MODULE 3: REAL-TIME SERVICES

WHERE IS MY BUS? - Real-Time Communication & Notifications

Project: Where Is My Bus? - Smart Bus Tracking System
Developer: Arpit Anand (23BCS12710)
Module: 3 of 6 - Real-Time Services
Technology Stack: WebSocket, STOMP, Firebase Cloud Messaging, Redis
Status: Planning & Documentation Phase

1. OVERVIEW

Module 3 implements **real-time communication features** that make Where Is My Bus? truly responsive. This module provides:

- **Real-Time Location Updates** - Live bus tracking on map
- **WebSocket Communication** - Instant messaging between passengers and drivers
- **Push Notifications** - Alerts for bus arrivals, delays, and updates
- **Live Status Updates** - Trip status changes, check-ins, and announcements
- **Real-Time Analytics** - Live dashboard updates for admins

2. TECHNOLOGY STACK

2.1 Core Technologies

Technology	Purpose
Spring WebSocket	WebSocket server implementation
STOMP Protocol	Messaging protocol over WebSocket
SockJS	WebSocket fallback for older browsers
Redis	In-memory data store for session management
Firebase Cloud Messaging (FCM)	Push notifications to mobile devices
Spring Cache	Caching for frequently accessed data

2.2 Architecture Pattern

```
Frontend (React)
    ↓ WebSocket Connection
WebSocket Server (Spring Boot)
    ↓ Pub/Sub
```

```
Redis (Message Broker)
    ↓ Push Notifications
Firebase Cloud Messaging
    ↓
Mobile Devices
```

3. WEBSOCKET ARCHITECTURE

3.1 Connection Flow

```
1. User opens app/website
2. Frontend establishes WebSocket connection
3. Backend authenticates WebSocket connection (JWT)
4. Connection stored in Redis with userId
5. User subscribes to relevant channels
6. Real-time updates flow through WebSocket
7. Connection maintained until user closes app
```

3.2 WebSocket Endpoints

Connection URL:

```
ws://localhost:8080/ws
wss://api.whereismybus.com/ws (production)
```

STOMP Topics (Subscribe):

Topic	Purpose	Who Subscribes
`/topic/bus/{busId}/location`	Real-time bus location	All passengers tracking bus
`/topic/route/{routeId}/buses`	All buses on route	Passengers viewing route
`/user/queue/messages`	Private messages	Passenger/Driver
`/user/queue/notifications`	Personal notifications	All authenticated users
`/topic/admin/dashboard`	Live dashboard stats	Admin only
`/topic/trip/{tripId}/updates`	Trip status updates	Driver + Admin

STOMP Destinations (Send):

Destination	Purpose	Who Sends
`/app/bus/location/update`	Update bus location	Driver
`/app/message/send`	Send message	Passenger/Driver
`/app/trip/checkin`	Check-in at stop	Driver
`/app/trip/status`	Update trip status	Driver

4. REAL-TIME BUS TRACKING

4.1 Location Update Flow

```
Driver's phone GPS
    ↓ (every 5 seconds)
Frontend sends location
    ↓ WebSocket
Backend validates location
    ↓
Updates MongoDB
    ↓
Publishes to Redis
    ↓ WebSocket broadcast
All subscribed passengers receive update
    ↓
Map marker moves in real-time
```

4.2 Driver Sends Location Update

Frontend (Driver App) sends via WebSocket:

```
// Using STOMP.js
stompClient.send("/app/bus/location/update", {}, JSON.stringify({
  busId: "bus_789",
  location: {
    latitude: 28.7041,
    longitude: 77.1025,
    accuracy: 12,
    speed: 35,
    heading: 180
  },
  timestamp: "2025-11-14T14:00:00Z"
}));
```

Backend processes and broadcasts:

```
@MessageMapping("/bus/location/update")
@SendTo("/topic/bus/{busId}/location")
public BusLocationUpdate updateBusLocation(
    @DestinationVariable String busId,
    BusLocationUpdateRequest request
) {
    // Validate location
    // Update in database
    // Return to all subscribers
    return new BusLocationUpdate(
        busId,
        request.getLocation(),
        System.currentTimeMillis()
    );
}
```

Passengers receive update:

```
// Using STOMP.js
stompClient.subscribe("/topic/bus/bus_789/location", (message) =&gt; {
  const locationUpdate = JSON.parse(message.body);

  // Update map marker
  updateBusMarker(locationUpdate.location);

  // Recalculate ETA
  calculateETA(locationUpdate.location);
});
```

4.3 Location Update Message Format

```
{
  "busId": "bus_789",
  "busNumber": "HR26Q4321",
  "location": {
    "latitude": 28.7041,
    "longitude": 77.1025,
    "accuracy": 12,
    "speed": 35,
    "heading": 180
  },
  "currentStop": {
    "stopId": "stop_sector17",
    "stopName": "Sector 17 Plaza"
  },
  "nextStop": {
    "stopId": "stop_sector22",
    "stopName": "Sector 22 Market",
    "eta": 5
  },
  "occupancy": "MEDIUM",
  "timestamp": "2025-11-14T14:00:00Z"
}
```

5. REAL-TIME MESSAGING

5.1 One-to-One Chat (Passenger ↔ Driver)

Passenger sends message:

```
stompClient.send("/app/message/send", {}, JSON.stringify({
  receiverId: "driver_456",
  busId: "bus_789",
  content: "Will the bus reach Sector 17 in 5 minutes?",
  messageType: "TEXT"
}));
```

Backend processes and delivers:

```
@MessageMapping("/message/send")
@SendToUser("/queue/messages")
public ChatMessage sendMessage(
    ChatMessageRequest request,
    Principal principal
) {
    // Save to database
    ChatMessage message = messageService.saveMessage(request, principal.getName());

    // Send to receiver
    messagingTemplate.convertAndSendToUser(
        request.getReceiverId(),
        "/queue/messages",
        message
    );

    // Also send push notification if offline
    if (!isUserOnline(request.getReceiverId())) {
        pushNotificationService.sendMessageNotification(request.getReceiverId(), message)
    }

    return message;
}
```

Driver receives message:

```
stompClient.subscribe("/user/queue/messages", (message) =&gt; {
  const chatMessage = JSON.parse(message.body);

  // Display in chat UI
  addMessageToChat(chatMessage);

  // Play notification sound
  playNotificationSound();

  // Show badge count
  incrementUnreadCount();
});
```

5.2 Message Format

```
{
  "messageId": "MSG_20251114_001",
  "sender": {
    "userId": "passenger_123",
    "name": "Arjun Kumar",
    "role": "PASSENGER"
  },
  "receiver": {
    "userId": "driver_456",
    "name": "Rajesh Sharma",
    "role": "DRIVER"
  },
  "busId": "bus_789",
  "content": "Will the bus reach Sector 17 in 5 minutes?",
  "messageType": "TEXT",
  "status": "DELIVERED",
  "sentAt": "2025-11-14T14:05:00Z"
}
```

5.3 Read Receipts

When driver reads message:

```
stompClient.send("/app/message/read", {}, JSON.stringify({
  messageId: "MSG_20251114_001"
}));
```

Passenger receives read receipt:

```
stompClient.subscribe("/user/queue/message-receipts", (receipt) =&gt; {
  const { messageId, readAt } = JSON.parse(receipt.body);

  // Update message status to "READ"
  updateMessageStatus(messageId, "READ", readAt);
});
```

6. TRIP STATUS UPDATES

6.1 Trip Lifecycle Events

Driver starts trip:

```
stompClient.send("/app/trip/start", {}, JSON.stringify({
  busId: "bus_789",
  routeId: "route_42"
}));
```

Backend broadcasts:

```
@MessageMapping("/trip/start")
@SendTo("/topic/route/{routeId}/trips")
public TripStatusUpdate startTrip(
    @DestinationVariable String routeId,
    TripStartRequest request
) {
    Trip trip = tripService.startTrip(request);

    return new TripStatusUpdate(
        trip.getTripId(),
        "IN_PROGRESS",
        trip.getActualStartTime()
    );
}
```

All passengers tracking route receive update:

```
stompClient.subscribe("/topic/route/route_42/trips", (update) =&gt; {
  const tripUpdate = JSON.parse(update.body);

  // Show "Bus started" notification
  showNotification(`Bus ${tripUpdate.busNumber} started trip`);

  // Enable tracking
  enableBusTracking(tripUpdate.tripId);
});
```

6.2 Stop Check-In Updates

Driver checks in at stop:

```
stompClient.send("/app/trip/checkin", {}, JSON.stringify({
  tripId: "TRIP_20251114_001",
  stopId: "stop_sector17",
  location: {
    latitude: 30.7409,
    longitude: 76.7794,
    accuracy: 10
  }
}));
```

Backend validates and broadcasts:

```
@MessageMapping("/trip/checkin")
@SendTo("/topic/trip/{tripId}/updates")
public StopCheckInUpdate checkIn(
    @DestinationVariable String tripId,
    CheckInRequest request
) {
    // Validate location within 100m of stop
    validateLocationProximity(request.getLocation(), request.getStopId());

    // Record check-in
    StopCheckIn checkIn = tripService.checkInAtStop(tripId, request);

    return new StopCheckInUpdate(
        checkIn.getStopName(),
        checkIn.getActualArrival(),
        checkIn.getDelay(),
        checkIn.getNextStopEta()
    );
}
```

Passengers waiting at stop receive notification:

```
stompClient.subscribe("/topic/trip/TRIP_20251114_001/updates", (update) =&gt; {
  const checkIn = JSON.parse(update.body);

  if (checkIn.stopId === userWaitingStopId) {
    // Show arrival notification
    showNotification(`Bus arrived at ${checkIn.stopName}!`);

    // Vibrate phone
    navigator.vibrate(200);
  }
});
```

6.3 Trip Status Update Format

```
{
  "tripId": "TRIP_20251114_001",
  "busNumber": "HR26Q4321",
  "routeNumber": "42",
  "status": "IN_PROGRESS",
  "currentStop": {
    "stopId": "stop_sector17",
    "stopName": "Sector 17 Plaza",
    "arrivedAt": "2025-11-14T14:15:00Z",
    "delay": 2
  },
  "nextStop": {
    "stopId": "stop_sector22",
    "stopName": "Sector 22 Market",
    "eta": 5
  },
  "completedStops": 8,
  "totalStops": 15,
  "timestamp": "2025-11-14T14:15:30Z"
}
```

7. PUSH NOTIFICATIONS

7.1 Firebase Cloud Messaging (FCM) Setup

Dependencies (pom.xml):

```
&lt;dependency&gt;
    &lt;groupId&gt;com.google.firebase&lt;/groupId&gt;
    &lt;artifactId&gt;firebase-admin&lt;/artifactId&gt;
    &lt;version&gt;9.2.0&lt;/version&gt;
&lt;/dependency&gt;
```

Configuration:

```
@Configuration
public class FirebaseConfig {

    @PostConstruct
    public void initialize() {
        FileInputStream serviceAccount =
            new FileInputStream("firebase-credentials.json");

        FirebaseOptions options = FirebaseOptions.builder()
            .setCredentials(GoogleCredentials.fromStream(serviceAccount))
            .build();

        FirebaseApp.initializeApp(options);
    }
}
```

7.2 Notification Types

Type	Trigger	Recipient	Priority
BUS_ARRIVING	Bus within 5 min of stop	Passengers waiting	HIGH
BUS_DELAYED	Bus >10 min delayed	Passengers waiting	HIGH
TRIP_STARTED	Trip starts	Route followers	NORMAL
TRIP_COMPLETED	Trip ends	Route followers	LOW
MESSAGE_RECEIVED	New message	Recipient (if offline)	HIGH
REPORT_APPROVED	Admin approves report	Report submitter	NORMAL
REPORT_REJECTED	Admin rejects report	Report submitter	NORMAL
REWARD_EARNED	User earns badge/points	User	NORMAL
ACCOUNT_SUSPENDED	Account banned	User	HIGH

7.3 Send Push Notification

Backend Service:

```
@Service
public class PushNotificationService {

    public void sendBusArrivingNotification(String userId, BusArrivalInfo info) {
        // Get user's FCM token from database
        String fcmToken = userRepository.findById(userId)
            .orElseThrow()
            .getFcmToken();

        // Build notification
        Message message = Message.builder()
            .setToken(fcmToken)
            .setNotification(Notification.builder()
                .setTitle("Bus Arriving Soon!")
                .setBody("Bus " + info.getBusNumber() + " arriving at " +
                         info.getStopName() + " in " + info.getEta() + " minutes")
                .build())
            .putData("type", "BUS_ARRIVING")
            .putData("busId", info.getBusId())
            .putData("stopId", info.getStopId())
            .putData("eta", String.valueOf(info.getEta()))
            .setAndroidConfig(AndroidConfig.builder()
                .setPriority(AndroidConfig.Priority.HIGH)
                .setNotification(AndroidNotification.builder()
                    .setSound("default")
                    .setChannelId("bus_arrivals")
                    .build())
                .build())
            .setApnsConfig(ApnsConfig.builder()
                .setAps(Aps.builder()
                    .setSound("default")
                    .setBadge(1)
                    .build())
                .build())
            .build();

        // Send notification
        String response = FirebaseMessaging.getInstance().send(message);
        log.info("Successfully sent notification: " + response);
    }
}
```

7.4 Frontend - Register for Notifications

Web (React):

```
// Request permission
const messaging = firebase.messaging();

messaging.requestPermission()
```

```
  .then(() =&gt; {
    return messaging.getToken();
  })
  .then((fcmToken) =&gt; {
    // Send token to backend
    api.post('/users/fcm-token', { fcmToken });
  });

// Listen for foreground messages
messaging.onMessage((payload) =&gt; {
  console.log('Message received:', payload);

  // Show notification
  showNotification(payload.notification.title, payload.notification.body);
});
```

Android (React Native):

```
import messaging from '@react-native-firebase/messaging';

async function requestUserPermission() {
  const authStatus = await messaging().requestPermission();
  const enabled =
    authStatus === messaging.AuthorizationStatus.AUTHORIZED ||
    authStatus === messaging.AuthorizationStatus.PROVISIONAL;

  if (enabled) {
    console.log('Authorization status:', authStatus);
    getFCMToken();
  }
}

async function getFCMToken() {
  const fcmToken = await messaging().getToken();

  // Send to backend
  api.post('/users/fcm-token', { fcmToken });
}

// Listen for notifications
messaging().onMessage(async remoteMessage =&gt; {
  console.log('FCM Message:', remoteMessage);

  // Show local notification
  PushNotification.localNotification({
    title: remoteMessage.notification.title,
    message: remoteMessage.notification.body,
  });
});
```

7.5 Notification Payload Example

```
{
  "notification": {
    "title": "Bus Arriving Soon!",
    "body": "Bus HR26Q4321 arriving at Sector 17 Plaza in 3 minutes"
  },
  "data": {
    "type": "BUS_ARRIVING",
    "busId": "bus_789",
    "busNumber": "HR26Q4321",
    "stopId": "stop_sector17",
    "stopName": "Sector 17 Plaza",
    "eta": "3"
  },
  "android": {
    "priority": "HIGH",
    "notification": {
      "sound": "default",
      "channelId": "bus_arrivals",
      "icon": "ic_bus_notification",
      "color": "#0EA5E9"
    }
  },
  "apns": {
    "payload": {
      "aps": {
        "sound": "default",
        "badge": 1
      }
    }
  }
}
```

8. REDIS FOR SESSION & CACHING

8.1 Redis Setup

Dependencies (pom.xml):

```
&lt;dependency&gt;
    &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
    &lt;artifactId&gt;spring-boot-starter-data-redis&lt;/artifactId&gt;
&lt;/dependency&gt;
&lt;dependency&gt;
    &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
    &lt;artifactId&gt;spring-boot-starter-cache&lt;/artifactId&gt;
&lt;/dependency&gt;
```

Configuration:

```
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.password=
spring.redis.timeout=60000
spring.cache.type=redis
```

8.2 Use Cases

8.2.1 WebSocket Session Management

Store active WebSocket connections in Redis:

```
@Service
public class WebSocketSessionService {

    @Autowired
    private RedisTemplate&lt;String, String&gt; redisTemplate;

    public void registerSession(String userId, String sessionId) {
        // Store user's session ID
        redisTemplate.opsForValue().set(
            "ws:session:" + userId,
            sessionId,
            Duration.ofHours(24)
        );

        // Add to active users set
        redisTemplate.opsForSet().add("ws:active-users", userId);
    }

    public boolean isUserOnline(String userId) {
        return redisTemplate.hasKey("ws:session:" + userId);
    }

    public String getUserSession(String userId) {
        return redisTemplate.opsForValue().get("ws:session:" + userId);
    }
}
```

8.2.2 Caching Bus Locations

Cache frequently accessed bus locations:

```
@Service
public class BusLocationService {

    @Autowired
    private RedisTemplate&lt;String, BusLocation&gt; redisTemplate;

    @Cacheable(value = "busLocations", key = "#busId")
    public BusLocation getBusLocation(String busId) {
        // Check Redis first
        BusLocation cached = redisTemplate.opsForValue().get("bus:location:" + busId);

        if (cached != null) {
            return cached;
        }

        // If not in cache, get from MongoDB
        BusLocation location = busRepository.findById(busId)
            .orElseThrow()
            .getCurrentLocation();

        // Cache for 10 seconds
        redisTemplate.opsForValue().set(
            "bus:location:" + busId,
            location,
            Duration.ofSeconds(10)
        );

        return location;
    }

    @CacheEvict(value = "busLocations", key = "#busId")
    public void updateBusLocation(String busId, BusLocation location) {
        // Update in MongoDB
        busRepository.updateLocation(busId, location);

        // Redis cache automatically evicted by @CacheEvict
    }
}
```

8.2.3 Rate Limiting

Implement rate limiting using Redis:

```
@Service
public class RateLimiterService {

    @Autowired
    private RedisTemplate&lt;String, Integer&gt; redisTemplate;

    public boolean isAllowed(String userId, String action, int maxAttempts, Duration winc
        String key = "ratelimit:" + action + ":" + userId;

        Integer attempts = redisTemplate.opsForValue().get(key);

        if (attempts == null) {
            // First attempt
            redisTemplate.opsForValue().set(key, 1, window);
            return true;
        }

        if (attempts &gt;= maxAttempts) {
            return false; // Rate limit exceeded
        }
        // Increment attempts
        redisTemplate.opsForValue().increment(key);
        return true;
    }
}
```

Usage in controller:

```
@PostMapping("/reports")
public ResponseEntity submitReport(
    @RequestBody ReportRequest request,
    Principal principal
) {
    // Rate limit: 10 reports per hour
    if (!rateLimiterService.isAllowed(
        principal.getName(),
        "submit-report",
        10,
        Duration.ofHours(1)
    )) {
        throw new RateLimitExceededException("Too many reports. Try again later.");
    }

    // Process report...
}
```

8.2.4 Real-Time Analytics

Store real-time stats in Redis for admin dashboard:

```
@Service
public class RealtimeAnalyticsService {

    @Autowired
    private RedisTemplate&lt;String, Object&gt; redisTemplate;

    public void incrementActiveBuses() {
        redisTemplate.opsForValue().increment("analytics:active-buses");
    }

    public void incrementActiveTrips() {
        redisTemplate.opsForValue().increment("analytics:active-trips");
    }

    public void addOnlineUser(String userId) {
        redisTemplate.opsForSet().add("analytics:online-users", userId);
    }

    public RealtimeStats getRealtimeStats() {
        return RealtimeStats.builder()
            .activeBuses(getLong("analytics:active-buses"))
            .activeTrips(getLong("analytics:active-trips"))
            .onlineUsers(redisTemplate.opsForSet().size("analytics:online-users"))
            .build();
    }
}
```

Broadcast to admin dashboard:

```
@Scheduled(fixedRate = 5000) // Every 5 seconds
public void broadcastRealtimeStats() {
    RealtimeStats stats = realtimeAnalyticsService.getRealtimeStats();

    messagingTemplate.convertAndSend(
        "/topic/admin/dashboard",
        stats
    );
}
```

9. LIVE ADMIN DASHBOARD

9.1 Real-Time Statistics

Admin subscribes to dashboard updates:

```
// Admin dashboard connects
stompClient.subscribe("/topic/admin/dashboard", (message) =&gt; {
  const stats = JSON.parse(message.body);

  // Update dashboard cards
  updateDashboardStats({
    activeBuses: stats.activeBuses,
    activeTrips: stats.activeTrips,
    onlineUsers: stats.onlineUsers,
    pendingReports: stats.pendingReports
  });

  // Update charts
  updateCharts(stats);
});
```

9.2 Dashboard Update Format

```
{
  "activeBuses": 42,
  "activeTrips": 38,
  "onlineUsers": 1250,
  "onlinePassengers": 1100,
  "onlineDrivers": 35,
  "pendingReports": 15,
  "systemHealth": {
    "database": "HEALTHY",
    "redis": "HEALTHY",
    "webSocket": "HEALTHY"
  },
  "recentActivity": [
    {
      "type": "TRIP_STARTED",
      "busNumber": "HR26Q4321",
      "timestamp": "2025-11-14T14:20:00Z"
    }
  ],
  "timestamp": "2025-11-14T14:20:05Z"
}
```

10. ERROR HANDLING & RECONNECTION

10.1 WebSocket Connection Lost

Frontend handles disconnection:

```
stompClient.onWebSocketClose = () =&gt; {
  console.log('WebSocket disconnected');

  // Show offline indicator
  showOfflineIndicator();

  // Attempt reconnection
  reconnectWebSocket();
};

function reconnectWebSocket() {
  let attempts = 0;
  const maxAttempts = 5;
  const delay = 2000; // 2 seconds

  const reconnect = setInterval(() =&gt; {
    attempts++;

    console.log(`Reconnection attempt ${attempts}/${maxAttempts}`);

    connectWebSocket()
      .then(() =&gt; {
        clearInterval(reconnect);
        hideOfflineIndicator();
        showNotification('Reconnected successfully');
      })
      .catch(() =&gt; {
        if (attempts &gt;= maxAttempts) {
          clearInterval(reconnect);
          showError('Failed to reconnect. Please refresh.');
        }
      });
  }, delay);
}
```

10.2 Message Delivery Guarantee

Implement message queue for offline users:

```
@Service
public class MessageDeliveryService {

    public void sendMessage(ChatMessage message) {
        String receiverId = message.getReceiverId();

        if (webSocketSessionService.isUserOnline(receiverId)) {
            // Send via WebSocket
            messagingTemplate.convertAndSendToUser(
                receiverId,
                "/queue/messages",
                message
            );
        } else {
            // Queue message in Redis
            redisTemplate.opsForList().rightPush(
                "message-queue:" + receiverId,
                message
            );

            // Send push notification
            pushNotificationService.sendMessageNotification(receiverId, message);
        }
    }

    public void deliverQueuedMessages(String userId) {
        String queueKey = "message-queue:" + userId;

        // Get all queued messages
        List&lt;ChatMessage&gt; messages = redisTemplate.opsForList().range(queueKey, 0,

        if (messages != null &amp;&amp; !messages.isEmpty()) {
            // Send all queued messages
            messages.forEach(message -&gt; {
                messagingTemplate.convertAndSendToUser(
                    userId,
                    "/queue/messages",
                    message
                );
            });

            // Clear queue
            redisTemplate.delete(queueKey);
        }
    }
}
```

11. PERFORMANCE OPTIMIZATION

11.1 Connection Pooling

```
# WebSocket configuration
spring.websocket.max-connections=10000
spring.websocket.buffer-size=8192
spring.websocket.max-idle-timeout=300000
```

11.2 Message Compression

Enable compression for large messages:

```
@Configuration
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic", "/queue")
              .setTaskScheduler(taskScheduler())
              .setHeartbeatValue(new long[]{10000, 10000}); // 10 sec heartbeat

        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws")
                .setAllowedOriginPatterns("*")
                .withSockJS()
                .setWebSocketEnabled(true)
                .setStreamBytesLimit(512 * 1024) // 512 KB
                .setHttpMessageCacheSize(1000);
    }
}
```

11.3 Throttling Location Updates

Only broadcast location if significant change:

```
public void updateBusLocation(String busId, Location newLocation) {
    Location lastLocation = getLastBroadcastedLocation(busId);

    // Only broadcast if moved &gt; 50 meters
    double distance = calculateDistance(lastLocation, newLocation);

    if (distance &gt; 50 || hasBeenTooLong(busId, 30)) { // 30 seconds
        // Broadcast
        messagingTemplate.convertAndSend(
            "/topic/bus/" + busId + "/location",
            newLocation
        );

        // Update cache
        updateLastBroadcastedLocation(busId, newLocation);
    }
}
```

12. SECURITY CONSIDERATIONS

12.1 WebSocket Authentication

Authenticate WebSocket connection:

```
@Configuration
public class WebSocketSecurityConfig {

    @Bean
    public WebSocketMessageBrokerConfigurer webSocketAuthenticationConfig(
        JwtService jwtService
    ) {
        return new WebSocketMessageBrokerConfigurer() {

            @Override
            public void configureClientInboundChannel(ChannelRegistration registration) {
                registration.interceptors(new ChannelInterceptor() {

                    @Override
                    public Message preSend(Message message, MessageChannel channel) {
                        StompHeaderAccessor accessor =
                            MessageHeaderAccessor.getAccessor(message, StompHeaderAccesso

                        if (StompCommand.CONNECT.equals(accessor.getCommand())) {
                            // Extract JWT from header
                            String token = accessor.getFirstNativeHeader("Authorization")

                            if (token != null &amp;&amp; token.startsWith("Bearer ")) {
                                token = token.substring(7);

                                // Validate JWT
                                if (jwtService.validateToken(token)) {
                                    String userId = jwtService.getUserIdFromToken(token);
                                    accessor.setUser(() -&gt; userId);
                                } else {
                                    throw new IllegalArgumentException("Invalid token");
                                }
                            }
                        }

                        return message;
                    }
                });
            }
        };
    }
}
```

Frontend sends token:

```
const socket = new SockJS('http://localhost:8080/ws');
const stompClient = Stomp.over(socket);

stompClient.connect(
  { Authorization: `Bearer ${accessToken}` },
  (frame) =&gt; {
    console.log('Connected:', frame);
  },
  (error) =&gt; {
    console.error('Connection error:', error);
  }
);
```

12.2 Authorization

Validate user permissions before broadcasting:

```
@MessageMapping("/bus/location/update")
public void updateBusLocation(
    @DestinationVariable String busId,
    BusLocationUpdateRequest request,
    Principal principal
) {
    // Check if user is driver assigned to this bus
    User user = userRepository.findById(principal.getName()).orElseThrow();

    if (!user.getRole().equals(Role.DRIVER)) {
        throw new UnauthorizedException("Only drivers can update bus location");
    }

    if (!user.getDriverDetails().getAssignedBusId().equals(busId)) {
        throw new UnauthorizedException("Not assigned to this bus");
    }

    // Proceed with update...
}
```

13. MONITORING & LOGGING

13.1 WebSocket Metrics

Track WebSocket connections:

```
@Component
public class WebSocketEventListener {

    @Autowired
    private RedisTemplate&lt;String, String&gt; redisTemplate;

    @EventListener
    public void handleWebSocketConnectListener(SessionConnectedEvent event) {
        StompHeaderAccessor headerAccessor = StompHeaderAccessor.wrap(event.getMessage())
        String userId = headerAccessor.getUser().getName();

        log.info("User connected: " + userId);

        // Increment connection count
        redisTemplate.opsForValue().increment("metrics:ws-connections");
    }

    @EventListener
    public void handleWebSocketDisconnectListener(SessionDisconnectEvent event) {
        StompHeaderAccessor headerAccessor = StompHeaderAccessor.wrap(event.getMessage())
        String userId = headerAccessor.getUser().getName();

        log.info("User disconnected: " + userId);

        // Decrement connection count
        redisTemplate.opsForValue().decrement("metrics:ws-connections");
    }
}
```

13.2 Logging

```
# Logging configuration
logging.level.org.springframework.messaging=DEBUG
logging.level.org.springframework.websocket=DEBUG
logging.level.com.whereismybus=INFO
```

14. TESTING STRATEGY

14.1 WebSocket Integration Tests

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class WebSocketIntegrationTest {

    @LocalServerPort
    private int port;
    @Test
    public void testBusLocationUpdate() throws Exception {
        WebSocketStompClient stompClient = new WebSocketStompClient(
            new SockJsClient(Arrays.asList(new WebSocketTransport(new StandardWebSocketCl
        );

        StompSession session = stompClient
            .connect("ws://localhost:" + port + "/ws", new StompSessionHandlerAdapter() {
            .get(1, TimeUnit.SECONDS);

        // Subscribe to topic
        session.subscribe("/topic/bus/bus_789/location", new StompFrameHandler() {
            @Override
            public Type getPayloadType(StompHeaders headers) {
                return BusLocationUpdate.class;
            }

            @Override
            public void handleFrame(StompHeaders headers, Object payload) {
                BusLocationUpdate update = (BusLocationUpdate) payload;
                assertEquals("bus_789", update.getBusId());
            }
        });

        // Send location update
        session.send("/app/bus/location/update", new BusLocationUpdateRequest());

        Thread.sleep(1000); // Wait for broadcast
    }
}
```

14.2 Push Notification Tests

Mock Firebase for testing:

```
@Test
public void testPushNotification() {
    // Mock FCM
    FirebaseMessaging mockMessaging = mock(FirebaseMessaging.class);
    when(mockMessaging.send(any(Message.class)))
        .thenReturn("mock-message-id");

    // Test notification send
    pushNotificationService.sendBusArrivingNotification("user_123", busArrivalInfo);

    verify(mockMessaging, times(1)).send(any(Message.class));
}
```

15. DEPLOYMENT CONFIGURATION

15.1 Production Settings

```
# WebSocket
spring.websocket.max-connections=50000
spring.websocket.buffer-size=16384

# Redis
spring.redis.host=${REDIS_HOST}
spring.redis.port=6379
spring.redis.password=${REDIS_PASSWORD}
spring.redis.ssl=true

# Firebase
firebase.credentials.path=${FIREBASE_CREDENTIALS_PATH}
```

15.2 Load Balancing

For horizontal scaling with multiple servers:

```
Load Balancer
    ↓
Server 1 (WebSocket)
    ↓
Redis Pub/Sub (Shared message broker)
    ↓
Server 2 (WebSocket)
```

Enable sticky sessions at load balancer level.

16. SUCCESS CRITERIA

✅ WebSocket server configured and running
✅ Real-time bus location updates working
✅ One-to-one messaging functional
✅ Push notifications sending successfully
✅ Trip status updates broadcasting
✅ Redis caching implemented
✅ Session management working
✅ Rate limiting configured
✅ WebSocket authentication working
✅ Reconnection logic implemented
✅ Admin dashboard live updates
✅ Performance optimization done
✅ Integration tests passing

17. INTEGRATION WITH MODULES 1 & 2

```
Module 1 (Database)
    ↓
MongoDB stores persistent data
    ↓
Module 2 (APIs)
    ↓
REST APIs for CRUD operations
    ↓
Module 3 (Real-Time)
    ↓
WebSocket for live updates
    ↓
Redis for caching &amp; sessions
    ↓
FCM for push notifications
```

Next Steps:

1. Complete backend implementation (Modules 1-3)
2. Integrate with existing frontend (Modules 4-5)
3. End-to-end testing
4. Deploy to production (Module 6)

Congratulations! You now have complete documentation for **Modules 1, 2, and 3** of the Where Is My Bus? backend system.