Backend Integration Manager - Deployment Guide

This document provides step-by-step instructions for deploying the Backend Integration Manager application. The application consists of two main components:

1. **Frontend**: A React application deployed on Vercel
2. **Backend**: A Node.js Express API deployed on AWS using Docker and ECS

Table of Contents

Prerequisites

Before deploying the application, ensure you have the following:

General Requirements

- Git repository access
- Node.js 16.x or higher
- npm 8.x or higher

Frontend Deployment Requirements

- Vercel account
- Vercel CLI installed (`npm install -g vercel`)

Backend Deployment Requirements

- AWS account with appropriate permissions
- AWS CLI installed and configured
- Docker installed
- jq installed (for deployment script)

Environment Variables

Frontend Environment Variables

Variable	Description	Example
`VITE_API_URL`	URL of the backend API	`https://api.backend-integration-manager.com`
`VITE_JWT_EXPIRY`	JWT token expiry time	`30m`
`VITE_ENABLE_ANALYTICS`	Enable/disable analytics	`true`

Backend Environment Variables

Variable	Description	Example
`NODE_ENV`	Environment (development, production)	`production`
`PORT`	Port on which the server runs	`3000`
`JWT_SECRET`	Secret key for JWT authentication	`your-secure-jwt-secret`
`JWT_EXPIRY`	JWT token expiry time	`30m`
`REFRESH_TOKEN_EXPIRY`	Refresh token expiry time	`7d`
`CORS_ORIGIN`	Allowed CORS origin	`https://backend-integration-manager.vercel.`
`SQUARE_API_KEY`	Square API key	`square_api_key`
`SENDGRID_API_KEY`	SendGrid API key	`sendgrid_api_key`
`TWILIO_ACCOUNT_SID`	Twilio Account SID	`twilio_account_sid`
`TWILIO_AUTH_TOKEN`	Twilio Auth Token	`twilio_auth_token`

Frontend Deployment (Vercel)

Step 1: Prepare the Frontend for Deployment

1. Ensure your frontend code is ready for production:

```
# Install dependencies
npm install

# Build the application
npm run build
```

2. Verify that the build completes successfully and check the `dist` directory for the built files.

Step 2: Configure Vercel

1. Create a `vercel.json` file in the project root (if not already present):

```
{
  "version": 2,
  "builds": [
    {
      "src": "package.json",
      "use": "@vercel/static-build",
      "config": { "distDir": "dist" }
    }
  ],
  "routes": [
    { "handle": "filesystem" },
    { "src": "/api/(.*)", "dest": "https://api.backend-integration-manager.com/api/$1"
    { "src": "/.*", "dest": "/index.html" }
  ],
  "env": {
    "VITE_API_URL": "@vite_api_url",
    "VITE_JWT_EXPIRY": "30m",
    "VITE_ENABLE_ANALYTICS": "true"
  }
}
```

> **Note**: Update the API URL in the routes section to match your actual backend API URL.

Step 3: Set Up Environment Variables in Vercel

1. Log in to your Vercel account and navigate to your project.
2. Go to the “Settings” tab and then “Environment Variables”.
3. Add the following environment variables:
   - `VITE_API_URL`: URL of your backend API
   - `VITE_JWT_EXPIRY`: JWT token expiry time
   - `VITE_ENABLE_ANALYTICS`: Enable/disable analytics
4. For sensitive values, use Vercel Secrets:

   ```
   vercel secrets add vite_api_url https://api.backend-integration-manager.com
   ```

Step 4: Deploy to Vercel

1. Deploy using the Vercel CLI:

   ```
   # Login to Vercel if not already logged in
   vercel login

   # Deploy the application
   vercel --prod
   ```

2. Alternatively, you can set up automatic deployments from your Git repository:
   - Connect your Git repository to Vercel
   - Configure the build settings
   - Set up the environment variables

- Enable automatic deployments on push to the main branch

3. After deployment, Vercel will provide you with a URL for your application.

Backend Deployment (AWS)

Step 1: Prepare the Backend for Deployment

1. Ensure your backend code is ready for production.
2. Create a `Dockerfile` in the backend project root (if not already present):

```
FROM node:18-alpine

WORKDIR /app

# Install dependencies
COPY package*.json ./
RUN npm ci --only=production

# Copy application code
COPY . .

# Set environment variables
ENV NODE_ENV=production
ENV PORT=3000

# Expose the application port
EXPOSE 3000

# Start the application
CMD ["node", "server.js"]
```

3. Create a `docker-compose.yml` file for local testing (if not already present):

```
version: '3.8'

services:
  backend:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=development
      - PORT=3000
      - JWT_SECRET=your_jwt_secret_key_here
      - JWT_EXPIRY=30m
      - REFRESH_TOKEN_EXPIRY=7d
      - CORS_ORIGIN=http://localhost:5173
      - SQUARE_API_KEY=your_square_api_key
      - SENDGRID_API_KEY=your_sendgrid_api_key
      - TWILIO_ACCOUNT_SID=your_twilio_account_sid
      - TWILIO_AUTH_TOKEN=your_twilio_auth_token
    volumes:
      - .:/app
      - /app/node_modules
    command: npm run dev
    restart: unless-stopped

  redis:
    image: redis:alpine
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
    restart: unless-stopped

volumes:
  redis-data:
```

Step 2: Store Sensitive Information in AWS Parameter Store

1. Store sensitive environment variables in AWS Parameter Store:

```
# Store JWT secret
aws ssm put-parameter \
  --name "/backend-integration-manager/production/JWT_SECRET" \
  --value "your-secure-jwt-secret" \
  --type "SecureString" \
  --region us-east-1

# Store Square API key
aws ssm put-parameter \
  --name "/backend-integration-manager/production/SQUARE_API_KEY" \
  --value "your-square-api-key" \
  --type "SecureString" \
  --region us-east-1

# Store SendGrid API key
aws ssm put-parameter \
  --name "/backend-integration-manager/production/SENDGRID_API_KEY" \
  --value "your-sendgrid-api-key" \
  --type "SecureString" \
  --region us-east-1

# Store Twilio Account SID
aws ssm put-parameter \
  --name "/backend-integration-manager/production/TWILIO_ACCOUNT_SID" \
  --value "your-twilio-account-sid" \
  --type "SecureString" \
  --region us-east-1

# Store Twilio Auth Token
aws ssm put-parameter \
  --name "/backend-integration-manager/production/TWILIO_AUTH_TOKEN" \
  --value "your-twilio-auth-token" \
  --type "SecureString" \
  --region us-east-1
```

Step 3: Deploy Using the Deployment Script

1. Navigate to the backend project directory:

   ```
   cd backend
   ```

2. Run the deployment script:

   ```
   ../aws/deploy.sh --region us-east-1 --env production
   ```

   > **Note**: The script will:
   >
   > - Create an ECR repository if it doesn't exist
   > - Build and push the Docker image to ECR
   > - Deploy the CloudFormation stack
   > - Output the API endpoint URL

3. Wait for the deployment to complete. This may take 5-10 minutes.

Step 4: Verify the Deployment

1. Test the API endpoint:

   ```
   curl https://your-api-endpoint.amazonaws.com/api/health
   ```

2. If the health check is successful, the API is deployed correctly.

CORS Configuration

Backend CORS Configuration

The backend API is configured to allow requests from the frontend domain. This is set using the `CORS_ORIGIN` environment variable.

1. Ensure the `CORS_ORIGIN` environment variable is set correctly in your AWS CloudFormation template or deployment script.
2. The backend should include CORS middleware:

```
const cors = require('cors');

// CORS configuration
const corsOptions = {
  origin: process.env.CORS_ORIGIN,
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'],
  allowedHeaders: ['Content-Type', 'Authorization'],
  credentials: true,
  maxAge: 86400 // 24 hours
};

app.use(cors(corsOptions));
```

Frontend API Requests

1. The frontend should include the appropriate headers in API requests:

```
// In integrationService.js
const headers = {
  'Content-Type': 'application/json',
  'Authorization': `Bearer ${localStorage.getItem('token')}`
};

const response = await axios.get(`${API_BASE_URL}/adapters`, { headers });
```

Security Considerations

JWT Authentication

1. **Token Storage**:
   - Store tokens securely using HttpOnly cookies or session storage instead of localStorage to mitigate XSS risks.
   - Implement token refresh mechanism for better security.
2. **Token Validation**:
   - Validate tokens on every request.
   - Check expiration, signature, and issuer.
3. **HTTPS**:
   - Always use HTTPS in production to encrypt token transmission.

API Security

1. **Rate Limiting**:
   - Implement rate limiting to prevent abuse.

```
const rateLimit = require('express-rate-limit');

const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  message: 'Too many requests from this IP, please try again after 15 minutes'
});

app.use('/api/', apiLimiter);
```

2. **Input Validation**:
   - Validate all input data using a library like Joi or express-validator.
3. **Security Headers**:
   - Implement security headers using Helmet.

```
const helmet = require('helmet');
app.use(helmet());
```

Troubleshooting

Frontend Deployment Issues

1. **Build Failures**:
   - Check the build logs in Vercel for errors.
   - Ensure all dependencies are correctly installed.
   - Verify that environment variables are correctly set.
2. **API Connection Issues**:
   - Verify that the `VITE_API_URL` is correctly set.
   - Check CORS configuration on the backend.
   - Ensure the API is accessible from the frontend domain.

Backend Deployment Issues

1. **CloudFormation Stack Creation Failures**:
   - Check the CloudFormation events in the AWS Console for error messages.
   - Verify that all required parameters are correctly set.
   - Ensure that the IAM user has sufficient permissions.
2. **Container Startup Issues**:
   - Check the ECS task logs in CloudWatch.
   - Verify that all environment variables are correctly set.
   - Ensure the container has access to all required resources.
3. **API Not Responding**:
   - Check the health of the ECS service.
   - Verify that the security group allows traffic on the required port.
   - Check the load balancer target group health.

Common Issues and Solutions

1. **CORS Errors**:
   - Ensure the `CORS_ORIGIN` environment variable is correctly set.
   - Verify that the frontend is making requests to the correct API URL.
   - Check that the backend CORS middleware is correctly configured.
2. **JWT Authentication Failures**:
   - Verify that the `JWT_SECRET` is the same in both frontend and backend.
   - Check that tokens are correctly stored and included in requests.
   - Ensure token expiration times are correctly set.
3. **Database Connection Issues**:
   - Check database connection strings.
   - Verify that the database is accessible from the ECS task.
   - Ensure database credentials are correctly set.

For additional support, please contact the development team or refer to the project documentation.