**Data driven systems**

**5CM506**

**Logbook**

**100670504**

**GITHUB -** [kas-rabur/NHS-PRS](https://github.com/kas-rabur/NHS-PRS)

(my zip file was too large for the submission point, lecturer informed)

[Structured data (Microsoft SQL Server) 1](#_Toc708205544)

[ERD diagram 2](#_Toc811066785)

[Entity tables 2](#_Toc1025480030)

[Attributes and design choice justification 3](#_Toc786976303)

[Relationships and cardinality 4](#_Toc145888723)

[Unstructured data/ noSQL (MongoDB) 5](#_Toc1514817312)

[NoSQL schema 5](#_Toc2001708688)

[Entity tables 9](#_Toc1507163505)

[Attributes and design choice justification 10](#_Toc372461909)

[Scalability 11](#_Toc1925311519)

[API specifications and Security 11](#_Toc1107843619)

[Endpoint list 12](#_Toc1344918021)

[Authentication flow 15](#_Toc619260794)

[Mock-ups 15](#_Toc387358015)

[Authentication, authorisation and security plan 18](#_Toc196766321)

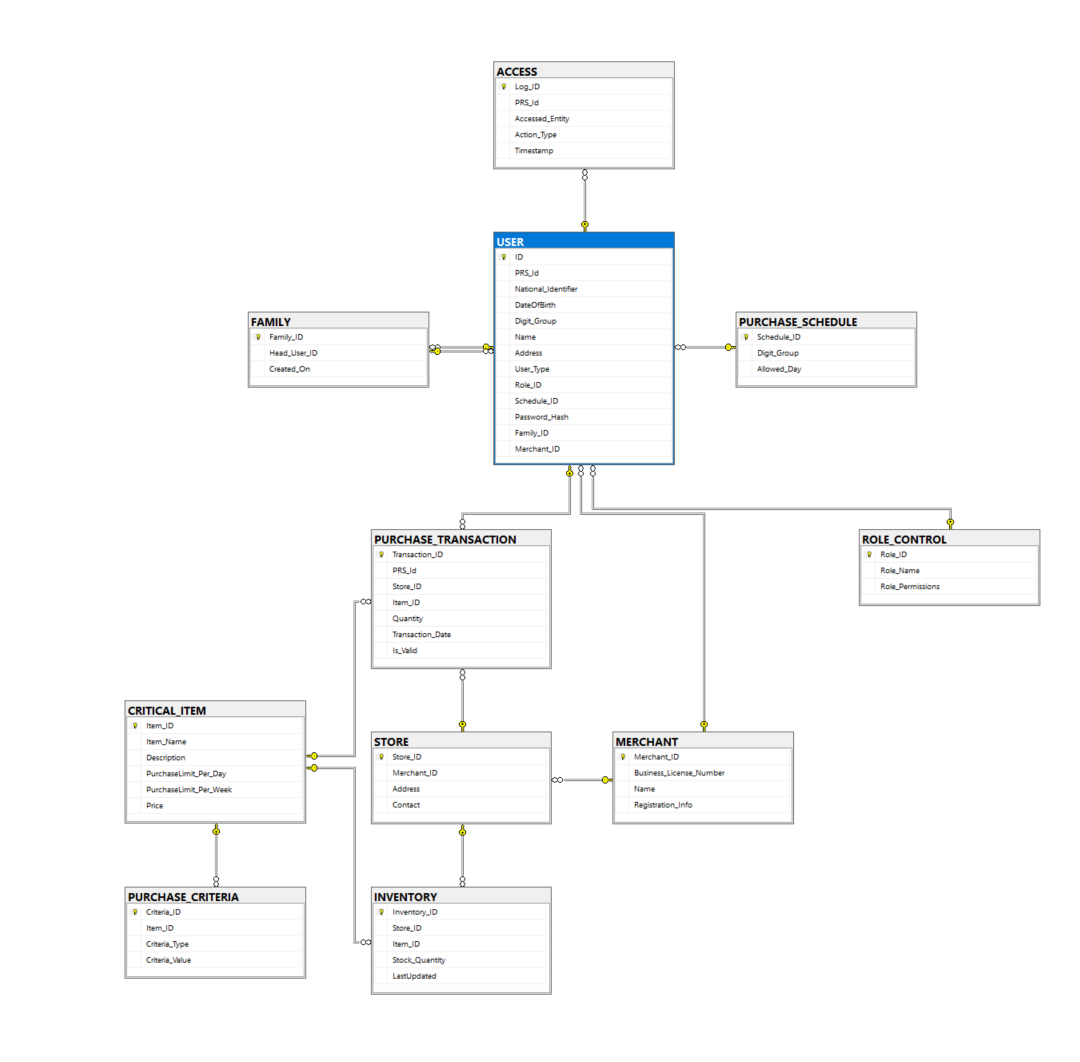
[Access controls, auditing and data protection 18](#_Toc1670603275)

[Data integrity 18](#_Toc2110873657)

[Security Plan 19](#_Toc940952378)

# Structured data (Microsoft SQL Server)

## ERD diagram



## Entity tables

|  |  |  |
| --- | --- | --- |
| **Entity** | **PK** | **Role in PRS** |
| USER | PRS\_ID | Holds every individual’s core profile and credentials. |
| FAMILY | Family\_ID | Groups users into household units (for shared schedules). |
| ACCESS | Log\_ID | Audit‐log of every access event (who did what when) |
| ROLE\_CONTROL | Role\_id | Defines each user’s permissions. |
| PURCHASE\_SCHEDULE | Schedule\_ID | Maps digit-group → allowed shopping days. |
| CRITICAL\_ITEM | Item\_ID | Catalogue of rationed items (with daily/weekly caps). |
| PURCHASE\_CRITERIA | Criteria\_ID | Overrides on per-item rules (e.g. age, location). |
| STORE | Store\_ID | Physical outlets where users redeem PRS quotas. |
| MERCHANT | Merchant\_ID | Business entity owning one or more stores. |
| INVENTORY | Inventory\_ID | Stock levels of each item at each store. |
| PURCHASE\_TRANSACTION | Transaction\_ID | Records each purchase attempt (valid or invalid). |

## Attributes and design choice justification

**USER.National\_Identifier** is defined as a fixed-length CHAR(11) so every identifier fits exactly. This eliminates padding or truncation issues and keeps storage overhead to a minimum.

**USER.Password\_Hash** stores only the hashed password rather than plaintext credentials. Even if the table is compromised, attackers cannot recover users’ real passwords.

**CRITICAL\_ITEM.PurchaseLimit\_Per\_Day** and **PurchaseLimit\_Per\_Week** live directly on the item table. Embedding these limits here means any part of the application can enforce rationing rules without additional checks.

**PURCHASE\_TRANSACTION.Is\_Valid** flags each transaction as true or false instead of rejecting invalid attempts outright. This approach preserves a complete history of all purchase attempts and makes it easy to analyze patterns of abuse or error.

**INVENTORY.LastUpdated** uses a UTC DATETIME timestamp. Consistent worldwide timing simplifies queries like “show me stock changes in the last 24 hours” without worrying about local time zones.

**FAMILY.Head\_User\_ID** points to the main account in each household. That lets you generate family-wide reports and apply permissions hierarchically, while still linking every member record.

**ROLE\_CONTROL.Role\_Permissions** is stored as a JSON object (or a SET) so you can add or remove permissions dynamically. This flexibility avoids frequent schema changes as access requirements evolve.

## Relationships and cardinality

**USER to ACCESS** (one-to-many) Each user can generate multiple access logs. Every log entry belongs to exactly one user. We do not cascade deletes so that audit trails remain intact even if a user account is removed.

**FAMILY to USER** (one-to-many) A family may include several users, but each user belongs to at most one family. The Family\_ID on USER is nullable to accommodate single-person households.

**PURCHASE\_SCHEDULE to USER** (one-to-many) Users inherit their allowed shopping days from the PURCHASE\_SCHEDULE table based on digit group. Changing schedule rules in one place immediately affects all linked users.

**ROLE\_CONTROL to USER** (one-to-many) Each user is assigned exactly one role, which determines their permissions. Swapping roles requires only updating the foreign key in the USER table.

**MERCHANT to STORE** (one-to-many) A merchant can own multiple stores. We enforce ON DELETE RESTRICT so that a merchant cannot be removed while they still have active storefronts.

**STORE to INVENTORY** (one-to-many) Every store maintains stock records for many items. A composite key of (Store\_ID, Item\_ID) guarantees there is only one inventory record per item in each store.

**CRITICAL\_ITEM to PURCHASE\_CRITERIA** (one-to-many) Items may have several special rules—such as age or regional overrides—but each rule entry links back to exactly one item. This keeps the core item list clean and focused.

**STORE and CRITICAL\_ITEM linked by PURCHASE\_TRANSACTION** (many-to-many) The PURCHASE\_TRANSACTION table bridges stores and items. Each record specifies which store sold which item, in what quantity, and whether it passed validation.

# Unstructured data/ noSQL (MongoDB)

## NoSQL schema

Schema JSON -

{

"$jsonSchema": {

"bsonType": "object",

"required": [

"\_id",

"dose",

"fhirPayload",

"prsId",

"updatedAt",

"vaccinationDate",

"vaccineName"

],

"properties": {

"\_id": {

"bsonType": "objectId"

},

"createdAt": {

"bsonType": "date"

},

"dose": {

"bsonType": "string"

},

"fhirPayload": {

"bsonType": "object",

"properties": {

"lotNumber": {

"bsonType": "string"

},

"manufacturer": {

"bsonType": "object",

"properties": {

"display": {

"bsonType": "string"

}

},

"required": [

"display"

]

},

"occurrenceDateTime": {

"bsonType": "string"

},

"patient": {

"bsonType": "object",

"properties": {

"identifier": {

"bsonType": "object",

"properties": {

"system": {

"bsonType": "string"

},

"value": {

"bsonType": "string"

}

},

"required": [

"system",

"value"

]

}

},

"required": [

"identifier"

]

},

"protocolApplied": {

"bsonType": "array",

"items": {

"bsonType": "object",

"properties": {

"doseNumberPositiveInt": {

"bsonType": "int"

}

},

"required": [

"doseNumberPositiveInt"

]

}

},

"resourceType": {

"bsonType": "string"

},

"status": {

"bsonType": "string"

},

"vaccineCode": {

"bsonType": "object",

"properties": {

"coding": {

"bsonType": "array",

"items": {

"bsonType": "object",

"properties": {

"code": {

"bsonType": "string"

},

"display": {

"bsonType": "string"

},

"system": {

"bsonType": "string"

}

},

"required": [

"code",

"display",

"system"

]

}

}

},

"required": [

"coding"

]

}

},

"required": [

"occurrenceDateTime",

"resourceType",

"status",

"vaccineCode"

]

},

"prsId": {

"bsonType": "string"

},

"updatedAt": {

"bsonType": "date"

},

"vaccinationDate": {

"bsonType": "date"

},

"vaccineName": {

"bsonType": "string"

},

"verified": {

"bsonType": "bool"

}

}

}

}

## Entity tables

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **BSON Type** | **Required** | **Role in PRS** |
| \_id | ObjectId | Yes | Unique document identifier |
| createdAt | Date | No | Timestamp when the record was first inserted |
| updatedAt | Date | Yes | Timestamp of the last update (audit and sync) |
| prsId | String | Yes | Links this record to a specific individual’s PRS-Id |
| vaccinationDate | Date | Yes | When the dose was administered (for eligibility & timeline) |
| dose | String | Yes | Label or number of the dose (e.g. “1st”, “2nd”) |
| vaccineName | String | Yes | Human-readable name of the vaccine (for reporting & displays) |
| fhirPayload | Object | Yes | Full HL7 FHIR Immunization JSON |
| fhirPayload.lotNumber | String | No | Manufacturer’s lot/batch number |
| fhirPayload.manufacturer.display | String | Yes (inner) | Manufacturer name from FHIR payload |
| fhirPayload.occurrenceDateTime | String | Yes (inner) | FHIR’s recorded date/time of the immunization |
| fhirPayload.status | String | Yes (inner) | FHIR status (e.g. “completed”) |
| fhirPayload.resourceType | String | Yes (inner) | Should always be “Immunization” |
| fhirPayload.vaccineCode.coding | Array | Yes (inner) | FHIR coding array for vaccine type (code, display, system) |
| fhirPayload.patient.identifier.system | String | Yes (inner) | FHIR patient identifier system URI |
| fhirPayload.patient.identifier.value | String | Yes (inner) | FHIR patient identifier value |
| fhirPayload.protocolApplied | Array | No | FHIR protocol details (e.g. doseNumberPositiveInt) |
| verified | Bool | No | Manual-review flag (has this record been approved?) |

## Attributes and design choice justification

**id (ObjectId, required)** MongoDB’s built-in ObjectId ensures each record is unique and indexed. This makes lookups, sharding, and any joins with other collections (for example, audit logs) both reliable and fast.

**createdAt\_ (Date, optional) and updatedAt (Date, required)** createdAt records when the document was first inserted—handy for debugging or back-fills. updatedAt is set on every write, so any change (manual verification, schema migration, data correction) is timestamped and easily sorted.

**prsId\_ (String, required)** Moving the user’s PRS identifier to a top-level field means queries across millions of documents—such as “show me every dose for this person”—run in milliseconds without scanning the full payload.

**vaccinationDate\_ (Date, required)** Storing the immunization date as a Date type allows true date-range queries (“who needs a shot next week?”) and avoids string-parsing headaches or time zone ambiguities.

**dose\_ (String, required)** Labels like “1st,” “2nd,” or “booster” live as simple strings. That keeps dashboards and reports readable while letting dose terminology stay flexible.

**vaccineName\_ (String, required**) A vaccine name (for example, “Moderna mRNA-1273”) is pulled out of the nested FHIR coding and stored at the root. This speeds up UI rendering and reporting without digging through deep JSON.

**fhirPayload\_ (Object, required)** The entire HL7 FHIR Immunization JSON is kept here preserving all data

**verified\_ (Bool, optional)** A single flag indicating whether the record has passed manual review. Keeping it at the top level means verification status is instantly accessible in queries and dashboards, without a separate table.

## Scalability

The system is designed to grow smoothly as more people use it or more data is added. Whether it's processing large numbers of vaccination records or supporting busy times with lots of users, the system can handle increased demand without slowing down. This is because of these features:

**Flexible Data Storage:** Vaccination records are saved in MongoDB, which allows information to be stored in a way that’s easy to adjust over time. This helps the system stay organised even as more records are added. At the same time, important user and merchant details are managed in SQL Server, which is set up to handle lots of users at once.

**Easy to Scale Up:** The system’s backend is built so that new servers can be added quickly whenever needed. If a lot of people try to use the system at the same time, it can easily create extra copies of itself to share the workload. This keeps things running smoothly for everyone.

**Fast Data Uploads:** When large files of vaccination records are uploaded, the system doesn't try to process everything at once. Instead, it accepts the file quickly and processes it in the background. This keeps the website responsive and avoids delays for users.

**Quick Searches:** The system uses smart ways to organise and search through the data. This means that looking up vaccination records or checking how many have been verified is very fast, even if there are thousands of records.

**Caching for Speed:** Some frequently used information, like store stock levels or user schedules, is kept in temporary memory (called caching). This allows the system to answer common questions very quickly without needing to search the database each time.

# API specifications and Security

## Endpoint list

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Path** | **Method** | **Purpose** | **Request Body (JSON)** | **Success Response** | **Error Responses** |
| /api/register | POST | Create a new user and issue a PRS-Id | nationalId (string), dob (string, ISO date), name, password, roleId | {"message":"User registered","prsId":<string>} (201) | 400: missing fields 500: registration error |
| /api/login | POST | Authenticate and return JWT | nationalId, password | {"message":"Login successful","token":<jwt>} (200) | 400: missing fields 401: invalid credentials |
| /api/findNearestSuppliers | POST | Return nearest stores with distance from user | prsId | [{ "storeId":<string>, "address":<string>, "distance":<number> }, …] (200) | 404: user location not found |
| /api/getUserVaccRecord | POST | Fetch a user’s vaccination record | prsId | Full record object (200) | 400: missing prsId 404: record not found |
| /api/addFamilyMember | POST | Add a family member under a user’s PRS-Id | prsId, nationalId, name, dob, address, userType | {"success":true,"message":"Family member added"} (201) | 400: missing fields 500: add error |
| /api/getFamilyMembers | POST | List all family members for a PRS-Id | prsId | [{ member }, …] (200) | 400: missing prsId 404: not found |
| /api/removeFamilyMember | POST | Remove a family member | prsId, familyMemberId | {"success":true,"message":"Family member removed"} (200) | 400: missing fields 500: removal error |
| /api/updateAddress | POST | Update a user’s address | prsId, address | {"success":true,"message":"Address updated"} (200) | 400: missing fields or update error |
| /api/updatePassword | POST | Change a user’s password | prsId, oldPassword, newPassword, confirmPassword | {"success":true,"message":"Password updated"} (200) | 400: missing fields 400: passwords do not match 400: update error |
| /api/get\_allowed\_critical\_items | POST | Retrieve items a user may purchase today | prsId | {"success":true,"items":[…]} (200) | 400: missing prsId 400: retrieval error |
| /api/get\_allowed\_day | POST | Get which day a user may buy restricted items | prsId | {"success":true,"allowedDay":<string>} (200) | 400: missing prsId 404: no rule found |
| /api/upload\_vaccination | POST | Save a batch of FHIR vaccination records | full FHIR Immunization Bundle JSON | {"success":true} (200) | 400: no JSON provided 500: save error |
| /api/merchant/dashboard-data | POST | Merchant dashboard summary | prsId | { "sales":…, "orders":…, "products":…, "stockLevels":…, "purchaseRestrictions":…, "vaccinationStats":…, "business\_info":{…} } (200) | 400: missing prsId 403: not a merchant account |
| /api/merchant/update-stock | POST | Update stock quantity for an item | merchantId, itemId, newQuantity | {"success":true} (200) | 400: update error |
| /api/merchant/updateVerifyRecord | POST | Change verification status of a vaccination record | prsId, recordID, verified\_status | {"success":true,"message":"Verification record updated"} (200) | 400: missing fields 500: update error |
| /api/merchant/getAllVaccinationRecords | POST | List all vaccination records in MongoDB for merchant | *(no body required)* | {"success":true,"records":[…]} (200) | 500: retrieval error |
| /api/gov/purchase-limits | GET | Return every item’s per-day purchase limit | *none* | {"items":[{"item":string,"limit":number},…]} (200) | 500: server error |
| /api/gov/schedules | GET | Return digit-group → allowed-day mappings | *none* | {"schedules":[{"digitgroup":string,"allowedDay":string},…]} (200) | 500: server error |
| /api/gov/alerts | GET | Return real-time limit-exceedance alerts | *none* | {"alerts":[{"message":string,"timestamp":string},…]} (200) | (none, stub always 200) |
| /api/gov/merchants | GET | List all merchants and their license info | *none* | {"merchants":[{"merchantId":string,"businessLicense":string,…},…]} (200) | 500: server error |
| /api/gov/compliance | GET | Store-by-store compliance status (“Compliant”/“Non-Compliant”) | *none* | {"statuses":[{"location":string,"status":string},…]} (200) | 500: server error |
| /api/gov/stock | GET | Per-store and aggregate stock levels for critical items | *none* | {"stock":[{"storeId":string,"itemId":string,"item\_name":string,"quantity":number},…]} (200) | 500: server error |
| /api/gov/vacc/records | POST | Fetch FHIR vaccination records (optionally scoped by PRS) | {"prsId":string} *or* {} for all records | {"records":[…]} (200) | 500: server error |

## Authentication flow

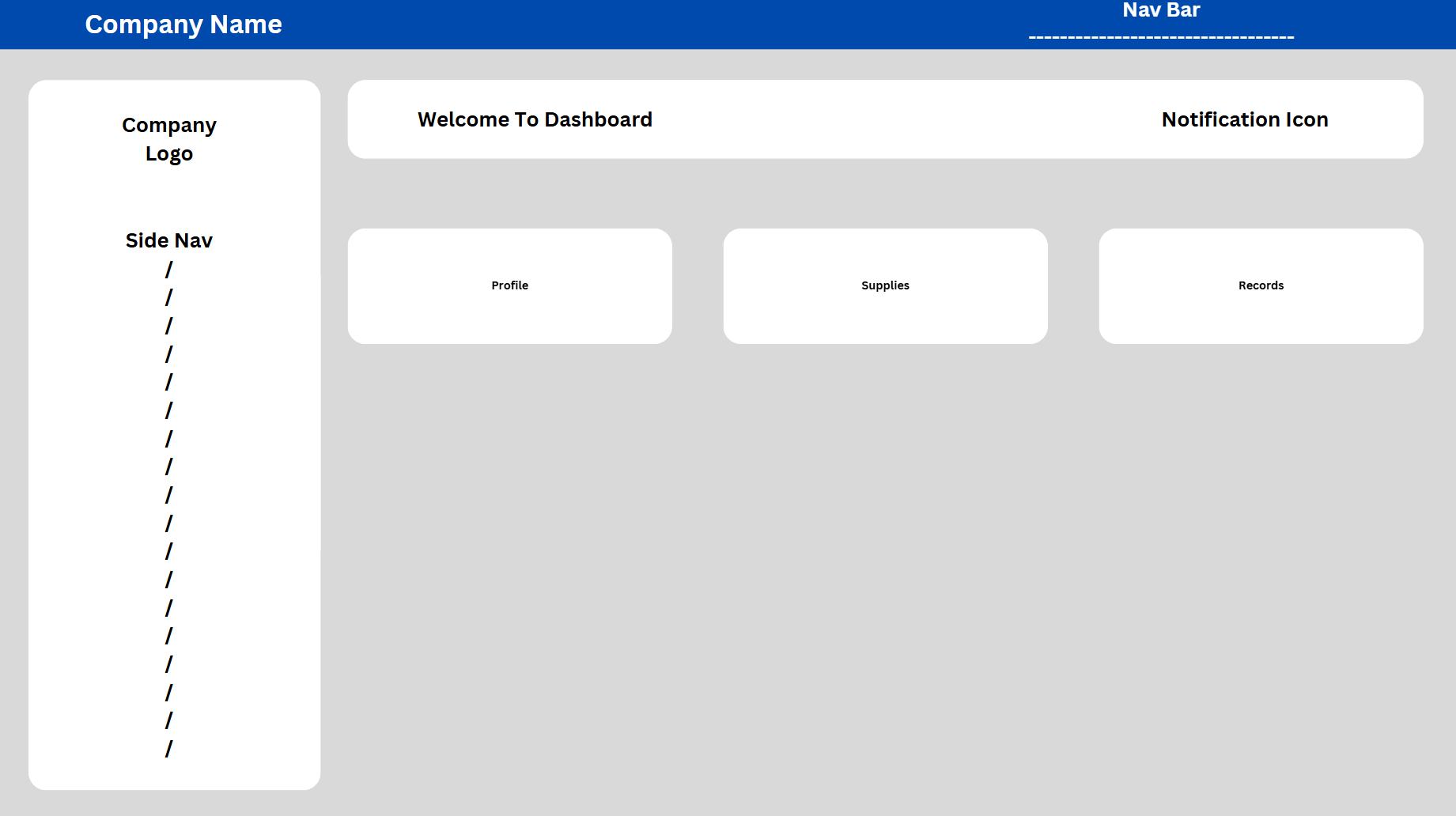
Clients send a POST request to /api/login with nationalId and password. On successful authentication, the server responds with a JWT (JSON Web Token) signed using the application’s secret key. The token expires after six hours. All protected endpoints require this token in the Authorization header and embedding identity and role in the token removes the need for a database lookup on every request.

JWT Structure:

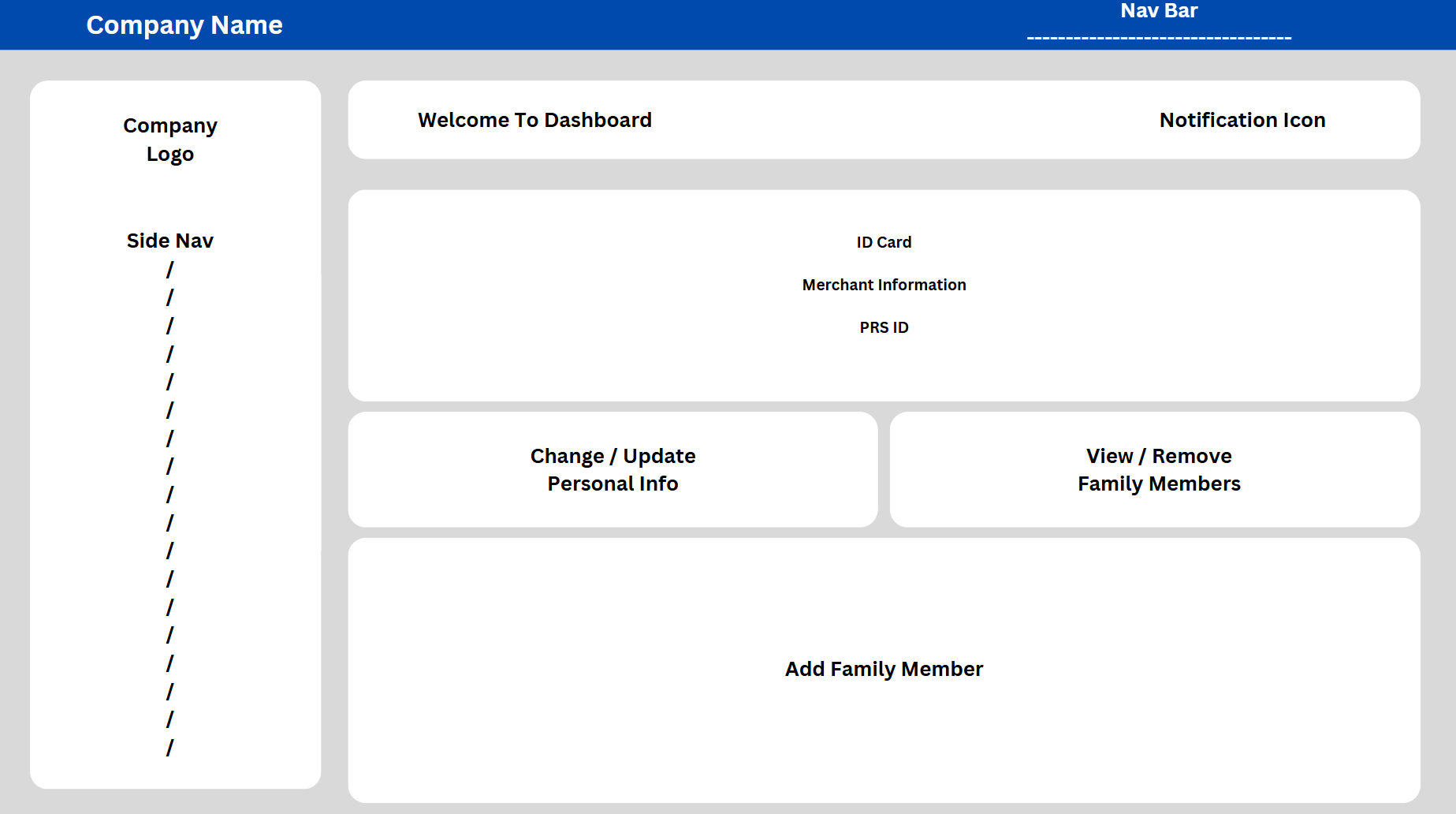
* prs\_id – user’s PRS identifier
* role – one of public, merchant, gov\_offical, admin
* name – user’s display name
* DOB – date of birth (for eligibility logic)
* merchantId – only present for merchant accounts
* exp – UTC timestamp when the token expires

# Mock-ups

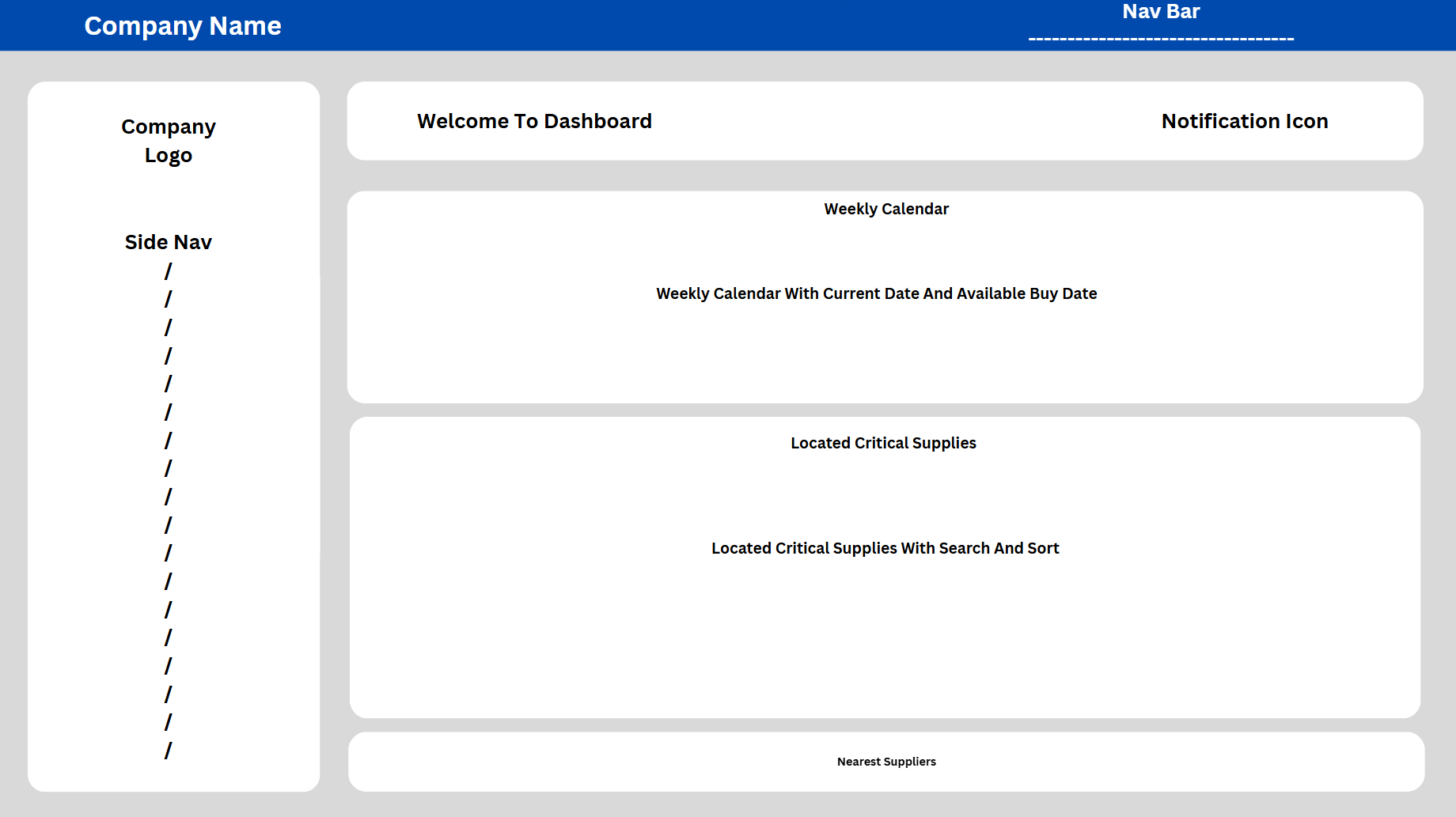
User-Home

* 

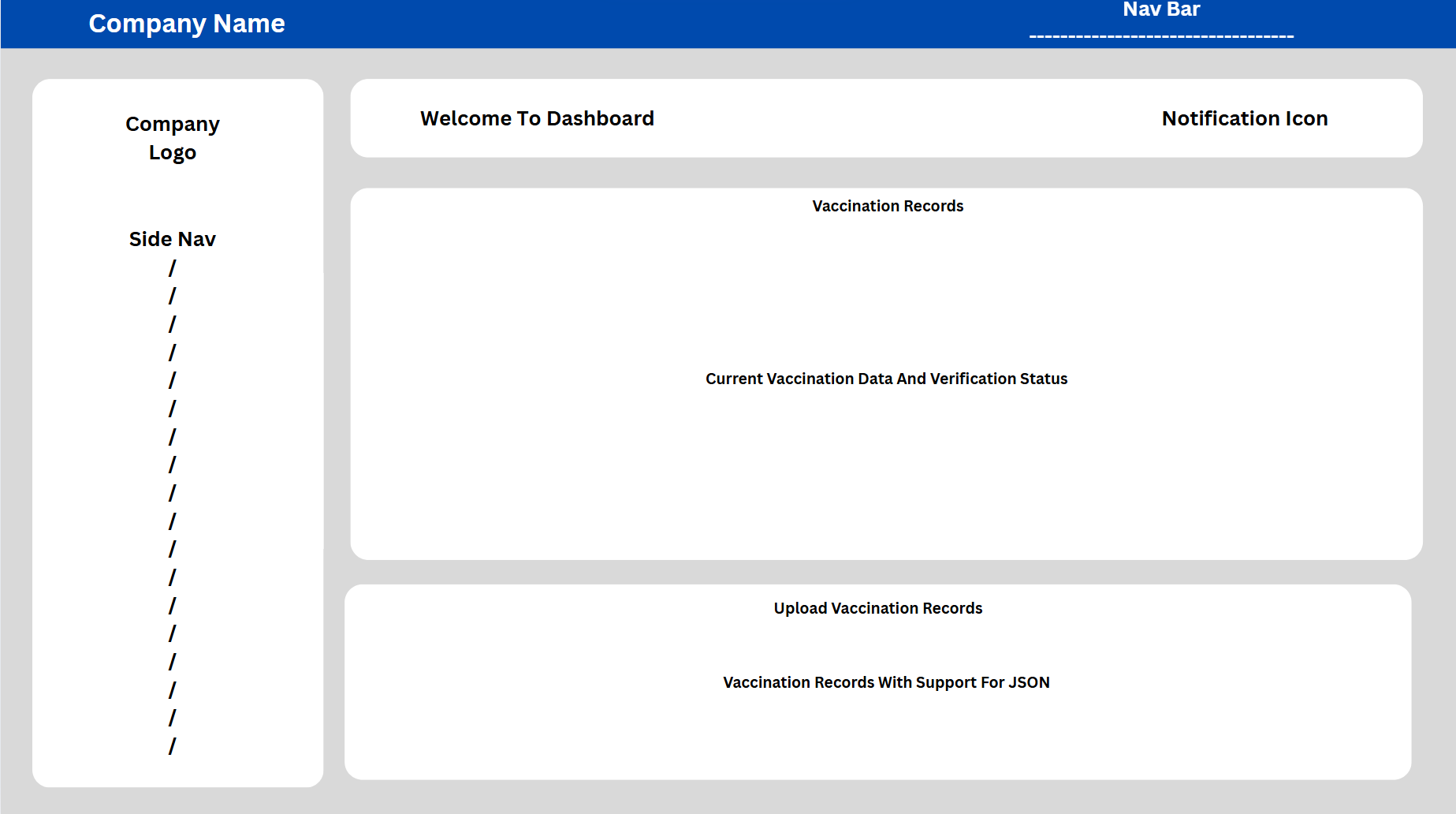
User-Profile

* 

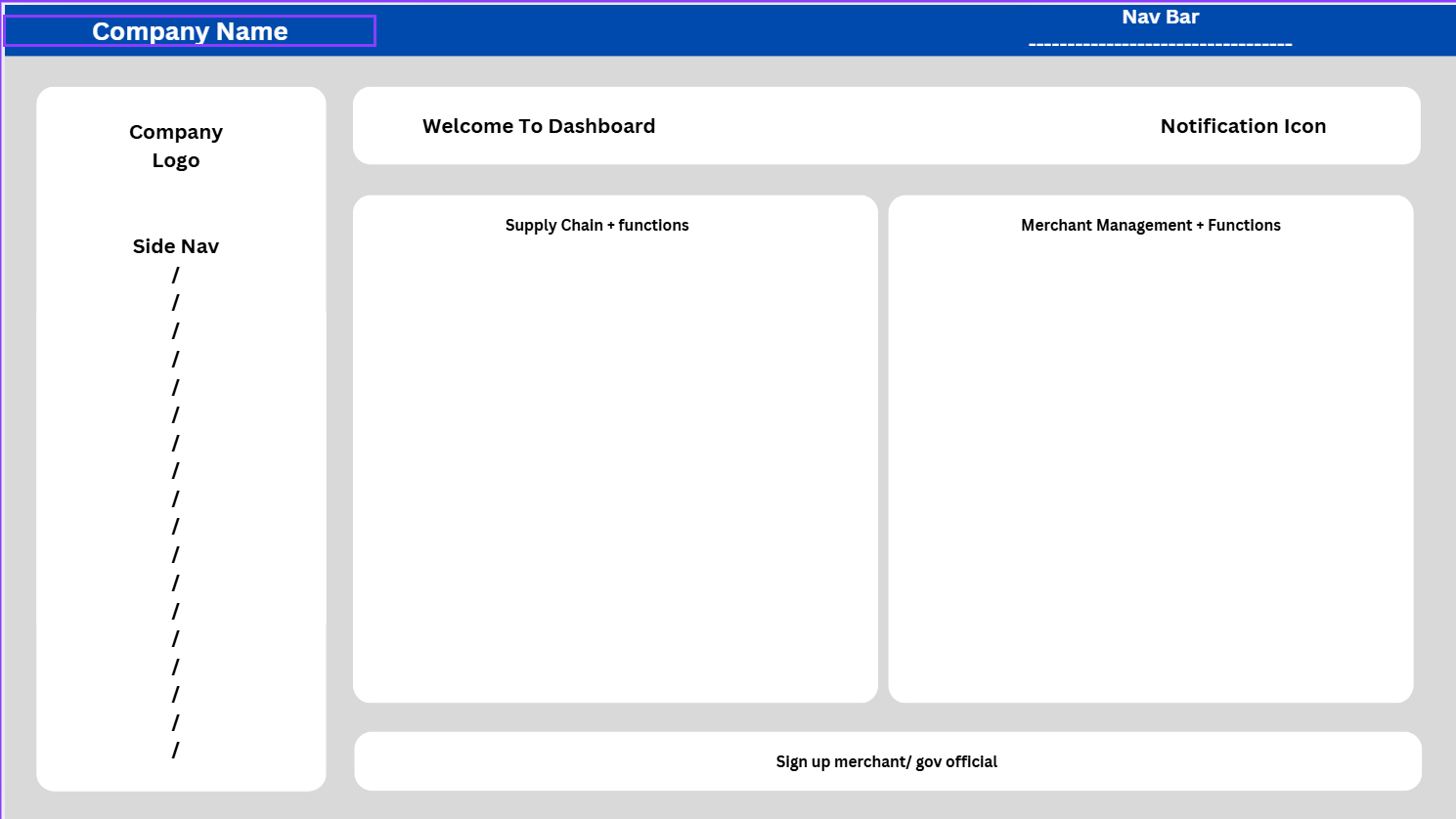
User-Suppliers

* 

User-Records

* 

Gov-Dashboard

* 

# Authentication, authorisation and security plan

## Access controls, auditing and data protection

The system includes clear access controls to make sure that each user can only see or do what they are allowed to. For example, public users can view their own vaccination status and purchase limits, while merchants can manage stock and view customer vaccine data, but only if it relates to their store. Government officials have access to high-level data and reports but can’t change personal records.

Passwords are stored safely using hashing (bcrypt), so even if someone got into the system, they couldn’t read users’ actual passwords. Users must log in before accessing protected parts of the system, and their identity is checked using secure tokens.

To support accountability, the system also keeps track of actions behind the scenes. This means it can log who uploaded vaccination files, when stock levels were changed, or who verified vaccine records. These logs help with troubleshooting and can be used to investigate if something goes wrong.

By limiting what each type of user can do, storing passwords securely, and recording important actions, the system fulfils the requirements for access control and auditing.

## Data integrity

The system is designed to make sure that all the information it stores is accurate, consistent, and protected from accidental or intentional damage. For example, when a vaccination record is uploaded, it checks that the data follows the correct format and that key fields like PRS ID, dose number, and vaccine name are included before saving it to the database.

To avoid duplicate or conflicting entries, the system saves each vaccine shot separately and checks that the patient’s ID is valid. The MongoDB database allows flexible storage of structured vaccine data while still enforcing key fields needed for reporting and analysis.

When a vaccine record is verified or stock levels are changed, the system also updates timestamps so it’s easy to track when changes were made. This helps keep the data reliable and traceable over time.

## Security Plan

Keeping data safe is a key part of how the system is built and managed. Several layers of protection are in place to guard personal and health information.

**Password Protection:** User passwords are never stored as plain text. They are hashed using strong encryption (bcrypt), which makes them useless to anyone even if the database were accessed.

**Token-Based Access:** Once a user logs in, they receive a secure token (JWT) that gives them access to the system. This means credentials aren’t passed around repeatedly, reducing risk.

**Role-Based Permissions:** Each user is assigned a role (like public user, merchant, or government staff), and only sees what they need to. This prevents unauthorized actions or data access.

**Input Checks:** All data sent into the system is carefully checked and cleaned. This stops attackers from injecting malicious code or trying to break the system.

**Database Rules:** The MongoDB database uses a validation schema to make sure only correctly formatted data gets stored. It rejects anything that doesn’t meet the rules.

**Audit Logs:** Key changes, like manual verifications, are tracked. This creates a record of who did what and when, helping catch mistakes or misuse.

**Secure Connections:** All communication between the client and server happens over HTTPS, which keeps data encrypted while it moves over the internet.