SWATS Software Design Document

Team 5: Aaron Hong, Alex Gjurich, Paul O'Meara, Jonathan Helms, Collin Rausch

1. Project Overview

Warehouses are busy places and have a large logistical overhead that needs to be accounted for. Most warehouse management systems (WMS) are able to keep an order moving properly, but when orders get stored for immediate pickup, issues can arise. Most systems do not include functions for storing orders (dispensing orders) to hand to customers.

Completed orders stored within the warehouse can be tracked on paper, but the possibility for an order to leave the premises without proper sign out procedures increases as the scope of the business increases. The dealers from Caterpillar, a company that specializes in construction machinery and equipment machinery, are a prime target of interest for our project as they manage orders throughout the dispensing process through paper invoices.

With proper tracking using our proposed system, Separate Warehouse Archival Tracking System (SWATS) warehouses are now able to figure out if a customer has received their product along with confirmation. Included with that the customer can track where the order has been.

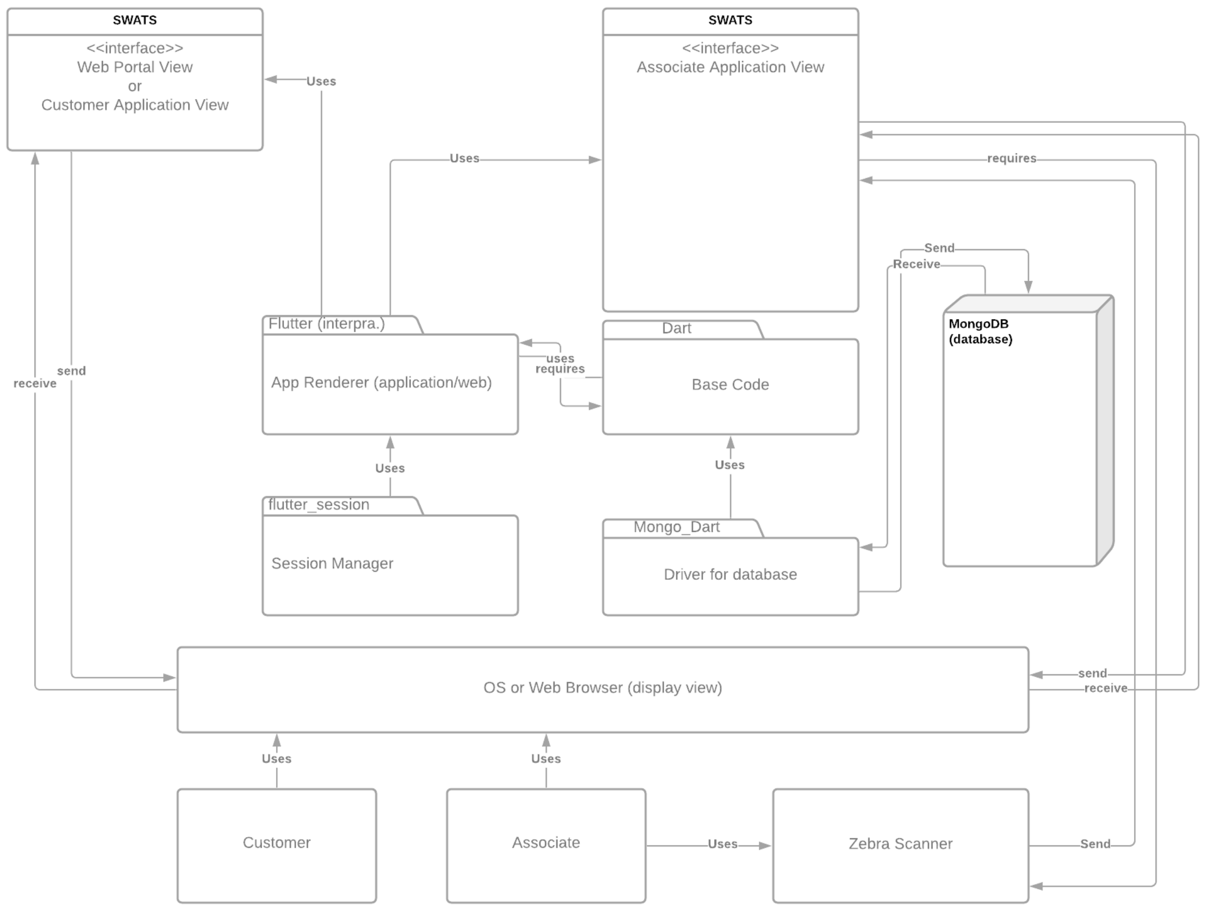
SWATS also solves the problem of not having to manually write down product information. This leads to more efficient shipments and will help eliminate human error or products being lost. Overall this saves the company more time and money. Through the use of barcode scanners and a webportal, warehouse employees can scan in and bin orders all while having an electronic log of where that order has been and where it currently is.

2. Architectural Overview

For our application we have decided on using Flutter which is written in Dart and MongoDB as our backend database. We wanted to have an approach that would function on many platforms, mobile and web. This led us to choosing the Flutter platform which natively cross complies onto Android and iOS platforms, including functioning on the web as well. We discussed writing the application in many different platforms including native android code, and even pure java using JavaFX. These platforms didn't offer the flexibility we desired within the application allowing many different forms of hardware to run our application.

For our backend we are using a MongoDB database running on Mongo Atlas, a hosted database provider. Mongo provides an easy document store with just as easy retrieval; this allows up to make quick and consistent updates to the database without worrying about consistency issues and allows for extra data to be entered on a per transaction basis as needed. Other database platforms were discussed but we decided on MongoDB as it allows for freeform data to be entered into a collection without the need to follow a specific prescribed schema.

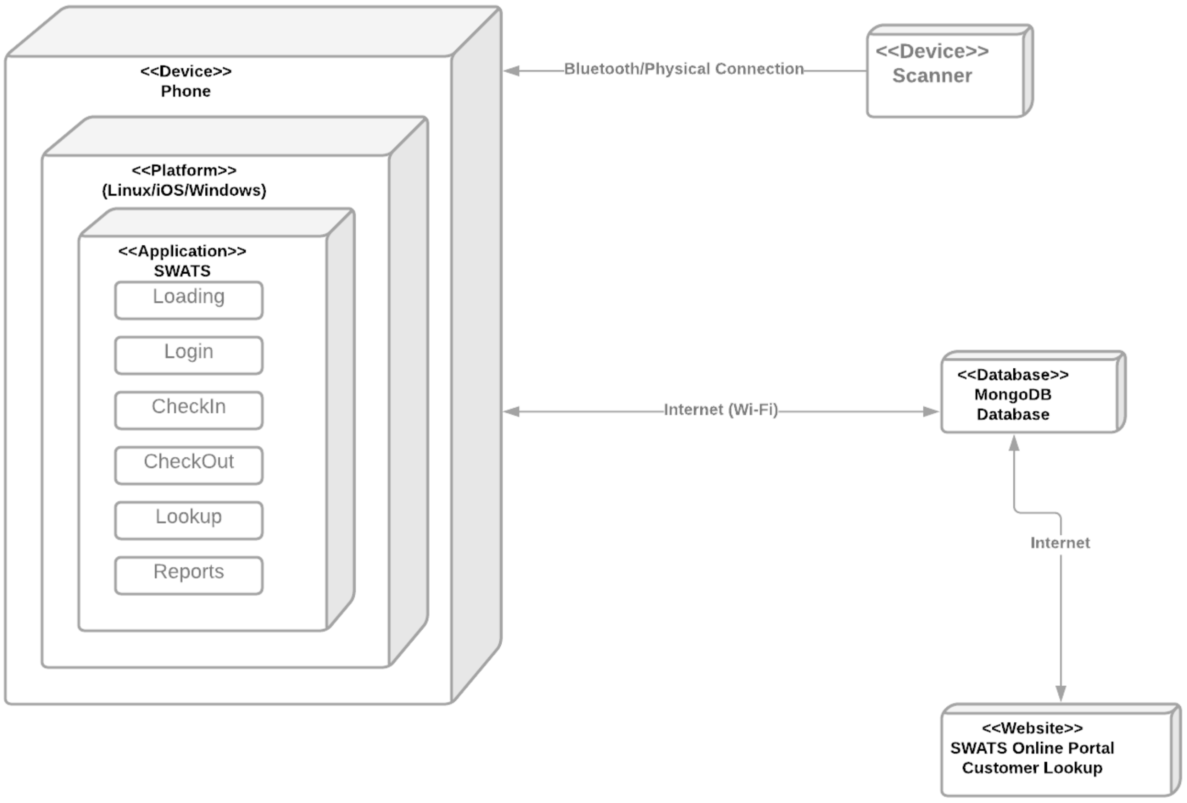
2.1 Subsystem Architecture



For our subsystem dependency diagram above, we use multiple components for both rendering, coding, and user interaction. All interaction comes from the user Operating system (OS) or Web Browser depending on which view needs to be used. Web portal view is used by the customers of the company which interacts with the Application View, while the Application View is used by the employee of the company that is using the SWATS application. The SWATS application itself requires the usage of a barcode scanner for certain functions. These functions will wait for a response from a barcode scanner or action from the Associate. Flutter is our software project main renderer and holds multiple packages which require the Dart programming language. Flutter_session package will handle information about user sessions which can aid in connection with the database and views of the application. Dart itself is a coding language that SWATS uses to create the views. This language uses a package called mongo_dart and this will interact with the Mongo database. The database is used to store customer data, employee data, product data, and order data that the company needs.

The overall architectural design our team chose for SWATS is a facade style described above. This is because the Customers and Associates don't directly modify each component of the system, but instead the application does. Customers and Associates do have an interaction with their respected view from their specific device. This makes our product easy to view and navigate so that Customers and Associates don't get confused. Of course other parts of the subsystem are designed in a different architectural style; for instance the MongoDB database, but SWATS is mostly a facade-styled application. The next section will describe how SWATS is deployed and how it interacts with the associated hardware and database services.

2.2 Deployment Architecture



Our application, SWATS, has several functions to help interact both inside and outside the application. All of the functions will interface with our database server (MongoDB) via an internet connection. Since SWATS has integrated cross platform functionality with Flutter, there are a multitude of operating systems that will be able to run SWATS, (including browsers within those operating systems). The OS itself will most likely run on a phone for easy use and access during day-to-day warehouse operations. The scanner device connected to the phone via a Bluetooth or a physical connection will be able to read an external barcode and print a unique string of numbers and letters to the application for use in interacting with the database.

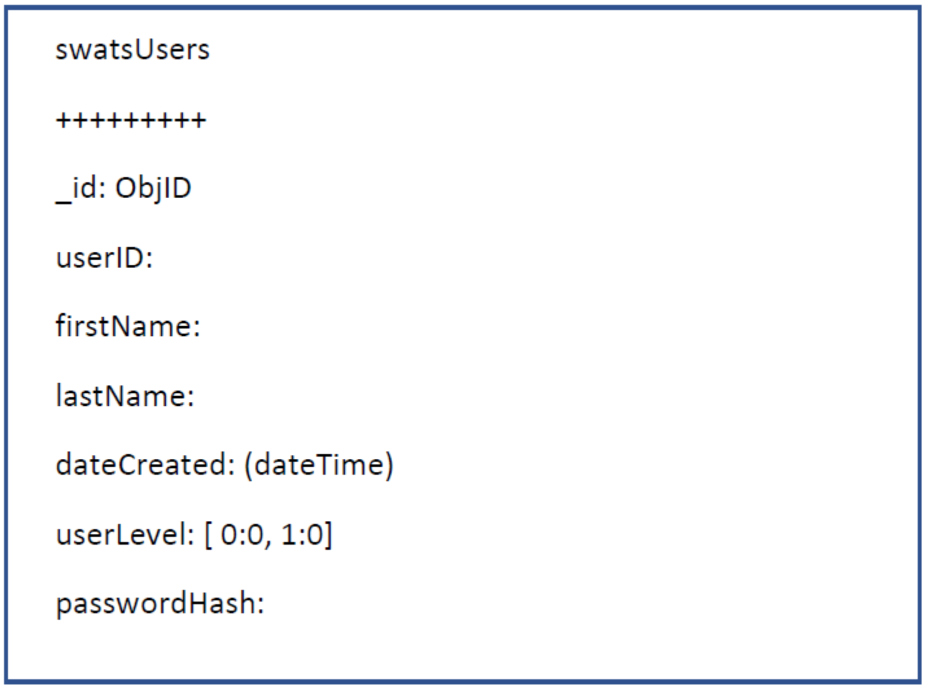
The protocols we will use will be for interacting and manipulating data within MongoDB. These include MongoDB’s database drivers, SecureShell (SSH), and HTTP for our online portal for customer lookup. For the following section, we will explain how our database will store and process information.

2.3 Persistent Data Storage

For our data storage solution, we are employing the use of MongoDB as a document store. All our login information along with order and customer data are stored within separate collections within the SWATS database. The format of the collections is as follows:

```
Orders

+++++++++++

_id: ObjID

documentNumber:

customerAccountNumber:

dateEntered: (Date)

binLocation:

orderStatus: 0/1 - true/false - in/out
```

```
swatsCustomers

+++++++++++++

_id: ObjID

customerName:

customerAccountNumber:
```



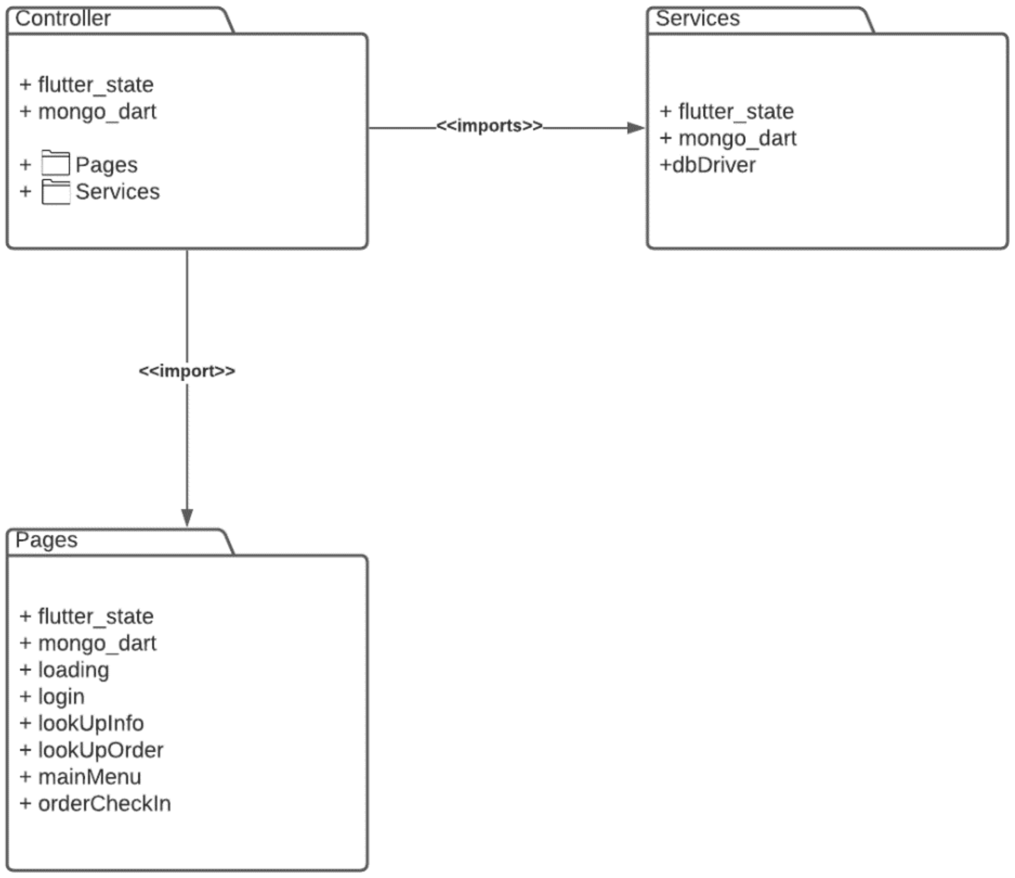
```
swatsUsers

+++++++++

_id: ObjID

userID:

firstName:

lastName:

dateCreated: (dateTime)

userLevel: [ 0:0, 1:0]

passwordHash:
```

2.4 Global Control Flow

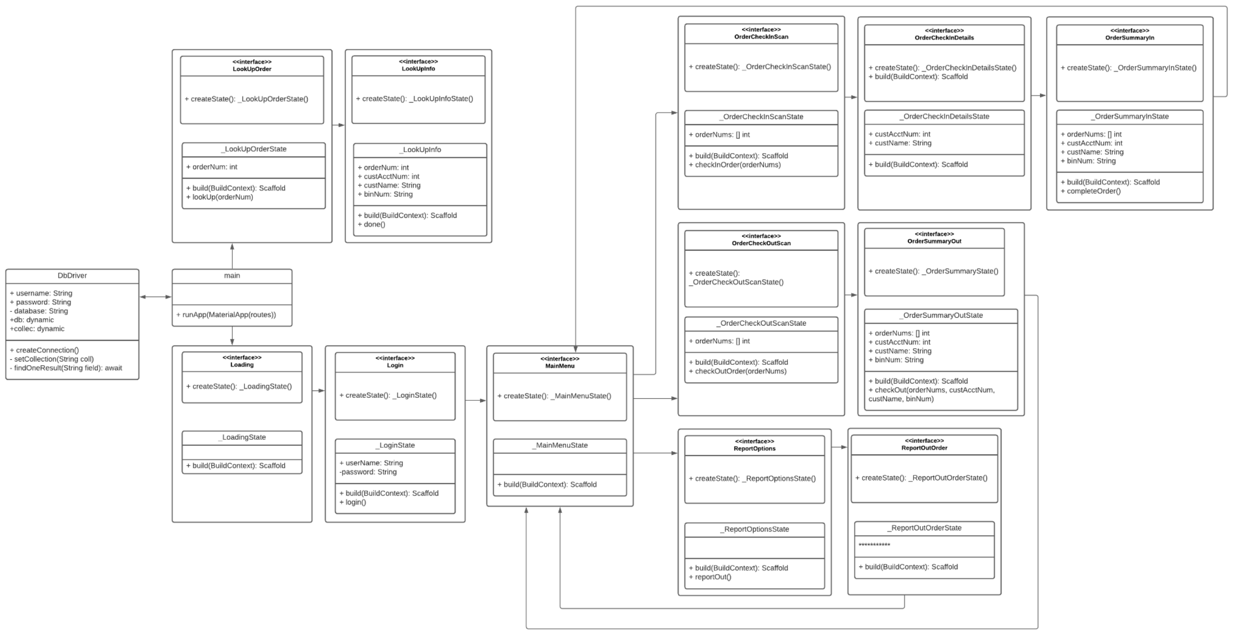
The generalized control flow of the system is an event driven system, because our program needs input from both its user and external device and services. Unlike procedural programming which doesn't need an event and is just a sequence of method calls without input. Our software product used time dependency for certain functions. For example, SWATS will not use timers to test connection to a barcode scanner since the barcode scanner acts as a keyboard. The MongoDB database, and for security a auto logout based on inactivity. SWATS will not need concurrency due to the database handling concurrency on the server side. None of our features do not need concurrency due to this app running on separate devices which use separate scanners. In the next section we will discuss all of the sections above in further detail, showing how each module, class, and methods interact.

3. Detailed System Design

The main controller package consists of imports for needed libraries along with the run method of the application which sets the base Material Design application, this package then imports the two other needed packages, the pages and services packages. The pages package houses all the layouts and state logic needed to display each page of the application. The services package houses all the services needed to support logic within the controller, such as providing a database object to make outside calls against.



3.1 Static View

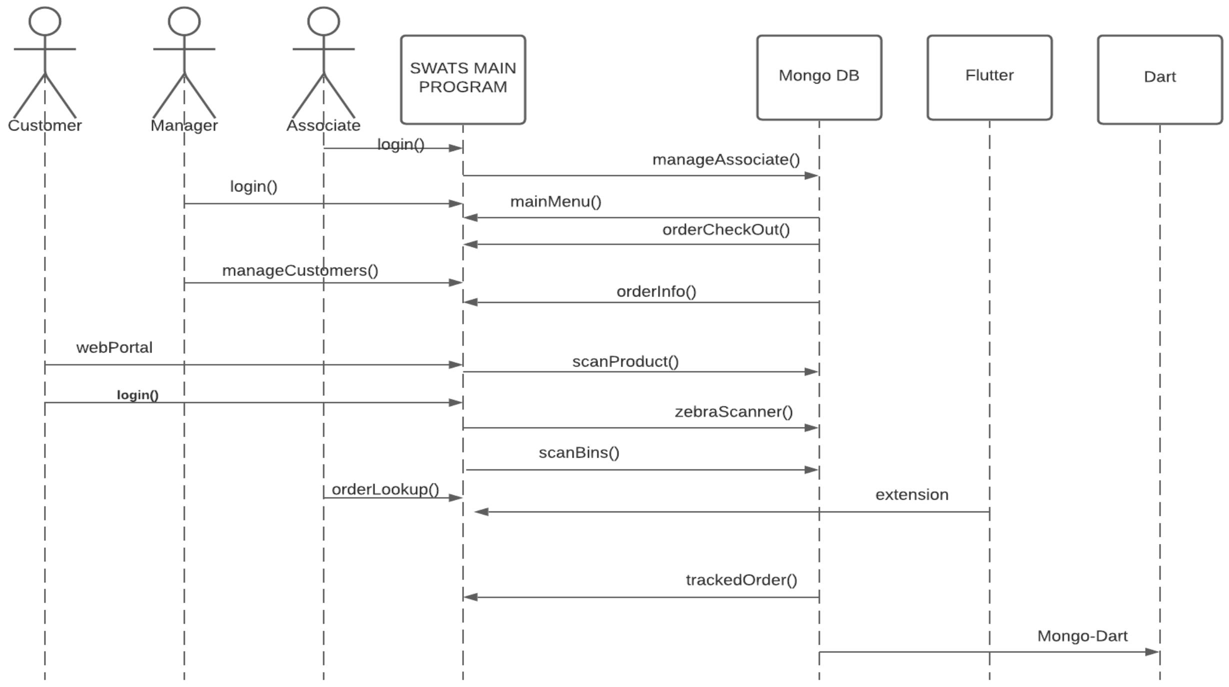


<<interface>> LookUpOrder
+ createState(): _LookUpOrderState()
_LookUpOrderState
+ orderNum: int
+ build(BuildContext): Scaffold
+ lookUp(orderNum)
<<interface>> LookUpInfo
+ createState(): _LookUpInfoState()
_LookUpInfo
+ orderNum: int
+ custAcctNum: int
+ custName: String
+ binNum: String
+ build(BuildContext): Scaffold
+ done()
DbDriver
+ username: String
+ password: String
- database: String
+db: dynamic
+collec: dynamic
+ createConnection()
- setCollection(String coll)
- findOneResult(String field): await
main
+ runApp(MaterialApp(routes))
<<interface>> Loading
+ createState(): _LoadingState()
_LoadingState
+ build(BuildContext): Scaffold
<<interface>> Login
+ createState(): _LoginState()
_LoginState
+ userName: String
-password: String
+ build(BuildContext): Scaffold
+ login()
<<interface>> MainMenu
+ createState(): _MainMenuState()
_MainMenuState
+ build(BuildContext): Scaffold
<<interface>> OrderCheckInScan
+ createState(): _OrderCheckInScanState()
_OrderCheckInScanState
+ orderNums: [] int
+ build(BuildContext): Scaffold
+ checkInOrder(orderNums)
<<interface>> OrderCheckInDetails
+ createState(): _OrderCheckInDetailsState()
+ build(BuildContext): Scaffold
_OrderCheckInDetailsState
+ custAcctNum: int
+ custName: String
+ build(BuildContext): Scaffold
<<interface>> OrderSummaryIn
+ createState(): _OrderSummaryInState()
_OrderSummaryInState
+ orderNums: [] int
+ custAcctNum: int
+ custName: String
+ binNum: String
+ build(BuildContext): Scaffold
+ completeOrder()
<<interface>> OrderCheckOutScan
+ createState(): _OrderCheckOutScanState()
_OrderCheckOutScanState
+ orderNums: [] int
+ build(BuildContext): Scaffold
+ checkOutOrder(orderNums)
<<interface>> OrderSummaryOut
+ createState(): _OrderSummaryState()
_OrderSummaryOutState
+ orderNums: [] int
+ custAcctNum: int
+ custName: String
+ binNum: String
+ build(BuildContext): Scaffold
+ checkOut(orderNums, custAcctNum, custName, binNum)
<<interface>> ReportOptions
+ createState(): _ReportOptionsState()
_ReportOptionsState
+ build(BuildContext): Scaffold
+ reportOut()
<<interface>> ReportOutOrder
+ createState(): _ReportOutOrderState()
_ReportOutOrderState
****
+ build(BuildContext): Scaffold

Expanded View of Diagram: [Link]

The decomposition of the classes is described in the UML diagram above. The basic breakdown of the classes and the connection between them follows the general format of a Flutter and Dart application, where each page has a class to create the interface and it connects to a state class which holds the methods and parameters for what the page does. There is also a class which is used to connect to the MongoDB collections that each of the other classes will have available to them. There is also a separation of classes for the web portal and the mobile application that will connect to the scanner, and this is because the web portal will only be accessed by the customers to check orders, so there is no need for them to login and see information only warehouse associates will need. A dynamic view of the diagram is displayed below in section 3.2.

3.2 Dynamic View


Customer
Manager
Associate
SWATS MAIN PROGRAM
Mongo DB
Flutter
Dart
login()
manageAssociate()
login()
mainMenu()
orderCheckOut()
manageCustomers()
orderInfo()
webPortal
scanProduct()
login()
zebraScanner()
scanBins()
orderLookup()
extension
trackedOrder()
Mongo-Dart