# Blue WhatsApp Bot

Technical Documentation

Blue WhatsApp Bot Team

May 11, 2025

# Contents

# List of Figures

# List of Tables

# Part I

# English Documentation

# Chapter 1

# Introduction

## 1.1 Overview

The Blue WhatsApp Bot is an automated reservation system that integrates with WhatsApp to provide a seamless booking experience for users. The system allows users to make reservations for trips through WhatsApp messages, with features including real-time capacity management, automated responses, and a comprehensive backoffice interface.

## 1.2 Purpose

The primary purpose of this documentation is to provide a comprehensive guide to the Blue WhatsApp Bot system, including its architecture, components, and implementation details. This documentation serves as a reference for developers, system administrators, and other stakeholders involved in the maintenance and development of the system.

## 1.3 System Features

The Blue WhatsApp Bot includes the following key features:

- WhatsApp-based reservation system
- Real-time trip capacity management
- Automated conversation handling
- Backoffice interface for reservation management
- Daily capacity reset automation
- Log management system
- Multi-language support
- Error handling and logging

## 1.4 Technical Stack

The system is built using the following technologies:

- **Backend Framework:** .NET Core
- **Database:** SQL Server
- **ORM:** Entity Framework Core
- **WhatsApp Integration:** WhatsApp Business API
- **Background Jobs:** Quartz.NET
- **Logging:** Custom logging implementation
- **API Documentation:** Swagger/OpenAPI

## 1.5 Documentation Structure

This documentation is organized into the following sections:

- **System Architecture:** Overview of the system's architecture and design patterns
- **Core Components:** Detailed description of the core system components
- **API Layer:** Documentation of the API endpoints and controllers
- **Database Design:** Database schema and relationships
- **Reservation System:** Detailed explanation of the reservation workflow
- **WhatsApp Integration:** Integration details with WhatsApp
- **Background Jobs:** Description of automated tasks and scheduling
- **Deployment:** Deployment instructions and requirements

## 1.6 Getting Started

To get started with the Blue WhatsApp Bot system:

1. Review the system requirements in the Deployment chapter
2. Set up the development environment
3. Configure the WhatsApp Business API integration
4. Set up the database and run migrations
5. Configure the background jobs
6. Deploy the application

For detailed instructions on each of these steps, please refer to the respective chapters in this documentation.

# Chapter 2

# System Architecture

## 2.1 Architecture Overview

The Blue WhatsApp Bot follows a clean architecture pattern, separating the system into distinct layers with clear responsibilities and dependencies. The architecture is designed to be maintainable, testable, and scalable.

## 2.2 Architecture Layers

### 2.2.1 Core Layer

The innermost layer containing the business logic and domain models.

- **Models:** Core domain entities and value objects
- **Interfaces:** Repository and service contracts
- **Enums:** Domain-specific enumerations
- **Utils:** Utility classes and helpers

### 2.2.2 Boundaries Layer

The layer responsible for implementing the interfaces defined in the Core layer.

- **Persistence:** Database context and configurations
- **Repositories:** Implementation of repository interfaces
- **Models:** Database entity models
- **Configurations:** Entity Framework configurations

### 2.2.3 API Layer

The outermost layer handling HTTP requests and external integrations.

- **Controllers:** API endpoints and request handling
- **Hubs:** SignalR hubs for real-time communication
- **Extensions:** Application configuration and middleware
- **Views:** Backoffice interface views

## 2.3 Design Patterns

### 2.3.1 Repository Pattern

Used for data access abstraction:

```
public interface IReservationRepository
{
    Task<IEnumerable<CoreReservation>> GetAllReservationsAsync();
    Task<CoreReservation?> GetReservationByIdAsync(int id);
    Task<CoreReservation> CreateReservationAsync(CoreReservation
   reservation);
    // ... other methods
}
```

### 2.3.2 Dependency Injection

Used throughout the application for loose coupling:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddScoped<IReservationRepository,
   ReservationRepository>();
        services.AddScoped<ITripRepository, TripRepository>();
        // ... other registrations
    }
}
```

### 2.3.3 SignalR Hub Pattern

Used for real-time communication:

```
public class ReservationsHub : Hub
{
    private readonly IReservationRepository _reservationRepository;

    public ReservationsHub(IReservationRepository reservationRepository
   )
    {
        _reservationRepository = reservationRepository;
    }

    // ... hub methods
}
```

## 2.4 Data Flow

### 2.4.1 Reservation Process

1. WhatsApp message received
2. Message processed by conversation handler
3. Reservation created in database
4. Real-time update sent to backoffice
5. Confirmation sent to user

### 2.4.2 Capacity Management

1. Trip capacity checked before reservation
2. Capacity updated in real-time
3. Daily reset at 23:50
4. Notifications sent when capacity changes

## 2.5 Cross-Cutting Concerns

### 2.5.1 Logging

Implemented using a custom logging interface:

```
public interface IAppLogger
{
    void LogInfo(string message);
    void LogError(string message);
    void LogWarning(string message);
}
```

### 2.5.2 Error Handling

Global error handling middleware:

```
public class ErrorHandlingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly IAppLogger _logger;

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error: {ex.Message}");
            // ... error handling logic
        }
    }
}
```

## 2.6 Security

### 2.6.1 Authentication

- JWT-based authentication
- Role-based authorization
- Secure password hashing

### 2.6.2 Data Protection

- HTTPS enforcement
- Input validation
- SQL injection prevention
- XSS protection

## 2.7 Performance Considerations

### 2.7.1 Caching

- In-memory caching for frequently accessed data
- Response caching for static content
- Distributed caching support

### 2.7.2 Database Optimization

- Indexed queries
- Efficient joins
- Connection pooling
- Query optimization

## 2.8 Scalability

### 2.8.1 Horizontal Scaling

- Stateless design
- Load balancing support
- Distributed caching
- Database sharding capability

### 2.8.2 Vertical Scaling

- Resource optimization
- Connection pooling
- Memory management
- Thread pool configuration

# Chapter 3

# Core Components

## 3.1 Core Components Overview

The Blue WhatsApp Bot system consists of several core components that work together to provide the reservation functionality. Each component has a specific responsibility and follows the single responsibility principle.

## 3.2 Reservation System

### 3.2.1 Reservation Model

The core reservation model that represents a booking in the system:

```
public class CoreReservation
{
    public int Id { get; set; }
    public string CustomerName { get; set; }
    public string PhoneNumber { get; set; }
    public int HotelId { get; set; }
    public DateTime ReservationDate { get; set; }
    public int ScheduleId { get; set; }
    public int TripId { get; set; }
    public string? AdditionalDetails { get; set; }
    public DateTime CreatedTime { get; set; }
    public DateTime ModifiedTime { get; set; }
    public bool IsActive { get; set; }
}
```

### 3.2.2 Reservation Repository

Interface defining the data access operations for reservations:

```
public interface IReservationRepository
{
    Task<IEnumerable<CoreReservation>> GetAllReservationsAsync();
    Task<CoreReservation?> GetReservationByIdAsync(int id);
    Task<CoreReservation> CreateReservationAsync(CoreReservation
   reservation);
    Task UpdateReservationAsync(CoreReservation reservation);
    Task DeleteReservationAsync(int id);
    Task<IEnumerable<CoreReservation>> GetReservationsByTripId(int
   tripId);
}
```

## 3.3 Trip Management

### 3.3.1 Trip Model

The core trip model representing a scheduled trip:

```
public class CoreTrip
{
    public int Id { get; set; }
    public string TripName { get; set; }
    public int RouteId { get; set; }
    public bool IsActiveForToday { get; set; }
    public int MaxCapacity { get; set; }
    public int CurrentReservations { get; set; }
    public int RemainingCapacity => MaxCapacity - CurrentReservations;
    public DateTime CreatedTime { get; set; }
    public DateTime ModifiedTime { get; set; }
    public bool IsActive { get; set; }
}
```

### 3.3.2 Trip Repository

Interface for trip-related data operations:

```
public interface ITripRepository
{
    Task<IEnumerable<CoreTrip>> GetAllTripsAsync();
    Task<CoreTrip?> GetTripByIdAsync(int id);
    Task<CoreTrip> CreateTripAsync(CoreTrip trip);
    Task UpdateTripAsync(CoreTrip trip);
    Task DeleteTripAsync(int id);
    Task UpdateTripStatusAsync(int id, bool isActiveForToday);
}
```

## 3.4 Conversation Handling

### 3.4.1 Conversation State

Model representing the current state of a user conversation:

```
public class ConversationState
{
    public string UserNumber { get; set; }
    public string CurrentStep { get; set; }
    public Dictionary<string, string> UserData { get; set; }
    public DateTime LastInteraction { get; set; }
}
```

### 3.4.2 Message Creator

Interface for creating different types of messages:

```
public interface IMessageCreator
{
    CoreMessageToSend CreateWelcomeMessage(string userNumber);
    CoreMessageToSend CreateDatePromptMessage(string userNumber);
    CoreMessageToSend CreateSchedulePromptMessage(string userNumber);
    CoreMessageToSend CreateHotelPromptMessage(string userNumber);
    CoreMessageToSend CreateConfirmationMessage(string userNumber,
   CoreReservation reservation);
    CoreMessageToSend CreateTripFullMessage(string userNumber);
}
```

## 3.5 Logging System

### 3.5.1 Logger Interface

Interface for system-wide logging:

```
public interface IAppLogger
{
    void LogInfo(string message);
    void LogError(string message);
    void LogWarning(string message);
    void LogDebug(string message);
}
```

### 3.5.2 Logger Implementation

Custom implementation of the logging interface:

```
public class AppLogger : IAppLogger
{
    private readonly string _logDirectory;

    public AppLogger(string logDirectory)
    {
        _logDirectory = logDirectory;
    }

    public void LogInfo(string message)
    {
        WriteLog("INFO", message);
    }

    public void LogError(string message)
    {
        WriteLog("ERROR", message);
    }

    // ... other logging methods
}
```

## 3.6 Background Jobs

### 3.6.1 Job Interface

Base interface for all background jobs:

```
public interface IJob
{
    Task Execute(IJobExecutionContext context);
}
```

### 3.6.2 Job Implementations

Implementation of specific background jobs:

- `DeleteOldFoldersJob`: Manages log file cleanup
- `ResetTripCapacitiesJob`: Resets trip capacities daily

## 3.7 SignalR Integration

### 3.7.1 Reservations Hub

Real-time communication hub for reservations:

```
public class ReservationsHub : Hub
{
    private readonly IReservationRepository _reservationRepository;
    private readonly ITripRepository _tripRepository;

    public async Task GetReservations()
    {
        var reservations = await _reservationRepository.
    GetAllReservationsAsync();
        await Clients.Caller.SendAsync("ReceiveReservations",
    reservations);
    }

    // ... other hub methods
}
```

## 3.8 Error Handling

### 3.8.1 Error Types

Custom exception types for different error scenarios:

```
public class ReservationException : Exception
{
    public ReservationException(string message) : base(message) { }
}

public class TripCapacityException : Exception
```

```
{
    public TripCapacityException(string message) : base(message) { }
}
```

### 3.8.2 Error Handler

Global error handling component:

```
public class ErrorHandler
{
    private readonly IAppLogger _logger;

    public ErrorHandler(IAppLogger logger)
    {
        _logger = logger;
    }

    public async Task HandleError(Exception ex)
    {
        _logger.LogError($"Error: {ex.Message}");
        // ... error handling logic
    }
}
```

## 3.9 Configuration

### 3.9.1 App Settings

Configuration model for application settings:

```
public class AppSettings
{
    public string WhatsAppApiKey { get; set; }
    public string WhatsAppApiUrl { get; set; }
    public string ConnectionString { get; set; }
    public int DefaultTripCapacity { get; set; }
    public string LogDirectory { get; set; }
}
```

### 3.9.2 Configuration Provider

Service for accessing configuration values:

```
public interface IConfigurationProvider
{
    AppSettings GetAppSettings();
    string GetConnectionString();
    string GetWhatsAppApiKey();
}
```

# Chapter 4

# API Layer

## 4.1 API Overview

The Blue WhatsApp Bot API provides endpoints for managing reservations, trips, and the backoffice interface. The API follows RESTful principles and uses SignalR for real-time updates.

## 4.2 Controllers

### 4.2.1 Reservations Controller

Handles reservation-related HTTP requests:

```
[ApiController]
[Route("api/[controller]")]
public class ReservationsController : ControllerBase
{
    private readonly IReservationRepository _reservationRepository;
    private readonly IAppLogger _logger;

    [HttpGet]
    public async Task<ActionResult<IEnumerable<CoreReservation>>>
   GetReservations()
    {
        var reservations = await _reservationRepository.
   GetAllReservationsAsync();
        return Ok(reservations);
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<CoreReservation>> GetReservation(int
    id)
    {
        var reservation = await _reservationRepository.
   GetReservationByIdAsync(id);
        if (reservation == null)
            return NotFound();
        return Ok(reservation);
    }

    [HttpPost]
    public async Task<ActionResult<CoreReservation>> CreateReservation(CoreReservation reservation)
    {
        try
        {
            var created = await _reservationRepository.CreateReservationAsync(reservation);
            return CreatedAtAction(nameof(GetReservation), new { id = created.Id }, created);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error creating reservation: {ex.Message}");
            return BadRequest(ex.Message);
        }
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateReservation(int id, CoreReservation reservation)
    {
        if (id != reservation.Id)
            return BadRequest();

        try
        {
            await _reservationRepository.UpdateReservationAsync(reservation);
            return NoContent();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error updating reservation: {ex.Message}");
            return BadRequest(ex.Message);
        }
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteReservation(int id)
    {
        try
        {
            await _reservationRepository.DeleteReservationAsync(id);
            return NoContent();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error deleting reservation: {ex.Message}");
            return BadRequest(ex.Message);
        }
    }
}
```

### 4.2.2 Trips Controller

Manages trip-related operations:

```
[ApiController]
[Route("api/[controller]")]
public class TripsController : ControllerBase
{
    private readonly ITripRepository _tripRepository;
    private readonly IAppLogger _logger;

    [HttpGet]
    public async Task<ActionResult<IEnumerable<CoreTrip>>> GetTrips()
    {
        var trips = await _tripRepository.GetAllTripsAsync();
        return Ok(trips);
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<CoreTrip>> GetTrip(int id)
    {
        var trip = await _tripRepository.GetTripByIdAsync(id);
        if (trip == null)
            return NotFound();
        return Ok(trip);
    }

    [HttpPost]
    public async Task<ActionResult<CoreTrip>> CreateTrip(CoreTrip trip)
    {
        try
        {
            var created = await _tripRepository.CreateTripAsync(trip);
            return CreatedAtAction(nameof(GetTrip), new { id = created.
   Id }, created);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error creating trip: {ex.Message}");
            return BadRequest(ex.Message);
        }
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateTrip(int id, CoreTrip trip)
    {
        if (id != trip.Id)
            return BadRequest();

        try
        {
            await _tripRepository.UpdateTripAsync(trip);
            return NoContent();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error updating trip: {ex.Message}");
            return BadRequest(ex.Message);
        }
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteTrip(int id)
    {
        try
        {
            await _tripRepository.DeleteTripAsync(id);
            return NoContent();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error deleting trip: {ex.Message}");
            return BadRequest(ex.Message);
        }
    }

    [HttpPut("{id}/status")]
    public async Task<IActionResult> UpdateTripStatus(int id, [FromBody
   ] bool isActiveForToday)
    {
        try
        {
            await _tripRepository.UpdateTripStatusAsync(id,
   isActiveForToday);
            return NoContent();
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error updating trip status: {ex.Message
   }");
            return BadRequest(ex.Message);
        }
    }
}
```

## 4.3 SignalR Hubs

### 4.3.1 Reservations Hub

Real-time communication for reservations:

```
public class ReservationsHub : Hub
{
    private readonly IReservationRepository _reservationRepository;
    private readonly ITripRepository _tripRepository;
    private readonly IAppLogger _logger;

    public ReservationsHub(
        IReservationRepository reservationRepository,
        ITripRepository tripRepository,
        IAppLogger logger)
    {
        _reservationRepository = reservationRepository;
        _tripRepository = tripRepository;
        _logger = logger;
    }

    public async Task GetReservations()
    {
        try
        {
            var reservations = await _reservationRepository.GetAllReservationsAsync();
            await Clients.Caller.SendAsync("ReceiveReservations", reservations);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error getting reservations: {ex.Message}");
            throw;
        }
    }

    public async Task GetTrips()
    {
        try
        {
            var trips = await _tripRepository.GetAllTripsAsync();
            await Clients.Caller.SendAsync("ReceiveTrips", trips);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error getting trips: {ex.Message}");
            throw;
        }
    }

    public async Task SaveReservation(CoreReservation reservation)
    {
        try
        {
            var trip = await _tripRepository.GetTripByIdAsync(reservation.TripId);
            if (trip == null)
                throw new Exception("Trip not found");

            if (trip.CurrentReservations >= trip.MaxCapacity)
                throw new TripCapacityException("Trip is at full capacity");

            var created = await _reservationRepository.CreateReservationAsync(reservation);
            await Clients.All.SendAsync("ReservationCreated", created);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error saving reservation: {ex.Message}");
            throw;
        }
    }
}
```

## 4.4 API Endpoints

### 4.4.1 Reservations

- `GET /api/reservations` - Get all reservations
- `GET /api/reservations/id` - Get reservation by ID
- `POST /api/reservations` - Create new reservation
- `PUT /api/reservations/id` - Update reservation
- `DELETE /api/reservations/id` - Delete reservation

### 4.4.2 Trips

- `GET /api/trips` - Get all trips
- `GET /api/trips/id` - Get trip by ID
- `POST /api/trips` - Create new trip
- `PUT /api/trips/id` - Update trip
- `DELETE /api/trips/id` - Delete trip
- `PUT /api/trips/id/status` - Update trip status

## 4.5 SignalR Methods

### 4.5.1 Reservations Hub

- `GetReservations()` - Get all reservations
- `GetTrips()` - Get all trips
- `SaveReservation(reservation)` - Save new reservation

## 4.6 Error Handling

### 4.6.1 HTTP Status Codes

- `200 OK` - Successful operation
- `201 Created` - Resource created
- `204 No Content` - Operation successful, no content
- `400 Bad Request` - Invalid request
- `404 Not Found` - Resource not found
- `500 Internal Server Error` - Server error

### 4.6.2 Error Responses

Error responses follow this format:

```
{
    "error": {
        "message": "Error message",
        "details": "Detailed error information"
    }
}
```

## 4.7 Authentication

### 4.7.1 JWT Authentication

The API uses JWT tokens for authentication:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
    TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = Configuration["Jwt:Issuer"],
            ValidAudience = Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(Configuration["Jwt:Key"]))
        };
    });
```

## 4.8 API Documentation

### 4.8.1 Swagger Integration

The API includes Swagger documentation:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "Blue WhatsApp Bot API",
        Version = "v1",
        Description = "API for the Blue WhatsApp Bot system"
    });
});
```

### 4.8.2 API Versioning

The API supports versioning:

```
services.AddApiVersioning(options =>
{
    options.DefaultApiVersion = new ApiVersion(1, 0);
    options.AssumeDefaultVersionWhenUnspecified = true;
    options.ReportApiVersions = true;
});
```

# Chapter 5

# Database Design

# Chapter 6

# Reservation System

## 6.1 Reservation System Overview

The Blue WhatsApp Bot's reservation system is designed to handle trip bookings through WhatsApp conversations, providing a seamless experience for users while maintaining accurate capacity management.

## 6.2 Reservation Process

### 6.2.1 Conversation Flow

The reservation process follows a structured conversation flow:

1. **Welcome Message**: Initial greeting and introduction to the booking system
2. **Date Selection**: User selects the desired trip date
3. **Schedule Selection**: User chooses from available time slots
4. **Hotel Selection**: User selects their pickup location
5. **Confirmation**: System confirms the booking details
6. **Completion**: Reservation is created and confirmation is sent

### 6.2.2 State Management

The conversation state is managed through the `ConversationState` class:

```
public class ConversationState
{
    public string UserNumber { get; set; }
    public string CurrentStep { get; set; }
    public Dictionary<string, string> UserData { get; set; }
    public DateTime LastInteraction { get; set; }
}
```

## 6.3 Capacity Management

### 6.3.1 Trip Capacity

Each trip has a maximum capacity that is managed through the `CoreTrip` model:

```
public class CoreTrip
{
    public int Id { get; set; }
    public string TripName { get; set; }
    public int MaxCapacity { get; set; }
    public int CurrentReservations { get; set; }
    public int RemainingCapacity => MaxCapacity - CurrentReservations;
    // ... other properties
}
```

### 6.3.2 Capacity Reset

Trip capacities are automatically reset daily at 23:50 through the `ResetTripCapacitiesJob`:

```
public class ResetTripCapacitiesJob : IJob
{
    private readonly ITripRepository _tripRepository;
    private readonly IAppLogger _logger;

    public async Task Execute(IJobExecutionContext context)
    {
        try
        {
            var trips = await _tripRepository.GetAllTripsAsync();
            foreach (var trip in trips)
            {
                trip.CurrentReservations = 0;
                await _tripRepository.UpdateTripAsync(trip);
            }
            _logger.LogInfo("Trip capacities reset successfully");
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error resetting trip capacities: {ex.
    Message}");
            throw;
        }
    }
}
```

## 6.4 Reservation Validation

### 6.4.1 Capacity Check

Before creating a reservation, the system validates the trip capacity:

```
public async Task<CoreReservation> CreateReservationAsync(
    CoreReservation reservation)
{
    var trip = await _tripRepository.GetTripByIdAsync(reservation.
   TripId);
    if (trip == null)
        throw new ReservationException("Trip not found");

    if (trip.CurrentReservations >= trip.MaxCapacity)
        throw new TripCapacityException("Trip is at full capacity");

    // ... create reservation logic
}
```

### 6.4.2 Date Validation

The system ensures reservations are made for valid dates:

```
private bool IsValidReservationDate(DateTime date)
{
    return date.Date >= DateTime.Today &&
           date.Date <= DateTime.Today.AddDays(30);
}
```

## 6.5 Real-time Updates

### 6.5.1 SignalR Integration

Reservations are broadcast in real-time through SignalR:

```
public async Task SaveReservation(CoreReservation reservation)
{
    var created = await _reservationRepository.CreateReservationAsync(
   reservation);
    await Clients.All.SendAsync("ReservationCreated", created);
}
```

### 6.5.2 Capacity Updates

Trip capacity changes are immediately reflected:

```
public async Task UpdateTripCapacity(int tripId)
{
    var trip = await _tripRepository.GetTripByIdAsync(tripId);
    trip.CurrentReservations++;
    await _tripRepository.UpdateTripAsync(trip);
    await Clients.All.SendAsync("TripCapacityUpdated", trip);
}
```

## 6.6 Error Handling

### 6.6.1 Reservation Exceptions

Custom exceptions for reservation-related errors:

```
public class ReservationException : Exception
{
    public ReservationException(string message) : base(message) { }
}

public class TripCapacityException : Exception
{
    public TripCapacityException(string message) : base(message) { }
}
```

### 6.6.2 Error Recovery

The system includes mechanisms for handling failed reservations:

```
public async Task HandleFailedReservation(CoreReservation reservation,
    Exception ex)
{
    _logger.LogError($"Reservation failed: {ex.Message}");
    // Notify user of failure
    await _messageCreator.CreateErrorMessage(reservation.PhoneNumber);
    // Attempt to rollback any partial changes
    await RollbackReservationChanges(reservation);
}
```

## 6.7 Reservation Queries

### 6.7.1 Common Queries

The system provides various methods to query reservations:

```
public interface IReservationRepository
{
    Task<IEnumerable<CoreReservation>> GetReservationsByDate(DateTime
    date);
    Task<IEnumerable<CoreReservation>> GetReservationsByTripId(int
    tripId);
    Task<IEnumerable<CoreReservation>> GetReservationsByPhoneNumber(
    string phoneNumber);
}
```

### 6.7.2 Reporting

Reservation data can be aggregated for reporting:

```
public async Task<ReservationReport> GenerateDailyReport(DateTime date)
{
    var reservations = await _reservationRepository.
    GetReservationsByDate(date);
    return new ReservationReport
    {
        Date = date,
        TotalReservations = reservations.Count(),
        ReservationsByTrip = reservations.GroupBy(r => r.TripId)
            .ToDictionary(g => g.Key, g => g.Count())
    };
}
```

## 6.8 Best Practices

### 6.8.1 Reservation Management

- Always validate capacity before creating reservations
- Use transactions for atomic operations
- Implement proper error handling and recovery
- Maintain audit trails of all changes
- Send confirmations for all successful operations

### 6.8.2 Performance Considerations

- Use appropriate indexes for frequent queries
- Implement caching for static data
- Optimize database queries
- Monitor capacity thresholds
- Implement rate limiting for API endpoints

# Chapter 7

# WhatsApp Integration

## 7.1 WhatsApp Integration Overview

The Blue WhatsApp Bot integrates with the WhatsApp Business API to handle automated conversations and manage reservations. The integration provides a seamless experience for users while maintaining robust error handling and logging.

## 7.2 Message Handling

### 7.2.1 Message Types

The system handles various types of WhatsApp messages:

```
public enum MessageType
{
    Text,
    Image,
    Document,
    Location,
    Contact
}

public class WhatsAppMessage
{
    public string From { get; set; }
    public string To { get; set; }
    public MessageType Type { get; set; }
    public string Content { get; set; }
    public DateTime Timestamp { get; set; }
}
```

### 7.2.2 Message Processing

Messages are processed through a dedicated handler:

```
public class WhatsAppMessageHandler
{
    private readonly IMessageCreator _messageCreator;
    private readonly IAppLogger _logger;
    private readonly Dictionary<string, ConversationState>
   _conversationStates;

    public async Task HandleMessage(WhatsAppMessage message)
    {
        try
        {
            var state = GetOrCreateConversationState(message.From);
            await ProcessMessage(message, state);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Error processing message: {ex.Message}")
   ;
            await SendErrorMessage(message.From);
        }
    }

    private async Task ProcessMessage(WhatsAppMessage message,
   ConversationState state)
    {
        switch (state.CurrentStep)
        {
            case "Welcome":
                await HandleWelcomeStep(message, state);
                break;
            case "DateSelection":
                await HandleDateSelection(message, state);
                break;
            // ... other steps
        }
    }
}
```

## 7.3 Conversation Flow

### 7.3.1 Step Management

The conversation flow is managed through distinct steps:

```
public class ConversationStep
{
    public string Name { get; set; }
    public string Message { get; set; }
    public string NextStep { get; set; }
    public Func<string, bool> Validation { get; set; }
}

public class ConversationManager
{
    private readonly Dictionary<string, ConversationStep> _steps;

    public ConversationManager()
    {
        _steps = new Dictionary<string, ConversationStep>
        {
            {
                "Welcome",
                new ConversationStep
                {
                    Name = "Welcome",
                    Message = "Welcome to Blue WhatsApp Bot! Let's book your trip.",
                    NextStep = "DateSelection",
                    Validation = _ => true
                }
            },
            // ... other steps
        };
    }
}
```

### 7.3.2 Message Templates

Predefined message templates for different scenarios:

```
public interface IMessageCreator
{
    CoreMessageToSend CreateWelcomeMessage(string userNumber);
    CoreMessageToSend CreateDatePromptMessage(string userNumber);
    CoreMessageToSend CreateSchedulePromptMessage(string userNumber);
    CoreMessageToSend CreateHotelPromptMessage(string userNumber);
    CoreMessageToSend CreateConfirmationMessage(string userNumber, CoreReservation reservation);
    CoreMessageToSend CreateTripFullMessage(string userNumber);
    CoreMessageToSend CreateErrorMessage(string userNumber);
}
```

## 7.4 API Integration

### 7.4.1 WhatsApp API Client

Integration with WhatsApp Business API:

```
public class WhatsAppApiClient
{
    private readonly HttpClient _httpClient;
    private readonly string _apiKey;
    private readonly string _apiUrl;

    public async Task SendMessage(string to, string message)
    {
        var request = new
        {
            messaging_product = "whatsapp",
            to = to,
            type = "text",
            text = new { body = message }
        };

        var response = await _httpClient.PostAsJsonAsync($"{_apiUrl}/messages", request);
        response.EnsureSuccessStatusCode();
    }

    public async Task SendTemplate(string to, string templateName, Dictionary<string, string> parameters)
    {
        var request = new
        {
            messaging_product = "whatsapp",
            to = to,
            type = "template",
            template = new
            {
                name = templateName,
                language = new { code = "en" },
                components = parameters.Select(p => new
                {
                    type = "body",
                    parameters = new[] { new { type = "text", text = p.Value } }
                })
            }
        };

        var response = await _httpClient.PostAsJsonAsync($"{_apiUrl}/messages", request);
        response.EnsureSuccessStatusCode();
    }
}
```

## 7.5 Error Handling

### 7.5.1 API Errors

Handling WhatsApp API errors:

```
public class WhatsAppApiException : Exception
{
    public int StatusCode { get; }
    public string ErrorCode { get; }

    public WhatsAppApiException(string message, int statusCode, string errorCode)
        : base(message)
    {
        StatusCode = statusCode;
        ErrorCode = errorCode;
    }
}

public async Task HandleApiError(HttpResponseMessage response)
{
    var error = await response.Content.ReadFromJsonAsync<WhatsAppApiError>();
    throw new WhatsAppApiException(
        error.Message,
        (int)response.StatusCode,
        error.ErrorCode
    );
}
```

### 7.5.2 Retry Logic

Implementing retry mechanism for failed API calls:

```
public class WhatsAppMessageSender
{
    private readonly IWhatsAppApiClient _apiClient;
    private readonly IAppLogger _logger;
    private readonly IAsyncPolicy<HttpResponseMessage> _retryPolicy;

    public WhatsAppMessageSender(IWhatsAppApiClient apiClient,
   IAppLogger logger)
    {
        _apiClient = apiClient;
        _logger = logger;
        _retryPolicy = Policy<HttpResponseMessage>
            .Handle<HttpRequestException>()
            .Or<WhatsAppApiException>()
            .WaitAndRetryAsync(3, retryAttempt =>
                TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)));
    }

    public async Task SendMessageWithRetry(string to, string message)
    {
        await _retryPolicy.ExecuteAsync(async () =>
        {
            try
            {
                return await _apiClient.SendMessage(to, message);
            }
            catch (Exception ex)
            {
                _logger.LogError($"Error sending message: {ex.Message
   }");
                throw;
            }
        });
    }
}
```

## 7.6 Message Templates

### 7.6.1 Template Management

Managing WhatsApp message templates:

```
public class TemplateManager
{
    private readonly IWhatsAppApiClient _apiClient;
    private readonly IAppLogger _logger;

    public async Task CreateTemplate(string name, string content,
   string category)
    {
        var request = new
        {
            name = name,
            language = "en",
            category = category,
            components = new[]
            {
                new
                {
                    type = "BODY",
                    text = content
                }
            }
        };

        await _apiClient.CreateTemplate(request);
    }

    public async Task<List<Template>> GetTemplates()
    {
        return await _apiClient.GetTemplates();
    }
}
```

## 7.7 Best Practices

### 7.7.1 Message Handling

- Implement proper error handling for API calls
- Use retry mechanisms for transient failures
- Log all message interactions
- Validate message content before sending
- Implement rate limiting to prevent abuse

### 7.7.2 Conversation Management

- Maintain conversation state
- Implement timeout handling
- Provide clear user instructions
- Handle unexpected user inputs
- Implement fallback options

### 7.7.3 Security

- Secure API key storage
- Validate message sources
- Implement message encryption
- Monitor for suspicious activity
- Regular security audits

# Chapter 8

# Background Jobs

## 8.1 Overview

The Blue WhatsApp Bot system includes several background jobs that automate routine tasks. These jobs are implemented using Quartz.NET, a powerful job scheduling library for .NET applications.

## 8.2 Job Configuration

All background jobs are configured in the `CronJobExtensions.cs` file, which sets up the job scheduling and dependencies. The jobs are registered during application startup using the `ConfigureCronSchedulerJobs` extension method.

## 8.3 Available Jobs

### 8.3.1 Delete Old Folders Job

This job manages the system's log files by removing old log directories.

- **Schedule:** Weekly on Monday at 00:00
- **Purpose:** Cleans up old log files to prevent disk space issues
- **Implementation:** `DeleteOldFoldersJob` class

#### Implementation Details

```
public class DeleteOldFoldersJob : IJob
{
    private readonly IAppLogger _logger;

    public DeleteOldFoldersJob(IAppLogger logger)
    {
        _logger = logger;
    }

    Task IJob.Execute(IJobExecutionContext context)
    {
        string rootDirectory = Path.Combine(Directory.GetCurrentDirectory(), "logs");
        // ... implementation details ...
    }
}
```

### 8.3.2 Reset Trip Capacities Job

This job resets the capacity of all trips to their default value at the end of each day.

- **Schedule:** Daily at 23:50
- **Purpose:** Ensures trip capacities are reset for the next day
- **Implementation:** `ResetTripCapacitiesJob` class

#### Implementation Details

```
public class ResetTripCapacitiesJob : IJob
{
    private readonly IAppLogger _logger;
    private readonly ITripRepository _tripRepository;

    public ResetTripCapacitiesJob(IAppLogger logger, ITripRepository tripRepository)
    {
        _logger = logger;
        _tripRepository = tripRepository;
    }

    async Task IJob.Execute(IJobExecutionContext context)
    {
        try
        {
            _logger.LogInfo("Starting daily trip capacity reset job");
            var trips = await _tripRepository.GetAllTripsAsync();
            foreach (var trip in trips)
            {
                trip.MaxCapacity = 30;
                await _tripRepository.UpdateTripAsync(trip);
            }
            _logger.LogInfo($"Successfully reset capacities for {trips.Count()} trips");
        }
        catch (Exception ex)
        {
            _logger.LogError($"Failed to reset trip capacities. Error: {ex.Message}");
        }
    }
}
```

## 8.4 Job Configuration

The jobs are configured in the application startup using the following code:

```
internal static void ConfigureCronSchedulerJobs(this
    WebApplicationBuilder builder)
{
     builder.Services.AddQuartz(q =>
     {
         q.UseMicrosoftDependencyInjectionJobFactory();

         // Configure DeleteOldFoldersJob
         var deleteFoldersJobKey = new JobKey("DeleteOldFoldersJob");
         q.AddJob<DeleteOldFoldersJob>(opts => opts.WithIdentity(
    deleteFoldersJobKey));

         q.AddTrigger(opts => opts
             .ForJob(deleteFoldersJobKey)
             .WithIdentity("DeleteOldFoldersTrigger")
             .WithSchedule(CronScheduleBuilder.
    WeeklyOnDayAndHourAndMinute(DayOfWeek.Monday, 0, 0))
         );

         // Configure ResetTripCapacitiesJob
         var resetCapacitiesJobKey = new JobKey("ResetTripCapacitiesJob
    ");
         q.AddJob<ResetTripCapacitiesJob>(opts => opts.WithIdentity(
    resetCapacitiesJobKey));

         q.AddTrigger(opts => opts
             .ForJob(resetCapacitiesJobKey)
             .WithIdentity("ResetTripCapacitiesTrigger")
             .WithSchedule(CronScheduleBuilder.DailyAtHourAndMinute(23,
    50))
         );
     });

     builder.Services.AddQuartzHostedService(q => q.
    WaitForJobsToComplete = true);
}
```

## 8.5 Error Handling

All background jobs include comprehensive error handling and logging:

- Each job operation is wrapped in try-catch blocks
- Errors are logged using the `IAppLogger` interface
- Failed jobs are logged with detailed error messages
- The system continues to run even if a job fails

## 8.6 Monitoring

The system provides monitoring capabilities for background jobs:

- Job execution status is logged
- Success/failure counts are tracked
- Detailed error messages are available in the logs
- Job execution times are recorded

## 8.7 Adding New Jobs

To add a new background job to the system:

1. Create a new class implementing `IJob`
2. Implement the `Execute` method
3. Add the job configuration in `ConfigureCronSchedulerJobs`
4. Register any required dependencies
5. Configure the job schedule

## 8.8 Best Practices

When working with background jobs in the system:

- Always include proper error handling
- Log all important operations
- Use dependency injection for required services
- Keep job execution times reasonable
- Consider the impact on system resources
- Test jobs thoroughly before deployment

# Chapter 9

# How to Use

## 9.1 How to Use Overview

This chapter provides a comprehensive guide on how to use the Blue WhatsApp Bot system, from initial setup to managing reservations and monitoring the system.

## 9.2 Getting Started

### 9.2.1 Prerequisites

Before using the system, ensure you have:

- A valid WhatsApp Business API account
- SQL Server 2019 or later installed
- .NET 6.0 or later runtime
- IIS 10.0 or later configured
- Valid SSL certificate for HTTPS

### 9.2.2 Initial Setup

1. Configure the database connection string in `appsettings.json`
2. Set up WhatsApp API credentials
3. Configure logging directory
4. Set up JWT authentication keys

## 9.3 Managing Trips

### 9.3.1 Creating a Trip

To create a new trip:

1. Access the Trips management interface
2. Click "Create New Trip"
3. Fill in the required information:
    - Trip Name
    - Route ID
    - Maximum Capacity
4. Select available schedules
5. Save the trip

### 9.3.2 Managing Trip Schedules

To manage trip schedules:

1. Select the trip from the list
2. Click "Manage Schedules"
3. Add or remove available time slots
4. Set schedule status (active/inactive)

## 9.4 Reservation Process

### 9.4.1 WhatsApp Conversation Flow

The system follows this conversation flow:

1. User initiates conversation
2. System sends welcome message
3. User selects date
4. User selects time slot
5. User selects hotel
6. System confirms booking
7. Reservation is created

### 9.4.2 Example Conversation

```
Bot: Welcome to Blue WhatsApp Bot! Let's book your trip.
User: Hi
Bot: Please select a date for your trip (DD/MM/YYYY)
User: 15/05/2024
Bot: Available time slots:
     1. 08:00 AM
     2. 10:00 AM
     3. 02:00 PM
     Please select a time slot (1-3)
User: 2
Bot: Please select your pickup hotel:
     1. Hotel A
     2. Hotel B
     3. Hotel C
User: 1
Bot: Confirming your booking:
     Date: 15/05/2024
     Time: 10:00 AM
     Hotel: Hotel A
     Is this correct? (Yes/No)
User: Yes
Bot: Your reservation has been confirmed!
     Reservation ID: #12345
     Thank you for choosing Blue WhatsApp Bot!
```

## 9.5 Admin Interface

### 9.5.1 Viewing Reservations

To view reservations:

1. Access the Reservations dashboard
2. Use filters to search by:
   - Date
   - Hotel
   - Trip
   - Customer name
   - Phone number
3. View reservation details
4. Export data if needed

### 9.5.2 Managing Capacity

To manage trip capacity:

1. Access the Trips dashboard
2. Select the trip
3. View current capacity
4. Adjust maximum capacity if needed
5. Monitor remaining slots

## 9.6 Monitoring and Maintenance

### 9.6.1 System Health

Monitor system health through:

- Health check endpoint (/health)
- Log files in configured directory
- Database performance metrics
- API usage statistics

### 9.6.2 Backup and Recovery

Regular maintenance tasks:

1. Daily database backups
2. Log file rotation
3. Application file backups
4. SSL certificate renewal

## 9.7 Troubleshooting

### 9.7.1 Common Issues

- **WhatsApp API Connection Issues**
  - Check API credentials
  - Verify internet connection
  - Check API rate limits
- **Database Connection Issues**

  - Verify connection string
  - Check SQL Server status
  - Verify user permissions

- **Capacity Management Issues**
  - Check trip status
  - Verify schedule availability
  - Review reservation counts

### 9.7.2 Error Messages

Common error messages and solutions:

- "Trip is at full capacity" - Trip has reached maximum bookings
- "Invalid date format" - Use DD/MM/YYYY format
- "Schedule not available" - Selected time slot is not active
- "Hotel not found" - Selected hotel is not in the system

## 9.8 Best Practices

### 9.8.1 System Usage

- Regular monitoring of system health
- Daily backup verification
- Capacity planning
- User communication templates
- Error log review

### 9.8.2 Performance Optimization

- Regular database maintenance
- Log file cleanup
- Cache management
- API rate limit monitoring
- Resource usage tracking

# Chapter 10

# Deployment

## 10.1 Deployment Overview

The Blue WhatsApp Bot is designed to be deployed in a production environment with high availability and scalability. This chapter outlines the deployment process, requirements, and best practices.

## 10.2 System Requirements

### 10.2.1 Hardware Requirements

- **CPU**: Minimum 2 cores, recommended 4 cores
- **RAM**: Minimum 4GB, recommended 8GB
- **Storage**: Minimum 50GB SSD
- **Network**: Stable internet connection with minimum 10Mbps

### 10.2.2 Software Requirements

- **Operating System**: Windows Server 2019 or later
- **.NET Runtime**: .NET 6.0 or later
- **Database**: SQL Server 2019 or later
- **IIS**: Version 10.0 or later
- **SSL Certificate**: Valid SSL certificate for HTTPS

## 10.3 Deployment Process

### 10.3.1 Database Setup

1. Install SQL Server
2. Create new database

3. Run database migrations
4. Configure database user and permissions

```sql
-- Create database
CREATE DATABASE BlueWhatsAppBot;

-- Create database user
CREATE LOGIN BlueWhatsAppBotUser WITH PASSWORD = 'StrongPassword123!';
USE BlueWhatsAppBot;
CREATE USER BlueWhatsAppBotUser FOR LOGIN BlueWhatsAppBotUser;

-- Grant permissions
GRANT SELECT, INSERT, UPDATE, DELETE ON SCHEMA::dbo TO
    BlueWhatsAppBotUser;
```

### 10.3.2 Application Deployment

1. Publish application
2. Configure IIS
3. Set up application pool
4. Configure SSL
5. Set up logging

```xml
<!-- web.config -->
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="
    AspNetCoreModuleV2" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet" arguments=".\BlueWhatsapp.Api.dll"
    stdoutLogEnabled="true" stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

## 10.4 Configuration

### 10.4.1 Environment Variables

Required environment variables:

```json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=BlueWhatsAppBot;
    User Id=BlueWhatsAppBotUser;Password=StrongPassword123!;"
  },
  "WhatsAppApi": {
    "ApiKey": "your-api-key",
    "ApiUrl": "https://graph.facebook.com/v17.0/your-phone-number-id"
  },
  "Jwt": {
    "Key": "your-secret-key",
    "Issuer": "BlueWhatsAppBot",
    "Audience": "BlueWhatsAppBotUsers"
  },
  "Logging": {
    "LogDirectory": "C:\\Logs\\BlueWhatsAppBot"
  }
}
```

### 10.4.2 IIS Configuration

IIS application pool settings:

```
<applicationPools>
  <add name="BlueWhatsAppBot"
       autoStart="true"
       managedRuntimeVersion="v4.0"
       managedPipelineMode="Integrated">
    <processModel identityType="ApplicationPoolIdentity" />
  </add>
</applicationPools>
```

## 10.5 Monitoring

### 10.5.1 Logging Configuration

Configure logging in `appsettings.json`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    },
    "File": {
      "Path": "C:\\Logs\\BlueWhatsAppBot\\app.log",
      "Append": true,
      "MinLevel": "Information",
      "FileSizeLimitBytes": 10485760,
      "MaxRollingFiles": 10
    }
  }
}
```

### 10.5.2 Health Checks

Configure health checks in `Startup.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
```

```
        .AddSqlServer(Configuration["ConnectionStrings:
    DefaultConnection"])
        .AddCheck<WhatsAppApiHealthCheck>("WhatsApp API")
        .AddCheck<DiskStorageHealthCheck>("Disk Storage");
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseHealthChecks("/health", new HealthCheckOptions
    {
        ResponseWriter = async (context, report) =>
        {
            context.Response.ContentType = "application/json";
            var response = new
            {
                status = report.Status.ToString(),
                checks = report.Entries.Select(x => new
                {
                    name = x.Key,
                    status = x.Value.Status.ToString(),
                    description = x.Value.Description
                })
            };
            await context.Response.WriteAsJsonAsync(response);
        }
    });
}
```

## 10.6 Backup and Recovery

### 10.6.1 Database Backup

Configure SQL Server backup jobs:

```
-- Create backup job
USE msdb;
GO

EXEC dbo.sp_add_job
    @job_name = N'BlueWhatsAppBot_DailyBackup',
    @description = N'Daily backup of BlueWhatsAppBot database',
    @category_name = N'Database Maintenance',
    @owner_login_name = N'sa';
GO

EXEC dbo.sp_add_jobstep
    @job_name = N'BlueWhatsAppBot_DailyBackup',
    @step_name = N'Backup Database',
    @subsystem = N'TSQL',
    @command = N'BACKUP DATABASE BlueWhatsAppBot TO DISK = ''C:\Backups
    \BlueWhatsAppBot_$(ESCAPE_SQUOTE(DATE)).bak'' WITH INIT';
GO

EXEC dbo.sp_add_schedule
    @job_name = N'BlueWhatsAppBot_DailyBackup',
    @name = N'Daily Schedule',
    @freq_type = 4,
    @freq_interval = 1,
    @active_start_time = 010000;
GO
```

### 10.6.2 Application Backup

Configure application backup:

```
# Backup script
$backupPath = "C:\Backups\BlueWhatsAppBot"
$appPath = "C:\inetpub\wwwroot\BlueWhatsAppBot"
$date = Get-Date -Format "yyyy-MM-dd"

# Create backup directory
New-Item -ItemType Directory -Force -Path "$backupPath\$date"

# Backup application files
Copy-Item -Path "$appPath\*" -Destination "$backupPath\$date" -Recurse

# Backup logs
Copy-Item -Path "C:\Logs\BlueWhatsAppBot\*" -Destination "$backupPath\
    $date\Logs" -Recurse
```

## 10.7 Security

### 10.7.1 SSL Configuration

Configure SSL in IIS:

```
<system.webServer>
  <security>
    <requestFiltering>
      <requestLimits maxAllowedContentLength="30000000" />
    </requestFiltering>
  </security>
  <rewrite>
    <rules>
      <rule name="HTTP to HTTPS" stopProcessing="true">
        <match url="(.*)" />
        <conditions>
          <add input="{HTTPS}" pattern="^OFF$" />
        </conditions>
        <action type="Redirect" url="https://{HTTP_HOST}/{R:1}"
    redirectType="Permanent" />
      </rule>
    </rules>
  </rewrite>
</system.webServer>
```

### 10.7.2 Firewall Configuration

Configure Windows Firewall:

```
# Allow HTTP traffic
New-NetFirewallRule -DisplayName "Allow HTTP" -Direction Inbound -
    Protocol TCP -LocalPort 80 -Action Allow

# Allow HTTPS traffic
New-NetFirewallRule -DisplayName "Allow HTTPS" -Direction Inbound -
    Protocol TCP -LocalPort 443 -Action Allow
```

## 10.8 Maintenance

### 10.8.1 Log Rotation

Configure log rotation in `appsettings.json`:

```
{
  "Serilog": {
    "WriteTo": [
      {
        "Name": "File",
        "Args": {
          "path": "C:\\Logs\\BlueWhatsAppBot\\log-.txt",
          "rollingInterval": "Day",
          "retainedFileCountLimit": 31
        }
      }
    ]
  }
}
```

### 10.8.2 Database Maintenance

Configure database maintenance jobs:

```
-- Create maintenance job
USE msdb;
GO

EXEC dbo.sp_add_job
    @job_name = N'BlueWhatsAppBot_Maintenance',
    @description = N'Weekly database maintenance',
    @category_name = N'Database Maintenance',
    @owner_login_name = N'sa';
GO

-- Add maintenance steps
EXEC dbo.sp_add_jobstep
    @job_name = N'BlueWhatsAppBot_Maintenance',
    @step_name = N'Update Statistics',
    @subsystem = N'TSQL',
    @command = N'UPDATE STATISTICS BlueWhatsAppBot WITH FULLSCAN';
GO

EXEC dbo.sp_add_jobstep
    @job_name = N'BlueWhatsAppBot_Maintenance',
    @step_name = N'Rebuild Indexes',
    @subsystem = N'TSQL',
    @command = N'ALTER INDEX ALL ON BlueWhatsAppBot.dbo.Reservations
    REBUILD';
GO
```

## 10.9 Best Practices

### 10.9.1 Deployment

- Use deployment scripts for consistency
- Implement blue-green deployment
- Test deployment in staging environment
- Maintain deployment documentation
- Use version control for configuration

### 10.9.2 Monitoring

- Set up application monitoring
- Configure alert thresholds
- Monitor system resources
- Track API usage
- Monitor error rates

### 10.9.3 Security

- Regular security updates
- SSL certificate renewal
- Access control review
- Security audit logging
- Regular vulnerability scanning

# Part II

# Documentación en Español

# Chapter 11

# Introducción

# Chapter 12

# Arquitectura del Sistema

# Chapter 13

# Componentes Principales

# Chapter 14

# Capa de API

# Chapter 15

# Diseño de Base de Datos

# Chapter 16

# Sistema de Reservas

# Chapter 17

# Integración con WhatsApp

# Chapter 18

# Trabajos en Segundo Plano

# Chapter 19

# Guía de Uso

## 19.1 Descripción General de Uso

Este capítulo proporciona una guía completa sobre cómo utilizar el sistema Blue WhatsApp Bot, desde la configuración inicial hasta la gestión de reservas y el monitoreo del sistema.

## 19.2 Comenzando

### 19.2.1 Requisitos Previos

Antes de usar el sistema, asegúrese de tener:

- Una cuenta válida de WhatsApp Business API
- SQL Server 2019 o posterior instalado
- Runtime .NET 6.0 o posterior
- IIS 10.0 o posterior configurado
- Certificado SSL válido para HTTPS

### 19.2.2 Configuración Inicial

1. Configure la cadena de conexión a la base de datos en `appsettings.json`
2. Configure las credenciales de la API de WhatsApp
3. Configure el directorio de registro
4. Configure las claves de autenticación JWT

## 19.3 Gestión de Viajes

### 19.3.1 Creación de un Viaje

Para crear un nuevo viaje:

1. Acceda a la interfaz de gestión de viajes
2. Haga clic en "Crear Nuevo Viaje"
3. Complete la información requerida:
    - Nombre del Viaje
    - ID de Ruta
    - Capacidad Máxima
4. Seleccione los horarios disponibles
5. Guarde el viaje

### 19.3.2 Gestión de Horarios de Viaje

Para gestionar los horarios de viaje:

1. Seleccione el viaje de la lista
2. Haga clic en "Gestionar Horarios"
3. Agregue o elimine horarios disponibles
4. Establezca el estado del horario (activo/inactivo)

## 19.4 Proceso de Reserva

### 19.4.1 Flujo de Conversación de WhatsApp

El sistema sigue este flujo de conversación:

1. El usuario inicia la conversación
2. El sistema envía un mensaje de bienvenida
3. El usuario selecciona la fecha
4. El usuario selecciona el horario
5. El usuario selecciona el hotel
6. El sistema confirma la reserva
7. Se crea la reserva

### 19.4.2 Ejemplo de Conversación

```
Bot: ¡Bienvenido a Blue WhatsApp Bot! Vamos a reservar tu viaje.
Usuario: Hola
Bot: Por favor, selecciona una fecha para tu viaje (DD/MM/AAAA)
Usuario: 15/05/2024
Bot: Horarios disponibles:
     1. 08:00 AM
     2. 10:00 AM
     3. 02:00 PM
     Por favor, selecciona un horario (1-3)
Usuario: 2
Bot: Por favor, selecciona tu hotel de recogida:
     1. Hotel A
     2. Hotel B
     3. Hotel C
Usuario: 1
Bot: Confirmando tu reserva:
     Fecha: 15/05/2024
     Hora: 10:00 AM
     Hotel: Hotel A
     ¿Es correcto? (Sí/No)
Usuario: Sí
Bot: ¡Tu reserva ha sido confirmada!
     ID de Reserva: #12345
     ¡Gracias por elegir Blue WhatsApp Bot!
```

## 19.5 Interfaz de Administración

### 19.5.1 Visualización de Reservas

Para ver las reservas:

1. Acceda al panel de Reservas
2. Use filtros para buscar por:
   - Fecha
   - Hotel
   - Viaje
   - Nombre del cliente
   - Número de teléfono
3. Vea los detalles de la reserva
4. Exporte los datos si es necesario

### 19.5.2 Gestión de Capacidad

Para gestionar la capacidad de viajes:

1. Acceda al panel de Viajes
2. Seleccione el viaje
3. Vea la capacidad actual
4. Ajuste la capacidad máxima si es necesario
5. Monitoree los espacios restantes

## 19.6 Monitoreo y Mantenimiento

### 19.6.1 Estado del Sistema

Monitoree el estado del sistema a través de:

- Punto final de verificación de estado (/health)
- Archivos de registro en el directorio configurado
- Métricas de rendimiento de la base de datos
- Estadísticas de uso de la API

### 19.6.2 Respaldo y Recuperación

Tareas de mantenimiento regulares:

1. Respaldos diarios de la base de datos
2. Rotación de archivos de registro
3. Respaldos de archivos de la aplicación
4. Renovación del certificado SSL

## 19.7 Solución de Problemas

### 19.7.1 Problemas Comunes

- **Problemas de Conexión con la API de WhatsApp**
  - Verifique las credenciales de la API
  - Compruebe la conexión a internet
  - Verifique los límites de tasa de la API
- **Problemas de Conexión con la Base de Datos**

  - Verifique la cadena de conexión
  - Compruebe el estado de SQL Server
  - Verifique los permisos de usuario

- **Problemas de Gestión de Capacidad**
  - Verifique el estado del viaje
  - Compruebe la disponibilidad del horario
  - Revise los conteos de reservas

### 19.7.2 Mensajes de Error

Mensajes de error comunes y soluciones:

- "El viaje está al máximo de capacidad" - El viaje ha alcanzado el máximo de reservas
- "Formato de fecha inválido" - Use el formato DD/MM/AAAA
- "Horario no disponible" - El horario seleccionado no está activo
- "Hotel no encontrado" - El hotel seleccionado no está en el sistema

## 19.8 Mejores Prácticas

### 19.8.1 Uso del Sistema

- Monitoreo regular del estado del sistema
- Verificación diaria de respaldos
- Planificación de capacidad
- Plantillas de comunicación con usuarios
- Revisión de registros de errores

### 19.8.2 Optimización de Rendimiento

- Mantenimiento regular de la base de datos
- Limpieza de archivos de registro
- Gestión de caché
- Monitoreo de límites de tasa de la API
- Seguimiento del uso de recursos

# Chapter 20

# Implementación

# Part III

# Apéndices

# Appendix A

# API Reference

# Appendix B

# Database Schema

## B.1 Database Overview

The Blue WhatsApp Bot uses SQL Server as its database management system. The database is designed using Entity Framework Core, which provides a clean and maintainable way to interact with the database.

## B.2 Entity Relationships

The database consists of several interconnected entities that work together to manage trips, reservations, and schedules. Figure B.1 shows the entity relationship diagram for the database schema.

Figure B.1: Entity Relationship Diagram

## B.3 Core Entities

### B.3.1 Trip

The Trip entity represents a scheduled trip in the system.

```sql
CREATE TABLE Trips (
    Id INT PRIMARY KEY IDENTITY(1,1),
    TripName NVARCHAR(100) NOT NULL,
    RouteId INT NOT NULL,
    IsActiveForToday BIT NOT NULL DEFAULT 1,
    MaxCapacity INT NOT NULL DEFAULT 30,
    CreatedTime DATETIME2 NOT NULL DEFAULT GETDATE(),
    ModifiedTime DATETIME2 NOT NULL DEFAULT GETDATE(),
    IsActive BIT NOT NULL DEFAULT 1
)
```

### B.3.2 Reservation

The Reservation entity stores booking information.

```sql
CREATE TABLE Reservations (
    Id INT PRIMARY KEY IDENTITY(1,1),
    CustomerName NVARCHAR(100) NOT NULL,
    PhoneNumber NVARCHAR(20) NOT NULL,
    HotelId INT NOT NULL,
    ReservationDate DATE NOT NULL,
    ScheduleId INT NOT NULL,
    TripId INT NOT NULL,
    AdditionalDetails NVARCHAR(MAX),
    CreatedTime DATETIME2 NOT NULL DEFAULT GETDATE(),
    ModifiedTime DATETIME2 NOT NULL DEFAULT GETDATE(),
    IsActive BIT NOT NULL DEFAULT 1
)
```

### B.3.3 Hotel

The Hotel entity represents available hotels.

```sql
CREATE TABLE Hotels (
    Id INT PRIMARY KEY IDENTITY(1,1),
    HotelName NVARCHAR(100) NOT NULL,
    CreatedTime DATETIME2 NOT NULL DEFAULT GETDATE(),
    ModifiedTime DATETIME2 NOT NULL DEFAULT GETDATE(),
    IsActive BIT NOT NULL DEFAULT 1
)
```

### B.3.4 Schedule

The Schedule entity defines available time slots.

```sql
CREATE TABLE Schedules (
    Id INT PRIMARY KEY IDENTITY(1,1),
    TimeSlot TIME NOT NULL,
    CreatedTime DATETIME2 NOT NULL DEFAULT GETDATE(),
    ModifiedTime DATETIME2 NOT NULL DEFAULT GETDATE(),
    IsActive BIT NOT NULL DEFAULT 1
)
```

## B.4 Relationship Tables

### B.4.1 TripSchedule

Links trips with their available schedules.

```sql
CREATE TABLE TripSchedules (
    TripId INT NOT NULL,
    ScheduleId INT NOT NULL,
    PRIMARY KEY (TripId, ScheduleId),
    FOREIGN KEY (TripId) REFERENCES Trips(Id),
    FOREIGN KEY (ScheduleId) REFERENCES Schedules(Id)
)
```

## B.5 Indexes

The following indexes are created to optimize query performance:

```sql
-- Reservations table indexes
CREATE INDEX IX_Reservations_ReservationDate ON Reservations(
    ReservationDate);
CREATE INDEX IX_Reservations_HotelId ON Reservations(HotelId);
CREATE INDEX IX_Reservations_TripId ON Reservations(TripId);

-- Trips table indexes
CREATE INDEX IX_Trips_RouteId ON Trips(RouteId);
CREATE INDEX IX_Trips_IsActiveForToday ON Trips(IsActiveForToday);

-- TripSchedules table indexes
CREATE INDEX IX_TripSchedules_ScheduleId ON TripSchedules(ScheduleId);
```

## B.6 Constraints

The database includes the following constraints:

- Primary keys on all tables
- Foreign key relationships between related tables
- NOT NULL constraints on required fields
- Default values for created/modified timestamps
- Default value of 30 for Trip.MaxCapacity

## B.7 Data Types

The system uses the following data types:

- `INT` for IDs and numeric values
- `NVARCHAR` for text fields
- `DATE` for date-only values
- `TIME` for time-only values
- `DATETIME2` for timestamp fields
- `BIT` for boolean values

## B.8 Soft Delete

All main entities implement soft delete functionality:

- `IsActive` bit field
- Default value of 1 (true)
- Used to mark records as deleted without removing them

## B.9 Audit Fields

All main entities include audit fields:

- `CreatedTime`: When the record was created
- `ModifiedTime`: When the record was last modified
- Both fields use `DATETIME2` data type
- Default value of `GETDATE()`

# Appendix C

# Referencia de API

# Appendix D

# Esquema de Base de Datos