalStack**.
2. **CI/CD Pipeline**:
   * Use GitHub Actions, CodePipeline, or Jenkins to:
     + Build the frontend.
     + Deploy backend resources via IaC.
   * Example pipeline stages:
     + **Build**: Run tests and build Lambdas/frontend.
     + **Deploy**: Deploy IaC stack.

**Step 8: Monitoring and Optimization**

1. **AWS CloudWatch**:
   * Set up logging for Lambda functions.
   * Monitor DynamoDB table metrics.
2. **AWS X-Ray**:
   * Enable tracing to analyze request paths through Lambdas and Step Functions.

**Feature Enhancements**

**Frontend Enhancements:**

1. **User Authentication:**
   * Integrate **AWS Cognito** for user sign-up, login, and token-based authentication.
   * Secure API endpoints using Cognito’s JWT tokens.
2. **Product Search with Autocomplete:**
   * Implement a search feature using **AWS CloudSearch** or **DynamoDB’s Query API** with indexes.
3. **Responsive Design:**
   * Use frameworks like **TailwindCSS** or **Bootstrap** to ensure mobile-first design.
4. **Client-Side Analytics:**
   * Track user interactions using **Amazon Pinpoint**.

**Backend Enhancements:**

1. **Enhanced Product Catalog:**
   * Store **high-resolution product images** in **S3** and serve them via **CloudFront**.
   * Use **DynamoDB Streams** to notify changes in product details (e.g., stock updates).
2. **Real-Time Notifications:**
   * Send order status updates (e.g., "Shipped," "Delivered") to users using **AWS SNS** for SMS/Email or **AWS AppSync** for WebSocket-based real-time updates.
3. **Order Lifecycle Management:**
   * Add a **Retry and DLQ mechanism** in SQS for failed messages.
   * Implement **order cancellation and refunds** workflows via Step Functions.
4. **Advanced Payment Processing:**
   * Integrate with third-party payment providers like **Stripe** or **PayPal**.
   * Use **AWS Secrets Manager** to store and retrieve API keys securely.
5. **Inventory Forecasting:**
   * Implement predictive stock replenishment using **AWS Forecast**.
   * Trigger Lambda functions to notify suppliers when stock is low.
6. **Content Delivery Optimization:**
   * Use **CloudFront Functions** for A/B testing (e.g., testing two versions of a webpage).
   * Implement image optimization using **Lambda@Edge**.

**Data Pipeline Enhancements:**

1. **Order Analytics Dashboard:**
   * Use **AWS Glue** to process data and load it into **Amazon Redshift** for analytics.
   * Visualize analytics with **Amazon QuickSight** (e.g., sales trends, top-selling products).
2. **Event Archival:**
   * Archive order events and logs in **Amazon S3 Glacier** for cost-efficient long-term storage.

**Security Enhancements:**

1. **IAM Policies:**
   * Fine-grained access control for Lambda, S3, and DynamoDB using least-privilege IAM roles.
2. **Web Application Firewall (WAF):**
   * Protect the frontend by attaching **AWS WAF** to CloudFront to prevent SQL injection, XSS, and other attacks.
3. **Data Encryption:**
   * Enable **server-side encryption (SSE)** for DynamoDB tables.
   * Store sensitive data (e.g., customer PII) in **AWS KMS-encrypted S3 buckets**.
4. **API Security:**
   * Use **Amazon API Gateway** with custom throttling limits and request validation.

**Monitoring and Observability Enhancements:**

1. **Custom Dashboards:**
   * Create custom **CloudWatch dashboards** for monitoring:
     + Lambda invocations/errors.
     + DynamoDB throughput and latency.
     + SQS message delays.
   * Add **Alarms** for critical thresholds.
2. **Tracing:**
   * Enable **AWS X-Ray** across all Lambdas and Step Functions to trace request paths and latency.
3. **Audit Logs:**
   * Enable **AWS CloudTrail** for auditing all API and resource changes.

**Expanded Architecture**

**Data Flow and Features:**

1. **User Places an Order:**
   * **Frontend:** React/Angular app authenticates the user via Cognito.
   * **API Gateway:** Secures and routes requests to Lambda.
   * **DynamoDB:** Stores order details and triggers DynamoDB Streams.
2. **Order Processing Pipeline:**
   * **Step Functions Workflow:**
     + Validate order (Lambda).
     + Charge payment (3rd-party API).
     + Update inventory (DynamoDB).
     + Send notifications (SNS or AppSync).
3. **Product Search:**
   * Query DynamoDB via indexed attributes or leverage CloudSearch for full-text search.
4. **Data Analytics Pipeline:**
   * **Order Data:** Extracted by AWS Glue, transformed, and stored in Amazon Redshift.
   * **Reports:** Generated by Amazon QuickSight for business insights.
5. **Static Content Delivery:**
   * Hosted on S3, optimized by CloudFront with caching and WAF for security.

**Detailed Project Steps**

**Step 1: Set Up Core Services**

* **Frontend (React/Angular):**
  + Configure Cognito user pools for authentication.
  + Use API Gateway for secured API calls.
* **Backend:**
  + Define DynamoDB tables with GSIs for quick search.
  + Configure S3 and CloudFront for static files and product images.

**Step 2: Develop Lambda Functions**

1. **OrderProcessing Lambda** (handles order validation and queuing):

const AWS = require('aws-sdk');

const sqs = new AWS.SQS();

exports.handler = async (event) => {

const order = JSON.parse(event.body);

if (!order.productId || !order.quantity) {

throw new Error("Invalid order payload");

}

await sqs.sendMessage({

QueueUrl: process.env.ORDER\_QUEUE,

MessageBody: JSON.stringify(order),

}).promise();

return { statusCode: 200, body: JSON.stringify({ message: "Order queued" }) };

};

1. **UpdateInventory Lambda** (consumes SQS and updates stock):

const AWS = require('aws-sdk');

const dynamodb = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {

for (const record of event.Records) {

const order = JSON.parse(record.body);

await dynamodb.update({

TableName: 'Products',

Key: { productId: order.productId },

UpdateExpression: 'SET stock = stock - :qty',

ExpressionAttributeValues: { ':qty': order.quantity },

}).promise();

}

};

**Step 3: Set Up SQS**

* Create standard queues with DLQs for failed message retries.

**Step 4: Implement Analytics Pipeline**

* Use AWS Glue for ETL (extract DynamoDB and S3 logs).
* Configure QuickSight for real-time reporting dashboards.

**Step 5: Add Advanced Observability**

* Use AWS X-Ray for distributed tracing.
* Monitor CloudWatch for service health metrics.

**Step 6: Security**

* Enable WAF and encrypt sensitive data with KMS

############################-----------------------NODE-JS---------------#######################

**1. Project Setup (0.5 hours)**

1. **Initialize Project**:
   * Create a project directory: mkdir nodejs-capstone && cd nodejs-capstone.
   * Initialize with npm: npm init -y.
2. **Install Dependencies**:
   * Install required packages:

npm install express mongoose dotenv bcryptjs jsonwebtoken cookie-parser cors

npm install --save-dev nodemon

* + Explanation of dependencies:
    - **express**: Web framework.
    - **mongoose**: Database (MongoDB) interaction.
    - **dotenv**: Manage environment variables.
    - **bcryptjs**: Hash passwords.
    - **jsonwebtoken**: Handle authentication.
    - **cookie-parser**: Parse cookies.
    - **cors**: Handle cross-origin requests.
    - **nodemon**: Automatically restart server during development.

1. **Project Structure**:
   * Create folders:

mkdir routes models controllers middlewares config

* + File structure:

nodejs-capstone/

├── config/

│ └── db.js

├── controllers/

│ ├── authController.js

│ └── userController.js

├── middlewares/

│ └── authMiddleware.js

├── models/

│ └── User.js

├── routes/

│ ├── authRoutes.js

│ └── userRoutes.js

├── .env

├── app.js

├── package.json

└── README.md

1. **Set Up Scripts**:  
   Update package.json:

"scripts": {

"start": "node app.js",

"dev": "nodemon app.js"

}

**2. Database Connection (0.5 hours)**

1. **MongoDB Setup**:
   * Create a database on MongoDB Atlas or use a local MongoDB instance.
2. **Environment Variables**:
   * Add a .env file with:

PORT=3000

MONGO\_URI=mongodb+srv://<username>:<password>@cluster.mongodb.net/capstone?retryWrites=true&w=majority

JWT\_SECRET=your\_jwt\_secret

1. **Connect to MongoDB**:
   * In config/db.js:

const mongoose = require('mongoose');

const connectDB = async () => {

try {

await mongoose.connect(process.env.MONGO\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

console.log('MongoDB Connected');

} catch (err) {

console.error(err.message);

process.exit(1);

}

};

module.exports = connectDB;

* + Call connectDB() in app.js.

**3. Authentication System (2 hours)**

1. **User Model**:
   * In models/User.js:

const mongoose = require('mongoose');

const bcrypt = require('bcryptjs');

const UserSchema = new mongoose.Schema({

name: { type: String, required: true },

email: { type: String, required: true, unique: true },

password: { type: String, required: true },

});

UserSchema.pre('save', async function (next) {

if (!this.isModified('password')) return next();

const salt = await bcrypt.genSalt(10);

this.password = await bcrypt.hash(this.password, salt);

next();

});

module.exports = mongoose.model('User', UserSchema);

1. **Auth Routes**:
   * In routes/authRoutes.js:

const express = require('express');

const { registerUser, loginUser } = require('../controllers/authController');

const router = express.Router();

router.post('/register', registerUser);

router.post('/login', loginUser);

module.exports = router;

1. **Auth Controller**:
   * In controllers/authController.js:

const User = require('../models/User');

const jwt = require('jsonwebtoken');

const bcrypt = require('bcryptjs');

const generateToken = (id) => {

return jwt.sign({ id }, process.env.JWT\_SECRET, { expiresIn: '1h' });

};

exports.registerUser = async (req, res) => {

const { name, email, password } = req.body;

try {

const user = await User.create({ name, email, password });

res.status(201).json({ token: generateToken(user.\_id) });

} catch (error) {

res.status(400).json({ message: 'Registration failed', error });

}

};

exports.loginUser = async (req, res) => {

const { email, password } = req.body;

try {

const user = await User.findOne({ email });

if (!user || !(await bcrypt.compare(password, user.password))) {

return res.status(401).json({ message: 'Invalid credentials' });

}

res.status(200).json({ token: generateToken(user.\_id) });

} catch (error) {

res.status(400).json({ message: 'Login failed', error });

}

};

**4. Routing and Middleware (2 hours)**

1. **Auth Middleware**:
   * In middlewares/authMiddleware.js:

const jwt = require('jsonwebtoken');

const protect = (req, res, next) => {

const token = req.headers.authorization?.split(' ')[1];

if (!token) return res.status(401).json({ message: 'Not authorized' });

try {

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

req.user = decoded.id;

next();

} catch (err) {

res.status(401).json({ message: 'Invalid token' });

}

};

module.exports = { protect };

1. **User Routes**:
   * In routes/userRoutes.js:

const express = require('express');

const { getUserProfile } = require('../controllers/userController');

const { protect } = require('../middlewares/authMiddleware');

const router = express.Router();

router.get('/profile', protect, getUserProfile);

module.exports = router;

1. **User Controller**:
   * In controllers/userController.js:

const User = require('../models/User');

exports.getUserProfile = async (req, res) => {

try {

const user = await User.findById(req.user).select('-password');

res.status(200).json(user);

} catch (error) {

res.status(404).json({ message: 'User not found', error });

}

};

**5. Testing and Debugging (1.5 hours)**

1. Use **Postman** or **Thunder Client** to test endpoints:
   * Register: POST /api/auth/register
   * Login: POST /api/auth/login
   * Get Profile: GET /api/user/profile (with token).
2. Verify database interactions using MongoDB Compass.

**6. Final Touches (1 hour)**

1. Add **CORS configuration**:
   * In app.js:

const cors = require('cors');

app.use(cors());

1. Set up central error handling middleware.
2. Prepare a **README.md** with project details and usage instructions.