DEVELOPMENT OF TIMES-7 AUTHENTICATION GATEWAY

Client: **Times-7 Research Ltd.**
Author:

Tane Putaranui
Rebekka Jing
Laura Cui
Egan Ahmad

Master of Software Development
Victoria University of Wellington
27 February 2026

TABLE OF CONTENT

1. DEFINITIONS, ACRONYMS, & ABBREVIATIONS

ABBREVIATIONS	DEFINITIONS
API	Application Programming Interface
DB	Database
epc	Electronic Product Code
curl	Client-URL open-source command-line tool and library (libcurl) for transferring data across networks using various protocols, most commonly HTTP/HTTPS.
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IAS	IMPINJ Authentication Services
JSON	JavaScript Object Notation
NDJSON	Newline-delimited JSON
RAIN	Standards-based, wireless technology that connects trillions of everyday items to the internet, enabling businesses and consumers to identify, locate, authenticate, and engage each item.
RFID	Radio Frequency Identification
TTL	time-to-live
Webhook	lightweight, event-driven communication that automatically sends data between applications via HTTP

2. PROBLEM STATEMENT

Times-7 needs an authentication service that their customers can use to confirm whether an RFID tagged product is genuine. When a customer scans an item using their existing UHF RFID reader setup, the service should verify the tag's authenticity and the integrity of its encoded identity, then return a clear, trustworthy outcome such as authentic, suspicious, or unable to verify.

At present, RFID systems are mainly used for tracking and inventory, which does not address the growing risk of counterfeit goods and product substitution during shipping, storage, or handling. Customers need a reliable way to validate that the item they received is the same item that was originally tagged and shipped by the legitimate supplier, especially for high value or high fraud product categories.

This service must fit naturally into customer workflows such as warehouse receiving, distribution checkpoints, and retail intake. It should produce actionable results, record each verification event for audit and traceability, and support alerts or reporting so suspicious items can be identified quickly and handled appropriately.

3. EXECUTIVE SUMMARY

About Times-7

Times-7 is a New Zealand-based company focused on the design and manufacture of RAIN RFID antennas. It serves customers across multiple sectors, including retail, healthcare, logistics, and manufacturing, through a global distribution network.

Times-7 offers a wide portfolio of custom RAIN RFID antennas to support different deployment environments and operational requirements. Its products are used in RFID systems that require reliable tag reading performance across varied read ranges and installation contexts.

About the Application

This product is an RFID-based product authentication and traceability platform designed for Times-7 to offer as a value-added service alongside their antenna solutions. It connects to RAIN RFID reader infrastructure, such as Impinj readers, to ingest high-frequency tag inventory events, performs authentication checks through IAS services, and converts raw reads into trustworthy, auditable records of what was seen, when it was seen, and where it was seen. The system maintains an active view of nearby tags for real-time operations while also persisting historical logs for reporting, investigation, and compliance needs.

The platform is built as a modular gateway and dashboard solution. The backend provides a stable ingestion layer for long-lived reader streaming, event processing, and database updates, while the frontend presents a clean interface for search, monitoring, and operational visibility. To support development and deployments where physical infrastructure is unavailable, the solution includes simulation and debug tooling that can replay reader events and validate workflows end-to-end. The application was completed in close collaboration with the Times-7 engineering team to align technical decisions with real deployment constraints and customer use cases. Overall, the product enables customers to improve inventory visibility, reduce manual handling, strengthen anti-counterfeit controls, and achieve better supply-chain traceability through a scalable software layer that complements Times-7’s RFID hardware ecosystem.

4. TOOLS AND SOFTWARE

4.1 Tools

Name	Description
Expo	Open-source platform for making universal native apps for Android, iOS, and the web with JavaScript and React.
Uvicorn	Web server implementation for Python. Used to run FASTAPI
pip	Package manager for Python
npm	Package manager for the JavaScript programming language
curl	Open-source command-line tool and library for transferring data across networks using various protocols, most commonly HTTP/HTTPS. Used to verify data stream from endpoints
Supabase	an open-source Backend-as-a-Service platform designed as a modern, postgres SQL-based alternative to Firebase. Used to host database in this project
Swagger	Application we use to view IMPINJ Reader API

4.2 Programming Language

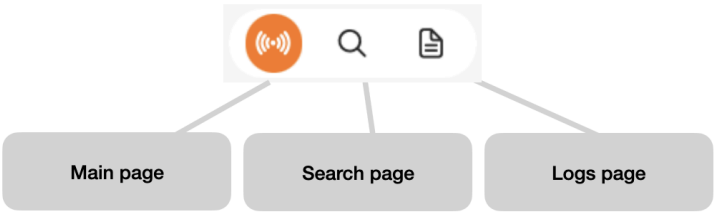
Name	Description
JavaScript	Used to develop frontend codes
Python	Used to develop backend codes
SQL	Language for managing, querying, and manipulating data within relational database management systems

4.3 Libraries/Framework

Name	Description
React Native	Javascript library used to develop frontend codes
FAST API	Web framework for building APIs in Python
Pydantic	Python library for data validation and settings management, utilizing Python type hints to enforce data structures at runtime
Python-dotenv	Python library used to load environment variables from .env file
HTTPX	Python library for HTTP client
AsyncStrorage	Local database for React Native
Fetch API	JavaScript interface for making HTTP request
Jest	Javascript testing framework

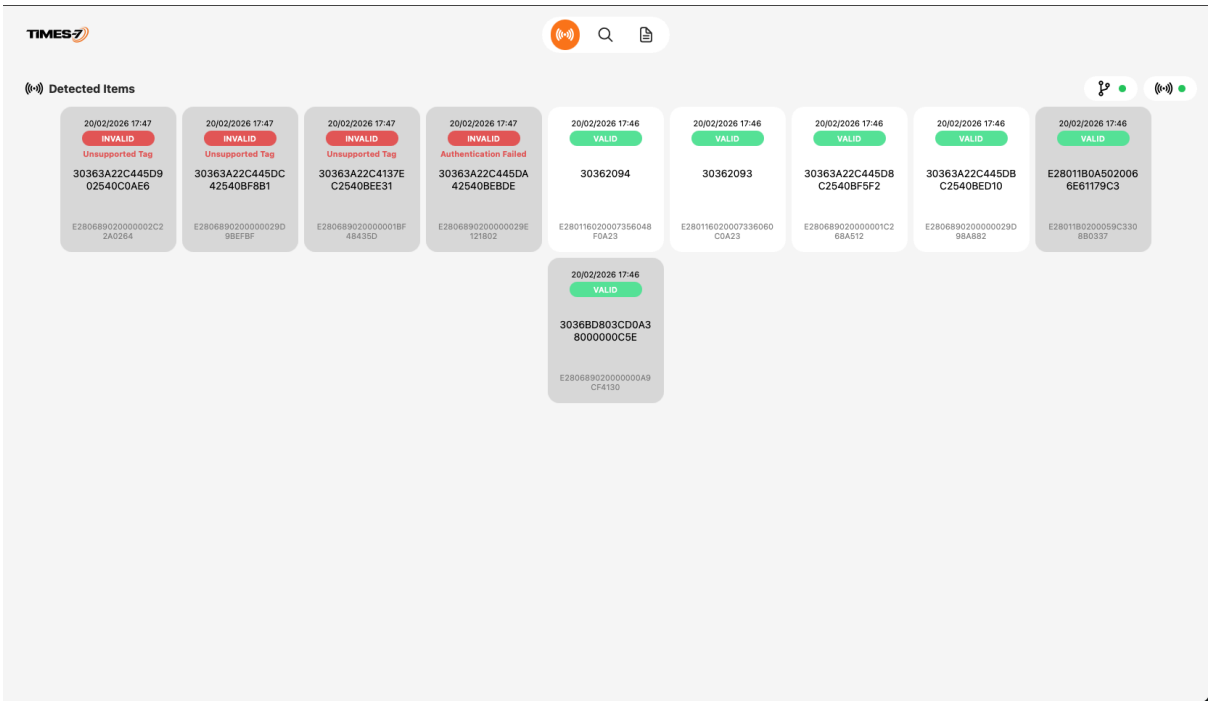
5. USER INTERFACE

5.1 Navigation

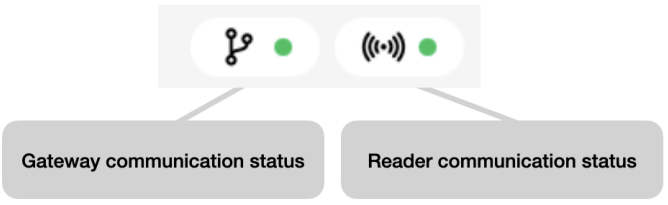


5.2 Main page

Display list of tags detected by the reader.



5.2.1 Communication Status



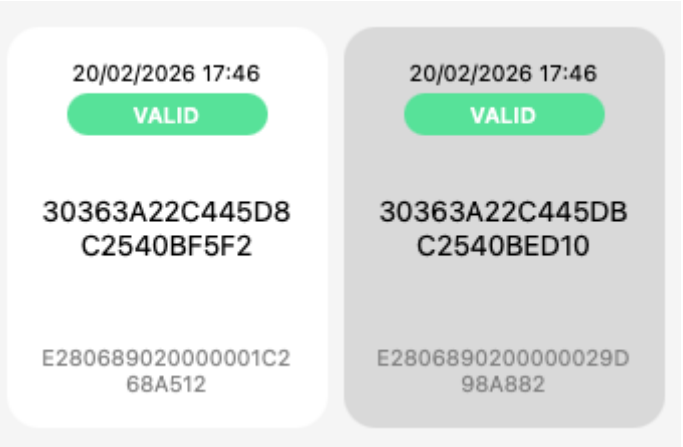
5.2.2 List of Detected Tags

5.2.3 Tag display card

5.2.3.1 Authentication status

VALID	Tag has been validated its authenticity and confirmed that it is a valid tag by IMPINJ Authentication Services (IAS)
INVALID Authentication Disabled	**Invalid - Authentication Disabled** Authentication response is not enabled in the reader. User needs to check reader configuration
INVALID Authentication Failed	**Invalid - Authentication Failed** Tag information has been verified by IMPINJ Authentication Services (IAS) but confirmed that it is not a valid IMPINJ tag.
INVALID Unsupported Tag	**Invalid - Unsupported Tag** Authentication response enabled in the reader but the tag is not supported by IMPINJ Authentication Services.

5.2.3.1 Registered Tags



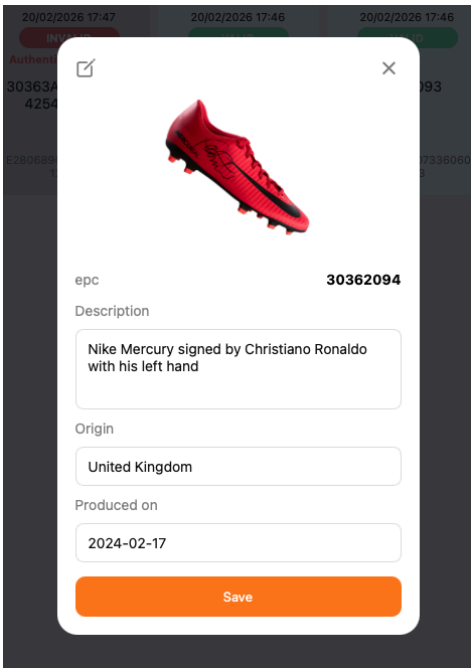
White card means the product information is registered in the Times-7 database.

Grey card means the product information is not registered in the Times-7 database.

5.2.3 View Product Information

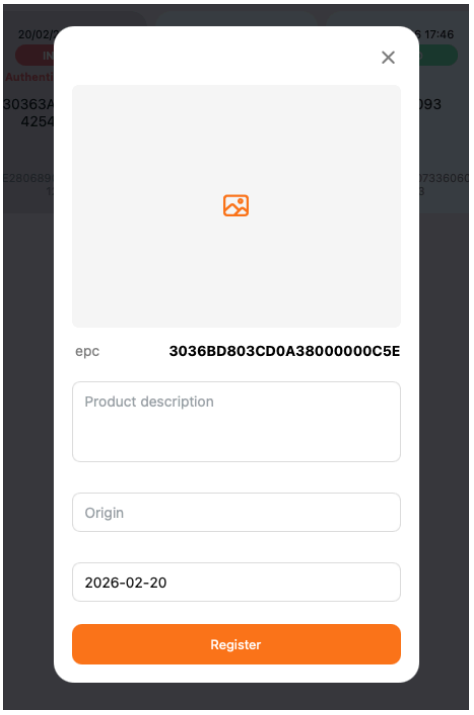
View product information by clicking on the registered card.

5.2.3 Edit Product Information



Edit product information by clicking on the edit button on the top left of the product card.

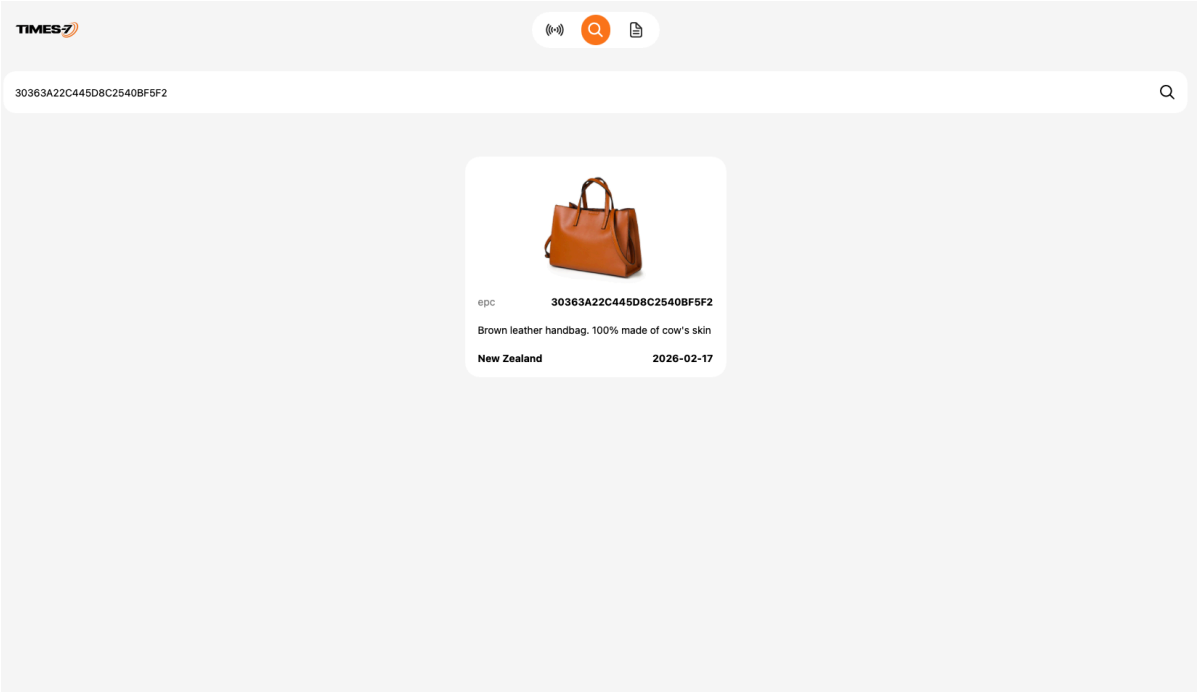
5.2.4 Product Registration



Register product into Times-7 database by clicking on unregistered (grey card) valid tag.

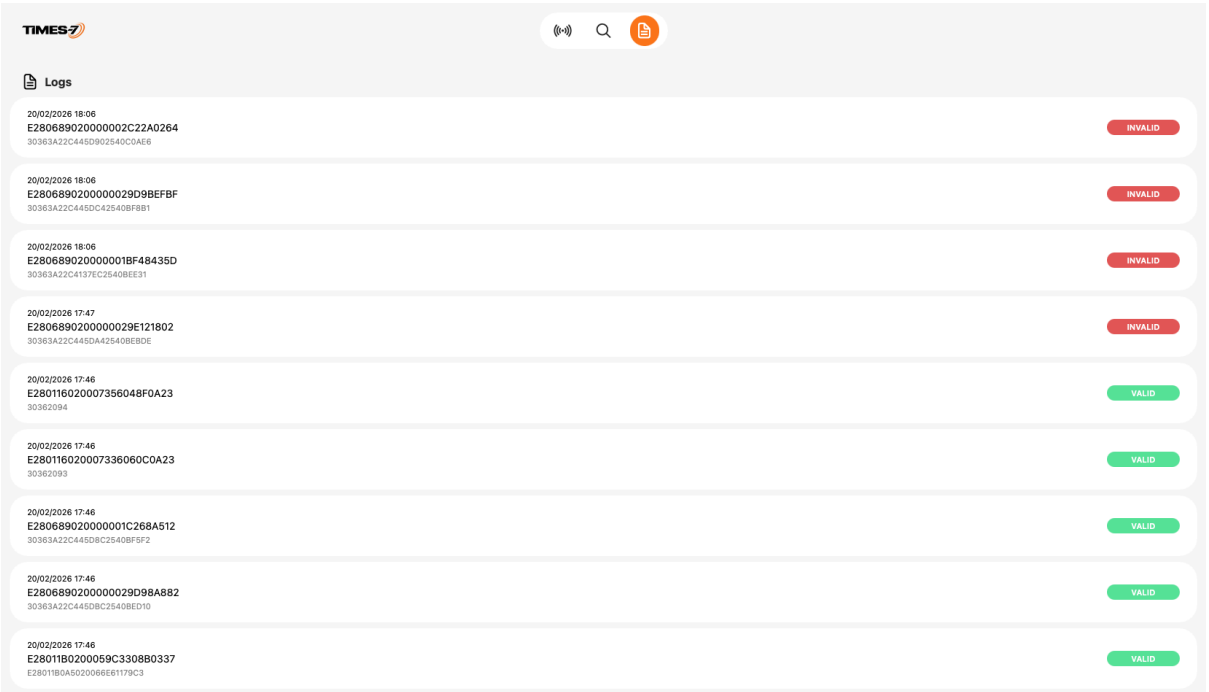
5.3 Search Page

Search registered products by entering epc number in the search bar.

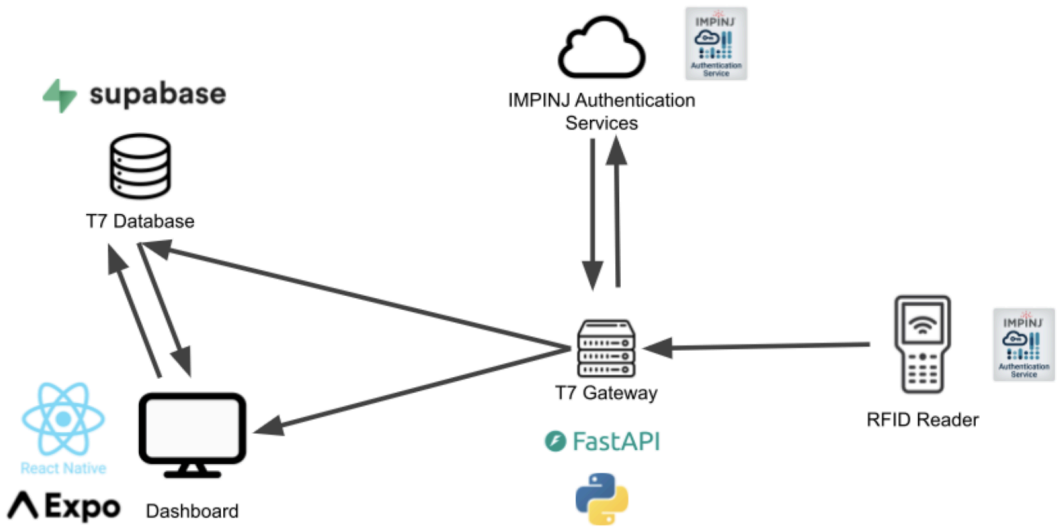


5.4 Logs Page

All scan activities are recorded and displayed in the Logs Page.



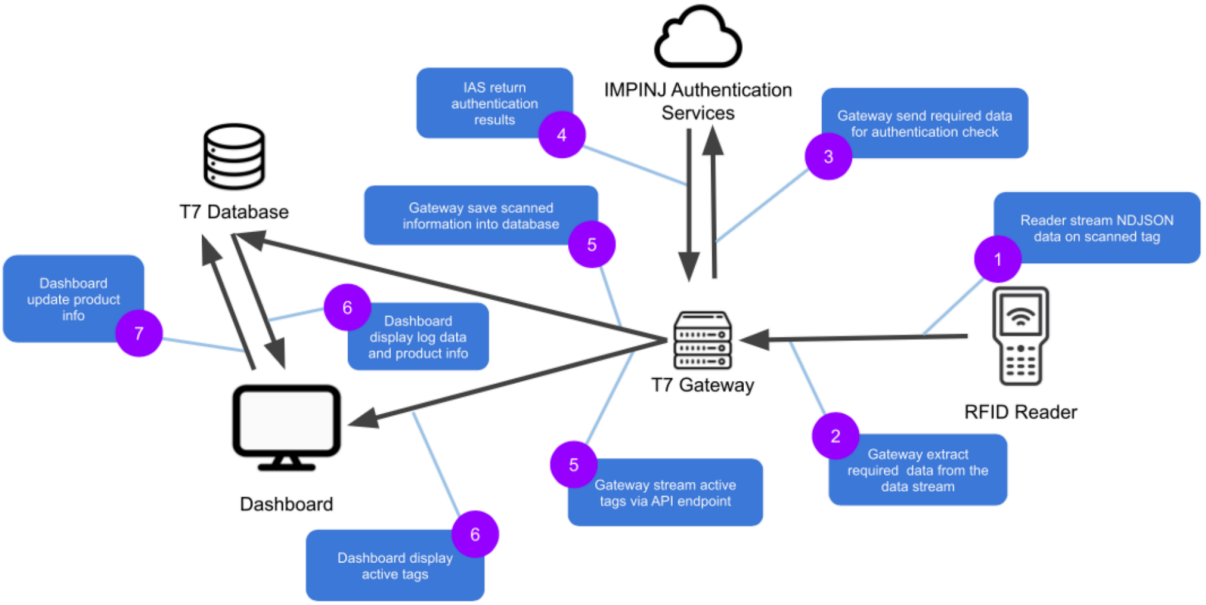
6. SYSTEM DESIGN



This architecture connects an RFID reader deployment to a Time-7 authentication and dashboard service. The RFID reader scans UHF tags in real time and sends tag events to the T7 Gateway, which is built with FastAPI and Python and acts as the central integration layer. When tag data is received, the Gateway coordinates the authentication step by communicating with Impinj Authentication Services in the cloud, then interprets the returned result to determine whether the tag is authentic, suspicious, or unable to verify. The Gateway also persists both the raw scan events and the authentication outcomes to the T7 Database hosted on Supabase, creating a reliable audit trail and enabling downstream analytics.

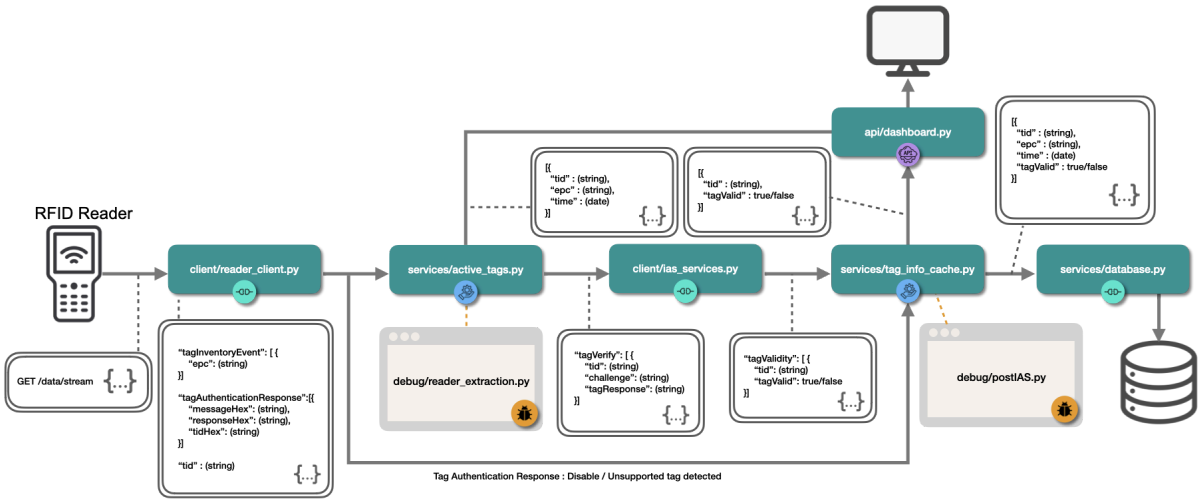
On the user side, the Dashboard application built with React Native and Expo provides the interface for real-time monitoring and operational workflows. It allows users to view current scanning activity, check authentication statuses, review historical logs, and retrieve or manage product information associated with tags. The Dashboard reads and writes data through Supabase for product records and event history, and can also rely on the Gateway for near real-time updates or API-driven views of the latest activity. Overall, the design keeps hardware and authentication complexity inside the Gateway, uses Impinj as the trusted authentication engine, stores authoritative records in Supabase, and exposes a clean user experience through the Dashboard.

6.1 Data Flow



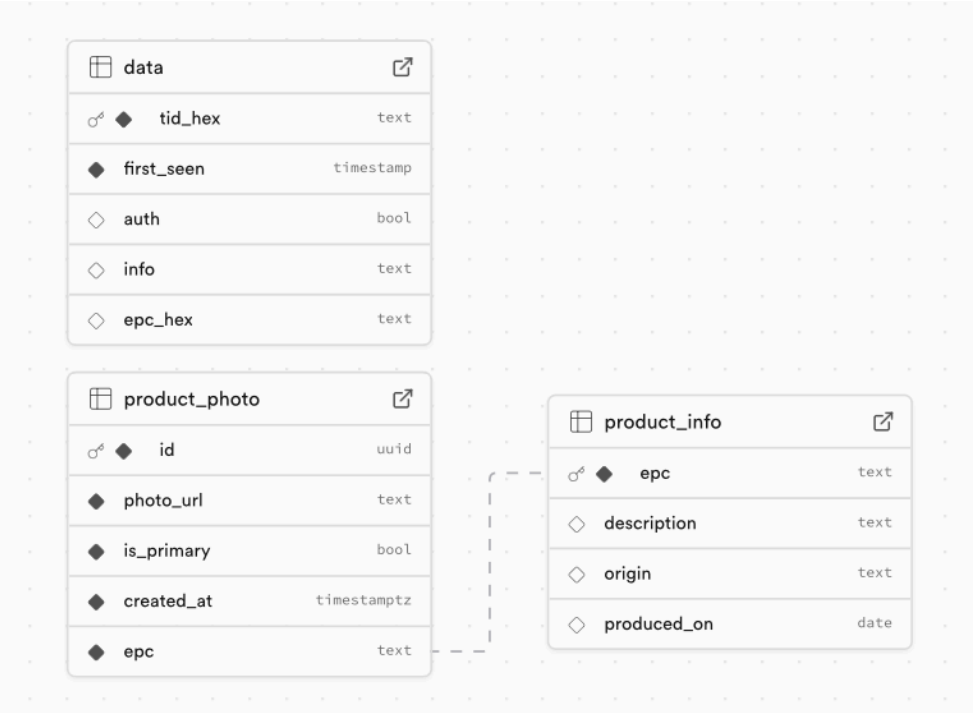
This architecture shows an end-to-end flow from tag scanning to authentication, storage, and live monitoring. First, the RFID reader streams tag inventory events in NDJSON format to the T7 Gateway (1). The Gateway parses the incoming stream and extracts only the fields needed for authentication and tracking, such as EPC/TID and timestamps (2). Using those extracted values, the Gateway sends an authentication request to Impinj Authentication Services (3), then receives the authentication decision back from IAS (4). After the result is returned, the Gateway records the scanned tag data and the authentication outcome into the T7 Database to create a persistent audit trail (5). In parallel, the Gateway exposes an API endpoint that provides the current “active tags” view so the Dashboard can display near real-time activity without relying solely on database polling (5–6). The Dashboard then presents both live active-tag monitoring and historical log views by reading from the Gateway and/or the database (6), and it also supports product management workflows where users can register or update product information, which is saved back into the database and reflected in the Dashboard views (7).

6.2 Backend Data Flow



The backend starts by consuming the RFID reader's /data/stream feed in client/reader_client.py, where it extracts the essential fields from each tag event, such as EPC, TID, and any authentication payload (message/response). The extracted tag IDs are then tracked in services/active_tags.py to maintain a live view of which tags are currently detected by the reader. For tags that include the required authentication data, the system calls Impinj Authentication Services via client/ias_services.py to perform validation and returns a tag validity result (true/false or an error if the request cannot be validated). That result is stored in services/tag_info_cache.py so the system can quickly pair each active tag with its latest validation outcome for the dashboard. The dashboard endpoint in api/dashboard.py combines active_tags with tag_info_cache to return a real-time view, while services/database.py persists scan and validation records into the T7 database for history and auditing. Two debug scripts, debug/reader_extraction.py and debug/postIAS.py, are provided to verify the reader parsing step and to test IAS requests/results independently.

6.3 Database Design



This database consists of three related tables. The product_info table stores the primary product record identified by the EPC and includes descriptive details, origin, and production date. The product_photo table stores one or more photos associated with each product and links to the product record through the EPC, including the photo location, a primary indicator, and the creation timestamp. The data table stores RFID tag observation records identified by the TID, including the first seen timestamp and optional fields for authentication status, additional information, and the associated EPC in hexadecimal form.

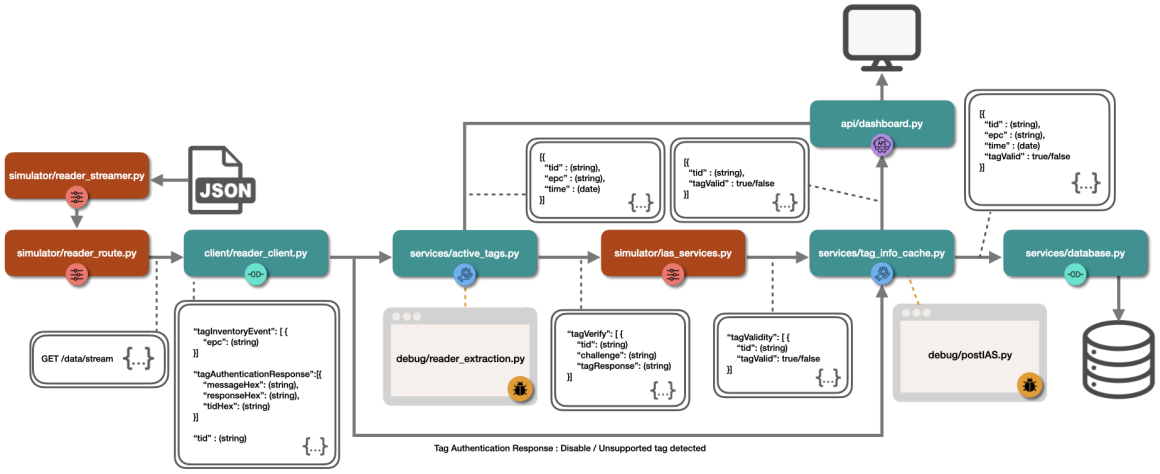
6.4 Simulation Mode

Simulation mode is created so we could test the data pipeline without the actual hardware or services. This could also be used for troubleshooting purposes during integration.

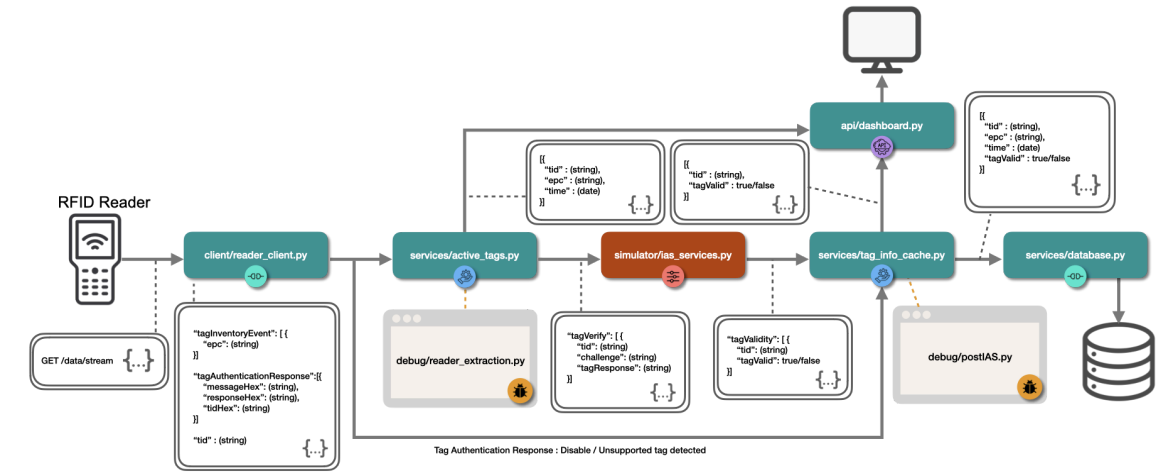
Instruction to configure simulation mode can be found in section 9.4.

Simulation mode available:

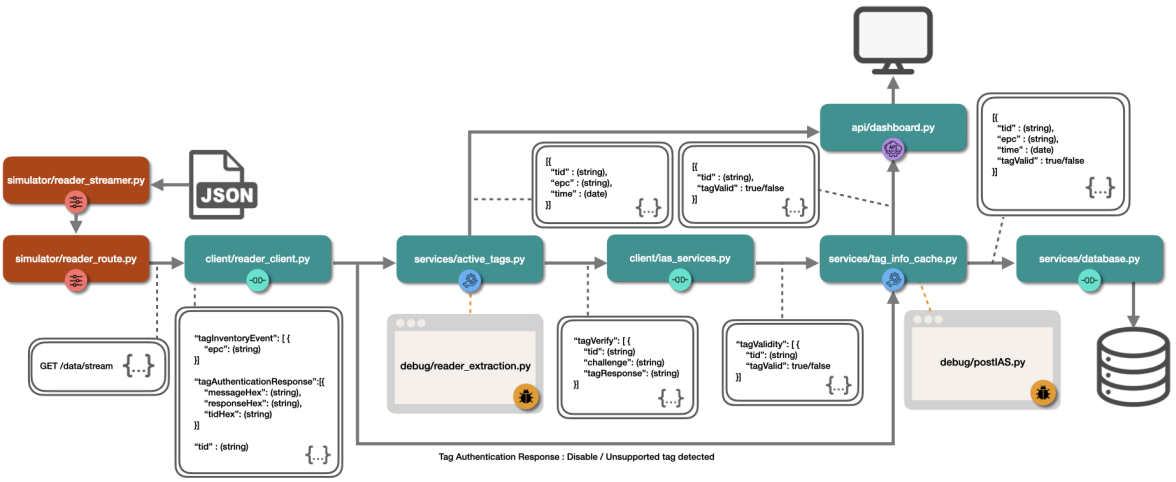
1. Full Simulation Mode (Reader Simulator + IAS Simulator)



2. IAS Simulation (Actual Reader + IAS Simulator)



3. Reader Simulation (Reader simulator + Actual IAS)



7. SOURCE CODE

7.1 Link to Source Code

Source code are made available in the link below:

https://github.com/egantech05/times7project.git

7.2 Source Code Folder Structure

The Folder Structure of this project is designed with intent to make every function to be modular so it can be easily maintained, easy to understand by system integrators and easy to add new features in the future. This application is divided by two main folder:

1. Front end

   Consist of user interface web applications used to interact with the authentication service in real time.

2. Back end (Times-7 Gateway)

   Consist of server application that powers the entire data pipeline. It is responsible for ingesting tag reads, performing authentication with IAS, save scan log into database and provide detected tags to front end.

7.2.1 Front End (Dashboard)

Folders/Files	Descriptions
Apps.js	Primary application access point
package.json	List of packages for this project. Run npm install to install all dependencies
.env.local	Store Supabase URL and KEY Store URL to Gateway This file will not be made available in git for data safety purposes
assets/	Store assets such as images
utils/	Reusable utilities/calculation/conversions

utils/formatDateTime.js	Provides two utility functions for handling dates. **formatDateTime(value)** takes an input timestamp, cleans it up, and ensures it has a timezone; if the input does not include one, it assumes UTC by converting it to an ISO-style string and appending Z. It then parses the value into a JavaScript Date and, if valid, formats it for New Zealand using the en-NZlocale and the Pacific/Auckland timezone in 24-hour time. **todayDate()** simply returns the current local date formatted as YYYY-MM-DD, with the month and day padded to two digits.
src/components/	Reusable UI components
/components/authStatus	UI component that displays an authentication state as a pill-shaped badge. It accepts an auth prop and renders either “VALID” or “INVALID,” switching its background color to green when authenticated and red when it is not.
/components/header	simple top header bar layout for the app. It renders a left section with the Times7 logo image, a centered section containing the **NavBar** component, and an empty right section reserved for future actions. It takes **navigationRef** and active as props and passes them into **NavBar** so the navigation can work and highlight the current active tab.
/components/itemCard	pressable display card component used to display tags on dashboard. It shows the formatted firstSeentime at the top, renders the AuthStatus pill for the auth result, and if the tag is invalid it can also display an error/info message in red. The middle of the card displays the epcHex, and the bottom shows the tidHex in smaller grey text. It supports an onPress handler for tapping the card, applies a pressed animation (slight scale/opacity), and can visually mark items as registered by switching the card background to white.
/components/logs	Renders each log in a readable list.

/components/navbar	App’s primary navigation strip. It renders the set of top-level nav buttons (tabs) and uses the **navigationRef** to trigger route changes when you click one, while the active prop is used to visually highlight which page is currently selected.
/components/navigation	Navigation module that centralizes how the app performs navigation actions. This file exports a shared **navigationRef** so things like **Header**/**NavBar** can trigger route changes from anywhere.
/components/newProduct	Implements the user interface for new product registration. It presents a structured form to capture key product details, performs basic input handling and validation, and provides clear actions to submit or cancel the creation process. Once submitted, the component passes the collected data to the relevant handler or database API integration so the new product can be persisted and reflected in the application state.
/components/searchBar	Reusable search input component. It renders a text entry field with fixed styling, captures user input changes, and forwards the query value through a callback so parent screens can apply filtering logic in real time.
/components/searchResults	Responsible for presenting the output of a search query in a clear, structured list. It receives the filtered or returned dataset from the parent screen, renders each matching item in a consistent card styling, and supports common interactions such as selecting an item to view details. The component also typically handles basic empty-state messaging so users are informed when no results match the current search criteria.
/components/statusDot	Reusable presentational component that displays a small circular indicator to provide a status at a glance. Its main exported function, typically StatusDot(...), receives a status-related prop (such as status, isActive, or connected) and renders a consistently sized dot with the appropriate styling applied. The component commonly includes a helper function (for example getStatusColor(...) to

	map the incoming status value to a specific color, and a styles definition (StyleSheet) that standardizes the dot's shape, size, and spacing while still allowing optional style overrides from the parent.
/components/submitButton	Reusable action button component for triggering submit-type operations. It exports the function **SubmitButton**, which requires the inputs **onPress** (a callback function invoked when the button is pressed) and **label** (string text displayed on the button), and also accepts optional inputs **color** (string) and **disabled** (boolean). The **SubmitButton** function returns a React element that renders an interactive button. When *disabled* is true, the button is non-interactive, and when pressed while enabled it executes the provided onPress handler.
/components/viewItem	Render a React element that displays detailed information for a selected record and provides an edit mode for updating its stored fields and associated photo. To use it, it requires **item** (the selected object used to identify and load the record data) and **onClose** (a callback to close the view). In edit mode, the component enables field updates and photo replacement; within this edit workflow, **pickImage** is used to select a new image, and **saveEdit** persists the edited fields and any newly selected image, then refreshes the displayed data.
src/data/	Connection to database
/data/supabase.js	Supabase configuration module that initializes and exports a ready-to-use Supabase client for the rest of the app. It requires the Supabase project URL and anon key (stored in .env.local file) and returns an initialized client instance that other files import to run database queries, authentication calls, and storage operations. The main function used in this file is **createClient** (called with the project URL and anon key), and the main exported value is **supabase** (the created client object that exposes methods like .from(), .auth, and .storage).
src/hooks/	Reusable state functions/hooks

hooks/useActiveTags.js	Exports the hook function **useActiveTags**, which polls the gateway for the latest active-tag scans and reader connection state, then exposes that data to components in a single return object. To use it, it accepts an optional input object { intervalMs } (number, default 1000) that controls the polling interval, and it returns { scans, gatewayStatus, error, readerConnected }, where scans is the normalized active-tag list, gatewayStatus reflects "connecting", "live", or "error", error holds any fetch error message, and readerConnected indicates whether the reader is currently connected. The main functions inside are **useActiveTags** (the exported hook) and **tick** (the internal async polling function that calls **fetchActiveTags** and **fetchReaderStatus**, updates state, and schedules repeated execution via setInterval).
src/pages/	Page layouts
pages/dashboard	Layout to display detected tags by reader
pages/logs	Layout to display logged scanned activity
pages/search	Layout to search product based on epc number
src/services/	Connection to external applications
services/gatewayClient.js	HTTP client wrapper for talking to the Time7 Gateway API, centralizing the base URL and the request/response parsing so the rest of the UI does not deal with raw fetch calls. Its main exported functions are **fetchActiveTags** and **fetchReaderStatus**: **fetchActiveTags** takes no required input and returns a Promise that resolves to the parsed JSON payload representing the current active-tag list, while **fetchReaderStatus** takes no required input and returns a Promise that resolves to the parsed JSON payload representing the current reader/gateway connection status. Both functions reject (or throw) on non-OK HTTP responses so callers can surface errors cleanly.
test	
src/pages/search/search.index.test.js	This test suite verifies the complete search workflow of the SearchPage component. It mocks the SearchBar and SearchResults components to

	isolate UI behavior and fully mocks the chained Supabase query sequence (from → select → eq → maybeSingle). The tests confirm that an empty query triggers no database calls, that the component searches in the correct order—first by tid, then by uppercase TID, then by epc, and finally uppercase EPC—and that successful hits are rendered correctly in the SearchResults component. When all lookups return null, the UI displays "Item not found," while Supabase errors surface visibly through the page's error handling. The suite relies on React Native Testing Library's render, fireEvent, and waitFor utilities to validate asynchronous state updates and user interactions.
src/hooks/useActiveTags.test.js	This test suite exercises the useActiveTags React hook end-to-end. It mocks the fetchActiveTags gateway client so the hook's behaviour can be driven deterministically, then uses React Testing Library's renderHook, act, and waitFor helpers together with Jest's fake timers. The tests verify the hook's full state machine: it starts in a connecting state, moves to live with populated scans on a successful fetch, and moves to error with a user-friendly message when the fetch rejects. They also confirm that polling happens on the configured interval, that timers are cleaned up correctly on unmount, and that no state updates occur after unmount even if an in-flight request resolves later. Edge cases such as null/undefined responses, error objects without a message field, interval changes (resetting the underlying setInterval), and recovery from an error state back to live on a subsequent successful fetch are all covered to ensure the hook remains robust under real-world conditions.

7.2.2 Backend (Gateway)

Folders/Files	Descriptions
main.py	Primary application access point
.env	Stores Supabase URL and KEY Store reader IP Address

	Store reader username and password
api/	Endpoints connections
api/dashboard.py	Defines the FastAPI dashboard routes that expose the gateway's live runtime state to the frontend. It creates an **APIRouter** named **router** and registers two GET endpoints: 1. **active_tags(request: Request)** at "/active-tags" 2. **reader_status(request: Request)** at "/reader-status". The required input for both functions is the FastAPI **Request** object (injected by FastAPI), and **active_tags** returns a **list[ScanResult]** by reading **request.app.state.active_tags.get_active()** and joining each active tag with cached IAS results from **request.app.state.tag_info_cache.get(t.tidHex)** ; this directly depends on other parts of the system that populate app.state.active_tags and app.state.tag_info_cache. **reader_status** returns a JSON object {"connected": <bool>} based on **request.app.state.reader_connected**, which is maintained elsewhere when the reader connection state changes.
clients/	Connection to external applications
clients/ias_services.py	Currently empty. Defines the interface layer for performing an IAS lookup. This shall be implemented once API for IAS available
client/reader_client.py	Reader-ingestion pipeline by defining the HTTP streaming client for the Impinj reader and the coroutine that consumes reader events to update application state used by the API. The main class is **ImpinjReaderClient**, where **__init__()** creates an authenticated httpx.AsyncClient, **stream_events()** requires no additional inputs and returns an **async generator** that yields parsed JSON events from GET {base_url}/data/stream, and **aclose()** closes the underlying HTTP client. The main orchestration function is **run_reader_stream(app)**, which reads READER_BASE_URL, READER_USER, and READER_PASSWORD from .env file, streams

	events via ImpinjReaderClient.stream_events, extracts tag fields (such as tidHex/epcHex), updates **app.state.active_tags** in order for api/dashboard.py:active_tags can return the live list, and uses **app.state.tag_info_cache** plus **clients/ias_services.py ias_lookup** to populate authentication results; it also calls **services/database.py upsert_latest_tag** to persist the latest tag state and updates **app.state.reader_connected** in order for api/dashboard.py:reader_status can report connection status.
supabase_client.py	Initializes and provides a shared Supabase client instance for database operations. Its main function is **get_supabase()**, which takes **no direct input arguments** but requires the environment variables **SUPABASE_URL** and **SUPABASE_SERVICE_ROLE_KEY** to be set. Used by other backend modules such as **services/database.py** to execute Supabase queries without re-initializing the client each time.
debug/	Script to debug data flow
debug/config.py	Configuration module used by the gateway's debug utilities to keep common settings in one place. It defines two module-level constants, **BASE_URL** (the root URL of the running gateway API) and **REFRESH_SECONDS** (the polling interval used by debug scripts). This file has **no functions or classes** and requires **no inputs** to use; other debug modules import **BASE_URL** and **REFRESH_SECONDS** directly (for example debug/postIAS.py) to build endpoint URLs and control refresh timing, so those values do not get duplicated across multiple debug tools.
debug/postIAS.py	Debug script used to exercise and verify the gateway's IAS-related endpoint by sending HTTP POST requests to the running backend. Its main function is **main()**, which takes no input arguments, builds the request URL using **debug/config.py BASE_URL**, repeatedly sends a POST request with a sample IAS payload, and prints/logs the response so developers can

	confirm if ias_services giving out results.
debug/reader_extraction.py	Debug utility that continuously polls the gateway's pre-IAS active-tag debug endpoint and prints the currently active tags to the terminal for quick verification during reader streaming. It sleeps using **REFRESH_SECONDS** between refreshes and handles KeyboardInterrupt to stop.
debug/routes.py	Debug API router and the endpoints that expose internal runtime state for development and troubleshooting. It exports an **APIRouter** named **router** and registers the handler functions **debug_active_tags** and **debug_post_ias**
models/	Reusable data structures
models/schema.py	Shared data structures as Pydantic model to maintain consistent structure and reusable by any functions
services/	Application internal logics
services/active_tags.py	Act as in-memory active-tag tracker that the gateway uses to maintain a live set of recently seen tags and automatically remove tags that have gone inactive. The main class is **ActiveTags** which will return an instance configured with the inactivity timeout; the key update method is **ActiveTags.sync_seen**, which takes a batch of currently seen tag IDs (and associated fields such as epcHex, messageHex, responseHex) plus an optional timestamp, updates internal state, and returns the set of newly added tag IDs. The retrieval method **ActiveTags.get_active()** returns the current active-tag list, and the maintenance method **ActiveTags.remove_inactive** remove tags whose last_seen exceeds the configured grace period. This service is updated by clients/reader_client.py run_reader_stream, and its output is consumed by api/dashboard.py active_tags and debug/routes.py debug_active_tags to serve live tag data to the frontend and debug tools.
services/database.py	Storing the latest tag state in Supabase, so live reader events can be recorded and later queried

	independently of in-memory state. Its main function is **upsert_latest_tag(**) which requires the inputs tag_id, seen_at, auth, and info, and returns None after performing an upsert into the database table that holds the “latest seen” record for each tag. This function depends on **clients/supabase_client.py get_supabase** to obtain the initialized Supabase client, and it is called by **clients/reader_client.py run_reader_stream** after IAS resolution and cache updates so the persisted record stays aligned with the in-memory ActiveTags state and the dashboard responses.
services/tag_info_cache.py	In-memory cache for IAS lookup results so the gateway does not repeatedly call IAS for the same tag within a short time window and so the dashboard can quickly enrich active tags with authentication metadata. The main class is **TagInfoCache** returns a cache instance configured with a TTL (time to live).The primary methods are **TagInfoCache.get()**, which takes a tag ID and returns the cached entry or None if missing/expired. **TagInfoCache.set()** stores a result for a tag and returns None. **TagInfoCache.prune()** removes expired entries and returns None. This cache is populated by **clients/reader_client.py run_reader_stream** after calling **clients/ias_services.py ias_lookup,** and it is read by **api/dashboard.py active_tags** and **debug/routes.py** to attach the latest auth status and message to each active tag in API responses.
simulators/	Application simulators for testing purposes
simulators/ias_services.py	Provides a simulated IAS implementation used when running the gateway in mock mode, allowing the pipeline to produce deterministic authentication outcomes without connecting to the real IAS service. The main function is **ias_lookup()**, which requires the input tag_id (an AuthPayload) and returns a 2-tuple (auth, info) where auth is a boolean authentication result and info is the corresponding message string. This function is called from the same reader-processing flow that calls the real clients/ias_services.py ias_lookup, and its output

	is stored in services/tag_info_cache.py TagInfoCache.set and then surfaced through api/dashboard.py active_tags and debug/routes.py so the frontend and debug tools behave consistently in simulation and production modes.
simulators/reader_route.py	FastAPI router that simulates an Impinj reader by exposing an endpoint that streams synthetic reader events in the same shape as the real /data/stream feed. It exports an **APIRouter** named **router** and the main handler function **reader_stream()**, which takes no explicit input arguments from callers and returns a **StreamingResponse** that emits NDJSON-formatted tag events over time. This route is used by **clients/reader_client.py ImpinjReaderClient.stream_events**, allowing the normal ingestion pipeline (run_reader_stream, services/active_tags.py, services/tag_info_cache.py, and api/dashboard.py) to be exercised end to end without connecting to physical reader hardware.
simulators/reader_streamer.py	FastAPI-based simulator for the reader's NDJSON event feed by exposing a streaming endpoint that serves lines from a local .ndjson file at a configurable rate. It defines **ndjson_line_stream()**, which takes loop and rate_hz, reads from **DATA_FILE** (the selected datastream*.ndjson), yields each line as UTF-8 bytes with an optional delay, and either repeats or stops based on loop. It also defines the route handler **data_stream()** registered on **router.get("/data/stream")**, which takes the same query inputs and returns a **StreamingResponse** wrapping ndjson_line_stream. This simulator is consumed by **clients/reader_client.py ImpinjReaderClient.stream_events**, which connects to /data/stream so the normal ingestion pipeline can be exercised without physical reader hardware.
test/	Unit test functions

test/test_active_tags.py	This test module validates the full behavior of the ActiveTags service, which is responsible for managing active RFID tag state and applying time-based expiry rules. The tests verify the _utc helper to ensure that naive datetimes are treated as UTC and that timezone-aware datetimes are correctly converted to UTC. They also exercise the core sync_seen() logic: detecting newly observed tag IDs, updating first_seen and last_seen timestamps consistently, and ensuring that repeated observations do not produce duplicate entries. The expiry mechanism is checked by advancing the logical clock beyond the configured grace period and confirming that inactive tags are removed. Finally, ordering behavior is validated through get_active(), confirming that active tags are sorted in descending order of their first_seen timestamps. This suite ensures the service behaves deterministically under different timing scenarios and maintains a correct, minimal representation of currently active tags.
test/test_reader_client.py	This unit-test module exercises the time7_gateway.clients.reader_client integration end-to-end in isolation. It first covers ImpinjReaderClient, asserting that the constructor stores the reader base URL, builds an httpx.AsyncClient with the correct Basic Auth credentials, and that stream_events hits the /data/stream endpoint, calls raise_for_status, skips empty lines, parses JSON correctly, and invokes the optional on_connect callback before yielding events; it also verifies that aclose closes the underlying async client. The handle_invalid_tag group checks that invalid tag events are routed through active_tags.sync_seen, that the tag info cache is updated with auth=False and an info message, and that upsert_latest_tag is called with the expected fields without raising exceptions. The run_reader_stream tests then drive the main async loop using patched ImpinjReaderClient.stream_events, a fake FastAPI app.state, and environment variables, to confirm all filtering and side-effect behaviour: non-tagInventory events and empty tidHex are

	ignored; missing tagAuthenticationResponse or empty responseHex are treated as invalid and never call ias_lookup; when inner tidHex is missing the outer tidHex is used; IAS cache hits bypass ias_lookup, cache misses call it exactly once and persist the result both in the cache and via upsert_latest_tag; valid events always call active_tags.sync_seen with the correct tidHex/epcHex mapping and construct AuthPayload with the right messageHex, responseHex, and tidHex. Finally, the suite asserts that app.state.reader_connected is toggled True on connect and back to False in the finally block, and that ImpinjReaderClient.aclose is awaited both on normal completion and when stream_events raises an exception, giving strong confidence that the reader client's networking, caching, database writes, and lifecycle management are all wired up correctly.
test/test_tag_info_cache.py	This unit-test module validates the behaviour of the TagInfoCache service used to cache IAS tag-auth results. It first checks the basic contract that a cache miss returns None by instantiating a fresh TagInfoCache and calling get on an unknown key. It then verifies the happy path by calling set with auth=True and an info string and asserting that a subsequent get returns the (auth, info) tuple exactly as stored. To exercise TTL expiry logic deterministically, the tests monkeypatch the datetime class inside the tag_info_cache module: they create a fake now() that returns a time 25 hours after an initial fetched_at timestamp, proving that get both returns None and deletes the entry once it is older than the configured 24-hour cache_ttl_hours. A complementary test sets now() to only 23 hours later and confirms that the value is still returned and the key remains in the internal _cache dict. Finally, the suite covers the snapshot API in both forms: when snapshot() is exposed as an instance method and when it is imported as a module-level function. In both cases it builds a cache with entries "b" and "a", asserts that the snapshot's count is 2, and that the items list is sorted by ID (["a", "b"]), demonstrating that callers can rely on a stable, ordered view of the cache

	contents.
utilities/	Consist of functions to simulate IAS services
utilities/simulate_encryption.py	Provides the mock cryptography used by the simulator tooling to generate a reproducible authentication response from a tag identifier and a challenge message. Its main function is **generate_response()** which requires the inputs tidHex and messageHex and returns a hex string representing the simulated responseHex. This function is called by simulators/ias_services to check if incoming tags have the same response.

8. INSTALLATIONS

8.1 Pre-requisites

Install the following software:

- Python 3.10+
- pip
- venv
- node.js
- npm

8.1 Front End

1. Install dependencies

   ```
   cd frontend
   npm install
   ```

2. Configure environment variables

   Create .env.local file inside:

   frontend/.env.local

   Add the following into .env.local file:

   ```
   EXPO_PUBLIC_SUPABASE_URL=your_supabase_project_url
   EXPO_PUBLIC_SUPABASE_PUBLISHABLE_KEY=your_supabase_anon_or_publishable_key
   EXPO_PUBLIC_GATEWAY_URL=http://localhost:8000
   ```

8.2 Back End

1. Create virtual environment

   ```
   cd backend/Time7_Gateway/time7_gateway
   Python -m venv .venv
   Source .venv/bin/activate
   ```

2. Install dependencies

   ```
   pip install -r requirement.txt
   ```

3. Configure environment variables
   Create .env file inside:

   backend/Time7_Gateway/time7_gateway/.env

   Add the following into .env file:

```
SUPABASE_URL=your_supabase_project_url
SUPABASE_SERVICE_ROLE_KEY=your_supabase_service_role_key
READER_BASE_URL=http://localhost:8000
READER_USER=root
READER_PASSWORD=impinj
```

8.3 Database

Setup the database with the following SQL code:

```
CREATE TABLE public.data (
  tid_hex text NOT NULL,
  first_seen timestamp without time zone NOT NULL,
  auth boolean,
  info text,
  epc_hex text,
  CONSTRAINT data_pkey PRIMARY KEY (tid_hex)
);
CREATE TABLE public.product_info (
  epc text NOT NULL,
  description text,
  origin text,
  produced_on date,
  CONSTRAINT product_info_pkey PRIMARY KEY (epc)
);
CREATE TABLE public.product_photo (
  id uuid NOT NULL DEFAULT gen_random_uuid(),
  photo_url text NOT NULL,
  is_primary boolean NOT NULL DEFAULT false,
  created_at timestamp with time zone NOT NULL DEFAULT now(),
  epc text NOT NULL,
  CONSTRAINT product_photo_pkey PRIMARY KEY (id),
  CONSTRAINT product_photo_epc_fkey FOREIGN KEY (epc)
REFERENCES public.product_info(epc)
```

```
);
```

9. RUNNING APPLICATION

9.1 Front End

Run the front end application by running the following command in the main folder:

```
cd frontend/
npm run start
```

9.2 Back End

Run the front end application by running the following command in the main folder:

```
cd backend/Time7_Gateway
```

Run the following if the connection log is not required. This will reduce the load on the unicorn:

```
uvicorn time7_gateway.main:app --log-level warning --no-access-log
```

Run the following if connection log is required:

```
uvicorn time7_gateway.main:app
```

9.3 Testing

This section describes how to execute the testing environment for both the backend and frontend, as well as a technical overview of each unit-test module included in the project.

9.3.1 Backend Testing

1. Navigate to the backend directory

```
cd backend/Time7_Gateway/time7_gateway
```

2. Run the backend test suite

```
pytest -v
```

9.3.2 Frontend Testing

1. Navigate to the frontend directory

```
cd frontend/
```

2. Run the frontend test suite

```
npm test
```

9.4 Simulation

9.4.1 Enable Reader Simulator

1. Configure URL for simulator:

   In **backend/Time7_Gateway/time7_gateway/.env,**

   Comment on actual reader URL to disable actual reader connection and remove comment on localhost (endpoint for reader simulator)

   ```
   #READER_BASE_URL=http://impinj-13-fb-57.local/api/v1
   READER_BASE_URL="http://localhost:8000"
   ```

   *vice versa to disable reader simulator to get data from actual reader
   *if connecting to the reader on Windows, remove .local at the end of the reader IP address. Include .local if running macOS.

2. Configure data source

   Ensure that data that need to be simulated are located in
   backend/Time7_Gateway/time7_gateway/simulators

   and update the path of the file accordingly in

   backend/Time7_Gateway/time7_gateway/simulators/reader_streamer.py

   ```
   DATA_FILE = Path(__file__).with_name("datastream1.ndjson")
   ```

9.4.2 Enable IAS Simulator

In **backend/Time7_Gateway/time7_gateway/main.py,**

Change IAS_MODE to “**mock”**

```
# IAS switch (mock vs real)
ias_mode = os.getenv("IAS_MODE", "mock")
app.state.ias_lookup = real_ias_lookup if ias_mode == "real" else mock_ias_lookup
```

*set IAS_MODE to “real” to use actual IAS services.

9.5 Debug Tool

9.5.1 Data Extraction

Created to confirm the results of data extraction from the reader stream. It is returning the tags that is stored in **active_tags**

backend/Time7_Gateway/time7_gateway/debug/reader_extraction.py

Command to run debug/reader_extraction.py

```
python -m time7_gateway.debug.reader_extraction
```

9.5.2 IAS Results

Created to check the results of IAS services. It is returning the tags that is stored in **tag_info_cache**

backend/Time7_Gateway/time7_gateway/debug/postIAS.py

Command to run debug/postIAS.py

```
python -m time7_gateway.debug.postIAS
```

10. COMPLETED USER STORIES/FEATURES

RFID Tag Ingestion

Capability	*Description*	*User*	*Story*	*Reason*
Live Reader Ingestion	**Ingests live RFID tag streams from an Impinj compatible reader**	Times-7 gateway system	I want to subscribe to a live RFID tag data stream from an Impinj-compatible reader	So that detected tags can be processed in real time for authentication
Payload Decoding	**Decodes raw RFID payloads into structured tag data**	Times-7 gateway system	I want to decode raw tag payloads into structured tag data	So that authentication requests can be generated correctly
Recorded Stream Support	**Supports replayed or simulated tag streams for development/testing**	Developer	I want to Ingest recorded RFID tag streams	So that development can continue without live hardware

IAS Authentication Handling

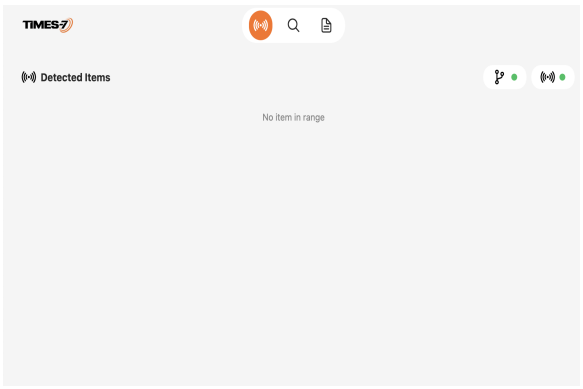
Capability	*Description*	*User*	*Story*	*Reason*
IAS Response Handling	**Processes authentication responses from IAS**	Times-7 gateway	I want to process authentication responses from IAS	So that authenticity results can be displayed and logged
Invalid Tag Detection	**Identifies invalid or unregistered tags and does not send them to IAS**	System user	I want to see when a scanned tag is invalid or unknown	So that unauthenticated products are immediately visible

Times-7 Dashboard

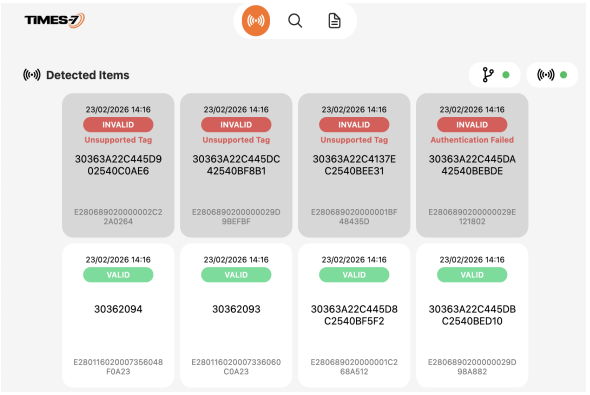
Capability	*Description*	*User*	*Story*	*Reason*
Live Item Display	**Displays real-time detected items from reader**	Times-7 Dashboard User	I want to see all currently detected items in real time	So that I can verify authenticity during scanning
Historical logs	**Stores and displays historical scan events**	Times-7 Dashboard User	I want to view previously scanned items	So that I have an audit trail of authentication events
EPC Search	**Allows users to search previously detected items by EPC**	Times-7 Dashboard User	I want to search for a specific item's authentication history	So that I can quickly retrieve its authentication history
Product Metadata Editing	**Allows users to edit product database information**	Times-7 Dashboard User	I want to associate product information with an EPC	So that authentication results are easier to interpret

Completed user stories:

Live Item Display:

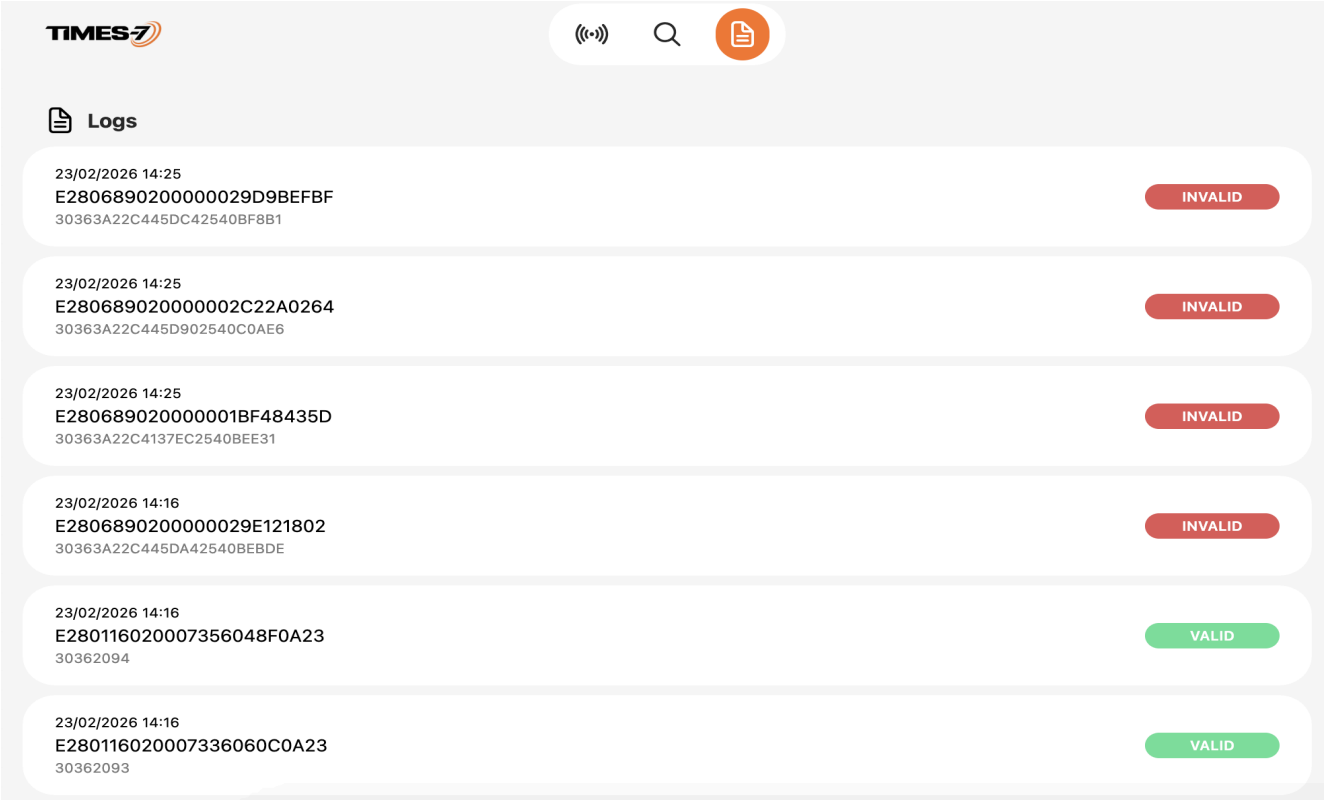


1. *No Items in range of reader*



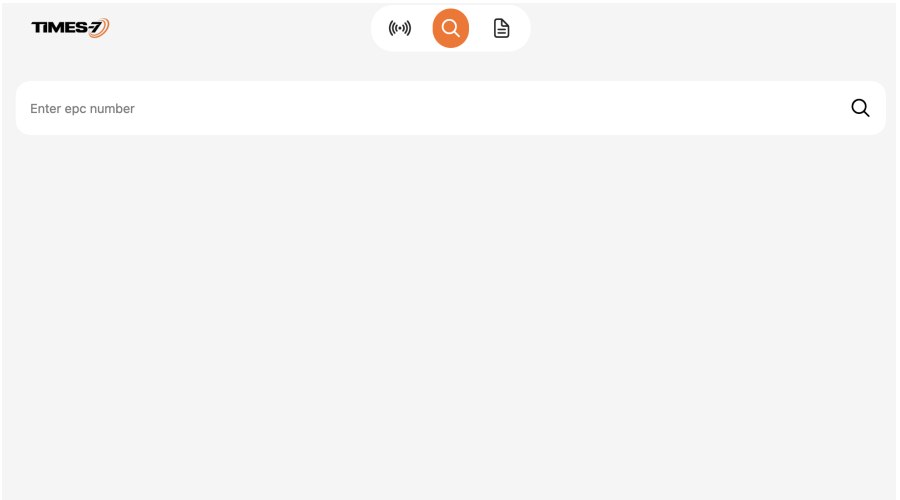
2. *Items detected in real time*

Historical Logs:

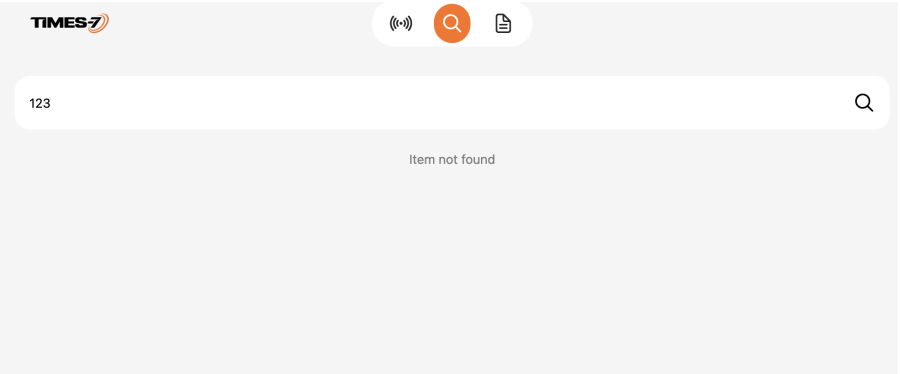


1. *User navigates to the logs page by clicking the 'document' icon in the navigation bar (highlighted in orange above)*

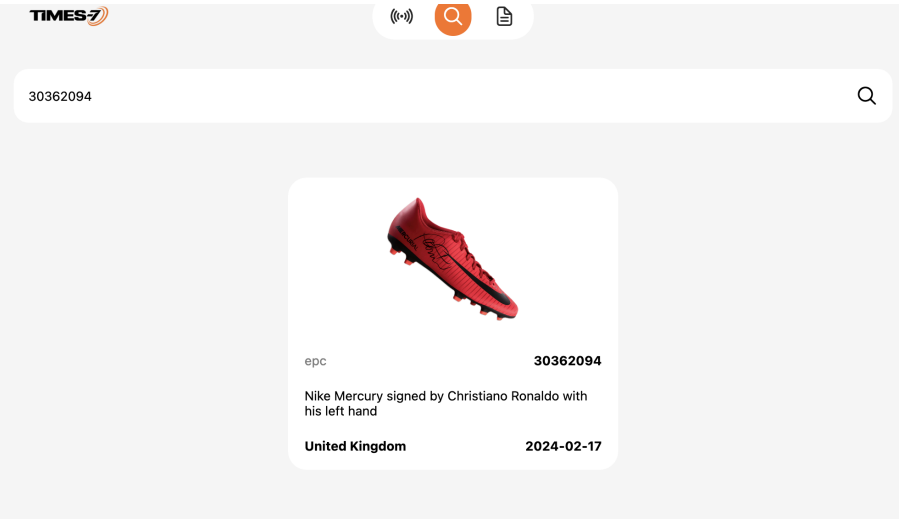
EPC Search:



1. *User navigates to the search page by clicking the 'magnifying glass' icon in the navigation bar (highlighted in orange above)*

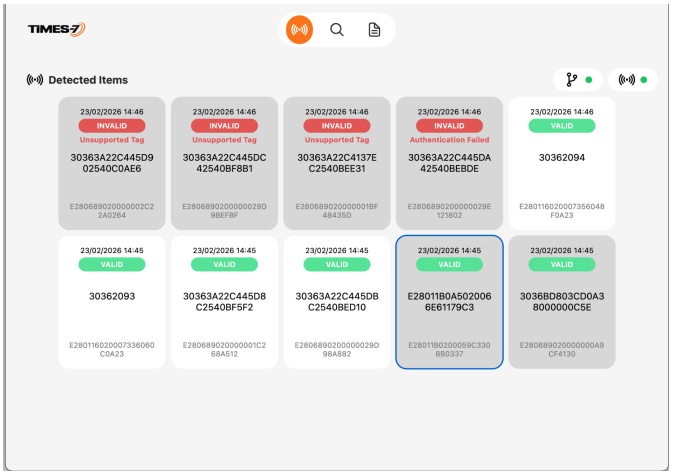


2. *User searches for an item that does not exist in the Times-7 dashboard database*

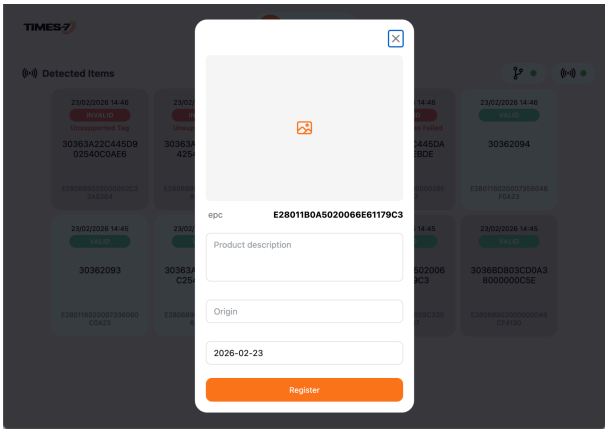


3. *User searches for an item that exists in the Times-7 dashboard database*

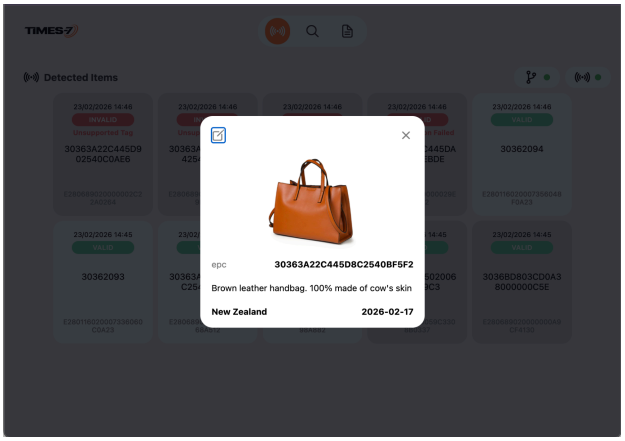
Product Metadata Editing:



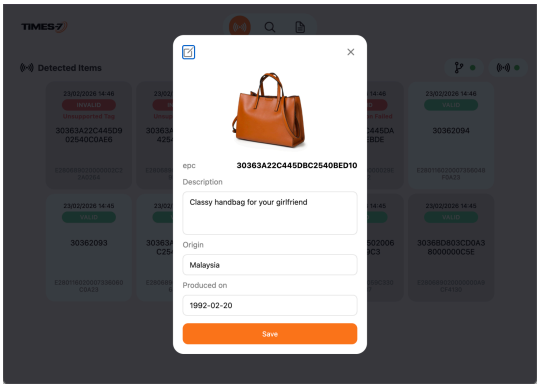
1. *User clicks a valid item box to edit information*



2. *‘Register item’ popup modal. This modal appears when there is no user-entered product info registered to the dashboard database*



3. *‘Existing item’ popup modal. This modal appears when the selected item has already been registered to the Times-7 dashboard database*



4. *‘Edit existing item’ modal. This appears when a user edits a registered item by clicking the ‘edit’ icon in the top left-hand corner of the ‘existing item’ modal*

11. INCOMPLETE USER STORIES/FEATURES

Capability	*Description*	*User*	*Story*	*Reason*
Direct IAS API Calls	**Gateway does not yet independently initiate direct IAS API calls**	Times-7 Gateway System	I want to send detected tagIDs to Impinj Authentication Service	So that product authenticity can be verified
External Authentication API	**No exposed public-facing authentication API for third-party integrations**	Future customer system	I want to retrieve authentication results via an API	So that I can integrate Times-7 authentication into my own systems

These features were not implemented due to unavailability of latest IAS API documentation and no access to IAS system.

12. VALIDATION OF TOP 3 TECHNICAL DECISIONS

12.1 Stabilizing Real-Time Tag Detection with In-Memory Buffer

The problem we encountered was that raw RFID inventory events arrive at a very high frequency and are often repetitive for the same tag within short time windows. Writing every event directly to the database created unnecessary load and amplified the impact of transient reads. On the user interface side, the dashboard was also prone to a flickering effect where tags appeared and disappeared rapidly due to brief gaps in reads, network jitter, or bursty scan patterns. As a result, the system could feel unstable even when the reader was functioning correctly, and the database workload increased without adding meaningful value to the operator.

To solve this, we introduced two in-memory buffers, ActiveTags and tag_info_cache. ActiveTags maintains a rolling view of which tags are currently present by tracking each tag's first seen and last seen timestamps and applying a short grace period before removal. This creates a stable tag presence signal for the dashboard, because a tag is not immediately removed during short read gaps, reducing UI flicker and improving operator trust in the detected tags screen. In parallel, tag_info_cache stores recently resolved tag metadata and authentication-related outcomes, which prevents repeated sightings of the same tag from triggering duplicate processing. This includes avoiding repeated IAS authentication checks for tags that have already been evaluated within the cache window, reducing unnecessary external calls and keeping authentication logic efficient under continuous scanning. Together, these components act as a controlled buffer between the high-frequency reader stream, the authentication workflow, and the database.

The result is a system that remains responsive while significantly reducing unnecessary database activity and redundant authentication work. The dashboard displays a smoother and more reliable set of detected tags, because active presence is derived from a consistent TTL-based model rather than raw event volatility. At the same time, cached tag data and controlled update patterns prevent repeated database writes, repeated IAS checks, and redundant lookups, which improves overall throughput and makes the platform more resilient under continuous scanning conditions.

12.2 Minimizing Unnecessary IAS Service Calls

The issue we identified was that the reader data stream does not always contain usable authentication information. In some configurations, authentication response is not enabled on the reader, so the stream will never include the required authentication payload. In other cases, the stream may include the authentication response structure but provide an empty responseHex, which indicates the tag is not compatible with IAS authentication. If the backend blindly calls IAS for every detected tag without validating these conditions, it generates unnecessary external requests and wasted processing. This is particularly important because IAS calls can be billable per request, so redundant or futile calls directly increase operational cost without improving authentication outcomes.

To address this, we implemented stream-based gating rules before invoking IAS. The backend first checks whether the authentication response fields exist in the incoming event; if they are absent, it is treated as a reader configuration issue where authentication response is not enabled, and the IAS workflow is skipped. If the fields exist and responseHex is present, the logic then evaluates its value: when responseHex is non-empty, the tag is considered to already have an authentication response available in the stream and therefore does not need an IAS authentication check. When responseHex is empty, the tag is treated as not IAS-compatible, so the system avoids making an IAS call that cannot succeed.

The result is fewer unnecessary IAS service calls and lower operational cost, while improving overall throughput and stability. The backend only invokes IAS when the stream indicates it is both required and meaningful, and it avoids wasting network calls on tags that already contain a response or cannot be authenticated due to incompatibility or missing reader-provided authentication data.

12.3 Dedicated Stream Termination to Improve Application Stability

The primary issue was that the RFID reader event feed is delivered through a long-lived streaming connection, whereas the FastAPI server is optimized for short request–response operations and is commonly restarted during development or scaled using multiple worker processes. When the API server also manages the /data/stream background task, the stream becomes tightly coupled to the application lifecycle. This coupling can result in duplicated stream consumers, interrupted connections during reloads or deployments, and unhandled background task failures, ultimately producing inconsistent event ingestion and unreliable dashboard updates.

To address this, the /data/stream feed was treated as the authoritative source of reader events and terminated within a dedicated stream-ingestion service that operates independently of the API process. This ingestion service maintains the persistent connection, implements reconnection and backoff logic, parses incoming events, and publishes normalized updates to shared storage. The FastAPI server then consumes this stored state to serve dashboard requests and monitoring endpoints, without directly owning the streaming connection.

As a result, the architecture became more reliable and scalable. Stream stability improved because the connection is managed by a single controlled process with explicit recovery behavior. In parallel, the API layer can be scaled horizontally without risking multiple competing stream consumers, and routine restarts or deployments no longer disrupt the event pipeline.

13. TECHNICAL CHALLENGES

13.1 High-frequency Event Ingestion

High-frequency event ingestion becomes difficult because RFID readers can produce a flood of repetitive inventory events for the same tag, often multiple times per second and across multiple antennas. If the application treats every raw event as equally important, the system quickly wastes resources doing the same work repeatedly. This leads to CPU pressure from parsing and validation, increased memory usage from excessive processing, and heavy database write amplification if each read is persisted, which can degrade performance exactly when the reader activity is highest.

The ingestion layer also needs to be resilient because the stream is typically a long-lived connection that must remain stable for long periods. Real deployments have disconnects, intermittent network issues, reader restarts, and timeout behavior that can cause naive implementations to crash, restart too aggressively, or accidentally spawn multiple stream consumers. A robust design requires explicit lifecycle ownership, reconnection logic with controlled backoff, and clear error handling so the system can distinguish between “no tags detected” and “stream failure.”

The core challenge is reducing redundancy without breaking the timing semantics that the application depends on, such as first-seen, last-seen, and presence or absence. Effective solutions usually introduce stateful aggregation by maintaining a tag map in memory, updating last-seen timestamps continuously, and emitting meaningful updates only when state changes or on a controlled cadence. This preserves correct presence behavior while preventing downstream bottlenecks, especially when database writes or authentication lookups are slower than the incoming event rate, and it keeps the stream consumer responsive under peak load.

13.2 Tags State Management

State management for active tags is technically challenging because the system must infer physical presence from intermittent, probabilistic observations. A single read only confirms that a tag was detected at a specific time, not that it remains within the read zone afterwards. Maintaining an accurate “active” view therefore requires a defined expiry model, typically implemented using a time-to-live policy with a configurable grace window, such as removing tags that have not been observed for two seconds. This approach must support consistent updates to first-seen and last-seen timestamps and provide deterministic rules for when a tag transitions between active and inactive states.

Correctness depends heavily on time handling and event ordering. Reader payloads may include multiple timestamp fields with different meanings, and events can be delayed, duplicated, or delivered out of order due to buffering and network conditions. The active-tag logic must normalize timestamps to a single reference standard, commonly UTC-aware datetimes, and avoid incorrect removals caused by late-arriving events or local timezone inconsistencies. Additionally, the removal mechanism must be robust against brief gaps in reads caused by RF interference or tag orientation changes, where a tag may remain physically present but temporarily undetected. Without careful TTL tuning and consistent timestamp normalization, the system can oscillate between active and non-active producing unstable dashboards and unreliable downstream processing.

The problem is compounded in multi-antenna or multi-reader deployments, where a single tag may be detected concurrently by several sources with different signal characteristics and update rates. Presence determination must therefore be reconciled across sources to avoid contradictory states, such as one antenna declaring the tag inactive while another continues to observe it. A reliable implementation typically maintains per-source last-seen timestamps and computes an aggregate active status based on the most recent observation across all sources, while preserving source-specific metadata for diagnostics and analytics. This reconciliation layer must also account for race conditions in concurrent updates and ensure that state transitions are applied atomically, so the system remains consistent under high event rates and parallel processing.

13.3 Authentication Workflow Integration Without Excessive Calls

Integrating an authentication workflow such as IAS introduces a separate reliability and performance dimension because authentication is inherently slower and more constrained than raw tag detection. Reader inventory events may arrive at very high frequency, whereas authentication services typically impose non-trivial latency and enforce rate limits or usage-based costs. The system must therefore treat authentication as a controlled operation rather than a per-event action. A primary technical decision is defining the trigger conditions for authentication, such as authenticating only on first sighting, on a defined refresh interval, or only when a tag’s state changes, while ensuring the chosen policy aligns with the required assurance level and the operational characteristics of the environment.

Preventing excessive calls requires a deliberate caching and deduplication strategy that remains correct under concurrency. If the same tag is repeatedly observed, a naive implementation may issue repeated authentication requests, creating a traffic surge that saturates the external service and degrades the ingestion pipeline. A robust approach typically includes a cache keyed by tag identity and relevant parameters, with explicit expiry rules and a mechanism to coalesce concurrent requests so only one in-flight authentication is performed per tag at a time. The implementation must also decide whether cached results are held only in

memory for low latency or persisted to a database to survive process restarts, and it must ensure cache invalidation is deterministic so stale or incorrect authentication outcomes are not reused beyond their acceptable lifetime.

The workflow must also define clear correctness rules for partial and failure scenarios, because authentication responses may be missing, malformed, or temporarily unavailable due to network errors or service side issues. The application must establish explicit status states such as authenticated, unauthenticated, unknown, or error, and ensure downstream components interpret these states consistently. Retrying logic must be bounded and policy driven to avoid amplifying transient failures into sustained load, and auditability must be preserved so operators can distinguish between a genuine authentication failure and an operational outage. These definitions are essential to maintaining predictable system behavior and preventing authentication from becoming a single point of instability in an otherwise real-time tagging pipeline.

13.4 Concurrency and Reliability in Async Systems

Concurrency and reliability are significant challenges in asynchronous systems because long-lived streaming workloads do not align naturally with request–response web frameworks. A reader stream is effectively a continuously running process, whereas FastAPI endpoints are short-lived and typically scaled by running multiple workers. If the streaming task is started implicitly within the API server lifecycle, it can be interrupted by reloads, duplicated across workers, or terminated when a worker is recycled. This creates instability where the system may appear operational but is no longer ingesting events reliably, or it may inadvertently create multiple concurrent consumers competing for the same reader stream.

Asynchronous execution also introduces failure modes that are easy to miss during development and difficult to diagnose in production. Background tasks can raise exceptions that are not propagated to the main application flow, resulting in silent stoppages unless explicit supervision and structured logging are implemented. Shared state structures such as active tag registries and authentication caches are particularly sensitive, because concurrent updates can cause race conditions, stale reads, and inconsistent state transitions. Under high event rates, small timing differences can produce non-deterministic behavior, including intermittent presence flapping, duplicate authentication calls, or conflicting database updates.

A reliable design requires an explicit ownership model and disciplined lifecycle management. The stream consumer should have a single, well-defined owner, typically a dedicated ingestion service or a single-worker process with predictable startup and shutdown hooks. Shared state must be protected through appropriate concurrency controls, such as using the event loop consistently, avoiding cross-thread mutation, and applying locks or queues where required to serialize

critical updates. This separation ensures that the API layer remains horizontally scalable for client requests, while the ingestion layer remains stable, observable, and deterministic in its handling of continuous streaming and state updates.

13.5 Development Constraints Due to Limited Access to Physical Hardware and IMPINJ Authentication Services

Testing and validation are significantly constrained when the development environment does not include access to the physical RFID reader and the live IAS authentication service. Without the reader, the application cannot be exercised against real streaming behaviour such as connection lifecycles, event burst patterns, duplicated reads, intermittent gaps, and the timing characteristics that determine active tag presence. This makes it difficult to verify that ingestion, deduplication, and TTL based presence logic behave correctly under realistic load, and it increases the risk that software which appears correct in unit tests will fail when exposed to real RF driven event patterns.

The absence of the live IAS service further limits verification of the authentication workflow and its operational safeguards. Authentication calls introduce latency, may be rate limited, and can fail in ways that materially affect system behaviour, including timeouts, malformed payloads, partial responses, or intermittent outages. Without the real service, it is not possible to accurately validate retry policies, caching correctness, request coalescing, and the end-to-end impact of authentication on throughput and responsiveness. This also prevents meaningful performance tuning, because the most critical bottleneck is often the external dependency rather than local processing.

To mitigate these constraints, the system must include simulation and diagnostic tooling that reproduces production like conditions with controlled inputs. A reader simulator can replay captured streams to emulate high frequency inventory events, connection interruptions, and representative payload variations, enabling deterministic testing of ingestion and presence logic. Similarly, mock IAS endpoints can emulate latency distributions, failure modes, and response structures, allowing validation of caching, rate control, and failure handling without relying on external availability. Complementary debug tools and endpoints provide visibility into internal state such as active tag maps, authentication cache entries, and pipeline timing, enabling engineers to validate correctness and troubleshoot behaviour in a repeatable manner before integrating with the real reader and IAS environment.

14. FUTURE ENHANCEMENT

14.1 GS1 Integration to Decode EPC

A future enhancement is to integrate GS1 decoding so that raw EPC values can be automatically translated into standardized GS1 identifiers and meaningful product attributes. This would reduce manual interpretation of tag data, improve data consistency across systems, and make search and reporting more accurate by storing both the raw EPC and its decoded GS1 representation.

14.2 Centralized Data Storage and Multi-Site Visibility

Moving scan logs, product metadata, and authentication history into the cloud would provide a unified view across all readers and deployment sites. This enables cross-site traceability, global inventory visibility, and centralized monitoring for distributed warehouses or retail networks.

14.3 Cloud Dashboard for Real-Time Monitoring

The Dashboard could be extended into a cloud-hosted web application, allowing users to log in from any device and gain visibility into authentication activity, recent scans, and operational status without requiring local installations.

14.5 Cloud API for Third-Party System Integration

Providing a public REST API would allow customers or partners to query authentication results, scan logs, and product metadata directly from their own systems, supporting ERP, WMS, and supply-chain applications.

14.6 Role-Based Access Control (RBAC) in the Cloud

A cloud-hosted system enables centralized user management, where different users (e.g., supplier, operator, auditor, admin) can be granted different levels of access to product registration, editing, or reporting functions.

14.7 Cloud Analytics and Insights

With centralized event data, the platform can provide higher-level analytics, such as scan frequency analysis, anomaly detection, usage trends, and product movement heatmaps, enhancing the long-term value of Times-7’s authentication offering.

15. TABLE OF CONTRIBUTIONS

Activity	Contributors
Product research and development	Tane, Rebekka, Laura, Egan
Dashboard architecture design	Laura, Egan
Design and development of live tag display	Egan
Design and development of product search function	Laura
Design and development of log page	Rebekka, Egan
Design of data pipeline	Rebekka, Tane, Egan
Live reading ingestion	Tane, Egan
Development of debug tool	Egan
IAS response handling	Tane
Data stream control function	Tane
Payload decoding	Tane
Design and implementation of database	Laura
Reader simulation tool	Tane, Egan
IAS Simulation tool	Tane
Stabilizing real time tag detection within memory buffer	Tane, Rebekka, Egan
Automated Testing	Rebekka