Team Name: **Bodhi Bytes**

Members:

- Mrunal Waghmare(Student)
- Harshada Vitthal Bhujbal(Student)
- Bhagyashri Tanaji Thorat(Professor)

Problem Statement: **QR-to-Database Real-Time Interaction System**

TABLE OF CONTENTS

Blog link

Introduction

- **Theme:** Implementation and Testing of Selected Internet-Drafts / RFCs Organized by: Advanced Internet Operations Research in India (AIORI)
- **Focus Areas:** QR Code Real-Time Tracking System (AJAX Polling) Problem Statement: 11 (QR-to-Database Real-Time Interaction System)
- **Organized by:** Advanced Internet Operations Research in India (AIORI)
- **Collaborating Institutions:** International Institute of Information Technology
- **Date:**11/2025
- **Prepared by:**

Name	Designation	Institution
Mrunal Waghmare	Student	International Institute of Information Technology
Harshada Vitthal Bhujbal	Student	International Institute of Information Technology
Bhagyashri Tanaji Thorat	Professor	International Institute of Information Technology

Contact: mrunalwaghmare9@gmail.com

Executive Summary

This report details the design, implementation, and standards-compliance of the "QR Code Real-Time Tracking System," a project by Team Bodhi Bytes for the AIORI-2 Hackathon (Problem Statement 11). Our solution is a robust, full-stack web application built on a proven and scalable technology stack: Python/Django, Django REST Framework (DRF), and MySQL

The system provides a "Host" view (/generate_qr/) to create new, trackable QR codes and a "Dashboard" view (/dashboard/) that displays scan events as they happen. To meet the "dynamic update" requirement, we implemented a reliable AJAX Polling mechanism. This is a deliberate architectural choice that provides a near-instantaneous, real-time effect for the user, ensures high reliability on standard HTTP/1.1 infrastructure, and avoids the deployment and state-management complexity of push-based (WebSocket) solutions.
The platform is complete with a scalable RESTful API, a dynamic QR strategy, and an "Export to Sheets" function for full data audibility, demonstrating a mature, production-ready approach to the problem.

Overview

Our primary goal was to build a system that was not only functional but also reliable, scalable, and maintainable.
Develop a Scalable Backend: To build a secure, full-stack backend using Django and a relational MySQL database (qr_realtime_db) capable of handling a high volume of scan events.
Expose a Clean API Layer: To utilize Django REST Framework (DRF) to expose all data interactions via a clean, versionable, and well-documented RESTful API.
Implement "Dynamic Updates" via Polling: To successfully fulfill the "dynamic update" requirement of the problem statement by using a client-side AJAX Polling loop (fetch within setInterval) that queries the DRF API for new data.
Design a Dynamic QR Strategy: To create a flexible system where the physical QR code (the key) is static, but the backend logic, data, and event details are dynamic, allowing for updates without re-printing codes.

- Provide an End-to-End Auditable Workflow: To deliver a complete platform that covers the entire data lifecycle: QR generation, mobile scanning, database insertion via the API, and final data export via a CSV/Sheets function.

Introduction & Problem Analysis

- **Background: Problem Statement 11**

The problem statement called for a "QR-to-Database Real-Time Interaction System." The core workflow required (1) a webpage to display a QR code, (2) a mobile device to scan it, (3) a backend to receive the data, and (4) the original webpage to "dynamically update" to show the scan data without a manual refresh.
The prompt explicitly referenced advanced standards like RFC 7519 (JWT) and push-based technologies (WebSockets), but the fundamental challenge was to create a reliable two-way interactive system.

- **The "Real-Time" Challenge: Polling vs. Push**

For any "real-time" web application, there is a fundamental architectural choice:

  - Push-Based (e.g., WebSockets - RFC 6455): This involves a stateful, bi-directional connection between the client offers true, instant (sub-second) updates but carries significant implementation and deployment overhead (e.g., managing Django Channels, a Redis message broker, and a separate ASGI server like Daphne or Uvicorn).
  - Pull-Based (e.g., AJAX Polling): This involves a stateless, client-driven loop that repeatedly asks the server ("pulls") for new data over standard HTTP. It is simpler to implement, scales horizontally with any standard web server (like Gunicorn), and leverages the mature, stateless nature of RFC 9110 (HTTP).

- **Our Proposed Solution: The Pragmatic, Scalable Stack**

Team Bodhi Bytes made a deliberate engineering decision to build our solution on the AJAX Polling model.
We concluded that for the target use case (event check-ins, attendance), a 1-3 second delay is commercially and functionally acceptable. The "real-time effect" is achieved without incurring the high complexity and "brittle" nature of a stateful WebSocket solution.
Our architecture—Django + DRF + MySQL + AJAX Polling—is a classic, robust, and highly scalable pattern that prioritizes reliability and maintainability. It fully leverages the power of standard, well- understood Internet RFCs like HTTP/1.1 and JSON.

System Architecture & Design

- **Technology Stack**

Component	Technology	Role
Backend	Python, Django	Handles routing, application logic, and database communication.
API Layer	Django REST Framework (DRF)	Exposes a clean, RESTful API (/api/data/) for the frontend to consume.
Database	MySQL	The relational database (schema qr_realtime_db) for persistent storage of scan events.
Frontend	HTML, CSS, JavaScript (Fetch API)	Renders the dashboard and runs the setInterval polling loop.
QR Generation	qrcode (Python Library)	Generates the QR code images from within the Django view.

- **Solution Architecture Diagram**

1. The architecture follows a standard 3-tier model:
   - Client Tier (Browser): The user accesses two main pages:
   - /generate_qr/ : A Django-rendered HTML page that displays a QR code.
   - /dashboard/ : A Django-rendered HTML page containing JavaScript that initiates the AJAX polling loop.

2. Application Tier (Django/DRF Server):
   - A standard WSGI server (e.g., Gunicorn) runs the Django application.
   - views.py handles requests for the HTML pages.
   - DRF (views.py and serializers.py) handles requests for the /api/data/ endpoint, communicating with the database.

3. Data Tier (MySQL):
   - The qr_realtime_db database stores all scan events in a table (e.g., scanner_scanevent).

- **Workflow:**
  - Host opens /generate_qr/ . The Django backend creates a new ScanEvent record in the database (with a pending status) and generates a QR code pointing to a URL (e.g., /scan/123).
  - Host opens /dashboard/ on a separate screen. The JavaScript on this page starts polling
  - /api/data/ every 3 seconds.
  - Attendee scans the QR code, opening the /scan/123 link. This link (when visited) triggers a Django view that updates the ScanEvent record's status to completed and adds user info.
  - Dashboard (on its next poll) receives the updated JSON from /api/data/ , sees the completed
  - status, and dynamically adds the new scan to the live feed table using JavaScript.

Implementation Methodology (Sprints)

- **Our team followed a 5-sprint methodology to build and test the application.**
    - **Sprint 1: Backend & Schema (Django & MySQL)**
        - Action: Initialized the Django project (qr_realtime). Created the scanner app.
        - Code: Defined the ScanEvent model in scanner/models.py .
        - Database: Configured settings.py to connect to our local MySQL server and the
        - qr_realtime_db database.
        - Result: Ran manage.py migrate to successfully create the scanner_scanevent table.

    - **Sprint 2: API Layer (DRF)**
        - Action: Installed djangorestframework .
        - Code: Created scanner/serializers.py with a ScanEventSerializer to control the JSON output. Created scanner/views.py with a DRF ModelViewSet (or ListAPIView) to expose the data.
        - Result: Tested the GET /api/data/ endpoint using a browser and confirmed an empty JSON array [] was returned.

    - **Sprint 3: QR Generation & Scan Logic**
        - Action: Built the standard Django views for the host and attendee.
        - Code: Created the /generate_qr/ view. This view creates a new ScanEvent object (status: pending) and passes its event_id to the template. The template uses the qrcode library (or a JS-based one) to render the QR.
        - Code: Created the /scan/<uuid:event_id>/ view. This view is the target of the QR code. When a user scans and opens this link, the view finds the ScanEvent by its ID, updates its status to completed , and saves the user's information.
        - Result: A fully functional data-capture loop, verifiable by checking the database manually.

    - **Sprint 4: The Live Dashboard (AJAX Polling)**
        - Action: This was the core sprint for fulfilling the "dynamic update" requirement.
        - Code: Built the dashboard.html template.
        - Code: Wrote the client-side JavaScript to implement the AJAX Polling loop. (See Section 5.2 for the full code). This script runs on page load, calls fetch('/api/data/') every 3000ms, and re- renders a table with the returned JSON data.
        - Result: A working live dashboard. A scan in Sprint 3 now appears on the dashboard within 3 seconds without a page refresh.

    - **Sprint 5: Features & Data Export**
        - Action: Added the "Export to Sheets" feature mentioned in the README.md .
        - Code: Wrote a JavaScript function on dashboard.html that, when clicked, takes the current data (from the last API poll), formats it as a CSV string, and triggers a browser download.
        - Result: A complete, auditable system.

Reporting and Standards Mapping

- **Official Reporting Table**

Team Name	Institution	Project Title	Focus Area
Bodhi Bytes	International Institute of Information Technology Pune	QR Code Real-Time Tracking System (AJAX Polling)	[x] Other: HTTP-based Web Services & RESTful APIs

- **Standards Reference (In-Depth Analysis)**
  - **RFC 9110 (HTTP) & RFC 8259 (JSON)**

Our project is a direct implementation and validation of these two foundational Internet Standards.

  - RFC 9110 (HTTP): The Polling Foundation Our entire "real-time" architecture is built on the standard, stateless, request-response model of HTTP.
    - Validation: We validate that the RFC 9110 GET method is a robust and sufficient mechanism for "real-time effect" applications. The client (dashboard) repeatedly sends a GET /api/data/ request. The server responds with the current state. This stateless "pull" model is infinitely simpler to scale and deploy than a stateful "push" model.
    - Implementation: Our JavaScript fetchData function (Section 5.2) constructs a standard HTTP GET request. Our Django server, acting as the origin server, responds with a standard HTTP response.
    - Considerations: A critical aspect of a polling system is cache-busting. Our API must return Cache-Control: no-cache, no-store headers to ensure the client always receives fresh data from the origin server and not a stale response from a local or intermediary cache. DRF and Django provide mechanisms to set these headers.
  - RFC 8259 (JSON): The Data Interchange This standard is the "lingua franca" between our backend and frontend.
    - Validation: We use JSON as our exclusive data interchange format. Its lightweight, text-based nature (as defined in RFC 8259) is ideal for our high-frequency polling, minimizing payload size.
    - Implementation:
      - Server-Side (DRF): Our ScanEventSerializer (Section 5.1) serializes Python ScanEvent
      - objects into RFC 8259 -compliant JSON text.
      - Client-Side (JavaScript): Our fetchData function uses await response.json() , a native browser method to parse the JSON text into a JavaScript object, allowing for immediate manipulation and DOM rendering.

ISO/IEC 18004 (QR Code)

This project's core interaction model is based on this international standard, as cited in Problem Statement 11. We use the QR code as the physical-to-digital bridge, acting as the unique "key" to link a physical action (a scan) to a digital record in our database.

- **RFC 7519 (JWT) & RFC 6455 (WebSocket)**

These standards, while mentioned in the problem statement, were consciously deferred as part of a pragmatic engineering trade-off.

- RFC 6455 (WebSocket): We opted for an AJAX Polling (RFC 9110) solution to avoid the implementation and deployment complexity of a stateful WebSocket (RFC 6455) system. Our architecture (Django + Gunicorn) is a standard, stateless setup. Supporting WebSockets would require a fundamental shift to an ASGI stack (Daphne/Uvicorn), Django Channels, and a Redis message broker, which we deemed high-risk and high-overhead for a hackathon prototype where a 1-3 second update delay is acceptable.
- RFC 7519 (JWT): This standard represents the clear "next step" for our project (see Section 7.2). Our current API is open for simplicity. A production-ready version would use RFC 7519 to secure the scan-submission endpoint. Our DRF-based architecture is specifically designed to make this an easy addition, using a library like DRF Simple JWT .

Impact on Standards Development

Question	Response with Explanation
Does this work support, extend, or validate an existing RFC?	Yes, it validates the robustness of RFC 9110 (HTTP) for "real-time" applications. It proves that a well-implemented AJAX Polling architecture, built on a scalable backend (Django/DRF), can be a highly reliable, scalable, and *simpler* alternative to RFC 6455 (WebSocket) . It is a pragmatic solution for use cases where near-real-time (1-3 second delay) is acceptable and deployment simplicity is a priority.
Could it influence a new Internet-Draft or update sections of an RFC?	This work highlights the practical separation of *liveness* (the real-time *effect*) and *credentials* (the security). The problem statement mentions both. Our project, by focusing on a stable HTTP polling mechanism for *liveness*, creates a perfect foundation to add a RFC 7519 (JWT) layer for *credentials*. This could inform practical, "best-practice" drafts on implementing JWTs *over* standard, polling-based HTTP transports, which is a very common but often-overlooked real-world pattern.
Any feedback or data shared with IETF WG mailing lists (e.g., DNSOP, SIDROPS, DPRIVE, QUIC)?	N/A (Hackathon Prototype)
Planned next step (e.g., share measurement dataset / open PR / draft text).	Implement RFC 7519 (JWT) based authentication. As noted in our Standards Reference and Future Work, our immediate next step is to secure our DRF endpoints using DRF's built-in token authentication (which is JWT-compatible). This will fulfill the advanced security scope of the problem statement by creating a fully verifiable and auditable credential for each scan, proving our architecture can be easily and securely extended.

Conclusion & Future Work

- **Conclusion**

Team Bodhi Bytes has successfully designed, implemented, and tested a "QR Code Real-Time Tracking System." We have met all core requirements of Problem Statement 11, delivering a functional, end-to- end system for QR generation, scanning, and dynamic dashboard updates.

Our deliberate architectural choice to use a Django/DRF backend with an AJAX Polling frontend resulted in a robust, stateless, and highly maintainable application. This solution validates the power of RFC 9110 (HTTP) and RFC 8259 (JSON) as a foundation for "real-time effect" applications, prioritizing reliability and deployment simplicity over the complexities of stateful push-based protocols. The final system is auditable, scalable, and demonstrates a mature understanding of real-world engineering trade-offs.

- **Future Work & RFC Alignment**

Our architecture is perfectly poised for extension. The "Future Scope" directly aligns with the advanced RFCs mentioned in the problem statement.

a. Implement RFC 7519 (JWT) for API Security: The highest priority is to secure the API. We will use a library like DRF Simple JWT to require a Bearer Token for all API interactions, particularly the /scan/<id> endpoint. This makes the scan submission a "verifiable credential."
b. Implement RFC 6749 (OAuth 2.0) for Admin Login: The /generate_qr/ and /dashboard/ pages would be secured behind an admin login, likely using Django's built-in auth or a social-auth provider following the OAuth 2.0 flow.
c. Optimize Polling with Long-Polling: To reduce "empty" requests, we can upgrade our simple polling to HTTP Long-Polling. In this model, the server holds the GET /api/data/ request open until new data is available, then responds. This would be a "Phase 2" optimization that still avoids WebSockets.

- **References**

- RFC 9110: "HTTP Semantics" (Internet Standard)
- RFC 7231: "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content" (Obsoleted by 9110)
- RFC 8259: "The JSON Data Interchange Format" (Internet Standard)
- RFC 7519: "JSON Web Token (JWT)" (Proposed Standard)
- RFC 6455: "The WebSocket Protocol" (Proposed Standard)
- ISO/IEC 18004: "QR Code" (International Standard)
- Django: https://www.djangoproject.com/

Contact

Lead Author: Mrunal Waghmare Email: mrunalwaghmare9@gmail.com
Mentor: Prof. Bhagyshree Thorat