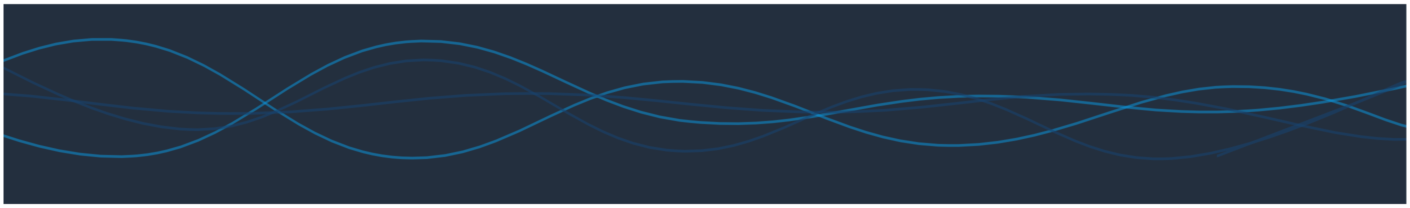
Deadbolt PCB Cloud migration

Primary Owner ots-dse-hw-team (LDAP)

Last modified 4 seconds ago by traykeli.



Deadbolt PCB Cloud Migration

Overview

Project Deadbolt is an advanced asset protection locker developed to provide a solution for preventing and tracking device shrinkage (loss, theft, breakage, etc.) at Amazon sites. The locker, which was developed in-house, utilizes advanced technologies to monitor and track the checkin and checkout activity of stored assets. In the event that an exception is triggered (for example, if an employee take 10 scanners from a locker), leadership is notified of who, what, and when. Site management can also view activity data from an externalized single-pane dashboard.

Current State

Deadbolt is currently deployed across xx sites. The system can reliably track the following:

- Amount of assets in a locker on a given day
- Historical data of employees' checkin/checkout activity
- Employee session time (the length of time needed to checkin or checkout an asset)
- General functional exceptions (charger board error codes, locker door left open, etc.)

Problems Statements & Opportunities

While the current iteration of Deadbolt (GEN2) has laid a great foundation, the system falls short in the following areas:

1. **The system is unable to establish a valid chain of custody for an asset.** Today, when an employee takes or returns an asset, the net number of assets exchanged is recorded by the system. That data is used to provide the maximum number of assets that a locker has stored for a given month. Over the course of several months, the trend yields a generic "device count reduction" metric (DCR) which represents the total amount of shrinkage a site has experienced. However, that metric is not granular enough to provide any insight into the cause of the shrinkage. A number of edge cases, such as: 1) when an employee checks in an asset that was originally checked out by a different employee or 2) when an asset moves between sites, are indistinguishable from one another. Ideally, each asset would have a unique identifier (e.g. serial number) which can be reliably tied to an employee during checkin/checkout. With that association in place, Deadbolt can provide an accurate device not returned metric (DNR) for stakeholders to provide coaching to associates on how to properly return an asset, or to track assets across sites. In the future, this tracking data could further extend the value that Deadbolt provides through tools that make strategic decisions about asset inventory and ordering.
2. **Our ability to monitor, access, and manage lockers remotely is limited.** In the current architecture, the ELO tablet (user interface) acts as an intermediary between the Deadbolt backend services and the locker hardware. This overloads the UI with responsibilities that range from managing the entire system state to continuously monitoring the hardware and reporting faults. Additionally, the system's operational logs are stored in multiple places (ELO Cloud, CloudWatch, various firmwares). There is an enormous opportunity to simplify the system architecture. Doing so will unlock specific features like firmware OTA updates, resets, and remote access. It will also greatly improve operational response time and cost structure as field techs would no longer need to be sent out to sites to troubleshoot basic issues.
3. **All locker decisions are supported only by locally available data.** Currently, all of the locker's decision making processes (e.g. which door to open) are performed locally on the user interface. This limits those processes to the data available on the local device, and, even at the scale of a dozen lockers, makes publishing changes and debugging issues difficult and time consuming. By connecting the hardware directly to AWS and lifting the business decision points into a distributed system, more data points can be considered and faster iteration / experimentation can be leveraged.
4. **Our hardware and supply chain flexibility is hampered.** The ELO tablets are expensive and bind the Deadbolt architecture to their specific hardware and software ecosystems. The field operation of lockers has been successfully executed with these devices, but their hardware constraints, subscription/licensing fees, and opinionated CMS tooling limit Deadbolt's growth potential by restricting the team from pursuing other cost conscious options.

Requirements

1. The Locker should effectively track devices (Assets) with a unique device ID (Asset ID).
2. Locker selection logic should be processed in the cloud.
3. Locker should function with any tablet
4. The System should be able to update the Locker's firmware OTA

Assumptions

- A Main Module will serve as the interface for all Charger Boards in a single Cabinet, with each shelf housing its own Charger board. This main Module will connect to IoT Core.
- The Main Module will be properly on-boarded/provisioned with IoT Core.
- An Asset ID will be provided to Iot Core via RFID device recognition on a Locker level
  - Asset ID will be prepended with the specific asset group (i.e. TC56, MC33, Ring, etc)
- Physical Locker Structure
  - One Locker can have many Cabinets
  - One Cabinet has one Main Module
  - One Main Module connects to many Charger Boards
  - One Charger Board exposes many Slots
  - Locker will have an RFID Badge reader to scan amazon blue badges
- The Main Module can communicate with the tablet via local HTTP

Key Concepts

AWS Services:

- IoT Core: Main form of communication between hardware devices and AWS Services more info
  - Rules: Enables the ability to interact with AWS Services more info
- Greengrass: Service used to facilitate the running and management of IoT Core devices more info

Main Terms:

- Main Module:
  - Serves as the interface between Charger boards and IoT Core.
- Charger Board:
  - Referred to as a Shelf at times. It is the PCB board that charges devices, senses proximity, and communicates to the Main Module regarding asset data.
- Asset:
  - Devices (TC56, Ring scanners, etc). A device will be called an Asset throughout this doc. Tracking of these Assets is the main prerogative of this design doc.

Team Services:

- Remote Device Management Service (RDMS):
  - Acts as an intermediary between API calls and IoT Core MQTT commands. For example, a door open command would be routed through the RDMS. Also facilitates the sending of SNS topic filters for any deadbolt messages sent via the Main Module.
- Device Provisioning Service (DPS)
  - Provides authentication for Deadbolt application
  - A solid option to facilitate IoT Core Main Module provisioning by generating and providing CA certs and keys enabling IoT Core on-boarding
- Deadbolt Backend:
  - Host storage of data and exposes API's for all functionality pertaining Deadbolt. Digests topics from RDMS and handles the storage/alarming of said data accordingly

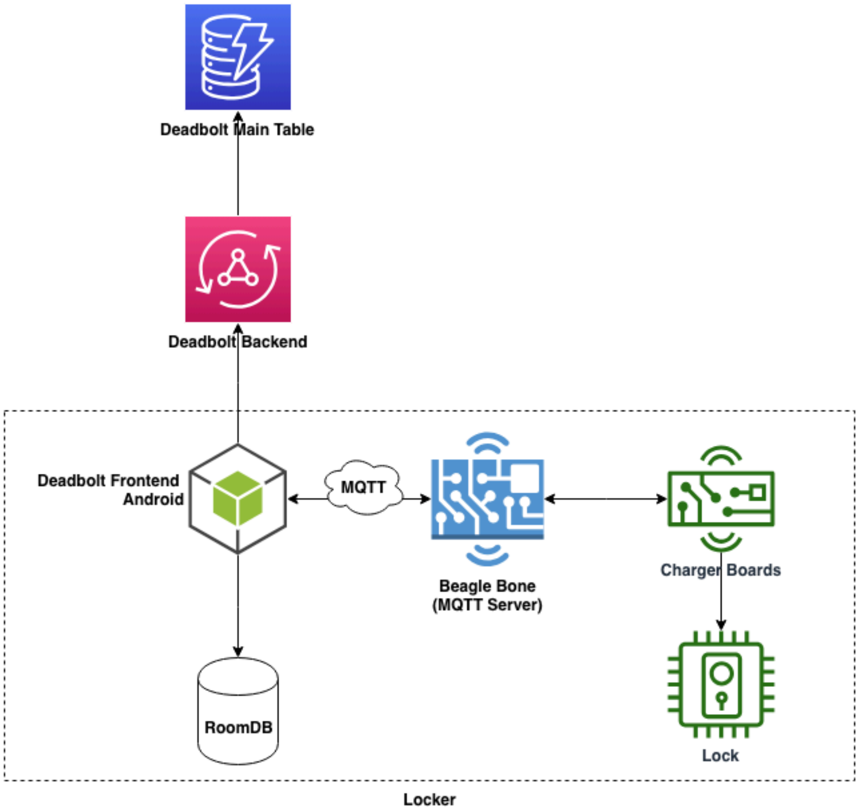
Languages:

- TypeScript: general cloud infrastructure.
- Java: integration of RDMS
- Android: updating the current tablet APK.

Proposed Solution

Old Architecture

As noted in prior sections, the only point of contact we have with the internet is via the ELO Tablet (running android as depicted below). All decisions are made through aggregating messages being passed from a single microcontroller hosting an MQTT broker. The Android application stores those messages in RoomDB and makes inferences on which door should be opened based on that data. The data is published in a round robin fashion (through the slots) in one second increments. If a command fails (i.e. door doesn't open) there is no real way to debug the root cause other than navigating to ELO CMS and digging through logs. Currently, this design has no API support for updating firmware or externally sending commands to the microcontroller. As such we are limited to the error messages the Charger Boards are reporting based off of the code supplied from our last vendor (USA Firmware). These error codes seldom provide and real insight and ultimately can notify us if a board goes offline.

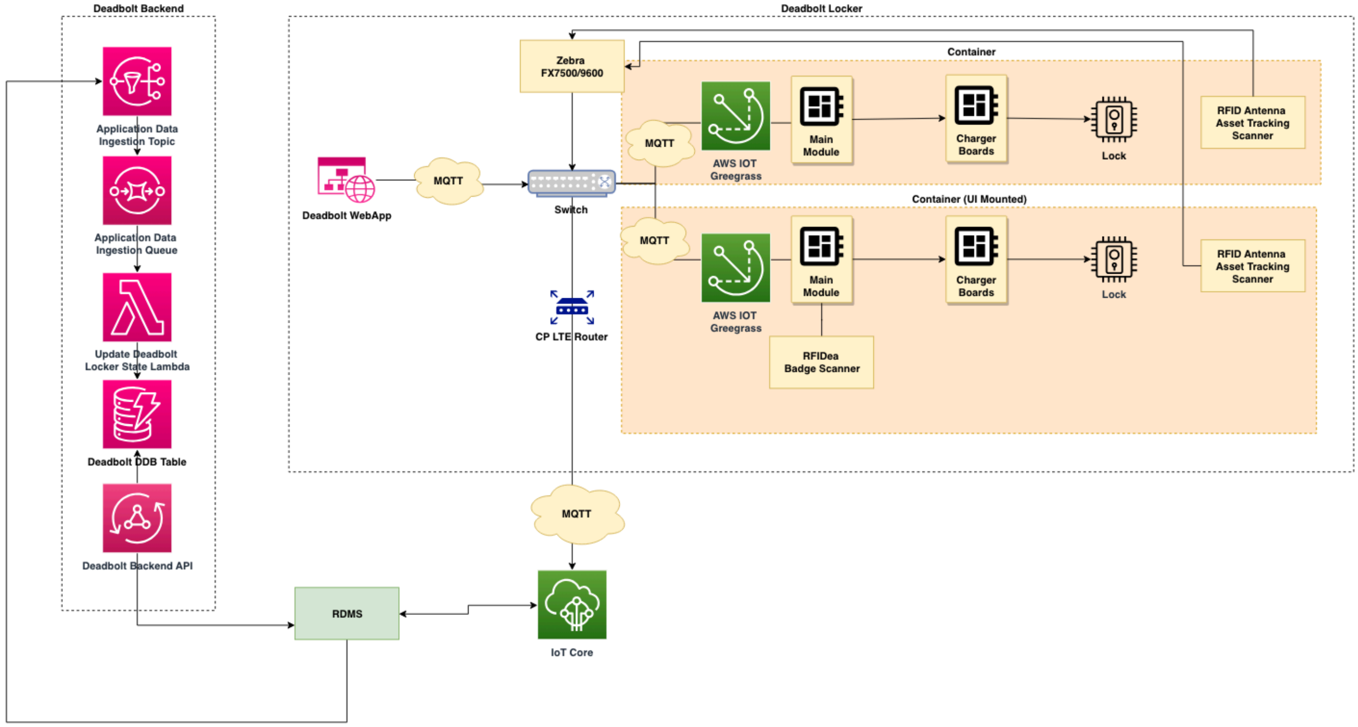


New Architecture

We will be replacing the old microcontroller with a newly engineered Main Module. The Main Module will receive signals/messages from the Charger Boards (where each Charger Board acts as a shelf) and ultimately format those raw signals/messages and send them to cloud via AWS IoT Core. It will be running linux with AWS IoT Greengrass installed.

Asset tracking will be enabled via RFID. Each Asset will have a unique RFID sticker attached grouped by a device type (e.g. deadbolttc_abc123). The RFID signal will communicate via switch to the Main Module and ultimately send that data up to the cloud to update the Deadbolt Main Table. While Asset tracking is a component for this document, lower level details on the subject will be further described in the Hardware/Firmware Deadbolt design review.

In order to support all the new capabilities of the Main Module and other Hardware components we will be leveraging RDMS and adding to the Deadbolt Backend. RDMS will handle the publishing of the messages from the Main Module and upon successful publishing of the message, the Deadbolt Backend system will subscribe to the respective SNS topic, manipulate the received data, and store it into a new Deadbolt DynamoDB table. This action results in the creation of a new row in the database. This row effectively represents a unique state that the Deadbolt API can query, thus enabling an informed decision about which locker door to select based on the required action (whether item return or retrieval).

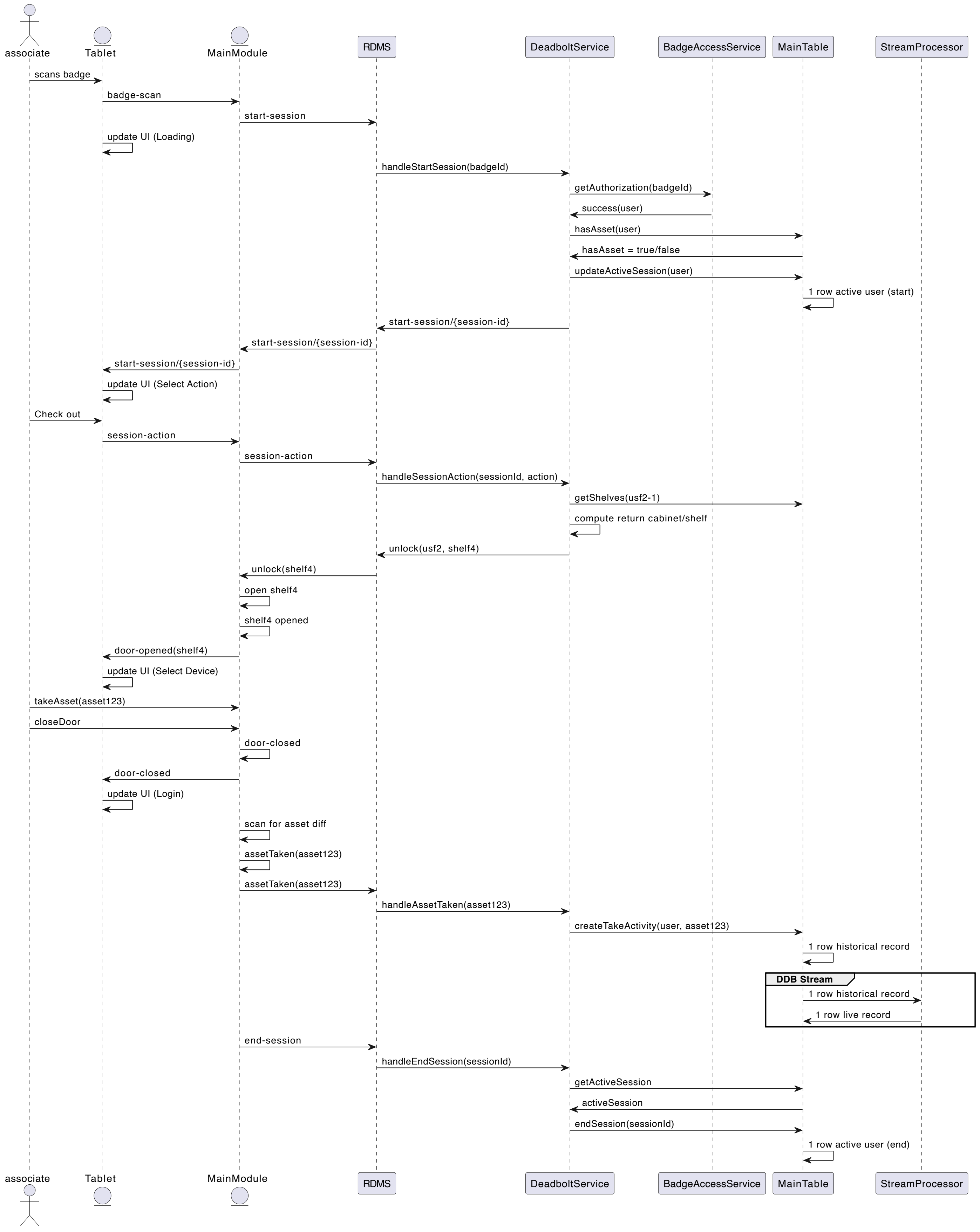


Design Inspector Source Architecture

Show

Take/Return Asset Flow

All pieces (Hardware, Firmware, and Software) are designed to perform this flow. This is the action that will happen 95% of the time (authenticated take/return of assets).

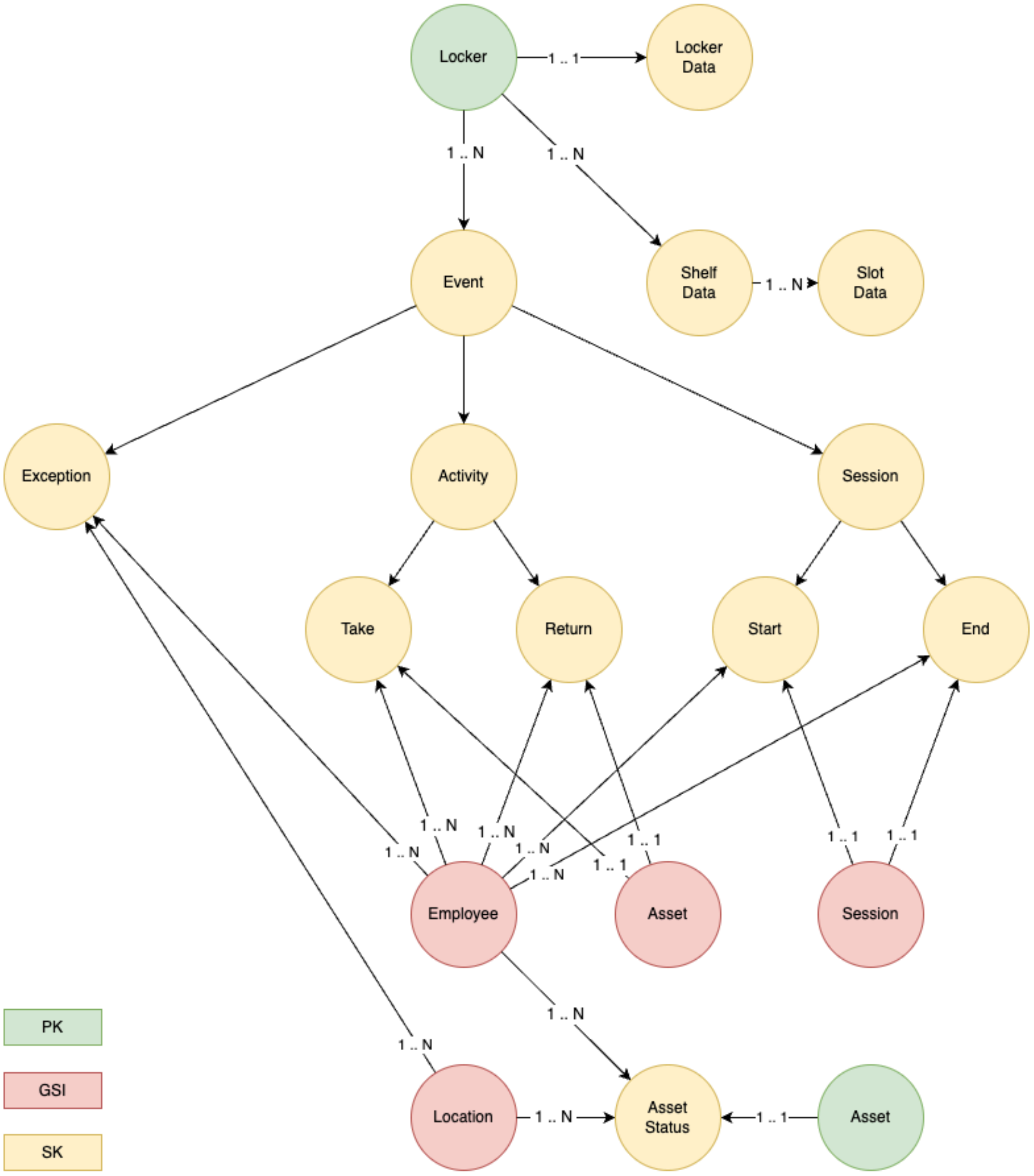


Actions: Pop Out | Edit | ?

Software Specific Domain

This part of the document is catered to software components that will need to be built.

Data Model



Data Storage

There will be three main Dynamo Document paradigms that will accomplish the following high level tasks:

- Real time Asset Tracking
  - Who took/returned the device last
  - Which locker was the device last taken/returned
- Historical trail of locker events (takes, returns, exceptions, sessions)
  - Which asset was taken, when, and by whom
  - Which locker threw the exception
  - Which user interacts with the locker and when
- Persisting the locker state
  - Number of devices
  - Slot level data
  - Shelf level data
  - Cabinet level data
  - Locker level data
  - Locker structure

Note: <someVariable> will be replaced with parameters, otherwise the string literal shown below will be the assumed value.

Real Time Asset Tracking

	pk	sk	gsi1pk	gsi1sk	gsi2pk	gsi2sk
Schema	ASSET#<type>#<assetId>	ASSET_STATUS	LOCATION#<location>	ASSET_STATUS#<returned|taken>#LOCKER#<location>#<lockerId>	EMPLOYEE#<employeeId>	ASSET_STATUS#taken
Example	ASSET#tc56#abc123	ASSET_STATUS	LOCATION#usf2	ASSET_STATUS#taken#LOCKER#usf2#1	EMPLOYEE#traykeli	ASSET_STATUS#taken

In this DDB paradigm a sparse index is used. When a locker detects a returned/taken asset, the assetId for the corresponding asset is used to update this row. Since it is deterministic (pk and sk are well defined and unchanging) we will only be updating the GSI's and the status attribute which holds information relevant to DNR numbers: where was the asset taken/returned and who returned it. Note that it is possible for gsi2 to be empty (when a manager manually takes devices out with the key), in which case the row would fall off of the GSI. To determine the DNR for a given location or locker we can simply run a begins_with 'ASSET_STATUS#taken' query on gsi1pk, it will return all assets which were not returned and who was last seen with said asset.

This also enables a simple method to determine whether a user has an asset checked out: run a query on gsi2pk with an sk of 'ASSET_STATUS#taken', if data is returned we know the asset that the user checked out hasn't been returned to a locker, otherwise the asset has been returned to the locker. Being that it is a sparse index, if a user gives their asset to a manager or another co-worker to return, the row is updated by removing the previously assigned employee on gsi2pk. With this paradigm in place we cover the scenarios below. We can also set alarms/exception/etc based off the timestamp (not shown in the row of data above but assume it exists). After X amount of hours, we signal a DNR (after 5 days or some number). We can save this lost trigger time at the locker level and place it in manager setting so managers can modify the time that they want to consider a device to be lost.

Scenario A: User1 takes an asset, User1 returns their own asset

Scenario B: User1 takes an asset, User 2 returns User1's asset

Scenario C: User1 takes an asset, loses it, goes to take an asset again *(DNR detected)*

Scenario D: User1 takes an asset, gives to User2 to return, User2 loses it, User1 goes to take an asset again *(DNR detected)*

Scenario E: User1 takes an asset, and never returns it *(DNR detected)*

Historical trail of locker events

	pk	sk	gsi1pk	gsi1sk	gsi2pk
Schema	LOCKER#<location>#<lockerId>	EVENT#<eventType>#<timestamp>	ASSET#<type>#<assetId>	<eventTypeDependent>#<timestamp>	EMPLOYEE#<employeeId>
Return Activity Example	LOCKER#usf2#1	EVENT#ACTIVITY#RETURN#1688147707000	ASSET#tc56#abc123	EVENT#ACTIVITY#RETURN#1688147707000	EMPLOYEE#trayk
Take Activity Example	LOCKER#usf2#1	EVENT#ACTIVITY#TAKE#1688147296000	ASSET#tc56#abc123	EVENT#ACTIVITY#TAKE#1688147296000	EMPLOYEE#trayk
Exception Example	LOCKER#usf2#1	EVENT#EXCEPTION#1688171473000	ASSET#tc56#xyz987	EVENT#EXCEPTION#1688171473000	EMPLOYEE#brisa
Session Start Example	LOCKER#usf2#1	EVENT#SESSION#START#1688171473000	SESSION#xyz987	EVENT#SESSION#START#1688171473000	EMPLOYEE#brisa
Session End Example	LOCKER#usf2#1	EVENT#SESSION#END#1688171473000	SESSION#xyz987	EVENT#SESSION#END#1688171473000	EMPLOYEE#brisa

Note: for exceptions it is possible to have an exception without an asset or an associated user (charger board malfunctions for instance) in which case those fields would be ASSET# and EMPLOYEE#

This paradigm enables the following:

- Get all activity by employee (can also be partitioned further via sk's: exceptions, returns, takes)
- Get all activity by locker (can also be partitioned further via sk's: exceptions, returns, takes)
- Get all activity by asset (can also be partitioned further via sk's: exceptions, returns, takes)

Persisting the locker state

	pk	sk	gsi1pk	gsi1sk	gsi2pk	gs2sk
Schema	LOCKER#<location>#<lockerId>	SHELF#<shelfNumber>#SLOT#<slotNumber>#DATA				
Locker Data	LOCKER#usf2#1	DATA	WHID#location	DATA	BUSINESSLINE#businessline	DATA

Shelf Data	LOCKER#usf2#1	SHELF#001#DATA	WHID#location	SHELF#001#DATA		
Slot Data	LOCKER#usf2#1	SHELF#001#SLOT#001#DATA	WHID#location	SHELF#001#SLOT#001#DATA		

Locker Data: Consists of data specific to the locker (lockerId, locker structure, dual device status, etc)

Cabinet Data: Consists of data specific to the Main Module (connection status, scanner attached, etc)

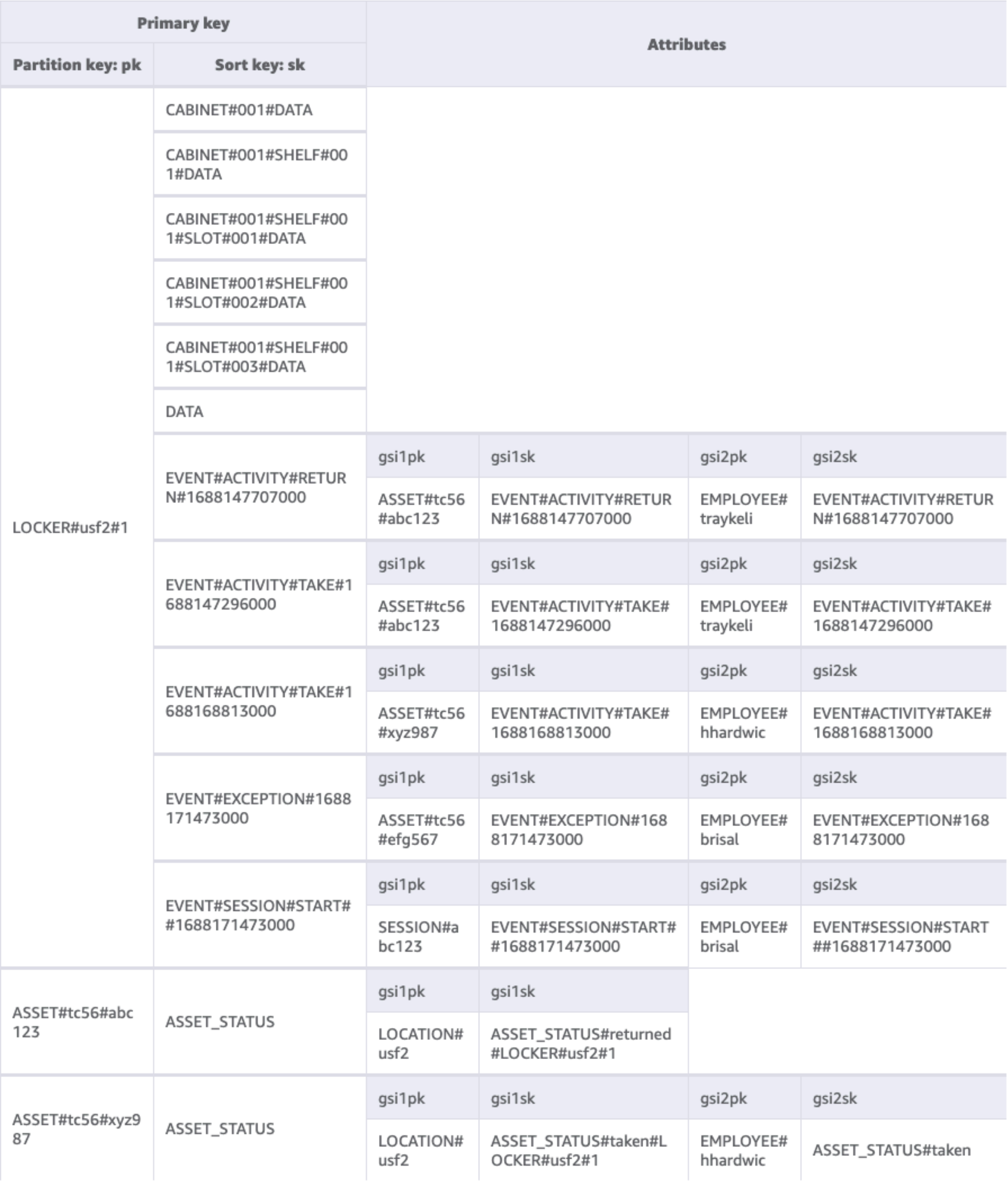
Shelf Data: Consists of data specific to the Charger Boards (connection status, MAC address, door status, etc)

Slot Data: Consists of data returned from the individual charging ports on the Charger Board (time on charge, device sensed, etc)

This data will be used in shelf selection criteria, broken device count, net device count, available slots, and other business logic (as needed):

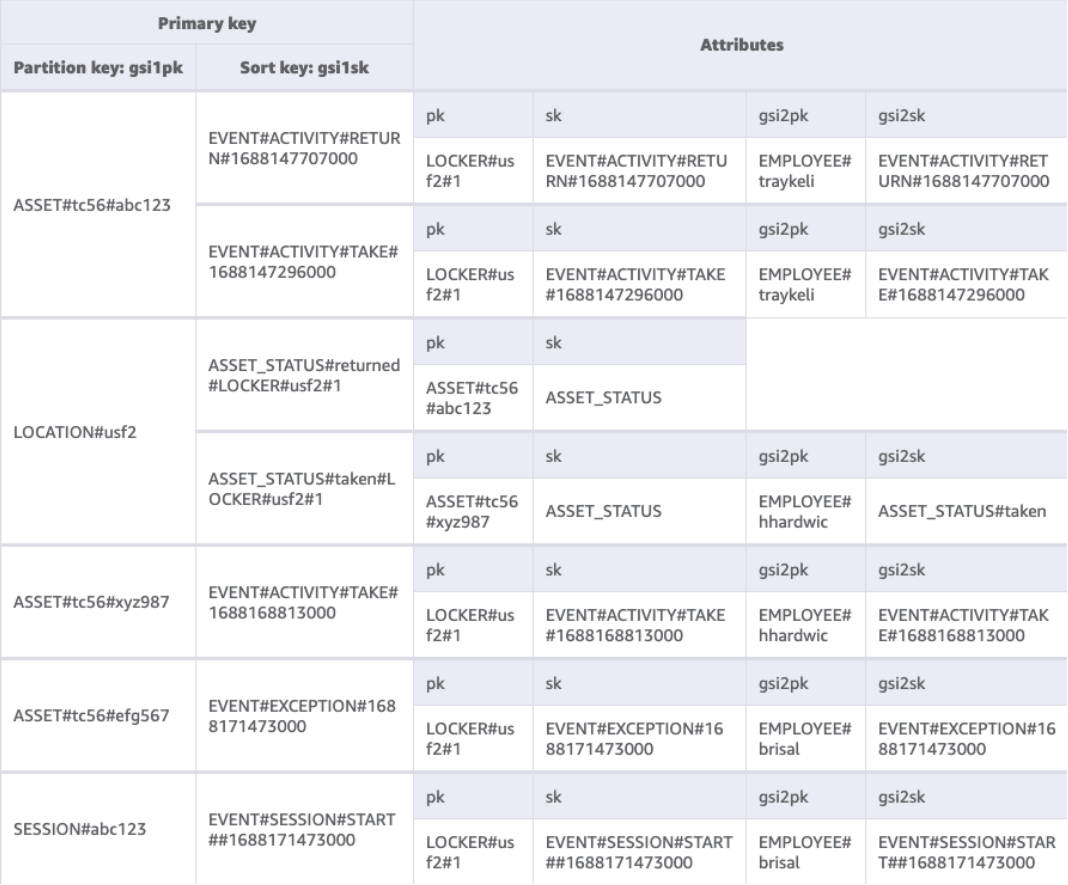
- Selection Criteria Single device:
  - Take Shelf: the shelf with the most available slots. Achieved by querying on pk, sorting on begins with CABINET#, and filtering with hasAsset = false. Then processing the result on the backend (return the shelf number with the least assets)
  - Return Shelf: the shelf with the most available slots. Achieved by querying on pk, sorting on begins with CABINET#, and filtering with hasAsset = true. Then processing the result on the backend (return the shelf number with the least assets)
  - Broken Return Shelf: the shelf labelled broken with most available slots. There will be n queries (based on the amount of broken shelves in locker) on pk, sorting on begins with CABINET#<cabinetNumber>#SHELF#<shelfNumber>#, and filtering with hasAsset = false.
- Selection Criteria Dual (n) devices:
  - These should be the same but passing in cabinets specified to the assets
  - Inferences can be made from asset tracking paradigm above (if user has two checked out assume they will want to return two, open two doors, and provide options otherwise tbd)

Deadbolt Main Table



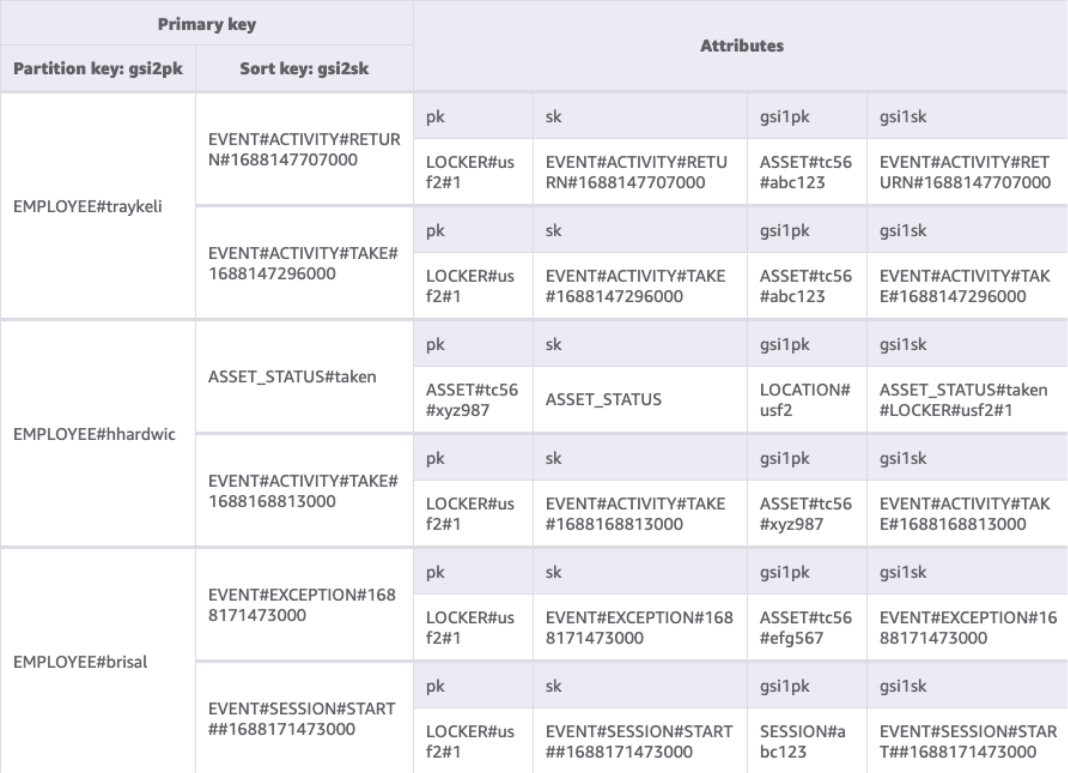
Primary key		Attributes			
Partition key: pk	Sort key: sk				
LOCKER#usf2#1	CABINET#001#DATA				
	CABINET#001#SHELF#001#DATA				
	CABINET#001#SHELF#001#SLOT#001#DATA				
	CABINET#001#SHELF#001#SLOT#002#DATA				
	CABINET#001#SHELF#001#SLOT#003#DATA				
	DATA				
	EVENT#ACTIVITY#RETURN#1688147707000	gsi1pk	gsi1sk	gsi2pk	gsi2sk
		ASSET#tc56#abc123	EVENT#ACTIVITY#RETURN#1688147707000	EMPLOYEE#traykeli	EVENT#ACTIVITY#RETURN#1688147707000
	EVENT#ACTIVITY#TAKE#1688147296000	gsi1pk	gsi1sk	gsi2pk	gsi2sk
		ASSET#tc56#abc123	EVENT#ACTIVITY#TAKE#1688147296000	EMPLOYEE#traykeli	EVENT#ACTIVITY#TAKE#1688147296000
	EVENT#ACTIVITY#TAKE#1688168813000	gsi1pk	gsi1sk	gsi2pk	gsi2sk
		ASSET#tc56#xyz987	EVENT#ACTIVITY#TAKE#1688168813000	EMPLOYEE#hhardwic	EVENT#ACTIVITY#TAKE#1688168813000
	EVENT#EXCEPTION#1688171473000	gsi1pk	gsi1sk	gsi2pk	gsi2sk
		ASSET#tc56#efg567	EVENT#EXCEPTION#1688171473000	EMPLOYEE#brisal	EVENT#EXCEPTION#1688171473000
	EVENT#SESSION#START##1688171473000	gsi1pk	gsi1sk	gsi2pk	gsi2sk
		SESSION#abc123	EVENT#SESSION#START##1688171473000	EMPLOYEE#brisal	EVENT#SESSION#START##1688171473000
ASSET#tc56#abc123	ASSET_STATUS	gsi1pk	gsi1sk		
		LOCATION#usf2	ASSET_STATUS#returned#LOCKER#usf2#1		
ASSET#tc56#xyz987	ASSET_STATUS	gsi1pk	gsi1sk	gsi2pk	gsi2sk
		LOCATION#usf2	ASSET_STATUS#taken#LOCKER#usf2#1	EMPLOYEE#hhardwic	ASSET_STATUS#taken

GSI 1



Primary key		Attributes			
Partition key: gsi1pk	Sort key: gsi1sk				
ASSET#tc56#abc123	EVENT#ACTIVITY#RETURN#1688147707000	pk	sk	gsi2pk	gsi2sk
		LOCKER#usf2#1	EVENT#ACTIVITY#RETURN#1688147707000	EMPLOYEE#traykeli	EVENT#ACTIVITY#RETURN#1688147707000
	EVENT#ACTIVITY#TAKE#1688147296000	pk	sk	gsi2pk	gsi2sk
		LOCKER#usf2#1	EVENT#ACTIVITY#TAKE#1688147296000	EMPLOYEE#traykeli	EVENT#ACTIVITY#TAKE#1688147296000
LOCATION#usf2	ASSET_STATUS#returned#LOCKER#usf2#1	pk	sk		
		ASSET#tc56#abc123	ASSET_STATUS		
	ASSET_STATUS#taken#LOCKER#usf2#1	pk	sk	gsi2pk	gsi2sk
		ASSET#tc56#xyz987	ASSET_STATUS	EMPLOYEE#hhardwic	ASSET_STATUS#taken
ASSET#tc56#xyz987	EVENT#ACTIVITY#TAKE#1688168813000	pk	sk	gsi2pk	gsi2sk
		LOCKER#usf2#1	EVENT#ACTIVITY#TAKE#1688168813000	EMPLOYEE#hhardwic	EVENT#ACTIVITY#TAKE#1688168813000
ASSET#tc56#efg567	EVENT#EXCEPTION#1688171473000	pk	sk	gsi2pk	gsi2sk
		LOCKER#usf2#1	EVENT#EXCEPTION#1688171473000	EMPLOYEE#brisal	EVENT#EXCEPTION#1688171473000
SESSION#abc123	EVENT#SESSION#START##1688171473000	pk	sk	gsi2pk	gsi2sk
		LOCKER#usf2#1	EVENT#SESSION#START##1688171473000	EMPLOYEE#brisal	EVENT#SESSION#START##1688171473000

GSI 2



Primary key		Attributes			
Partition key: gsi2pk	Sort key: gsi2sk				
EMPLOYEE#traykeli	EVENT#ACTIVITY#RETURN#1688147707000	pk	sk	gsi1pk	gsi1sk
		LOCKER#usf2#1	EVENT#ACTIVITY#RETURN#1688147707000	ASSET#tc56#abc123	EVENT#ACTIVITY#RETURN#1688147707000
	EVENT#ACTIVITY#TAKE#1688147296000	pk	sk	gsi1pk	gsi1sk
		LOCKER#usf2#1	EVENT#ACTIVITY#TAKE#1688147296000	ASSET#tc56#abc123	EVENT#ACTIVITY#TAKE#1688147296000
EMPLOYEE#hhardwic	ASSET_STATUS#taken	pk	sk	gsi1pk	gsi1sk
		ASSET#tc56#xyz987	ASSET_STATUS	LOCATION#usf2	ASSET_STATUS#taken#LOCKER#usf2#1
	EVENT#ACTIVITY#TAKE#1688168813000	pk	sk	gsi1pk	gsi1sk
		LOCKER#usf2#1	EVENT#ACTIVITY#TAKE#1688168813000	ASSET#tc56#xyz987	EVENT#ACTIVITY#TAKE#1688168813000
EMPLOYEE#brisal	EVENT#EXCEPTION#1688171473000	pk	sk	gsi1pk	gsi1sk
		LOCKER#usf2#1	EVENT#EXCEPTION#1688171473000	ASSET#tc56#efg567	EVENT#EXCEPTION#1688171473000
	EVENT#SESSION#START##1688171473000	pk	sk	gsi1pk	gsi1sk
		LOCKER#usf2#1	EVENT#SESSION#START##1688171473000	SESSION#abc123	EVENT#SESSION#START##1688171473000

Access Patterns

Assumptions:

1. Location is already apart of a given Locker ID. For instance, a locker with the Locker ID of usf2-5 entails that we are dealing with locker number 5 at the usf2 location. Thus, location can be derived from locker ID.
2. <someVariable> will be replaced with parameters, otherwise the string literal shown below will be the assumed value.
3. Any values dealing with times, dates, or ranges will be a Unix timestamp in milliseconds

Gets

Access Pattern	**Query Type**	**Required Parameters**
Get shelf with most devices by cabinet	pk, sk begins_with query and 1 filter (pk = LOCKER#<location>#<lockerId>, sk = CABINET#<cabinetNumber>#SHELF#, filter = where hasDevice is true)	location, lockerId, cabinetNumber
Get shelf with least devices by cabinet	pk, sk begins_with query and 1 filter (pk = LOCKER#<location>#<lockerId>, sk = CABINET#<cabinetNumber>#SHELF#, filter = where hasDevice is false)	location, lockerId, cabinetNumber
Get all slot data for a given shelf	pk, sk begins_with query (pk = LOCKER#<location>#<lockerId>, sk = CABINET#<cabinetNumber>#SHELF#<shelfNumber>#SLOT#)	location, lockerId, cabinetNumber, shelfNumber
Get locker	pk, sk equal_to query (pk = LOCKER#<location>#<lockerId>, sk = DATA)	location, lockerId
Get all taken assets by location (can be used for return as well)	gsi1pk gsi1sk begins_with query (gsi1pk = LOCATION#<location>, gsi1sk = ASSET_STATUS#taken)	location
Get all taken assets by locker (can be used for return as well)	gsi1pk, gsi1sk begins_with query (gsi1pk = LOCATION#<location>, gsi1sk = ASSET_STATUS#takenLOCKER#<location>#<lockerId>)	location, lockerId
Get all activity by asset ID	gsi1pk, gsi1sk begins_with query (gsi1pk = ASSET#<assetId>, gsi1sk = EVENT#ACTIVITY#)	assetId
Get asset count for a locker	pk, sk begins_with query and 1 filter (pk = LOCKER#<location>#<lockerId>, sk = CABINET#, filter = where hasDevice is true)	location, lockerId
Get slot count for a locker	pk, sk begins_with query and 1 filter (pk = LOCKER#<location>#<lockerId>, sk = CABINET#, filter = where hasDevice is false)	location, lockerId
Get broken Device count for a locker	pk, sk begins_with query and 1 filters (pk = LOCKER#<location>#<lockerId>, sk = CABINET#, filter = where isBrokenDeviceSlot equals true)	location, lockerId,
Get all exceptions by location after a given date	pk, sk greater_than query (pk = LOCKER#<location>#<lockerId>, sk = EVENT#EXCEPTION#<timestamp>)	location, timestamp
Get all exceptions by locker after a given date	pk, sk greater_than query (pk = LOCKER#<location>#<lockerId>, sk = EVENT#EXCEPTION#<timestamp>)	lockerId, timestamp
Get all activity by employee ID after a given date	gsi2pk, gsi2sk greater_than query (gsi2pk = EMPLOYEE#<employeeId>, gsi2sk = EVENT#ACTIVITY#<timestamp>)	employeeId, timestamp
Get all exceptions by employee ID after a given date	gsi2pk, gsi2sk greater_than query (gsi2pk = EMPLOYEE#<employeeId>, gsi2sk= EVENT#EXCEPTION#<timestamp>)	employeeId, timestamp
Get all sessions by locker after a given date	pk, sk greater_than query (pk = LOCKER#<location>#<lockerId>, sk = EVENT#SESSION#START#<timestamp>)	location, lockerId,
Get session by session ID	gsi1pk, gsi1sk equals query (gsi1pk = SESSION#<sessionId>, gsi1sk= EVENT#SESSION#)	sessionId
Get sessions by employee ID	gsi2pk, gsi2sk begins_with query (gsi2pk = EMPLOYEE#<employeeId>, gsi2sk = EVENT#SESSION#)	employeeId

Creates

Create Type	**Number of items created**	**Required Parameters**
Create Locker	1 Locker item, K Cabinet Item, N Shelf Items, K x N x M Slot Items => 1 + K + N + (K x N x M) items	location, lockerId, cabinetNumber, shelfNumber, slotNumber
Create Exception	1 Exception item	location, lockerId, timestamp
Create Take Activity	1 Event Item, 1 Asset Item	location, lockerId, timestamp, assetId, employeeId
Create Return Activity	1 Event Item, 1 Asset Item	location, lockerId, timestamp, assetId, employeeId

Updates

Update Type	**Items touched**	**Required Parameters**
Update Locker data	1 Locker Item (pk = LOCKER#<location>#<lockerId>, sk = DATA)	location, lockerId

Update Cabinet data	1 Cabinet Item (pk = LOCKER#<location>#<lockerId>, sk = CABINET#<cabinetNumber>#DATA)	location, lockerId, cabinetNumber
Update Shelf data	1 Shelf Item (pk = LOCKER#<location>#<lockerId>, sk = CABINET#<cabinetNumber>#SHELF#<shelfNumber>#DATA)	location, lockerId, cabinetNumber, shelfNumber
Update Slot data	1 Slot Item (pk = LOCKER#<location>#<lockerId>, sk = CABINET#<cabinetNumber>#SHELF#<shelfNumber>#SLOT#<slotNumber>#DATA)	location, lockerId, cabinetNumber, shelfNumber, slotNumber
Update Locker	1 Locker item, K Cabinet Items, N Shelf Items, K X N x M Slot Items => 1 + K + N + (K x N x M) items	location, lockerId, cabinetNumber, shelfNumbers, slotNumbers

APIs and Ingestion Message handling

API

/CreateLocker/{lockerId}

Request

```
{
  "locker_id": "usf2-1",
  "locker": {
    "lockerFriendlyName": "ye",
    "lockerNumber": 1,
    "whid": "lockergsi2",
    "businessLine": "bl1",
    "lockerType": "ring",
    "columnCount": 1,
    "columns": [
      {
        "mac": "1",
        "position": "1",
        "shelves": [
          {
            "shelfId": "shelfId",
            "column": 1,
            "macAddress": "macAddress1",
            "slotCount": 10,
            "deviceCount": 10,
            "deviceType": "ring",
            "shelfNumber": 1,
            "isScreen": false,
            "shelfHeight": 10,
            "isEnabled": true,
            "isDamagedDeviceShelf": false
          },
          {
            "shelfId": "shelfId2",
            "column": 1,
            "macAddress": "macAddress1",
            "slotCount": 10,
            "deviceCount": 10,
            "deviceType": "ring",
            "shelfNumber": 2,
            "isScreen": false,
            "shelfHeight": 10,
            "isEnabled": true,
            "isDamagedDeviceShelf": false
          }
        ]
      }
    ]
  }

}
```

Response

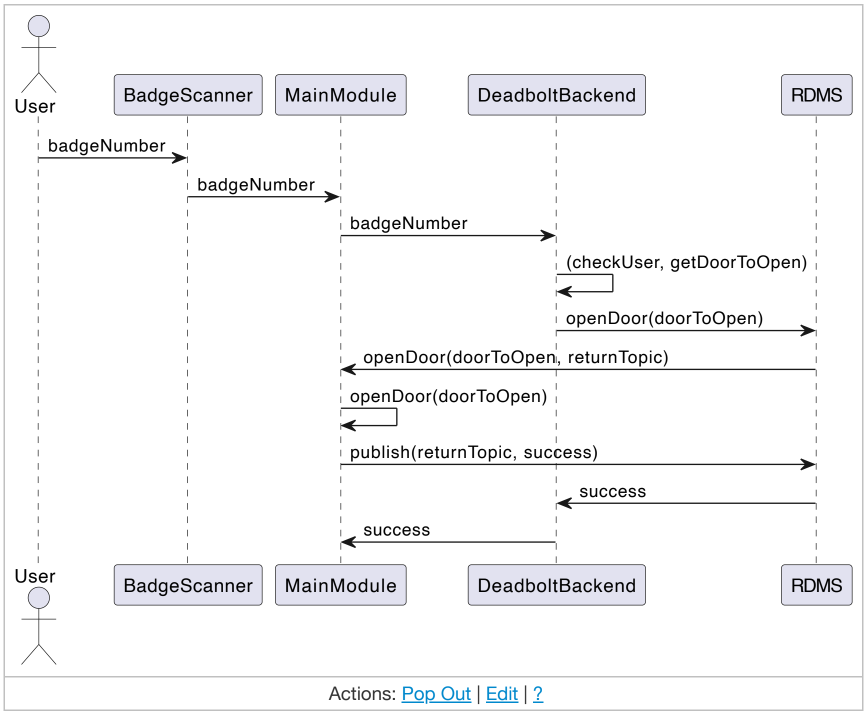
```
{
  "message": "locker created successfully",
  "status": 200,
}
```

Ingestion Message Handling:

RDMS and Deadbolt backend Sequence Diagram

Ingestion Message Handling calls involving RDMS follow this general flow:

Sequence Diagram



Actions: Pop Out | Edit | ?

RDMS and Main Module exchange

RDMS will send a payload below via MQTT:

```
{
    "mac": "fcc23d13afc8",
    "command": "unlatch",
    "details": {
        "return_id": "1234"
        "publish_response_topic": "08c71152-c552-42e7-b094-f510ff44e9cb"
    }
}
```

Leveraging RDMS's **SendCommandToDeviceRequest** an MQTT payload can be directed to the proper locker

```
final SendCommandToDeviceRequest request =
    SendCommandToDeviceRequest.builder()
        .application("deadbolt")
        .context("usf2")
        .deviceId("locker1-cabinet1")
        .command("unlatch")
        .payload(payload)
        .requestResponse(true)
        .build();
```

Note: the deviceId is the following combination locker<lockerNumber>-cabinet<cabinetNumber>.

The Main Module will then execute the command, and if successful it will publish the following payload to the `publish_response_topic`:

```
{
    "return_id": "1234",
    "message_id": "e460285b-547f-4d05-9672-d6ac8202bce3",
    "data": {
        "success": true
    }
}
```

Otherwise if the door does not open, an error payload will be published to the `publish_response_topic`:

```
{
    "return_id": "1234",
    "message_id": "e460285b-547f-4d05-9672-d6ac8202bce3",
    "data": {
        "success": false
        "error": {
           "message": "door latch did not open"
           "error_code": 1203
        }
    }
}
```

RDMS can forward the data from the payload back to the Deadbolt API and proper error handling and HTTP message transformation can be returned to the client.

OpenBrokenShelf

Request

```
{
  "locker_id": "usf2-1",
  "__metadata": {
    "timestamp": 1234567890,
  }
}
```

Response

```
{
  "message": "opening door <MAC>",
  "success": true,
}
```

OpenTakeShelf

Request

```
{
  "locker_id": "usf2-1",
  "__metadata": {
    "timestamp": 1234567890,
  }
}
```

Response

```
{
  "message": "opening door <MAC>",
  "success": true,
}
```

OpenReturnShelf

Request

```
{
  "locker_id": "usf2-1",
  "__metadata": {
    "timestamp": 1234567890,
  }
}
```

Response

```
{
  "message": "opening door <MAC>",
  "success": true,
}
```

GetChargingSlots

Request

```
{
  "locker_id": "usf2-1",
  "__metadata": {
    "timestamp": 1234567890,
  }
}
```

Response

```
{
    "message": "successfully retrieved charged slots",
    "success": true,
    "data": {
        "shelves": [
            {
                "mac": "fcc23d13afc8",
                "slots": [
                    {
                        "slot": 2,
                        "charge": "52mAh",
                        "time": "2min",
                        "status": "charging",
                        "current": "62mA",
                        "proximity": "present",
                        "proxtime": "2min"
                    },
                    {
                        "slot": 5,
                        "charge": "12mAh",
                        "time": "78min",
                        "status": "charging",
                        "current": "17mA",
                        "proximity": "present",
                        "proxtime": "78min"
                    },
                    {
                        "slot": 6,
                        "charge": "93mAh",
                        "time": "1min",
                        "status": "charging",
                        "current": "622mA",
                        "proximity": "present",
                        "proxtime": "1min"
                    },
                    {
                        "slot": 7,
                        "charge": "9mAh",
                        "time": "102min",
                        "status": "charging",
                        "current": "6mA",
                        "proximity": "present",
                        "proxtime": "102min"
                    }
                ]
            }
        ]
    }
}
```

GetEmptySlots

Request

```
{
  "locker_id": "usf2-1",
  "__metadata": {
    "timestamp": 1234567890,
  }
}
```

Response

```
{
    "message": "successfully got empty slots",
    "success": true,
    "data": {
        "shelves": [
            {
                "mac": "fcc23d13afc8",
                "slots": [
                    {
                        "slot": 1,
                        "charge": "0mAh",
                        "time": "0min",
                        "status": "unoccupied",
                        "current": "0mA",
                        "proximity": "absent",
                        "proxtime": "0min"
                    },
                    {
                        "slot": 3,
                        "charge": "0mAh",
                        "time": "0min",
                        "status": "unoccupied",
                        "current": "0mA",
                        "proximity": "absent",
                        "proxtime": "0min"
                    },
                    {
                        "slot": 4,
                        "charge": "0mAh",
                        "time": "0min",
                        "status": "unoccupied",
                        "current": "0mA",
                        "proximity": "absent",
                        "proxtime": "0min"
                    },
                    {
                        "slot": 8,
                        "charge": "0mAh",
                        "time": "0min",
                        "status": "unoccupied",
                        "current": "0mA",
                        "proximity": "absent",
                        "proxtime": "0min"
                    }
                ]
            }
        ]
    }
}
```

HandleBadgeScan

Request

```
{
  "badgeNumber": "29382225",
  "__metadata": {
    "timestamp": 1234567890,
  }
}
```

Response

```
{
   "data": {
      "getAuthorization": {
         "authorization": {
            "isManager": true,
            "userAlias": "traykeli"
          }
      }
   },
   "message": "Authorization Call Successful",
   "success": true,
}
```

Provisioning the System

AWS IoT Fleet Provisioning with Greengrass

Each Main Module will be manufactured with a static provisioning claim certificate, signed by a DPS subordinate CA. Upon first connection, this certificate is exchanged for a device certificate that will be used to connect to IoT Core during normal operation.

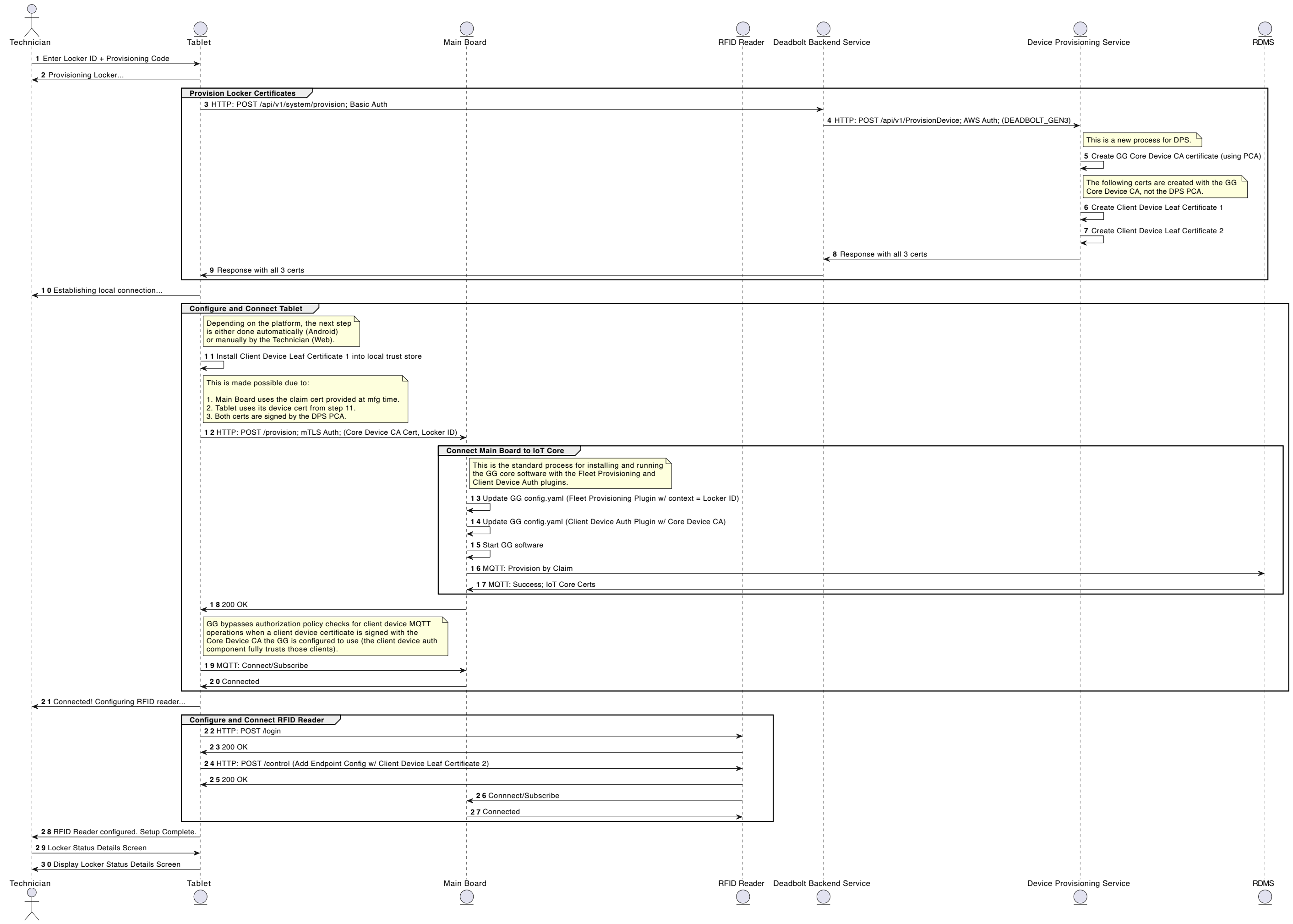
The first connection is facilitated by a setup flow initiated from the tablet interface. The tablet connects to the Main Module over an HTTPS configuration endpoint[1] and provides the Main Module with the locker information and Core Device CA[2]. The Core Device CA and the associated client device certificates for the system are acquired from DPS during the setup flow.

The software team will create the necessary roles, templates, and configuration details based on the target environment: (alpha, beta, gamma, prod) to enable provisioning.

[1] For security during setup, the HTTPS configuration endpoint utilizes mTLS. The server certificate used by the Main Module is the same claim certificate mentioned above and the client certificate used by the tablet is acquired through DPS.

[2] The Core Device CA is used for local client device communication between all devices on the local locker network.

Detailed Sequence Diagram



1 Enter Locker ID + Provisioning Code
2 Provisioning Locker...

Provision Locker Certificates

3 HTTP: POST /api/v1/system/provision; Basic Auth
4 HTTP: POST /api/v1/ProvisionDevice; AWS Auth; (DEADBOLT_GEN3)

This is a new process for DPS.

5 Create GG Core Device CA certificate (using PCA)

The following certs are created with the GG Core Device CA, not the DPS PCA.

6 Create Client Device Leaf Certificate 1
7 Create Client Device Leaf Certificate 2
8 Response with all 3 certs
9 Response with all 3 certs
10 Establishing local connection...

Configure and Connect Tablet

Depending on the platform, the next step is either done automatically (Android) or manually by the Technician (Web).

11 Install Client Device Leaf Certificate 1 into local trust store

This is made possible due to:

1. Main Board uses the claim cert provided at mfg time.
2. Tablet uses its device cert from step 11.
3. Both certs are signed by the DPS PCA.

12 HTTP: POST /provision; mTLS Auth; (Core Device CA Cert, Locker ID)

Connect Main Board to IoT Core

This is the standard process for installing and running the GG core software with the Fleet Provisioning and Client Device Auth plugins.

13 Update GG config.yaml (Fleet Provisioning Plugin w/ context = Locker ID)
14 Update GG config.yaml (Client Device Auth Plugin w/ Core Device CA)
15 Start GG software
16 MQTT: Provision by Claim
17 MQTT: Success; IoT Core Certs
18 200 OK

GG bypasses authorization policy checks for client device MQTT operations when a client device certificate is signed with the Core Device CA the GG is configured to use (the client device auth component fully trusts those clients).

19 MQTT: Connect/Subscribe
20 Connected
21 Connected! Configuring RFID reader...

Configure and Connect RFID Reader

22 HTTP: POST /login
23 200 OK
24 HTTP: POST /control (Add Endpoint Config w/ Client Device Leaf Certificate 2)
25 200 OK
26 Connnect/Subscribe
27 Connected
28 RFID Reader configured. Setup Complete.
29 Locker Status Details Screen
30 Display Locker Status Details Screen

Additional details on how the Fleet Provisioning configuration and how it will be implemented are documented here.

Firmware Specific Domain

The sections below are catered more towards the Firmware in the overall design.

Main Module / IoT Core Provisioning

Provisioning Steps (High Level)

1. Tablet subscribes to the following topics: `mainModule/+`, `provisioning`
2. Upon new client connection, all Main Modules send out their mac address along with a list of mac addresses of their Charger Boards to the `mainModule` topic:

   -
     ```
     {
        "mainModuleMac": <some_mac>,
        "chargerBoards": [<chargerBoardMac1>, <chargerBoardMac2>, ...]
     }
     ```
3. Tablet will publish door open commands to each of the mac addresses and build a json virtual mapping of the Locker
4. When the mapping is done, the Tablet will publish to a `provisionLocker` topic passing the locker ID and provisioning code, the Main Module will publish the result to the `provisioning` topic (basically a skeleton of the locker with no associated mac addresses).

   -
     ```
     // received from tablet
     {
        "lockerId": <some_id>,
        "provisionCode": <some_provision_code>
     }

     // returned from provisionLocker
     {
        lockerId: <some_id>,
        lockerType: <some_type>,
        columns: [
           {
              mac: "",
              position: <number>,
              shelves: [
                 {
                    column: <column>,
                    deviceType: <some_device_type>,
                    isScreen: false,
     ```

```
                    position: <number>,
                    shelfHeight: <number>,
                    slotCount: <number>,
                    isDamagedDeviceShelf: <boolean>,
                    macAddress: ""
                },
                ...
            ]
        },
        ...
    ]
    whid: <some_whid>,
    businessLine: <some_business_line>,
    maintenanceModePin: <number>,
    ... <tbd>
}
```

5. Upon a successful provisioning, the virtual mapping of the Locker will finally be used to update the virtual locker. Each Main Module will want to store their position and the lockerId locally, as those two values will be needed in the topics that they will be publishing in IoT Core.

Firmware communication

This section will describe the (high level) communication between the following relationships.

- Main Module and Charger Board
- Main Module and IoT Core
- Main Module and Tablet

Must ensure that we adhere to these limit restrictions for IoT Core. Note that the max topic limit is 8 (for MQTT5).

Current Communication - Android & Beagle Bone

1690555643437-151.png

Deadbolt Generation 2 (how it works today)

Android subscriptions to Beagle Bone

Topic	Message	How it's handled
“shelf-v1/<MAC>/status”	"online" | "(offline)"	publish rate: When it first establishes a connection / On change if "(offline)" => Tablet report exception via API call and updates locker state via RoomDB tablet persistence if "online" => update locker state via RoomDB tablet persistence
“shelf-v1/<MAC>/info”	“shelf-door:<position> cabinet-door:<position> fault:<CSV list of error codes>”	publish rate: When it first establishes a connection / On change if fault does not equal 00 or empty => parse error codes, if any are bad* throw exception via API call upon publish (on change) => convert to JSON object, update locker state via RoomDB tablet persistence layer <position> = “open” | “closed” | “not-connected”
“shelf-v1/<MAC>/slot<N>/info”	“status:<status> current:<current> time:<time on charge> charge:<charge delivered> proximity:<proximity> proxtime: <proxtime>"	publish rate: One second in round robin fashion (charger board will report on a different slot sequentially each second) upon publish (one second round robin) => convert to JSON object, update locker state via RoomDB tablet persistence layer <status> = "unoccupied" | "charging" | "fault" <current> = “1200mA” (current value range [0-3000] mA) <time on charge> = “45min” (number of minutes on charge, range [0-1080] minutes) <charge delivered> = “2234mAh” (charge in mAh delivered, range [0-5000] mAh) <proximity> = "absent" | "present" <proxtime> = "45min” (number of minutes on charge, range [0-1080] minutes)

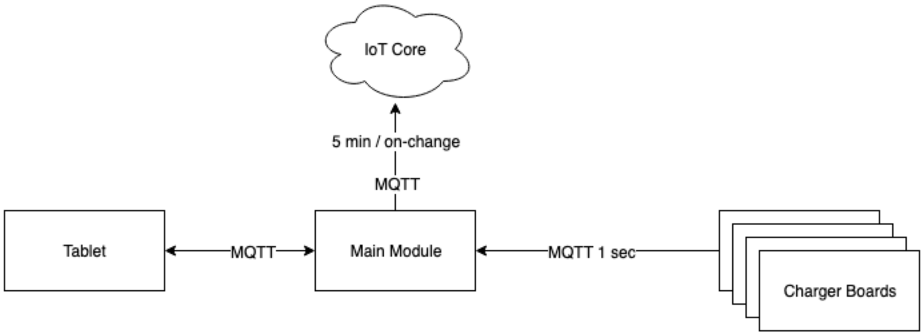
*bad was determined by USA Firmware as non-functional, some fault codes were benign (charger over currents for instance). Here are the codes that currently throw exceptions: 01, 02, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 52 - 57

Android publishes to Beagle Bone

Topic	Message	How it's handled

“shelf-v1/<MAC>/shelf-door-release”	"release"	validate door open via physical door confirmation & handled Charger Board will write the message “hold” when latch operation has completed. The shelf-v1/<MAC>/info topic will indicate shelf door position, or fault topic on failure. Upon receiving the message, Android updates the locker state via RoomDB or throws an API exception (if there was a fault)

Future Communication



Deadbolt Generation 3

Mainboard publishes to IoT Core

Publishes

- Cabinet level updates
  - topic: `dt/deadbolt/–/cabinet`
    - example: `dt/deadbolt/usf2–2/cabinet1`
  - message:

```
{ "data": { "status": , "fault": [] "assets": [] }, "meta–data": { "timestamp":, "mac": } }
```

  - publish rate: every 5 minutes
- Shelf level updates
  - topic: `dt/deadbolt/–/cabinet/shelf`
    - example: `dt/deadbolt/usf2–2/cabinet1/shelf1`
  - message:

```
{ "data": { "status": , "shelf–door":, "fault": [] }, "meta–data": { "timestamp": 1688745553, "last–updated":
, "mac": } }
```

publish rate: on-change * Slot level updates topic: `dt/deadbolt/–/cabinet/shelf/slot`

  - example: `dt/deadbolt/usf2–2/cabinet1/shelf1/slot5`
  - message:

```
{ "data": { "status":, "current":, "time":, "charge":, "proximity":, "proxtime": , }, "meta–data": { "time–
stamp":, } }
```

publish rate: every 5 minutes / on-change * Heartbeat topic: `dt/deadbolt/–/cabinet/heartbeat`

  - example: `dt/deadbolt/usf2–1/cabinet2/heartbeat`
  - message:

```
{ "data": { "status":, "firmware–version": , "mac": , "tbd": , "timestamp": } }
```

  - publish rate: every 5 minutes

Main Module subscriptions to IoT Core

- cmd
  - topic: `dt/deadbolt/–/cabinet/cmd`
    - example: `dt/deadbolt/usf2–1/cabinet2/cmd`
  - message: JSON data to facilitate command being sent from RDMS
    - example:

```
{ "data": { "cmd":"unlatch", "details": { "response_requested": true, "return_id": "1234", "return_topic":
"08c71152–c552–42e7–b094–f510ff44e9cb", "mac": "fcc23d15faa2", } } }
```

- cmd
  - `unlatch`: when this keyword is provided, a mac address of the shelf to unlatch (aka open) is provided, along with a topic to publish to whether the command was successful or not.
  - `update_charger_board_fw`: when this keyword is provided, updates the firmware of the charger board associated with the main board tbd.
  - `reset`: when this keyword is provided, reset the state of the locker (linux level on main board)
  - `send_error`: when this keyword is provided, sends an error that the tablet could reflect
  - tbd

Main Module and Peripheral MQTT connections

The Main Module will interact with different RFID readers via MQTT protocol as well (Zebra IoT connection docs here) , we will also need to have the tablet be listening to these events as they happen. Development will be done on the client (Android, Web App, etc.) to listen to said events via MQTT subscriptions. Assuming that as these RFID events happen and are published accordingly, the Tablet interface will have all the data it needs to update the UI.

Subscriptions

- badgeScan: topic to let the tablet know a badge scan has taken place
- badgeScanComplete: topic to let the tablet know the badge scan has finished
- doorOpened: topic to let the tablet know the door has just opened, along with the associated shelf, and slot data (for the UI)
- doorClosed: topic to let the tablet know the door just closed Publishes for provisioning
- provisionLocker: Tablet provide the locker Id and provision code
- updateLocker: Tablet verifies locker structure and data from client. MAC addresses are filled in to the virtual locker

Offline Mode

Upon scaling from tens to hundreds of lockers launched, it is a fair assumption that not all lockers will have reliable internet connectivity. Deadbolt will need a mechanism to provide customers access to assets even while there is no internet connection.

RFID Scanning enhancement

In production today, the default scanning method is used. The real First step would be to verify that this scanner only works with Amazon blue badges, if this is the case then scanning enhancement steps below can be skipped.

1. Deep dive on RFID scanning
    - only allow blue badges scan access
    - program executable resource: RFID program download
    - other team doc wikis: wiki1 wiki2

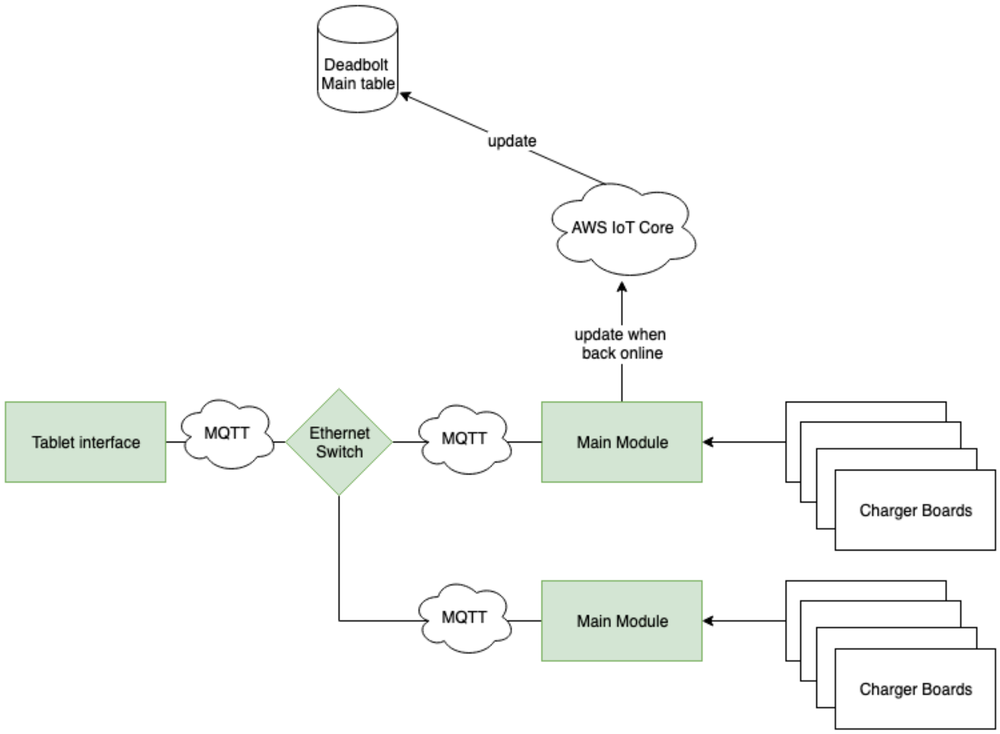
Rational for this is if the locker is offline, we want to ensure that only blue badges work with the scanner since that activity data and RFID number will be cached locally and sent up to our backend upon reconnecting to the internet

MQTT Local Server (Preferred Option)

The tablet will have direct ethernet connection to the Main Module. The Main module could then host a local MQTT server via greengrass listening for the following topics:

- getReturnDoor
    - door with least assets
- getTakeDoor
    - door with most assets
- getBrokenDoor
    - door assigned to broken assets
- getDeviceCount
    - total asset count
- getDevicesForShelf
    - used to update the UI

While offline, the MainModule would cache the activity locally and then send the historical activity data to the backend with a specific topic (meant to differentiate between normal active operations and offline operations).



Pros

- Leverages the existing MQTT broker (which will already be communicating with RFID)

Cons

- Must ensure that the offline topic is triggered for historical data

Purposed Solution Pros / Cons

Pros

- Remote access to lockers opens new possibilities
  - OTA FW updates, saves money without needing to send techs to update firmware
  - Locker updates/status
  - Health checks
- Removes the need for a specific tablet/screen
  - The android application will be making basic HTTP calls and no longer being hardwired to the PCBs. This opens new screen options, web portal options, etc.
- Sets a precedence for IoT hardware development in Rapid and Rural Logistics
  - There are many standards that exist for software best practices within amazon, the same is not true for Hardware/Firmware development
  - Pioneering the space allows our org to grow to new markets and add value to the company

Cons

- Complexity of managing multiple Cabinets in one Locker entity may reveal unexpected/unplanned bugs or behavior
- Rigid topic structure. If we were to have a shelf halved for instance (column wise) our topic structure wouldn't support it.

Spend

Note: this cost summary focuses only on AWS IoT costs given that its a new capability that the team hasn't investigated previously. Costs for Lambda compute, DynamoDB, and application integration services are expected to be within the norms of other GSF/SCAR services.

AWS IoT Monthly Deadbolt Cost

Service	Charges	Anticipated Monthly Device Usage (Launch)	Monthly Cost
Connectivity	$0.08 per 1M minutes of connection	**43,200 minutes** (connected 24/7)	$ 0.0035
Messages (MQTT and HTTP)	$1.00 per 1M messages*	8,640 device health messages (1 every 5 minutes) 8,640 update device count command (1 every 5 minutes) 1,020 update slot command (on-change est. 1 every 42 minutes) 1,020 door open commands (1 every 42 minutes) 1,020 get open/close door command (1 every 42 minutes) 8,640 update slot command (1 every 5 minutes) 1,020 update slot command (on-change est. 1 every 42 minutes)	$ 0.03

		30,000 total messages	
Rules	$0.15 per 1M rules triggered / actions applied	25,920 rules initiated (3 per health message) 25,920 actions applied (3 per health message) **51,840 total rules/actions**	$ 0.008
Events	$15.00 per 1M message evaluations	**8,640 rules initiated** (1 per health message)	$ 0.13
			Total: $ 0.17

Total Monthly Cost (various architecture and scale scenarios)

Number of Lockers	**Total Monthly Cost**
100	**$ 17.00**
500	**$ 85.00**
1000	**$ 170.00**
5000	**$ 850.00**

Cost Savings Implementations

1. Restructure the publish rate of the Main Module to IoT Core (currently in production PCB boards are publishing slot information every 1 second). Early designs talked about publishing slot data at a 15 second interval. In the new design 5 minutes coupled with on-change is the logical choice because:
    1. The monthly estimated IoT Core service cost drops from $63.22 (15 seconds) per locker to $0.03 (5 minutes). That is literally more than a 10,000x saving.
    2. We can perform all operations and collect all relevant metrics we do today with on-change at a Shelf level.
        - Shelf selection is based on the devices being there (we do not use the other metrics captured on a slot level [charge, time on charge, time on sensor, etc]). It would be pointless to aggregate gigabytes of slot data spending 10,000 times more, when it isn't used.
        - Device Count can also be optimized to only be on change (when proximity sensor recognizes that a device has been taken) we update via IoT Core
        - 15 minutes is sufficient to keep a cadence, prove out that a long running system is viable, prevents overly stale data, and we can always increase that rate when/if we find use cases in the future.
            - Pros
                - Old Monthly cost: $63.22 vs New Monthly cost: $0.01
                - Does not affect nor impact the overall system in anyway
                - Processing power used to deliver MQTT messages in a one second cadence will be distributed elsewhere (opening doors quicker, general responses, etc)
            - Cons
                - No longer have historical slot data on a one second level cadence (nor did we have a use case or reason for it)
2. We could further cost saving methods by leveraging Basic ingest as defined here. In doing this, we limit ourselves from IoT Core OOTB functionality (more info), but if we decide to never use these features we can mitigate all MQTT costs as a whole. This may be important if we need to increase the rate of MQTT publishes in the future.

Security

- This redesign of the system does not introduce new data (i.e. emails, etc)
- Communication between IoT core and Main Modules will be secured through the DPS
- Client access
    - From device via RFID scan (already in production), note that with the enhancement defined in offline mode, we can ensure that anyone with and RFID emitting device spoofing RFID numbers will not be able to enter the locker
    - From Deadbolt Backend post device provisioning via IAM roles (already in production)
- Given a data breach:
    - Open access to any deadbolt locker
    - Access to device data
    - Access to locker data

High Level Tech Scoping

The scope breakdown below uses the following t-shirt sizing to estimate the work required:

- XS - 1 dev week
- S - 2 dev weeks
- M - 4 dev weeks
- L - 8 dev weeks
- XL - 16 dev weeks

Work Item Number	Service/Component	Title	Description	Estimate	Dependency
1	Deadbolt Backend Service	DynamoDB Main Table 2.0	A new table created in GSF-HSE-DeadboltBackendServiceCDK code package.	XS	

2	Deadbolt Backend Service	Integrate with RDMS	Create a new topic that receives Deadbolt messages from RDMS.	XS	
3	Deadbolt Backend Service	Locker Heartbeat Ingestion	Create a queue + lambda that receives heartbeat messages from the integration topic and upserts the locker/cabinet/shelf/slot data into Main Table 2.0.	S	1, 2
4	Deadbolt Backend Service	Asset Event Ingestion Lambda	Create a queue + lambda that receives asset take/return (activity) messages from the integration topic and inserts them into Main Table 2.0.	S	1, 2
5	Deadbolt Backend Service	Offline Asset Event Ingestion Lambda	Create a queue + lambda that receives asset take/return (activity) messages from the integration topic which have been flagged as taken/returned when the device was offline. This lambda should perform additional validation to determine if the take/return was authorized. If the access should not have been granted, an exception message is sent to the SIM exception queue.	M	4
6	Deadbolt Backend Service	Exception Event Ingestion Lambda	Create a queue + lambda that receives locker exceptions messages from the integration topic and inserts them into Main Table 2.0.	S	1, 2
7	Deadbolt Backend Service	Deadbolt Shelf Selection Service	Create a service that queries Main Table 2.0 for different shelf doors (take, return, or broken). Take = most full shelf Return = most empty shelf Broken Take = most full broken shelf Broken Return = most empty broken shelf	XS	1, 2, 3
8	Deadbolt Backend Service	Deadbolt Asset Assignment Service	Create a service that queries Main Table 2.0 for the assets currently taken out by an employee. Returns: a list of Assets currently taken out by an employee.	XS	1, 2, 4
9	Deadbolt Backend Service	Handle Badge Scan Lambda	Create a lambda that listens to RDMS for Deadbolt badge scans. This lambda then invokes the existing authorization logic (which queries the Badge Access Service) and then blends data from the Shelf Selection and Asset Assignment services to determine what action to take. Based on that determination, it integrates with RDMS's Command Bridge to either send an unlock command for a specific locker door or a display error command.	S	7, 8
10	Deadbolt Data Pipeline	Update Andes Data Pipeline	Update the existing Andes data pipeline to consume the new Main Table 2.0 DDB stream.	M	1, 3, 4, (5), 6
11	Deadbolt UI Web Client	Initial Project Setup	Create the requisite Bindles, packages, and pipeline infrastructure for a new web application that builds and deploys to S3	XS	
12	Deadbolt UI Web Client	Implement New UI (Meridian) Screens	Port the existing UI screens from Android to React. During the port, implement the new Merdian-based UI. Screens to port: Home (Surge Flow) Home (Additional Options) Loading Select Device Settings [Others?]	L	11
13	Deadbolt UI Web Client	Local MQTT Communication Service (Design)	Design a local (browser based) service that can authenticate and subscribe to the Main Module's local MQTT broker. This design should define a protocol that enables: 1. The UI to be driven from certain screens to other screens when key messages are sent from the Main Module (e.g. the main module should be able to direct the UI from the loading screen to the Select Device screen). 2. The UI to query the Main Module for the status of a particular cabinet/shelf/slot for the purpose of refreshing the UI when on a particular screen (this includes RFID sensing through the Zebra local MQTT deployment option).	S	

14	Deadbolt UI Web Client	Local MQTT Communication Service (Implementation)	Implement a local (browser based) service that subscribes to the Main Module's local MQTT broker. This service should implement the design and integrate it into the actual UI (11).	M	11, 12, 13, 17
15	Deadbolt UI Web Client	Offline Mode Interface	Display a warning when the system is operating in Offline Mode.	XS	5, 11
16	Remote Device Management Service	Integrate Standard Detector Model	Integrate the standard RDMS detector model for Online/Offline transitions based on the Deadbolt locker heartbeat.	S	2, 3
17	Remote Device Management Service	Provisioning Policy/Role Infrastructure	Finalize the design of the policy, role, and provisioning templates needed to provision the lockers using IoT Fleet Management provisioning.	S	
18	Deadbolt Backend Service	API Gateway Locker Service	Create a Deadbolt 3.0 service using API Gateway + Lambda service for Deadbolt (to deprecate AppSync).	XS	
19	Deadbolt Backend Service	Locker Configuration Lambda	Create an API Gateway fronted Lambda that provides the ability to read the Locker's configuration from DDB and update the it through RDMS. May also need a "refresh" (read from device) and an "apply" to save a DDB configuration to the device directly.	M	1, 18
20	Deadbolt Backend Service	Integrate Load Testing	Integrate Hydra load testing w/ Deadbolt virtual devices.	S	
21	Device Provisioning Website	Deadbolt GEN 3.0 Support	Update the Device Provisioning Website to allow the configuration of the Deadbolt GEN3 locker.	M	19
22	Hardware/Firmware Integration	Hardware/Firmware Integration	Hardware/Firmware Integration	M	All

Roll Out Plan

Phase 1: IoT Core with old Hardware Prototype

Software team: will migrate the current shelf selection process to API calls instead of a local MQTT network. (Does not need to leave lab at AZA 10)

High-level tasks:

1. Firmware: Connect Beagle Bone to IoT core with greengrass using the current firmware running on the Charger Boards
2. Deadbolt Backend: Create the new Deadbolt Lambda, Deadbolt Dynamo DB v2, SNS ingestion topic, and SQS
3. Deadbolt Frontend: integrate new provisioning and change selection criteria from RoomDB to API calls
4. DPS: Update provisioning flow, return certs and keys mapped to specified AWS accounts (alpha, beta, prod) during provisioning step
5. RDMS: integrate with Deadbolt backend (Create necessary IoT Core rules, topics, and events)

Phase 2: Full integration of Main Module and Next Gen Charger Boards

The software team will focus on enabling OTA firmware updates, bugs, and testing APIs. The hardware will focus on updating the Charger Board FW (raw string to JSON, finalizing relevant data returned, etc) and other RFID enablement tasks

High-level tasks:

1. Firmware: Update firmware on Charger Boards (return JSON instead of strings, etc, Ovyl's choice)
2. Firmware: Finalize HTTP server communication with Tablet and RFID Scanner(s)
3. Deadbolt Frontend (WebApp): update locker mapping UI, MQTT subscriptions and publishes (getting RFID from HTTP instead of USB)
4. Deadbolt Provision Site: update default locker structures (Cabinet/Main Module)
5. RDMS: OTA firmware updates
6. Any carry over tasks not completed in phase 1

Phase 3: Monitoring and Deprecation

The software team will focus on Integration of HTTP LAN calls from the Main Module(s). The hardware team will focus on Main Module server hosting and provisioning.

High-level tasks:

1. RDMS: Heartbeat detector model (support multi Main Module)
2. Deadbolt Backend: Update data streams and Quicksight dashboards
3. Migrate old lockers in the field to new
4. Deprecate old code and libraries upon locker migrations
5. Any carry over tasks

Open Questions

1. Any ideas how to better handle multiple main modules?
2. Would it make sense to create a new API instead of reusing the old AppSync?
3. Pros and cons for storing a whole nested JSON structure for DDB as a locker representation vs what was presented?

Appendices

Tags: