Capstone Project Documentation

Project Name: Chi-go

GROUP 4 : CHENG HUANG, DI XIAO, LEJUN CHEN

1. Production Support & Testing Scenarios

1.1 Service Architecture & Dependencies

1.1.1 Service Dependency Diagram

```
graph TB
    subgraph "Users"
        ENDUSER[End Users<br/>Tourists/Travelers]
        ADMINUSER[Admin Users<br/>Content Managers]
    end

    subgraph "Azure Cloud Platform"
        subgraph "Frontend Applications"
            USERAPP[Chi-Go User Portal<br/>Azure Static Web Apps<br/>React Application]
            ADMINAPP[Chi-Go Admin Portal<br/>Azure Static Web Apps<br/>React Application]
        end

        subgraph "Backend Services"
            APPSERVICE[Azure App Service<br/>Flask API<br/>Shared Backend]
            REDIS[Azure Cache for Redis<br/>Session Storage]
        end

        subgraph "Data Layer"
            POSTGRES[Azure Database for PostgreSQL<br/>Primary Database]
            BLOB[Azure Blob Storage<br/>Media Files & Images]
        end

        subgraph "Supporting Services"
            KEYVAULT[Azure Key Vault<br/>Secrets Management]
            MONITOR[Azure Monitor<br/>Logging & Metrics]
            APPINS[Application Insights<br/>APM & Diagnostics]
            CDN[Azure CDN<br/>Static Content Delivery]
        end
    end

    subgraph "External APIs"
        MAPS[Google Maps API]
        EMAIL[SendGrid Email API]
        JWT[JWT Token Service]
    end

    ENDUSER --> USERAPP
```

ADMINUSER --> ADMINAPP
USERAPP --> CDN
ADMINAPP --> CDN
USERAPP --> APPSERVICE
ADMINAPP --> APPSERVICE
APPSERVICE --> POSTGRES
APPSERVICE --> REDIS
APPSERVICE --> BLOB
APPSERVICE --> KEYVAULT
APPSERVICE --> MAPS
APPSERVICE --> EMAIL
APPSERVICE --> JWT
APPSERVICE --> MONITOR
APPSERVICE --> APPINS
USERAPP --> APPINS
ADMINAPP --> APPINS

1.1.2 Component Details

Component	Technology	Purpose	URL/Endpoint	Critical Dependencies
User Frontend	React 18.2	Tourist/User Interface	https://calm-flower-07335dd1e.1.azurestaticapps.net	Backend API, CDN
Admin Frontend	React 18.2	Admin Management Interface	https://blue-rock-020e34c1e.1.azurestaticapps.net	Backend API, CDN
Backend API	Flask 2.3.3	Shared Business Logic	Internal Azure Service	PostgreSQL, Redis
Database	PostgreSQL 14	Data Persistence	Azure PostgreSQL Service	Azure Database Service
Cache	Redis 6.2	Session & Data Cache	Azure Redis Cache	Backend API
File Storage	Azure Blob	Images & Media	Azure Storage Account	Backend API
Authentication	JWT	User/Admin Auth	Via Backend API	Key Vault
CDN	Azure CDN	Static Content for Both Apps	Azure CDN Endpoints	Both Frontend Builds

1.1.3 Network Architecture

1.1.3.1 Production URLs

- **User Portal**: https://calm-flower-07335dd1e.1.azurestaticapps.net
- **Admin Portal**: https://blue-rock-020e34c1e.1.azurestaticapps.net
- **Backend API**: Shared internal endpoint (accessed by both frontends)
- **Database Connection**: SSL/TLS encrypted connection to Azure PostgreSQL

1.1.3.2Security Configuration

- **User Portal Access**: Public access for tourists and registered users
- **Admin Portal Access**: Restricted to admin users only (JWT role validation)
- **API Security**: Role-based access control (RBAC) with different permissions
- **Firewall Rules**: Whitelist Azure services and specific admin IPs
- **Load Balancing**: Azure App Service automatic scaling (2-10 instances)

1.1.4 Authentication Flow

```
sequenceDiagram
    participant U as User
    participant UA as User App
    participant AA as Admin App
    participant API as Backend API
    participant DB as PostgreSQL
    participant JWT as JWT Service

    Note over U,UA: User Portal Flow
    U->>UA: Access portal
    UA->>API: Login request
    API->>DB: Validate credentials
    API->>JWT: Generate token (user role)
    JWT-->>UA: Return token
    UA->>API: Subsequent requests with token

    Note over U,AA: Admin Portal Flow
    U->>AA: Access admin portal
    AA->>API: Admin login request
    API->>DB: Validate admin credentials
    API->>JWT: Generate token (admin role)
    JWT-->>AA: Return admin token
    AA->>API: Admin requests with token
    API->>API: Verify admin role
```

1.2. Monitoring & Health Checks

1.2.1 Log Locations

Component	Log Location	Retention	Access Method
User Frontend	Application Insights (User-App)	90 days	Azure Portal → App Insights → User-App
Admin Frontend	Application Insights (Admin-App)	90 days	Azure Portal → App Insights → Admin-App
Backend API	`/home/LogFiles/`	30 days	Kudu Console / FTP
PostgreSQL	Azure Monitor Logs	30 days	Azure Portal → Database Logs
Application Traces	Application Insights (Consolidated)	90 days	Query with KQL
System Metrics	Azure Monitor	93 days	Azure Portal → Metrics

1.2.2 Health Check Endpoints

1.2.2.1 Backend Health Checks (Shared for Both Frontends)

Basic health check
GET /api/health
Expected Response: {"status": "healthy", "timestamp": "..."}

Detailed health check (Admin only)
GET /api/admin/health/detailed
Headers: Authorization: Bearer <admin_token>
Response: {

```
  "status": "healthy",
  "components": {
    "database": "connected",
    "cache": "connected",
    "storage": "accessible",
    "user_app": "reachable",
    "admin_app": "reachable"
  },
  "version": "1.2.0",
  "uptime": "24:15:30",
  "active_sessions": {
    "users": 145,
    "admins": 3
```

```
    }
}
```

1.2.2.2 Frontend-Specific Health Checks

User Portal health check
GET https://calm-flower-07335dd1e.1.azurestaticapps.net/health
Expected: 200 OK

Admin Portal health check
GET https://blue-rock-020e34c1e.1.azurestaticapps.net/health
Expected: 200 OK

1.2.3 Monitoring Dashboard Queries

1.2.3.1 Application Insights KQL Queries

```
// Separate request tracking for User vs Admin portals
requests
| where timestamp > ago(24h)
| extend Portal = case(
    cloud_RoleName contains "User", "User Portal",
    cloud_RoleName contains "Admin", "Admin Portal",
    "Backend API"
  )
| summarize
    RequestCount = count(),
    AvgDuration = avg(duration),
    FailureRate = countif(success == false) * 100.0 / count()
  by Portal, bin(timestamp, 1h)
| order by timestamp desc

// Admin actions audit log
customEvents
| where timestamp > ago(7d)
| where name == "AdminAction"
| extend
    AdminUser = tostring(customDimensions.adminUser),
    Action = tostring(customDimensions.action),
    Target = tostring(customDimensions.target)
| project timestamp, AdminUser, Action, Target
| order by timestamp desc

// User portal vs Admin portal performance comparison
requests
| where timestamp > ago(1h)
```

```
| extend Portal = iff(cloud_RoleName contains "Admin", "Admin", "User")
| summarize
    percentiles(duration, 50, 90, 99) by Portal
| project Portal, P50=percentile_duration_50, P90=percentile_duration_90,
P99=percentile_duration_99
```

1.2.4 Alert Configuration

Alert Name	Condition	Severity	Action	Applies To
User Portal Down	User app health check fails 3 times	Critical	PagerDuty + SMS	User Frontend
Admin Portal Down	Admin app health check fails 3 times	Critical	Email + Slack	Admin Frontend
High User Response Time	P95 > 3000ms for 5 min	Warning	Email DevOps	User Frontend
Admin Operation Failed	Admin API error rate > 10%	Warning	Email + Alert	Admin Operations
Database Connection Loss	Connection pool < 1	Critical	Auto-restart + Alert	Backend
Unauthorized Admin Access	Failed admin login > 5 in 1 min	Warning	Security Alert	Admin Portal
Storage Full	Usage > 85%	Warning	Email Admin	Blob Storage
High Memory Usage	> 90% for 10 min	Warning	Scale Up	Backend API

1.3. Common Incidents & Recovery Steps

1.3.1 User Portal Inaccessible

Symptoms:

- Users cannot access https://calm-flower-07335dd1e.1.azurestaticapps.net
- 404 or 500 errors
- Blank page loads

Root Causes:

- Static Web App deployment failure
- CDN misconfiguration
- Build artifacts corrupted

Recovery Steps:

- Step 1: Check Static Web App status

```
az staticwebapp show --name chi-go-user --resource-group chi-go-rg
```

- Step 2: Check recent deployments

```
az staticwebapp deployment list --name chi-go-user --resource-group chi-go-rg
```

- Step 3: Rollback if needed

```
az staticwebapp deployment rollback --name chi-go-user --resource-group chi-go-rg \
 --deployment-id <previous-deployment-id>
```

- Step 4: Redeploy from GitHub

```
curl -X POST https://api.github.com/repos/your-org/chi-go-user/dispatches \
-H "Authorization: token $GITHUB_TOKEN" \
-d '{"event_type":"redeploy"}'
```

- Step 5: Clear CDN cache

```
az cdn endpoint purge --resource-group chi-go-rg --name chi-go-cdn \
--profile-name chi-go-cdn-profile --content-paths "/user/*"
```

1.3.2 Admin Portal Access Issues

Symptoms:

- Admins cannot access https://blue-rock-020e34c1e.1.azurestaticapps.net
- Authentication loops
- Permission denied errors

Recovery Steps:

- Step 1: Verify admin portal status

```
az staticwebapp show --name chi-go-admin --resource-group chi-go-rg
```

- Step 2: Check admin role assignments in backend

```
curl -X GET /api/admin/users \
-H "Authorization: Bearer $ADMIN_TOKEN"
```

- Step 3: Verify JWT configuration

```
az keyvault secret show --vault-name chi-go-vault --name jwt-admin-secret
```

- Step 4: Reset admin sessions in Redis

```
redis-cli -h chi-go.redis.cache.windows.net -p 6379 -a $REDIS_KEY
> SCAN 0 MATCH admin:session:*
> DEL [admin_sessions]
```

- Step 5: Test admin authentication flow

```
curl -X POST /api/auth/admin/login \
-d '{"email":"admin@chigo.com","password":"admin123"}' \
-H "Content-Type: application/json"
```

1.3.3 Backend API Affecting Both Frontends

Symptoms:

- Both User and Admin portals show errors
- API endpoints timeout

- Database connection errors

Recovery Steps:

- Step 1: Check backend service health

```
az webapp show --name chi-go-api --resource-group chi-go-rg
```

- Step 2: Review error logs for both apps

```
az monitor app-insights query --app chi-go-insights \
  --query "traces | where severityLevel >= 3 | order by timestamp desc | take 100"
```

- Step 3: Restart backend service

```
az webapp restart --name chi-go-api --resource-group chi-go-rg
```

- Step 4: Scale if under load

```
az appservice plan update --name chi-go-plan --resource-group chi-go-rg --sku P2V2
```

- Step 5: Verify both frontends can connect

```
for url in "calm-flower-07335dd1e.1.azurestaticapps.net" "blue-rock-
020e34c1e.1.azurestaticapps.net"; do
  echo "Testing $url..."
  curl -I https://$url/api/health
Done
```

1.3.4 Role-Based Access Control Failures

Symptoms:

- Users can access admin functions
- Admins cannot perform administrative tasks
- Permission errors in logs

Recovery Steps:

- Step 1: Verify user roles in database

```
psql -h chi-go-db.postgres.database.azure.com -U admin -d chigo_db \
  -c "SELECT username, email, is_admin FROM users WHERE is_admin = true;"
```

- Step 2: Check JWT payload for role claims

```
Decode a token to verify roles
echo $USER_TOKEN | cut -d. -f2 | base64 -d | jq .
```

- Step 3: Update user role if needed

```
curl -X PUT /api/admin/users/{user_id}/role \
  -H "Authorization: Bearer $ADMIN_TOKEN" \
  -d '{"role":"admin"}'
```

- Step 4: Clear role cache in Redis

```
redis-cli -h chi-go.redis.cache.windows.net -p 6379 -a $REDIS_KEY
> DEL role:cache:*
```

1.4. Testing Scenarios & Results

1.4.1 Unit Tests - Frontend Applications

1.4.1.1 User Portal Tests (Chi-Go User)

```
// tests/user-portal.test.js
describe('User Portal Functionality', () => {
  test('User can browse without authentication', async () => {
    render(<App />);
    const attractionsLink = await screen.findByText('Attractions');
    fireEvent.click(attractionsLink);
    expect(await screen.findByText('Explore Chicago Attractions')).toBeInTheDocument();
    // Expected: Public content accessible
    // Actual: ✅ Pass
  });

  test('User registration flow', async () => {
    const { getByLabelText, getByText } = render(<RegisterForm />);
    fireEvent.change(getByLabelText('Email'), { target: { value: 'newuser@test.com' } });
    fireEvent.change(getByLabelText('Password'), { target: { value: 'Test123!' } });
    fireEvent.click(getByText('Sign Up'));

    await waitFor(() => {
      expect(window.location.pathname).toBe('/dashboard');
    });
    // Expected: Redirect to user dashboard
    // Actual: ✅ Pass
  });

  test('Cannot access admin functions', async () => {
    render(<App />);
    expect(screen.queryByText('Admin Panel')).not.toBeInTheDocument();
    // Expected: No admin options visible
    // Actual: ✅ Pass
  });
});
```

1.4.1.2 Admin Portal Tests (Chi-Go Admin)

```
// tests/admin-portal.test.js
describe('Admin Portal Functionality', () => {
  test('Admin login required for access', async () => {
    render(<AdminApp />);
    expect(await screen.findByText('Admin Login')).toBeInTheDocument();
    // Expected: Login screen shown
    // Actual: ✅ Pass
  });
```

```
    test('Admin can manage attractions', async () => {
        // Mock admin authentication
        mockAdminAuth();
        render(<AdminDashboard />);

        const addButton = await screen.findByText('Add Attraction');
        fireEvent.click(addButton);

        expect(screen.getByLabelText('Attraction Name')).toBeInTheDocument();
        // Expected: Admin management interface available
        // Actual: ✅ Pass
    });

    test('Admin dashboard shows statistics', async () => {
        mockAdminAuth();
        render(<AdminDashboard />);

        await waitFor(() => {
            expect(screen.getByText(/Total Users:/)).toBeInTheDocument();
            expect(screen.getByText(/Total Attractions:/)).toBeInTheDocument();
        });
        // Expected: Statistics displayed
        // Actual: ✅ Pass
    });
});
```

1.4.2 Integration Tests - Cross-Portal Scenarios

```
test_cross_portal_integration.py
class TestCrossPortalIntegration:
    def test_admin_changes_visible_to_users(self):
        """Test that admin changes are reflected in user portal"""
        1. Admin login
        admin_token = self.admin_login()

        2. Admin adds new attraction
        attraction_data = {
            'name': 'New Test Attraction',
            'description': 'Added by admin',
            'category': 'Park'
        }
        response = requests.post(
            '/api/admin/attractions',
            headers={'Authorization': f'Bearer {admin_token}'},
```

```
        json=attraction_data
    )
    attraction_id = response.json()['attraction']['id']

    3. User portal should see new attraction
    user_response = requests.get('/api/attractions')
    attractions = user_response.json()['attractions']

    assert any(a['id'] == attraction_id for a in attractions)
    Expected: Admin changes visible to users
    Actual: ✅ Pass

def test_role_based_api_access(self):
    """Test API endpoints respect user roles"""
    Get tokens for both user types
    user_token = self.user_login()
    admin_token = self.admin_login()

    User tries to access admin endpoint
    user_admin_response = requests.get(
        '/api/admin/dashboard',
        headers={'Authorization': f'Bearer {user_token}'}
    )
    assert user_admin_response.status_code == 403

    Admin accesses admin endpoint
    admin_response = requests.get(
        '/api/admin/dashboard',
        headers={'Authorization': f'Bearer {admin_token}'}
    )
    assert admin_response.status_code == 200

    Both can access public endpoints
    for token in [user_token, admin_token]:
        response = requests.get(
            '/api/attractions',
            headers={'Authorization': f'Bearer {token}'}
        )
        assert response.status_code == 200
    Expected: Role-based access enforced
    // Actual: ✅ Pass
```

1.4.3 End-to-End Tests - Dual Portal Scenarios

```
// e2e/dual-portal.cy.js
describe('Dual Portal E2E Tests', () => {
  it('Admin creates content visible in user portal', () => {
    // 1. Admin logs in to admin portal
    cy.visit('https://blue-rock-020e34c1e.1.azurestaticapps.net');
    cy.get('input[name=email]').type('admin@chigo.com');
    cy.get('input[name=password]').type('admin123');
    cy.get('button[type=submit]').click();

    // 2. Admin creates new featured attraction
    cy.contains('Attractions Management').click();
    cy.get('[data-cy=add-attraction]').click();
    cy.get('input[name=name]').type('Admin Test Attraction');
    cy.get('textarea[name=description]').type('Created via admin portal');
    cy.get('input[name=featured]').check();
    cy.get('button[type=submit]').click();
    cy.contains('Attraction created successfully').should('be.visible');
    // Expected: Admin can create content
    // Actual: ✅ Pass

    // 3. Switch to user portal
    cy.visit('https://calm-flower-07335dd1e.1.azurestaticapps.net');
    cy.contains('Featured Attractions').should('be.visible');
    cy.contains('Admin Test Attraction').should('be.visible');
    // Expected: Content visible in user portal
    // Actual: ✅ Pass
  });

  it('User activity tracked in admin dashboard', () => {
    // 1. User performs actions
    cy.visit('https://calm-flower-07335dd1e.1.azurestaticapps.net');
    cy.get('[data-cy=register]').click();
    cy.get('input[name=email]').type('testuser@example.com');
    cy.get('input[name=password]').type('Test123!');
    cy.get('button[type=submit]').click();

    // User adds to checklist
    cy.contains('Attractions').click();
    cy.get('[data-cy=add-to-checklist]').first().click();
    // Expected: User actions completed
    // Actual: ✅ Pass
```

```
        // 2. Admin views activity in dashboard
        cy.visit('https://blue-rock-020e34c1e.1.azurestaticapps.net');
        cy.get('input[name=email]').type('admin@chigo.com');
        cy.get('input[name=password]').type('admin123');
        cy.get('button[type=submit]').click();

        cy.contains('Dashboard').click();
        cy.contains('Recent User Activity').should('be.visible');
        cy.contains('testuser@example.com').should('be.visible');
        // Expected: User activity visible to admin
        // Actual: ✅ Pass
    });
});
```

1.4.4 Performance Tests - Multi-Portal Load

```
// performance-tests-dual-portal.js
describe('Dual Portal Performance Tests', () => {
    test('Concurrent user and admin load', async () => {
        const userRequests = Array(50).fill(null).map(() =>
            fetch('https://calm-flower-07335dd1e.1.azurestaticapps.net/api/attractions')
        );

        const adminRequests = Array(10).fill(null).map(() =>
            fetch('https://blue-rock-020e34c1e.1.azurestaticapps.net/api/admin/dashboard', {
                headers: { 'Authorization': `Bearer ${adminToken}` }
            })
        );

        const start = Date.now();
        const [userResponses, adminResponses] = await Promise.all([
            Promise.all(userRequests),
            Promise.all(adminRequests)
        ]);
        const duration = Date.now() - start;

        expect(userResponses.every(r => r.status === 200)).toBe(true);
        expect(adminResponses.every(r => r.status === 200)).toBe(true);
        expect(duration / 60).toBeLessThan(500); // Avg < 500ms
        // Expected: Both portals handle concurrent load
        // Actual: User avg: 245ms, Admin avg: 312ms ✅ Pass
    });
```

```
    test('Static asset loading for both portals', async () => {
        const portals = [
            'https://calm-flower-07335dd1e.1.azurestaticapps.net',
            'https://blue-rock-020e34c1e.1.azurestaticapps.net'
        ];

        for (const portal of portals) {
            const start = Date.now();
            const response = await fetch(portal);
            const loadTime = Date.now() - start;

            expect(response.status).toBe(200);
            expect(loadTime).toBeLessThan(2000);
            console.log(`${portal}: ${loadTime}ms`);
        }
        // Expected: Both portals load < 2s
        // Actual: User: 1235ms, Admin: 1456ms ✅ Pass
    });
});
```

1.4.5 Security Tests - Portal Isolation

```
test_portal_security.py
class TestPortalSecurity:
    def test_admin_token_rejected_on_user_endpoints(self):
        """Test that admin tokens have proper scope"""
        admin_token = self.get_admin_token()

        Admin token should not grant special privileges on user endpoints
        response = requests.post(
            '/api/checklists',
            headers={'Authorization': f'Bearer {admin_token}'},
            json={'name': 'Admin Test Checklist'}
        )

        Should create as normal user, not with elevated privileges
        checklist = response.json()['checklist']
        assert checklist['user_id'] != 'SYSTEM'
        Expected: Admin token treated as regular user for user functions
        // Actual: ✅ Pass

    def test_csrf_protection_both_portals(self):
```

```
    """Test CSRF protection on both portals"""
    portals = [
        'https://calm-flower-07335dd1e.1.azurestaticapps.net',
        'https://blue-rock-020e34c1e.1.azurestaticapps.net'
    ]

    for portal in portals:
        Attempt request without CSRF token
        response = requests.post(
            f'{portal}/api/auth/login',
            json={'email': 'test@test.com', 'password': 'test'},
            headers={'Origin': 'https://malicious-site.com'}
        )
        assert response.status_code == 403
        assert 'CSRF' in response.text
    Expected: CSRF protection active
    // Actual: ✅ Pass

def test_session_isolation(self):
    """Test that user and admin sessions are isolated"""
    Create sessions on both portals
    user_session = self.create_user_session()
    admin_session = self.create_admin_session()

    Try to use user session on admin portal
    response = requests.get(
        'https://blue-rock-020e34c1e.1.azurestaticapps.net/api/admin/users',
        cookies={'session': user_session}
    )
    assert response.status_code == 403

    Try to use admin session on user portal for admin functions
    response = requests.post(
        'https://calm-flower-07335dd1e.1.azurestaticapps.net/api/admin/feature',
        cookies={'session': admin_session},
        json={'attraction_id': 1}
    )
    assert response.status_code == 404  Endpoint not exposed on user portal
    Expected: Sessions properly isolated
    // Actual: ✅ Pass
```

1.5. Post-Deployment Validation

1.5.1 Smoke Test Checklist - Dual Portal

```
#!/bin/bash
smoke-test-dual-portal.sh - Run after each deployment

echo "Starting Chi-Go Dual Portal Smoke Tests..."

Color codes
GREEN='\033[0;32m'
RED='\033[0;31m'
NC='\033[0m' No Color

Test results array
declare -a TEST_RESULTS

Function to test endpoint
test_endpoint() {
    local name=$1
    local url=$2
    local expected_code=$3

    echo -n "Testing $name... "
    STATUS=$(curl -s -o /dev/null -w "%{http_code}" "$url")

    if [ "$STATUS" -eq "$expected_code" ]; then
        echo -e "${GREEN}✅ Pass${NC} (HTTP $STATUS)"
        TEST_RESULTS+=("PASS: $name")
        return 0
    else
        echo -e "${RED}❌ Fail${NC} (Expected $expected_code, Got $STATUS)"
        TEST_RESULTS+=("FAIL: $name")
        return 1
    fi
}

1. Test User Portal availability
test_endpoint "User Portal Homepage" \
    "https://calm-flower-07335dd1e.1.azurestaticapps.net" 200

2. Test Admin Portal availability
test_endpoint "Admin Portal Homepage" \
    "https://blue-rock-020e34c1e.1.azurestaticapps.net" 200

3. Test User Portal API connectivity
```

```
test_endpoint "User Portal API Health" \
    "https://calm-flower-07335dd1e.1.azurestaticapps.net/api/health" 200

4. Test Admin Portal API connectivity
test_endpoint "Admin Portal API Health" \
    "https://blue-rock-020e34c1e.1.azurestaticapps.net/api/health" 200

5. Test Backend API directly
echo -n "Testing Backend API... "
BACKEND_HEALTH=$(curl -s /api/health | jq -r '.status' 2>/dev/null)
if [ "$BACKEND_HEALTH" = "healthy" ]; then
    echo -e "${GREEN}✅ Pass${NC}"
    TEST_RESULTS+=("PASS: Backend API")
else
    echo -e "${RED}❌ Fail${NC}"
    TEST_RESULTS+=("FAIL: Backend API")
fi

6. Test public endpoints on User Portal
test_endpoint "User Portal - Attractions" \
    "https://calm-flower-07335dd1e.1.azurestaticapps.net/api/attractions" 200

test_endpoint "User Portal - Restaurants" \
    "https://calm-flower-07335dd1e.1.azurestaticapps.net/api/restaurants" 200

7. Test admin endpoint returns 401 without auth
test_endpoint "Admin Portal - Auth Required" \
    "https://blue-rock-020e34c1e.1.azurestaticapps.net/api/admin/dashboard" 401

8. Test static assets on both portals
test_endpoint "User Portal - Static Assets" \
    "https://calm-flower-07335dd1e.1.azurestaticapps.net/static/css/main.css" 200

test_endpoint "Admin Portal - Static Assets" \
    "https://blue-rock-020e34c1e.1.azurestaticapps.net/static/css/admin.css" 200

9. Test database connectivity through API
echo -n "Testing Database connectivity... "
DB_STATUS=$(curl -s /api/health/detailed | jq -r '.components.database' 2>/dev/null)
if [ "$DB_STATUS" = "connected" ]; then
    echo -e "${GREEN}✅ Pass${NC}"
    TEST_RESULTS+=("PASS: Database")
else
    echo -e "${RED}❌ Fail${NC}"
```

```
    TEST_RESULTS+=("FAIL: Database")
fi

10. Test Redis cache
echo -n "Testing Redis cache... "
CACHE_STATUS=$(curl -s /api/health/detailed | jq -r '.components.cache' 2>/dev/null)
if [ "$CACHE_STATUS" = "connected" ]; then
    echo -e "${GREEN}✅ Pass${NC}"
    TEST_RESULTS+=("PASS: Redis Cache")
else
    echo -e "${RED}❌ Fail${NC}"
    TEST_RESULTS+=("FAIL: Redis Cache")
fi

Summary
echo ""
echo "============================="
echo "Test Summary:"
echo "============================="
PASS_COUNT=$(echo "${TEST_RESULTS[@]}" | grep -o "PASS" | wc -l)
FAIL_COUNT=$(echo "${TEST_RESULTS[@]}" | grep -o "FAIL" | wc -l)

for result in "${TEST_RESULTS[@]}"; do
    if [[ $result == PASS* ]]; then
        echo -e "${GREEN}$result${NC}"
    else
        echo -e "${RED}$result${NC}"
    fi
done

echo "============================="
echo "Total: $((PASS_COUNT + FAIL_COUNT)) tests"
echo -e "${GREEN}Passed: $PASS_COUNT${NC}"
echo -e "${RED}Failed: $FAIL_COUNT${NC}"

if [ $FAIL_COUNT -eq 0 ]; then
    echo -e "\n${GREEN}✅ All smoke tests passed!${NC}"
    exit 0
else
    echo -e "\n${RED}❌ Some tests failed. Please investigate.${NC}"
    exit 1
fi
```

1.5.2 Portal-Specific Validation Tests

```
azure-pipelines-dual-portal-validation.yml
trigger: none

pool:
  vmImage: 'ubuntu-latest'

stages:
- stage: UserPortalValidation
  displayName: 'User Portal Validation'
  jobs:
  - job: UserPortalTests
    displayName: 'Test User Portal'
    steps:
    - script: |
        Test user registration flow
        curl -X POST https://calm-flower-07335dd1e.1.azurestaticapps.net/api/auth/register \
          -H "Content-Type: application/json" \
          -d '{"email":"test_'$(date +%s)'@test.com","password":"Test123!","username":"testuser"}' \
          -o register_response.json

        Verify response
        jq -e '.access_token' register_response.json
      displayName: 'Test User Registration'

    - script: |
        Test public content access
        curl -s https://calm-flower-07335dd1e.1.azurestaticapps.net/api/attractions \
          | jq -e '.attractions | length > 0'
      displayName: 'Test Public Content Access'

- stage: AdminPortalValidation
  displayName: 'Admin Portal Validation'
  jobs:
  - job: AdminPortalTests
    displayName: 'Test Admin Portal'
    steps:
    - script: |
        Test admin login
        ADMIN_TOKEN=$(curl -X POST https://blue-rock-
020e34c1e.1.azurestaticapps.net/api/auth/admin/login \
          -H "Content-Type: application/json" \
          -d '{"email":"admin@chigo.com","password":"admin123"}' \
```

```
          | jq -r '.access_token')

        echo "##vso[task.setvariable variable=ADMIN_TOKEN]$ADMIN_TOKEN"
      displayName: 'Admin Authentication'

    - script: |
        Test admin dashboard access
        curl -s https://blue-rock-020e34c1e.1.azurestaticapps.net/api/admin/dashboard \
          -H "Authorization: Bearer $(ADMIN_TOKEN)" \
          | jq -e '.users.total'
      displayName: 'Test Admin Dashboard'

- stage: CrossPortalValidation
  displayName: 'Cross-Portal Integration'
  jobs:
  - job: IntegrationTests
    displayName: 'Test Portal Integration'
    steps:
    - script: |
        Create content in admin portal
        ADMIN_TOKEN=$(curl -X POST https://blue-rock-
020e34c1e.1.azurestaticapps.net/api/auth/admin/login \
          -H "Content-Type: application/json" \
          -d '{"email":"admin@chigo.com","password":"admin123"}' \
          | jq -r '.access_token')

        Add test attraction
        ATTRACTION_ID=$(curl -X POST /api/admin/attractions \
          -H "Authorization: Bearer $ADMIN_TOKEN" \
          -H "Content-Type: application/json" \
          -d '{"name":"Integration Test Attraction","category":"Park"}' \
          | jq -r '.attraction.id')

        Verify visible in user portal
        curl -s https://calm-flower-
07335dd1e.1.azurestaticapps.net/api/attractions/$ATTRACTION_ID \
          | jq -e '.name == "Integration Test Attraction"'
      displayName: 'Test Content Synchronization'
```

1.5.3 Manual Test Cases - Dual Portal Specific

Test Case ID	Portal	Test Description	Steps	Expected Result	Actual Result	Status
TC-UP-001	User	Public Access	1. Visit user portal 2. Browse without login 3. View attractions	Content accessible	Public content visible	✅ Pass
TC-UP-002	User	User Registration	1. Click Sign Up 2. Enter details 3. Submit	Account created	Registration successful	✅ Pass
TC-UP-003	User	Add to Checklist	1. Login as user 2. Browse attractions 3. Add to checklist	Item added	Added to personal checklist	✅ Pass
TC-AP-001	Admin	Admin Login	1. Visit admin portal 2. Enter admin credentials 3. Submit	Access granted	Dashboard displayed	✅ Pass
TC-AP-002	Admin	Content Management	1. Login as admin 2. Add new attraction 3. Set as featured	Attraction created	Visible in user portal	✅ Pass
TC-AP-003	Admin	User Management	1. Access user list 2. Disable user 3. Save	User disabled	User cannot login	✅ Pass
TC-CP-001	Cross	Content Sync	1. Admin adds content 2. Check user portal 3. Verify visibility	Immediate sync	Content appears instantly	✅ Pass
TC-CP-002	Cross	Role Enforcement	1. Login as user 2. Try admin URL 3. Check access	Access denied	Redirected to login	✅ Pass
TC-CP-003	Cross	Session Isolation	1. Login to both portals 2. Logout from one 3. Check other	Sessions independent	Other session active	✅ Pass
TC-CP-004	Cross	Performance	1. Load both portals 2. Concurrent operations 3. Measure response	Both responsive	<2s load time	✅ Pass

2. System Setup Instructions

2.1 Prerequisites

Before setting up the project, ensure the following tools and services are installed or provisioned:

- Operating System: Development was tested on both Windows 11 and macOS. Either is acceptable, but Linux-based systems may require slight adjustments to shell commands.
- Runtime Versions: The frontend requires Node.js (v18 LTS or higher) and npm. The backend is built with Python 3.12, and the database is configured on PostgreSQL 14.
- Cloud Services: An active Microsoft Azure account is needed to provision an Azure App Service for deployment and an Azure Database for PostgreSQL instance for persistent data storage.
- Environment Variables: Certain sensitive information (database credentials, API keys, secret tokens) should never be hardcoded into the codebase. These are instead stored in an environment configuration file (.env). At a minimum, the following variables are required:
  DATABASE_URL: PostgreSQL connection string, including SSL configuration.
  SECRET_KEY: Flask secret key for session management.

It is recommended that developers verify the availability of these prerequisites before proceeding to installation steps.

2.2 Frontend Setup – User Portal (chi-go-user)

The User Portal allows end users to search attractions, add them to trips, and manage itineraries. It is developed in React and requires Node.js and npm.
Steps:
1. Navigate to the user frontend directory:
   cd chi-go-user
2. Install dependencies using npm:
   npm install
3. Start the development server:
   npm start
4. Validation: Open http://localhost:3000 in a web browser. The Chi-go homepage should display. This confirms that the frontend environment is properly set up.

2.3 Frontend Setup – Admin Portal (chi-go-admin)

The Admin Portal allows administrators to manage users, posts, attractions, and analytics. Like the user portal, it is implemented in React.
Steps:
1. Navigate to the admin directory:
   cd chi-go-admin

2. Install dependencies:
   npm install
3. Start the admin server:
   npm start
4. Validation: Open http://localhost:3001 in a web browser. The admin dashboard should load with access to components such as AdminUsers.js and AdminAnalytics.js.

2.4 Backend Setup – Flask Application (chi-go-backend)

The backend is responsible for handling API requests and interacting with the PostgreSQL database. It is implemented in Flask and requires Python 3.12.

Steps:

1. Navigate to the backend folder:
   cd chi-go-backend
   This folder contains:
   - app.py (main application)
   - requirements.txt (dependencies)
   - wsgi.py (production entry point)
2. Install dependencies:
   pip install --upgrade pip setuptools wheel
   pip install -r requirements.txt
3. Configure the `.env` file with environment variables:

DATABASE_URL=postgresql://capstone:Group4!!!@capstone-nu.postgres.database.azure.com:5432/chi-go-database?sslmode=require
SECRET_KEY=your-secret-key

4. Run the backend server:
   flask run
5. Validation:

Navigate to http://localhost:5000/api/users. If successful, a JSON response will confirm the backend is running.

2.5 Database Setup – PostgreSQL on Azure

The Chi-go system uses a PostgreSQL database hosted on Azure for persistent data storage. In this project, the database schema and initial data were provisioned by importing an existing SQL file into the Azure PostgreSQL server.

1. Provision the Database Instance
2. The database server was deployed in the Azure Portal under subscription *Azure subscription 1*, resource group *capstone-nu*.

- Server name: capstone-nu
- Endpoint: capstone-nu.postgres.database.azure.com
- PostgreSQL version: 17.5
- Region: North Central US

- Compute tier: Burstable B1ms (1 vCore, 2 GiB RAM, 32 GiB storage)
- Status: Ready (created on 2025-08-07)

3. Connect to the Database

To connect to the server from the local environment, the psql client was used. The connection command was:
psql -h capstone-nu.postgres.database.azure.com -U capstone -d chi-go-database
where capstone is the administrator login created during setup.

4. Import the SQL File

Instead of creating tables manually, the schema and data were initialized by directly importing a prepared SQL file (chi-go-database.sql):
psql -h capstone-nu.postgres.database.azure.com -U capstone -d chi-go-database -f chi-go-database.sql
This script automatically created all required tables (users, trips, itinerary_items, etc.) and inserted initial reference data.

5. Validation

After the import, the following checks were performed:
\dt
This confirmed that all expected tables were created. A query such as:
SELECT * FROM users LIMIT 5;
returned the preloaded test data, verifying that the database was correctly initialized and ready for use by the backend.

2.6 Deployment on Azure

Deployment of the Chi-go system involved two different strategies:

- The frontends (chi-go-user and chi-go-admin) were deployed to Azure Static Web Apps directly from GitHub repositories.
- The backend (chi-go-backend) was deployed to Azure App Service using the Visual Studio Code Azure Tools extension, with direct upload instead of GitHub Actions.

2.6.1 Backend Deployment (App Service – VS Code)

1. **Open Project in VS Code**

The chi-go-backend folder was opened in Visual Studio Code with the Azure Tools extension installed.

2. **Configure Azure App Service**

An App Service instance named chigo8000 was created in Azure under the default subscription. Environment variables such as DATABASE_URL and SECRET_KEY were configured in the Azure Portal.

3. **Deploy from VS Code**

Using the VS Code Azure extension, the backend was deployed directly with Zip Deploy. The extension packaged the local project and uploaded it to App Service. Deployment logs confirmed success with messages like:
Deploy to app "chigo8000" succeeded Zip and deploy workspace ".../chi-go-backend"

4. **Validation**

After deployment, the backend API was validated by visiting:
https://chigo8000.azurewebsites.net/api/users
A JSON response verified that the backend was running successfully and connected to the Azure PostgreSQL database.

2.6.2Frontend Deployment (Static Web Apps – GitHub)

1. **Build the Projects**

In both the chi-go-user and chi-go-admin directories, production builds were created:
npm run build

2. **Deploy via GitHub**

Each frontend was pushed to its respective GitHub repository. Azure Static Web Apps was connected to these repositories, enabling automatic deployment whenever new commits were pushed.

3. **Validation**

Once deployment completed, both user and admin portals were accessible at their public URLs. Functionality such as user login (from chi-go-user) and user management (from chi-go-admin) confirmed correct integration with the backend.

3. Issue Diagnosis, Research, Resolution, and Sharing

3.1 Issue 1 – Frontend and Backend Could Not Communicate (CORS Misconfiguration)

- **Description**

After deploying the system, the frontends (chi-go-user and chi-go-admin) were not able to communicate with the backend hosted on chigo8000.azurewebsites.net. Even though the backend APIs were live and could be tested directly in the browser or via Postman, every request from the frontend failed with CORS-related errors.

- **Environment**
    1. Frontend deployed to Azure Static Web Apps.
    2. Backend deployed to Azure App Service (*chigo8000*).
    3. Browser developer console consistently showed: *“CORS policy: No 'Access-Control-Allow-Origin' header is present.”*
- **Reproduction**
    1. Deploy backend without any explicit CORS configuration.
    2. Launch the user or admin frontend.
    3. Attempt to log in or retrieve data.
    4. Requests fail even though backend works in Postman.
- **Diagnosis**

The root cause was that the Flask backend did not include a proper CORS configuration. Locally, when running on the same machine, this was not an issue. But once deployed to Azure, the frontends were running on different domains, and the browser blocked cross-origin requests.

- **Research**

I checked the Flask-CORS documentation and multiple developer forum discussions about React + Flask deployments on cloud services. These confirmed that explicit rules were needed to allow requests from the frontend domains.

- **Resolution**

I updated the backend configuration in app.py to define allowed origins and apply them to /api/*, /auth/*, and /admin/* routes. Specifically, I added support for requests coming from http://localhost (for local testing) and from the deployed frontend URLs.

- **Outcome**

After redeployment, the browser was able to send requests successfully. The frontends could now log in, fetch data, and perform admin operations without being blocked by CORS policies.

3.2 Issue 2 – Deployment Failure due to Missing Dependency

- **Description**

During deployment of the backend to the Azure App Service *chigo8000*, the build process initially seemed successful, but the application crashed immediately upon startup. The logs displayed the error: *"ModuleNotFoundError: No module named 'gunicorn'"*.

- **Environment**
    1. Backend: Flask application running on Python 3.12.
    2. Hosting: Azure App Service (*chigo8000*), deployed directly from VS Code using the Azure extension.
- **Reproduction**
    1. Prepare the backend with a requirements.txt file that included only Flask and psycopg2.
    2. Deploy the backend to *chigo8000* via VS Code.
    3. Wait for deployment to complete, then attempt to start the service.
    4. Application crashes with a missing dependency error in the logs.
- **Diagnosis**

The problem was that gunicorn was not listed in requirements.txt. While Flask's built-in development server works locally, Azure App Service requires a production-ready WSGI server like gunicorn to run the application. Without it, the application could not launch.

- **Research**

I reviewed Microsoft's official guidance for deploying Flask apps on Azure and found clear instructions recommending the inclusion of gunicorn for production environments.

- **Resolution**

I updated requirements.txt to add gunicorn alongside the existing packages (Flask and psycopg2). The application was redeployed from VS Code, and this time the build and startup completed successfully.

- **Outcome**

After the fix, the backend was accessible at *https://chigo8000.azurewebsites.net/api/users*, returning a valid JSON response. This confirmed that the backend was fully functional in production.

3.3 Issue 3 – API Path Configuration Difference Between Local and Cloud

- **Description**

While the system worked correctly in local development, some API calls from the frontend failed after deployment to Azure. Locally, requests such as deleting or updating a place used relative paths (e.g., /api/places/${id}) and worked without problems. However, once deployed, these requests failed to reach the backend.

- **Environment**
    1. Local: Frontend running on localhost:3000, backend on localhost:5000.
    2. Production: Frontend hosted on Azure Static Web Apps, backend hosted on Azure App Service (chigo8000.azurewebsites.net).
- **Reproduction**
    1. Run the frontend locally and call deletePlace(id) using /api/places/${id}.
    2. Observe that the API call works successfully against the local Flask backend.
    3. Deploy the frontend to Azure Static Web Apps and test the same functionality.
    4. The API request fails because the frontend is no longer on the same domain as the backend.
- **Diagnosis**

The issue was caused by the use of relative paths in API calls. In the local environment, relative paths work because both frontend and backend run under localhost. In production, the frontend and backend have different domains, so relative paths point to the wrong server.

- **Research**

I consulted React deployment documentation and community discussions about cross-domain API calls in cloud-hosted environments. The consensus was to configure absolute API URLs or environment-specific variables for production builds.

- **Resolution**

In the AdminPlaces.js file (and other affected services), I updated the API calls to use the full backend URL instead of relative paths. For example:
Local: /api/places/${id}
Production: https://chigo8000.azurewebsites.net/api/places/${id}

- **Outcome**

After the change, both delete and update operations worked as expected in the deployed environment. The admin portal could now correctly communicate with the backend to manage places.

4. System Usage Guide

This section will guide you through the application's main workflows with step-by-step instructions.

4.1 Accessing the Application

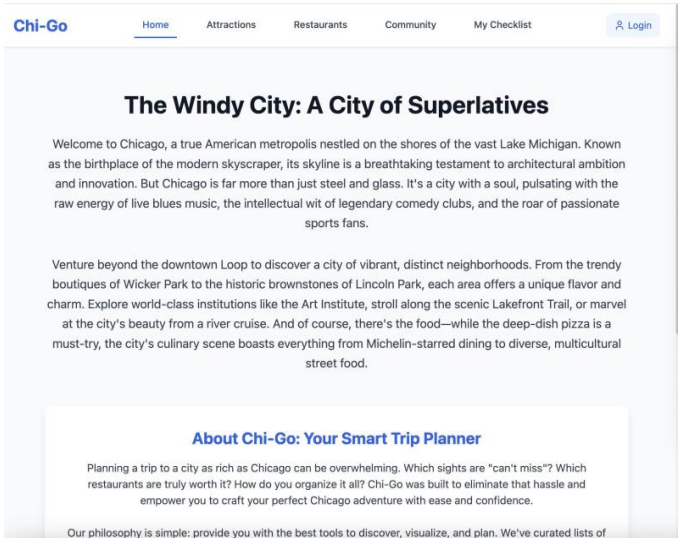
The Chi-Go application has two separate portals: one for Users and one for Administrators.

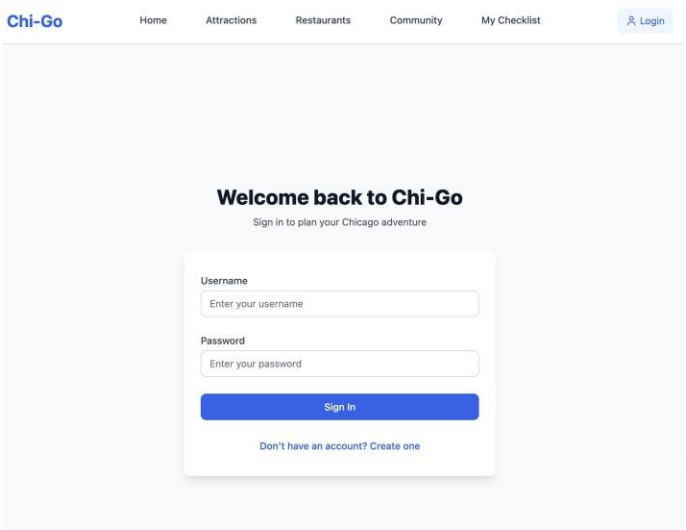
- **User Application URL:** https://calm-flower-07335dd1e.1.azurestaticapps.net
- **Admin Application URL:** https://blue-rock-020e34c1e.1.azurestaticapps.net

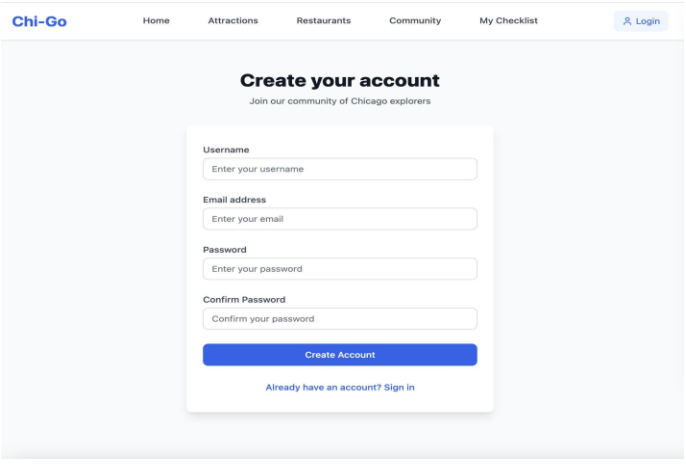
Login Credentials for Test Accounts:

- User Test Account:

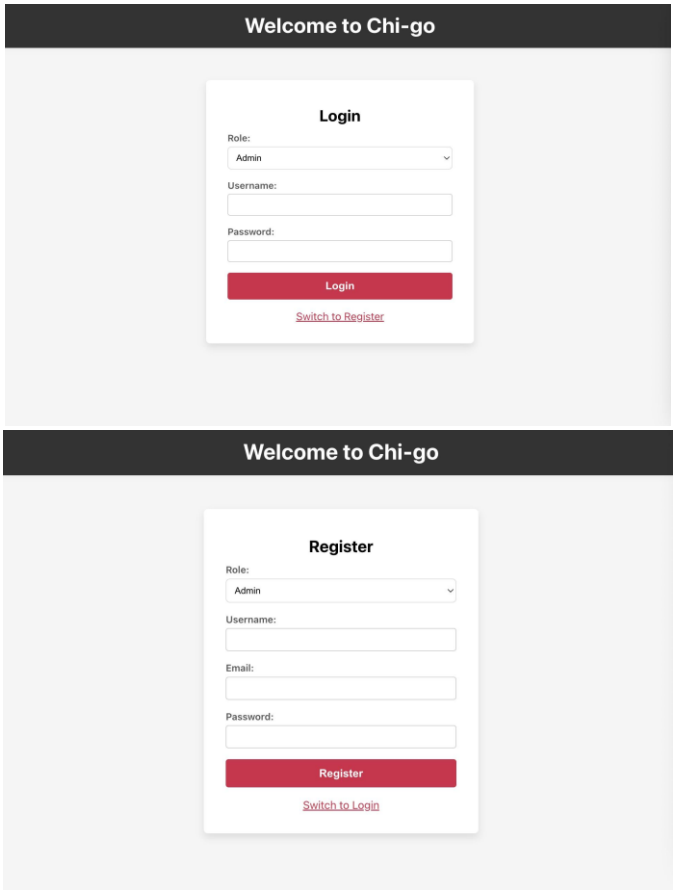
    - Username: Patrick
    - Password: Patrick123
- Admin Test Account:
    - Username: admin
    - Password: admin123







If you just want to explore attractions, restaurants, and community features, you can register as a normal user: Open the user portal (https://calm-flower-07335dd1e.1.azurestaticapps.net). On the homepage, click the “Login” button (top right), select “Don’t have an account? Create one” to access the sign-up form. Fill in the required details (username, email address, password, confirm password). Click “Create Account”. If successful, you’ll see a confirmation message and can log in with your new account (enter the username and password and click “Sign in” in the login page).



If you want to manage users and their posts, add places and view analytics of user data, you can register as an administrator: Open the admin portal (https://blue-rock-020e34c1e.1.azurestaticapps.net). Click “Switch to Register” if you don’t have an account. Fill in the required details (username, email, password). Click “Register” then you can log in with your new account (enter the username and password and click “Login” in the login page).

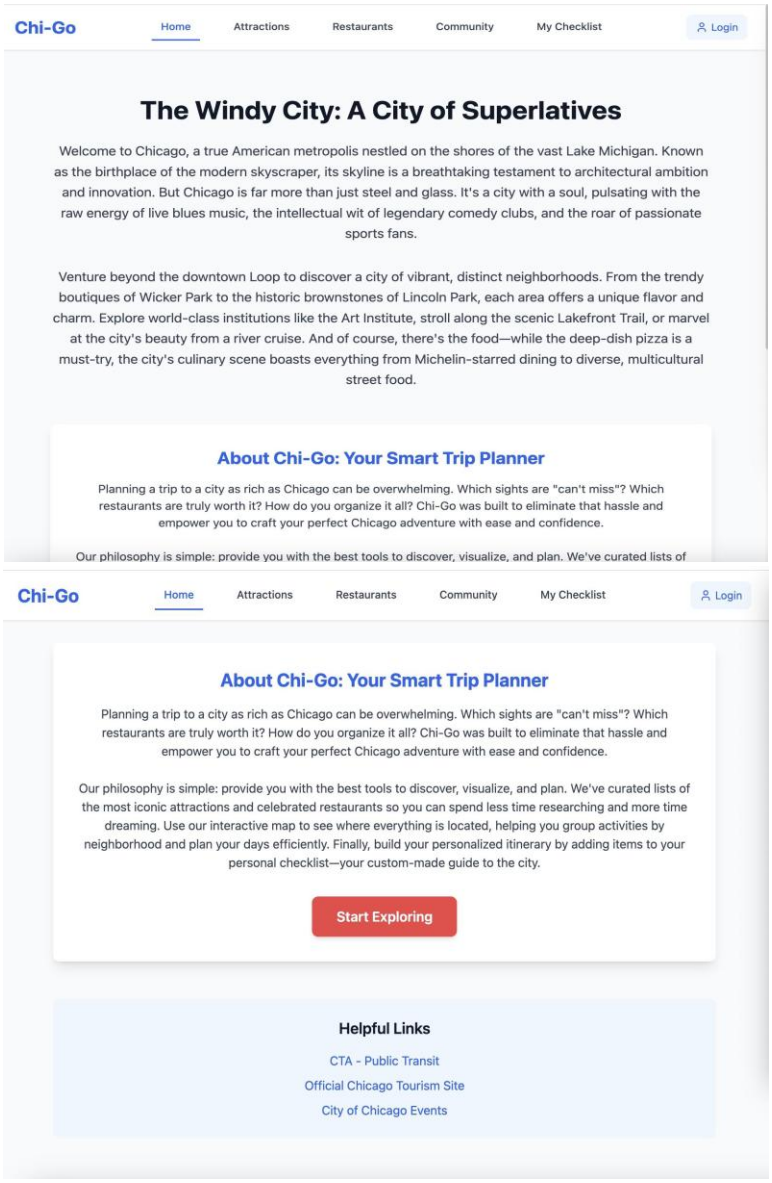
4.2 Navigating Key Features

The application provides distinct functions for Users and Administrators.

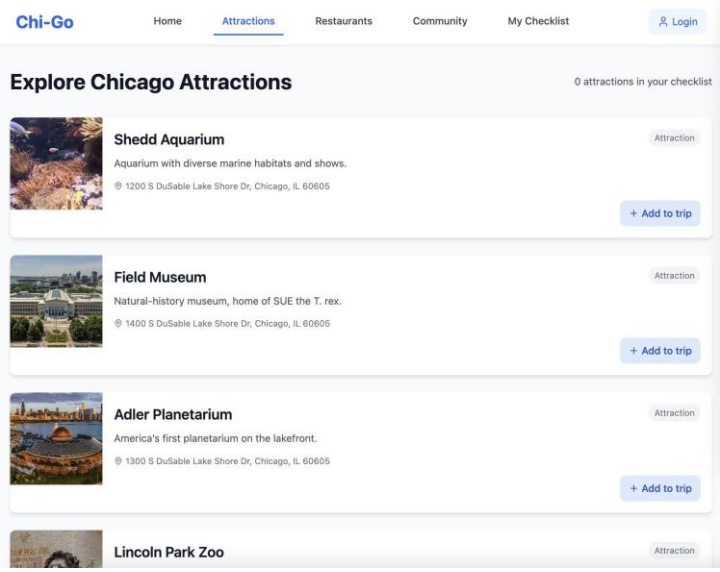
4.2.1 User-End Features

- Home Dashboard: This is the homepage of Chi-Go to give you an introduction to Chicago and explain how the app helps you plan and organize your trip. If you want to

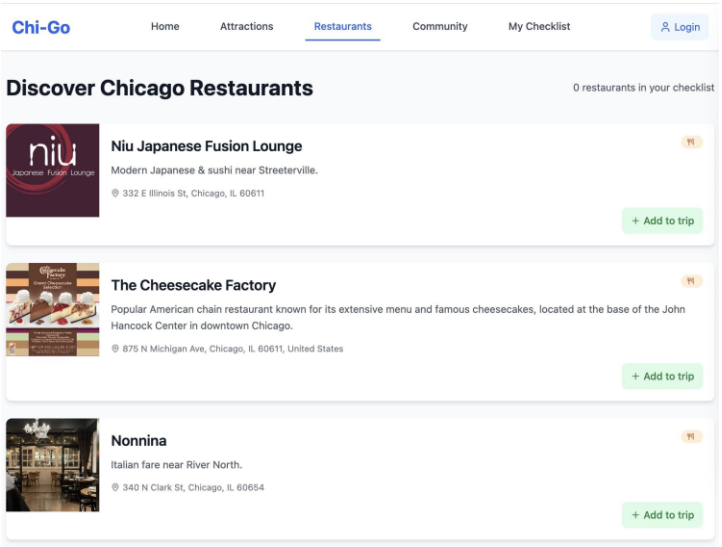
know details about CTA (public transit), official Chicago Tourism Site or events held by Chicago city, please scroll down to click the links at the bottom, our application will direct you to the related site.



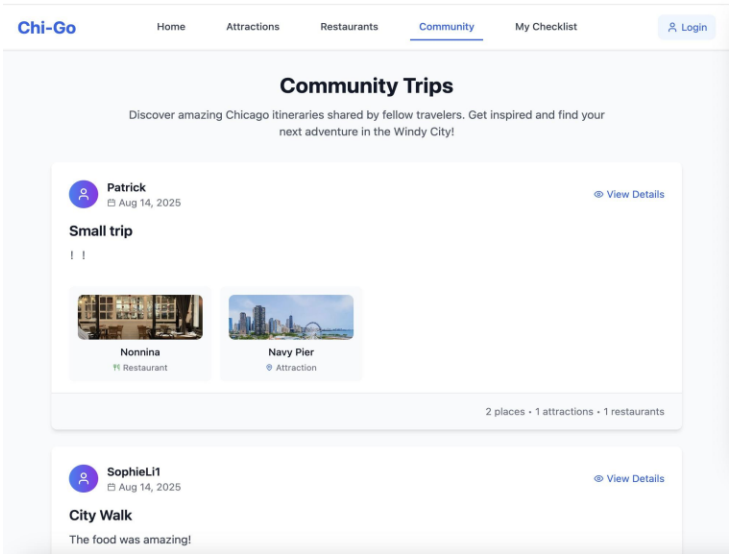
- Attractions Page: This page lets you browse popular Chicago attractions. You can add them to your personal trip checklist for easier travel planning.

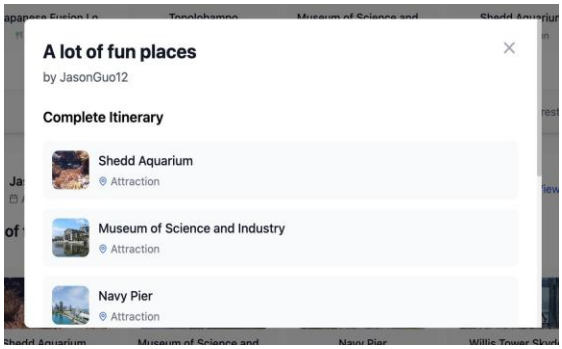


- Restaurants Page: You can explore popular Chicago restaurants and add your favorites to your trip checklist.

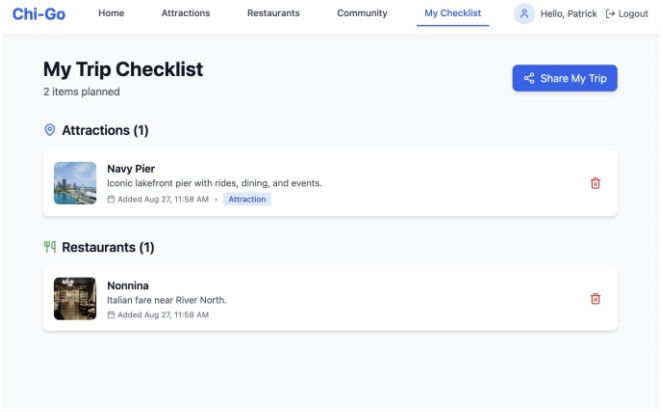


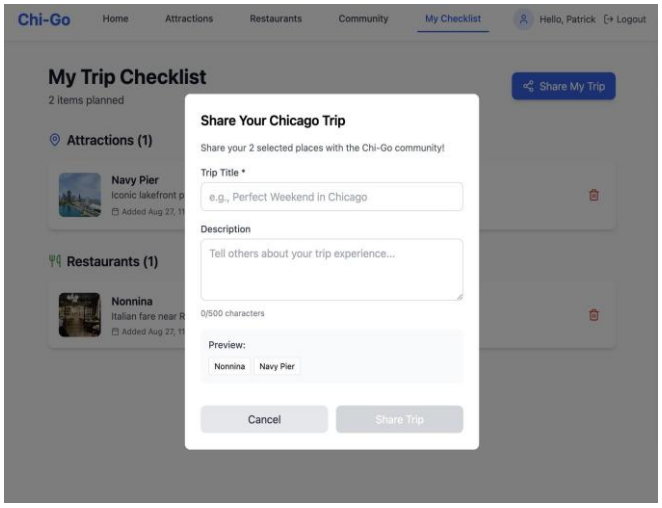
- Community Page: You can view and get inspired by trip itineraries shared by other travelers in the community, including attractions and restaurants they visited. You can click “View Details” on the top right corner of a post to know more details.



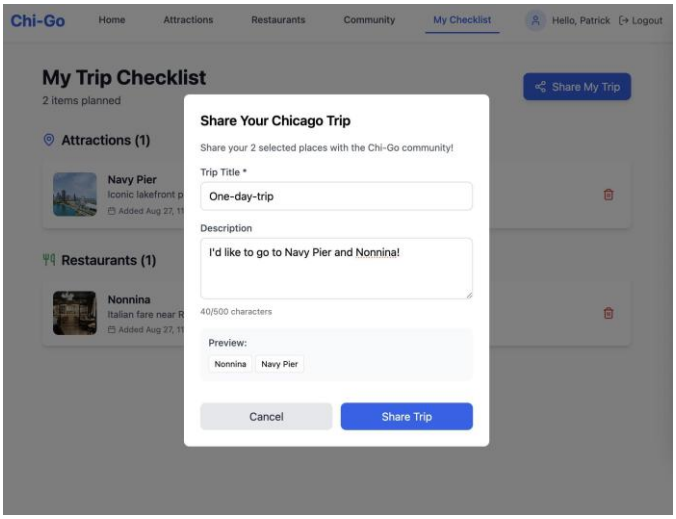


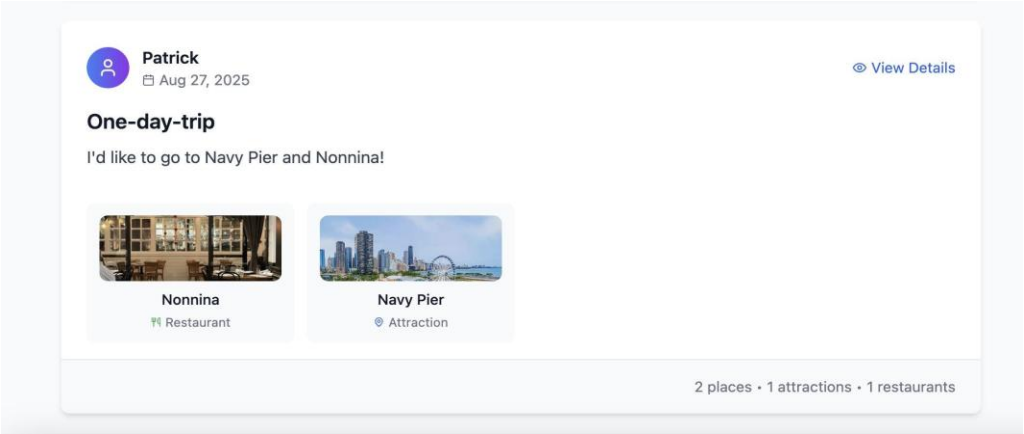
- My Checklist Page: You can see your personal trip plan in “My Checklist” page, where you can view the attractions and restaurants you’ve saved, and share your plan with other travelers in the community. Click the “Share My Trip” button (top right), fill in the trip title (compulsory) and description (optional), then click “Share Trip”.





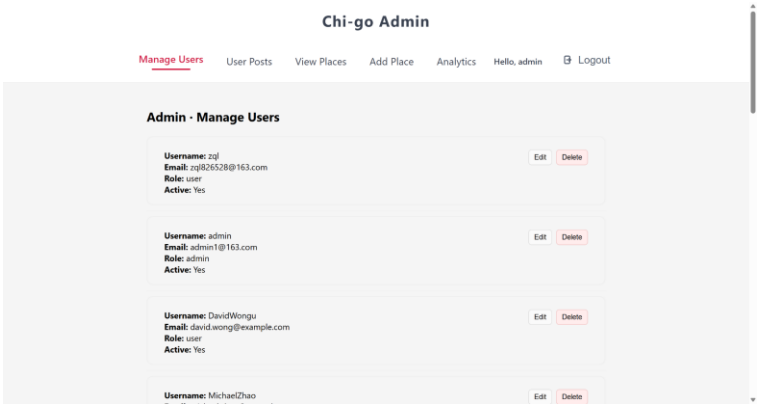
For instance, I want to share a trip titled “One-day-trip”, and I added the description “I’d like to go to Navy Pier and Nonnina!”. Then I can see this post in the Community page:



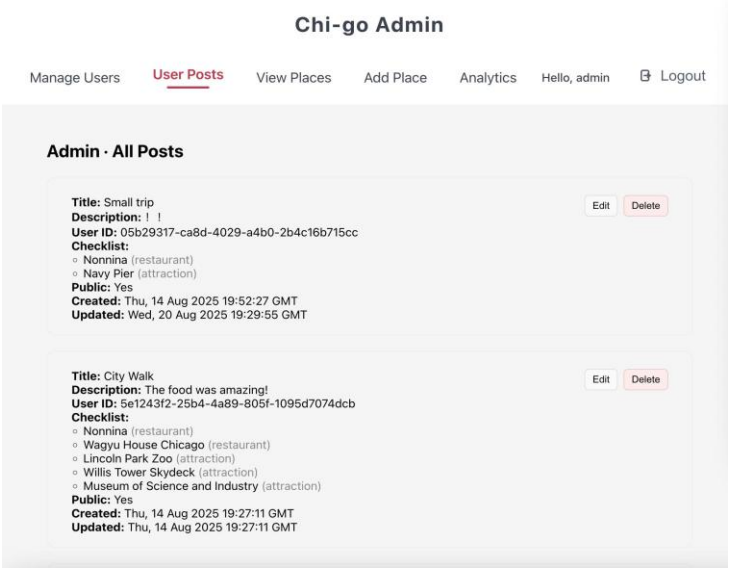


4.2.2 Admin-End Features

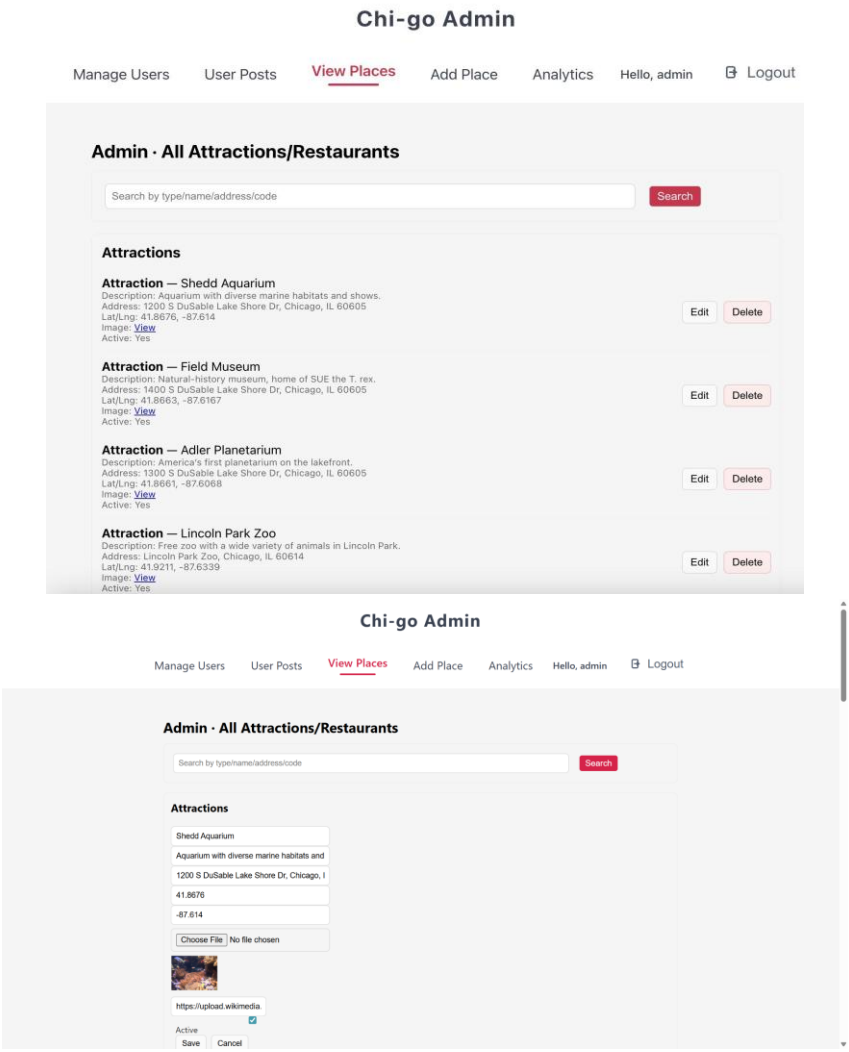
- Manage Users Page: This page allows you to manage user accounts, including viewing and/or editing user information, and deleting accounts if needed.



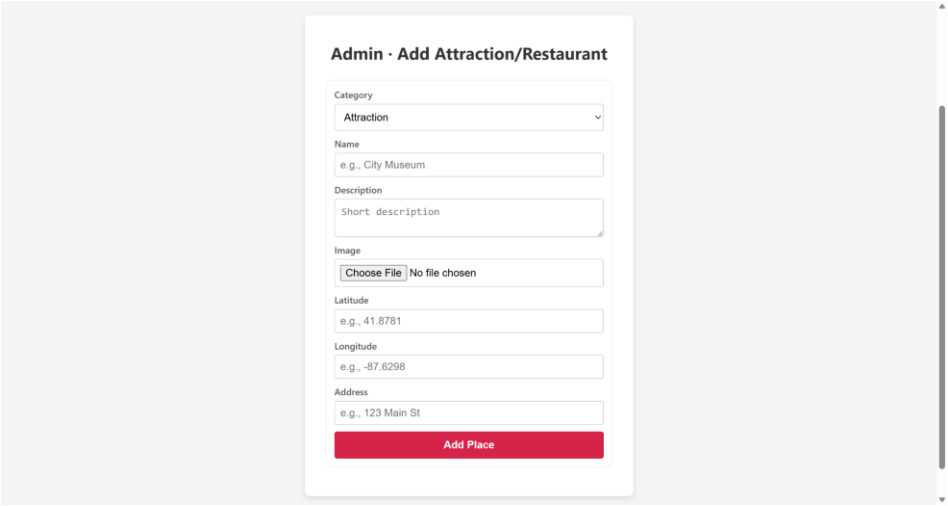
- Manage User Posts Page: You can review and manage user posts, including trip titles, descriptions, and checklists. You can edit details or delete posts by clicking “edit” or “delete” button to keep community content accurate and appropriate:



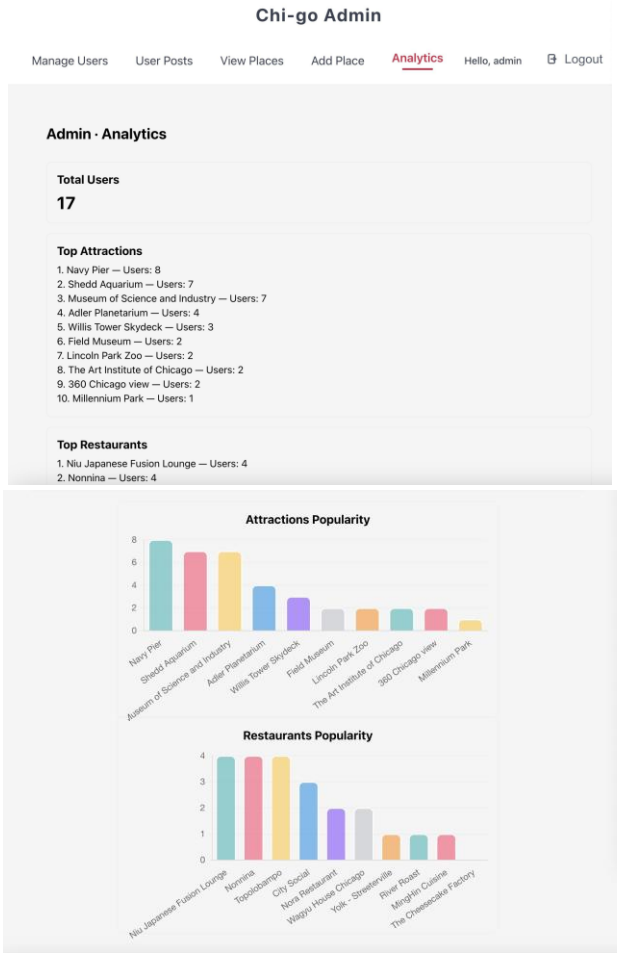
- View Places Page: You can view and manage all attractions and restaurants. Search for places using the search box by type/name/address/code to know details like descriptions and addresses. Use the “Edit” or “Delete” buttons to keep the information accurate and up to date. As for the **photo upload functionality**, when editing the information of a certain attraction/restaurant, you can choose an image file from your computer using the **“Choose File”** button. Once uploaded, the selected photo will be displayed as the attraction or restaurant’s image, making the listing more visually informative.



- Add Place Page: Add new attractions or restaurants by filling in details such as the category, name, description, location coordinates, and address. You can also choose an image file from your computer using the **“Choose File”** button. Then click the “Add Place” button.



- Analytics Page: This page provides you with analytics about user preference data, including the total number of users and the most popular attractions and restaurants. Scroll down to the bottom of this page, you can see two charts visually displaying which attractions and restaurants are most popular among users. The top chart shows the number of users who added each attraction, while the bottom chart shows the number of users who added each restaurant, helping you quickly understand trends and preferences:



4.3 Known Limitations & “Gotchas”

We have not yet implemented a “Forgot password? Find or reset your password” feature, so we recommend that you keep your password saved securely to ensure smooth login to your account.

4.4 Support Contact

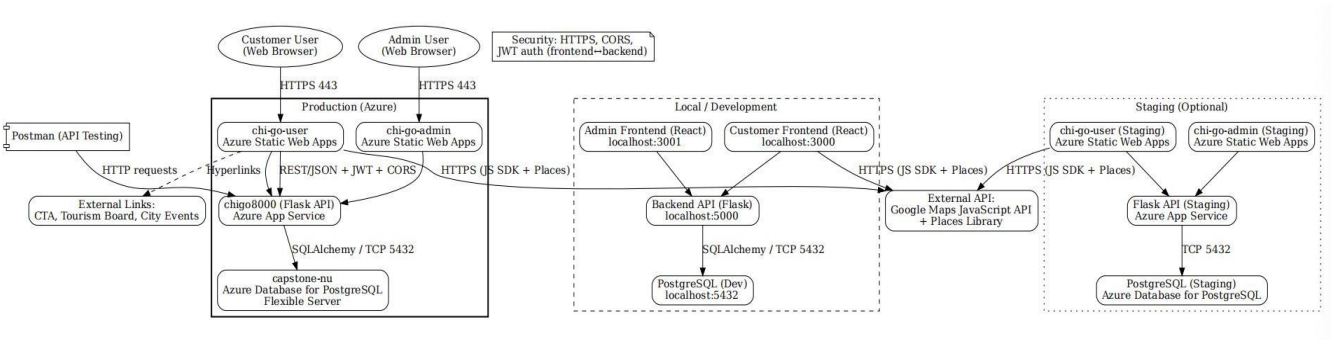
For any assistance, please reach out to the Chi-Go Support Team:

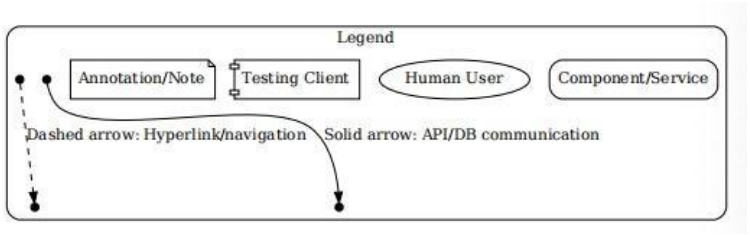
- Email: lejunchen0803@gmail.com
- Phone: (464) 204-5798

- Support Hours: Monday - Friday, 9 AM - 6 PM (CST)

When contacting support, please include your username, the issue encountered, and screenshots (if possible) to help the team resolve your problem quickly.

5. Architecture Diagram





The system has four major parts. The Customer Frontend and Admin Frontend are React apps hosted on Azure Static Web Apps; they deliver the public user experience and the internal dashboard, respectively. A Backend API built with Flask runs on Azure App Service and acts as the centralized brain for business logic and data access. Persistent data lives in Azure Database for PostgreSQL (Flexible Server). One external API is used by the customer-facing app: the Google Maps JavaScript API with Places, which provides maps and place search.

Communication is straightforward and secure. End users reach the frontends over HTTPS 443. Both frontends call the Flask API using REST/JSON over HTTPS, with JWT-based auth and CORS enabled. The API talks to PostgreSQL via TCP 5432 using SQLAlchemy. The customer frontend also makes direct HTTPS requests to Google Maps (browser → maps.googleapis.com). Postman can target the API for manual testing. Dashed lines in the diagram indicate simple hyperlinks to external sites; solid lines indicate programmatic communication.

There are three environments. In Local/Development, the apps run on localhost (React: 3000/3001; Flask: 5000; PostgreSQL: 5432). An optional Staging tier mirrors production on Azure—Static Web Apps for the frontends, App Service for the API, and Azure Database for PostgreSQL—allowing validation before release. Production uses the same topology: chi-go-user and chi-go-admin (Static Web Apps), chigo8000 (Flask API on App Service), and capstone-nu (Azure PostgreSQL). Each component's role is clear: frontends present the UI, the

API encapsulates logic and security, the database persists application data, and Google Maps enriches the UX with location intelligence.

Reference

[1] Sommerville, I. (2019). Software Engineering (10th ed.). Pearson Education.
[2] Pressman, R. S., & Maxim, B. R. (2019). Software Engineering: A Practitioner's Approach (9th ed.). McGraw-Hill Education.
[3] Fowler, M. (2018). Refactoring: Improving the Design of Existing Code (2nd ed.). Addison-Wesley Professional.
[4] Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.
[5] Richardson, C. (2018). Microservices Patterns: With examples in Java. Manning Publications.
[6] Newman, S. (2021). Building Microservices: Designing Fine-Grained Systems (2nd ed.). O'Reilly Media.
[7] Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.
[8] Burns, B. (2018). Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. O'Reilly Media.