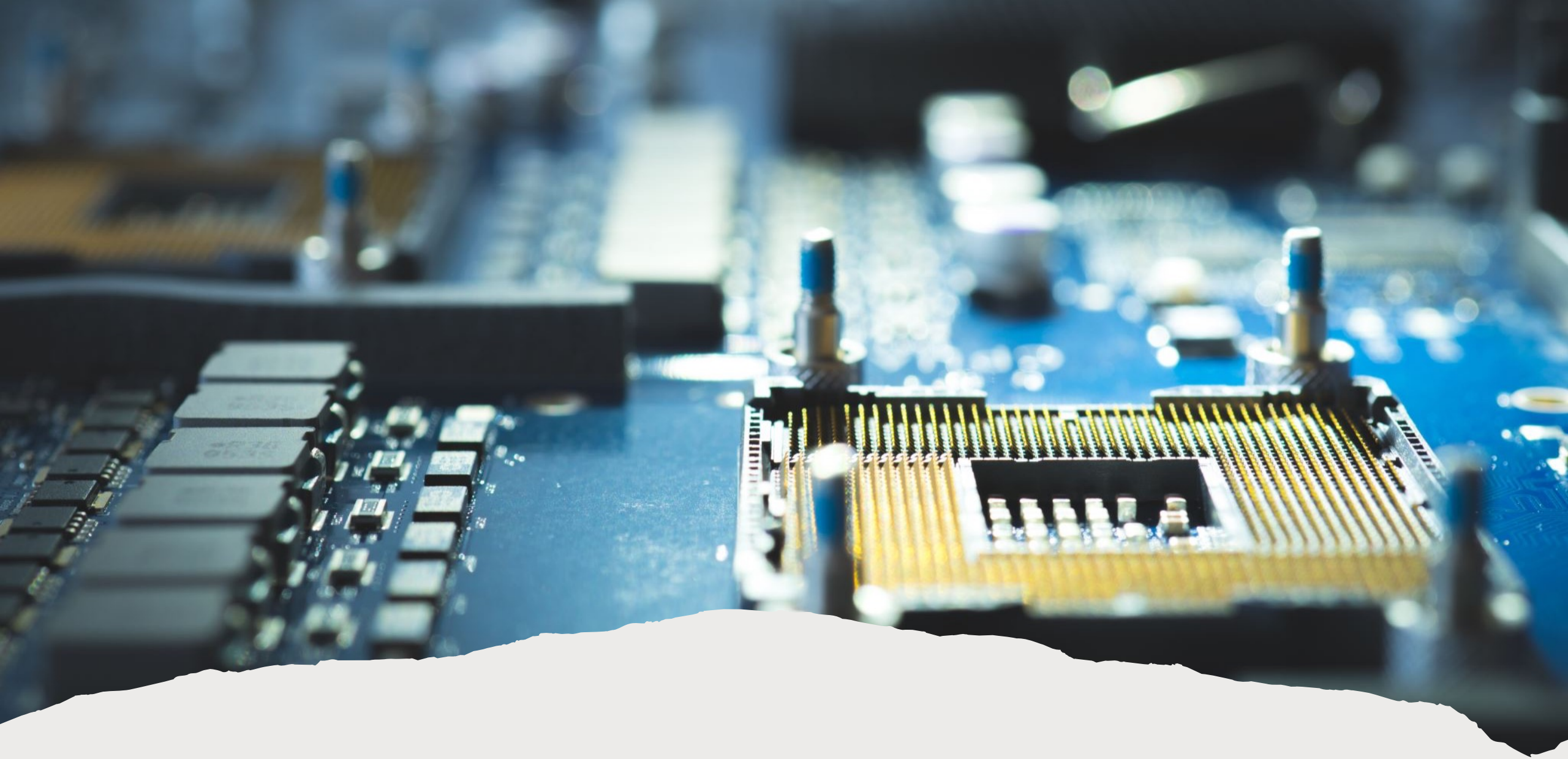
PRODUCT ENGINEERING MINDSET - WORKSHOP

Balaji Thiruvengadam

DAY 4 & 5 - AGENDA

- Recap of day 3
- Swiggy problem - Develop MVP
- Functional vs Non-functional
- How product demand evolves?
- Design Principles
- Architecture
- Translating customer problem into design



PRODUCT ENGINEERING – RECAP DAY3

HOW PRODUCT AND FEATURES TO BE DESIGNED FOR CUSTOMERS HAVING DIFFERENT VERSIONS?

- Key challenges
  - Customer can move between plans
  - Product upgrades to customers having different versions

- Configurability
  - Individual feature level enablement
  - Subscription based
  - Packaged model

	Free	Standard	Premium	Enterprise
	$0 Always free for 10 users Get started	**$7.50** per user (average) $75 a month Start trial	**$14.50** per user (average) $145 a month Start trial	Billed annually. Switch the Billing cycle to Annual to view Enterprise pricing. Contact us
	For small teams to plan and track work more efficiently	For growing teams focused on building more together	For organizations that need to scale how they collaborate and track work	For enterprises with global scale, security, and governance needs
Features				
User limit (per site)	10 users	20,000 users	20,000 users	20,000 users
Site limit	One	One	One	Unlimited
Scrum and Kanban boards	✓	✓	✓	✓
Backlog	✓	✓	✓	✓
Agile reporting	✓	✓	✓	✓
Customizable workflows	✓	✓	✓	✓
Apps and integrations ↗	✓	✓	✓	✓
Automation	Single project	Single project	Global and multi-project	Global and multi-project
Roadmaps	Basic	Basic	Advanced	Advanced
Dependency management	Basic	Basic	Advanced	Advanced
Capacity planning	-	-	✓	✓
Project archiving	-	-	✓	✓
Admin Controls				
Domain verification & account capture	✓	✓	✓	✓
Session duration management (desktop)	✓	✓	✓	✓
Project roles	-	✓	✓	✓
Advanced permissions	-	✓	✓	✓
Admin insights	-	-	✓	✓
Sandbox	-	-	✓	✓
Release tracks	-	-	✓	✓

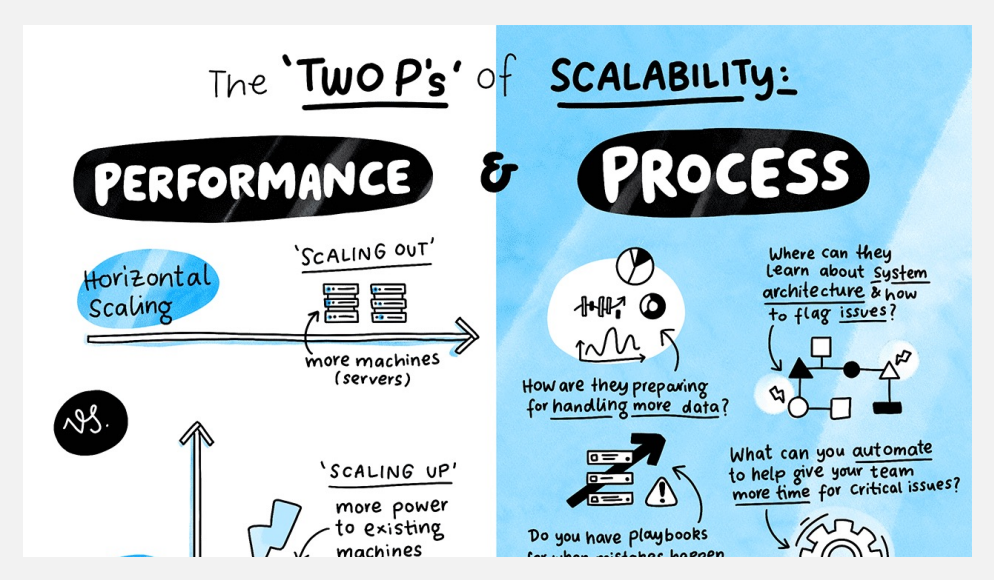
SWIGGY – FUNCTIONAL VS NFR

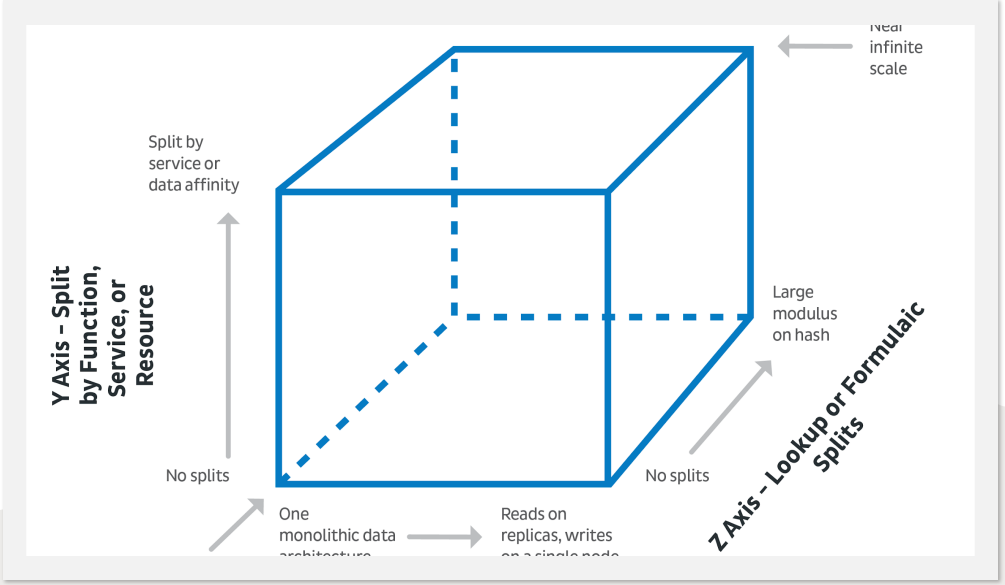
Let’s do together

HOW TO BUILD PRODUCT FOR SCALE

What is scalability?

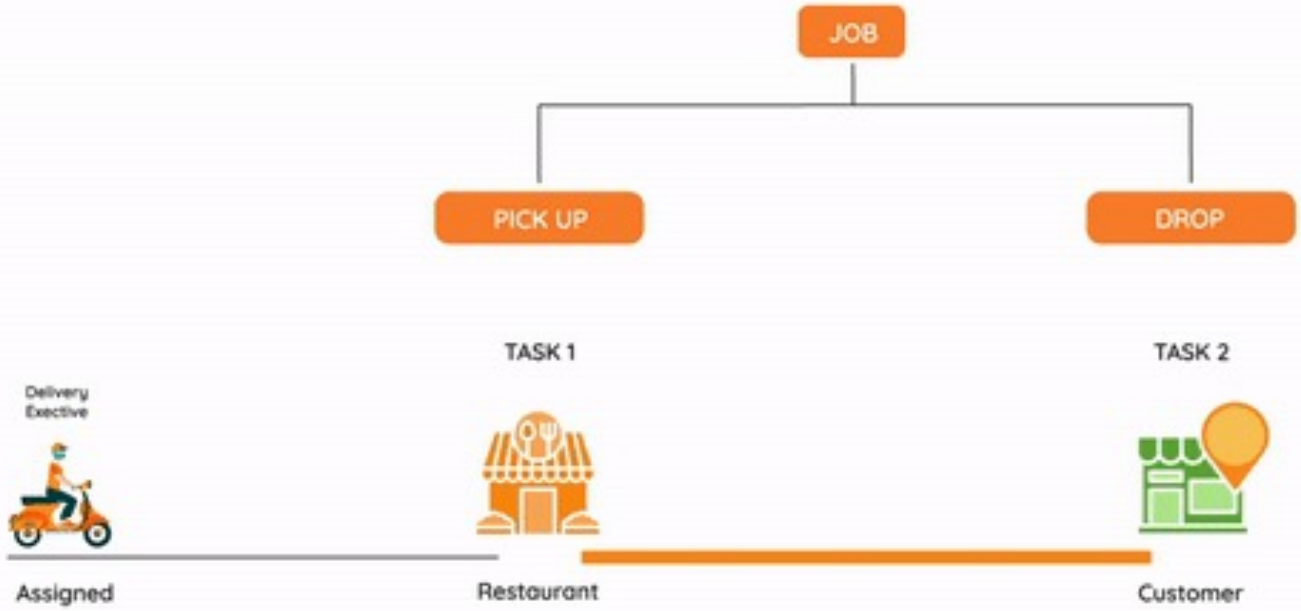
- Scalability is your ability to efficiently scale resources up or down to meet demand.
- Examples would include how well a hardware system performs when the number of users is increased, how well a database withstands growing numbers of queries, or how well an operating system performs on different classes of hardware.





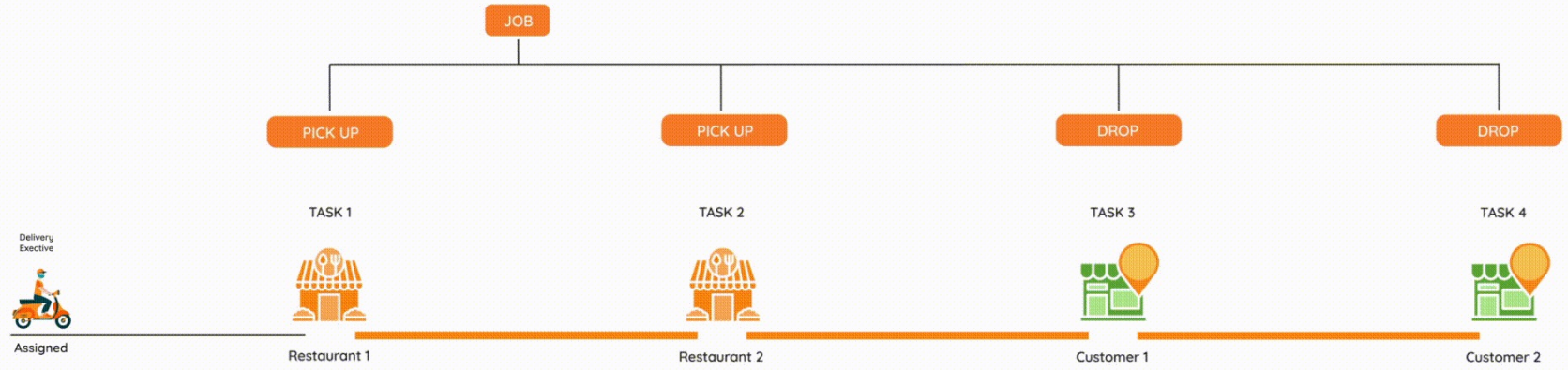
HOW REQUIREMENT EVOLVES - SINGLE ORDER

Delivery Executive is assigned to the trip. DE marks the arrived status update on pick_up task post arrival at the task location and updates the status completed post picking up the order. DE proceeds to drop location and updates the arrived and the completed status for the drop task.

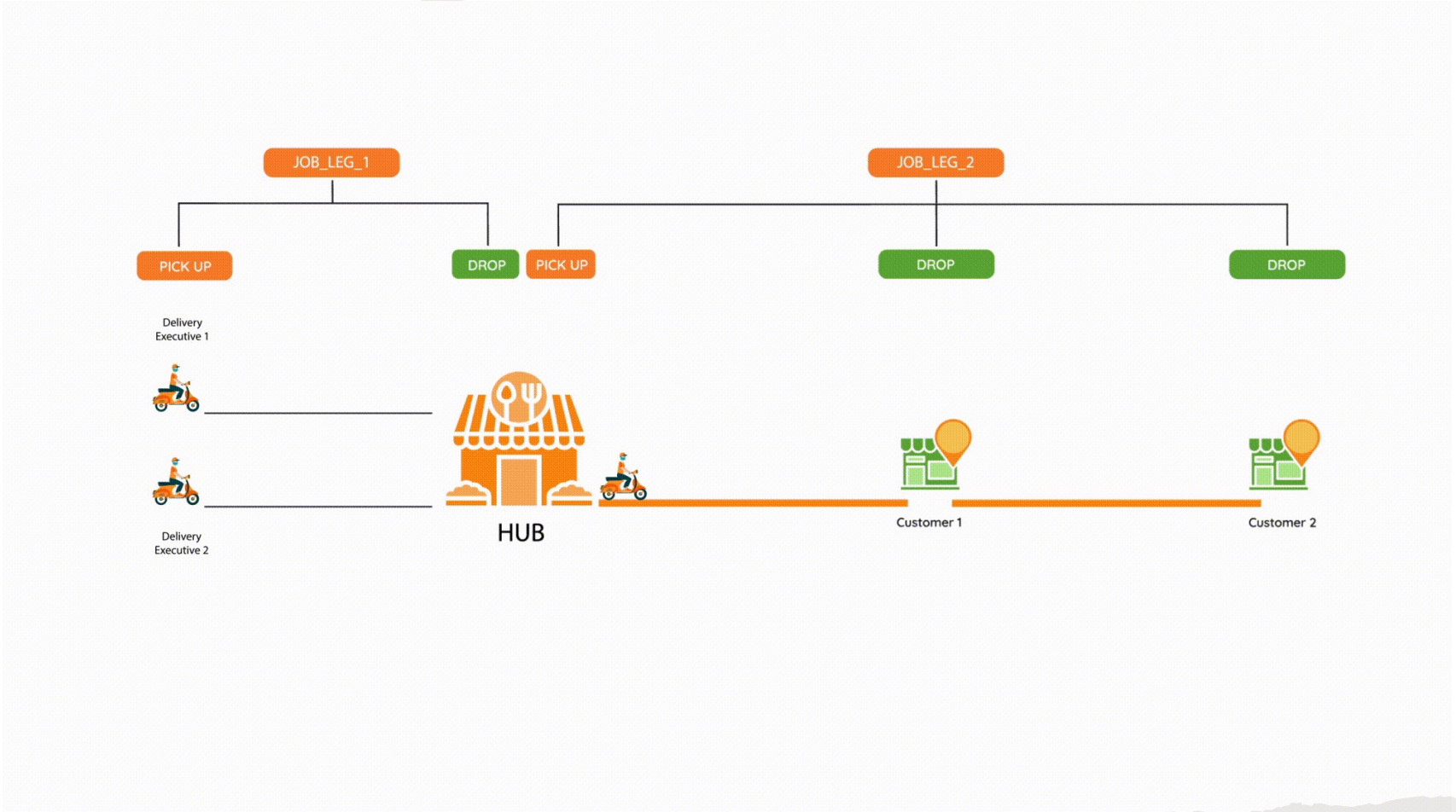


HOW REQUIREMENT EVOLVES - MERGED TRIP

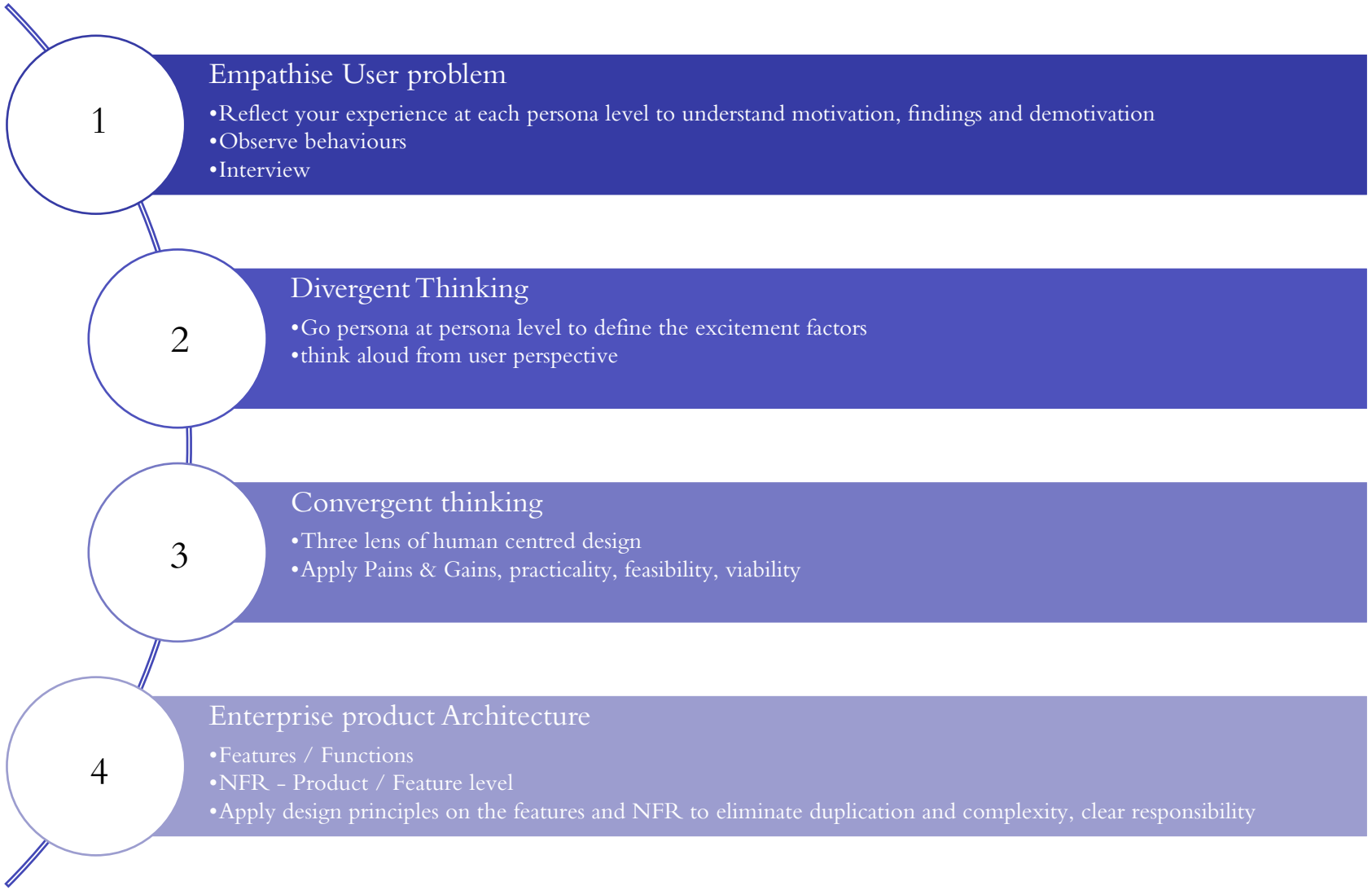
The delivery executive assigned to the trip will be executing the task in the sequence provided by the system.



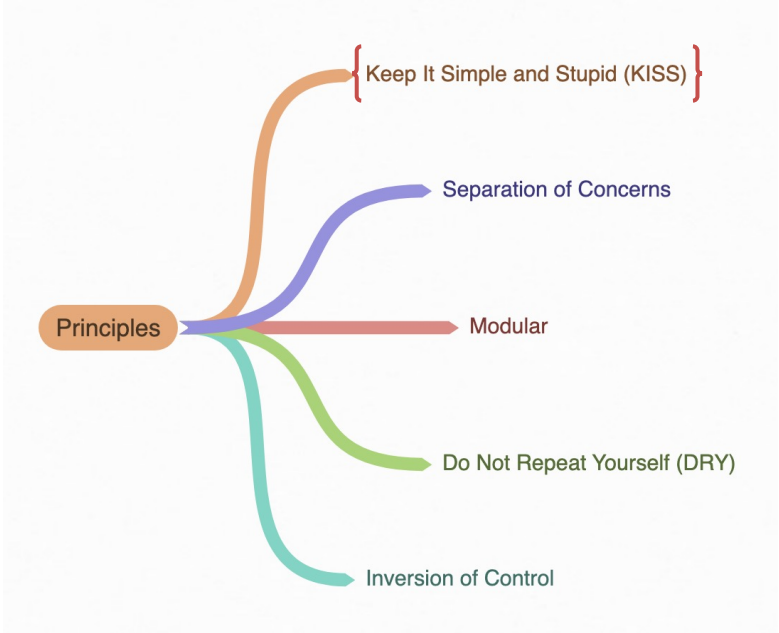
HOW REQUIREMENT EVOLVES - HUB & SPOKE

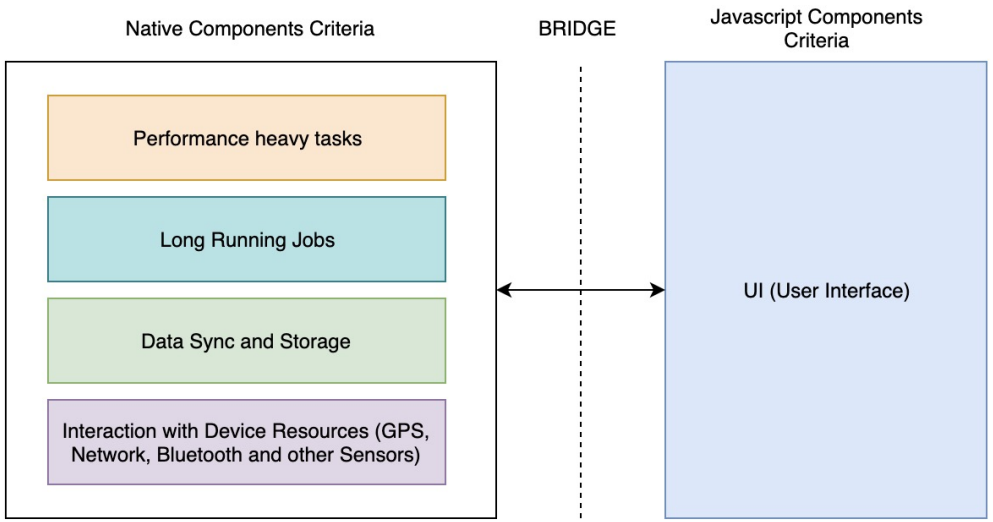


JOURNEY – DESIGN THINKING TO DESIGN PRINCIPLE



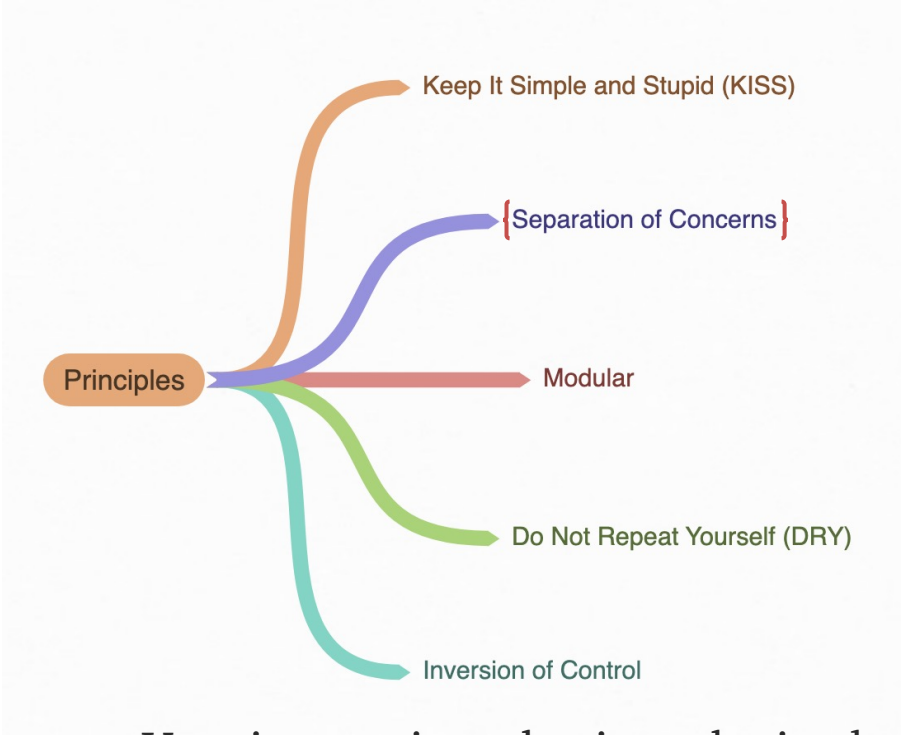
DESIGN PRINCIPLE - KISS

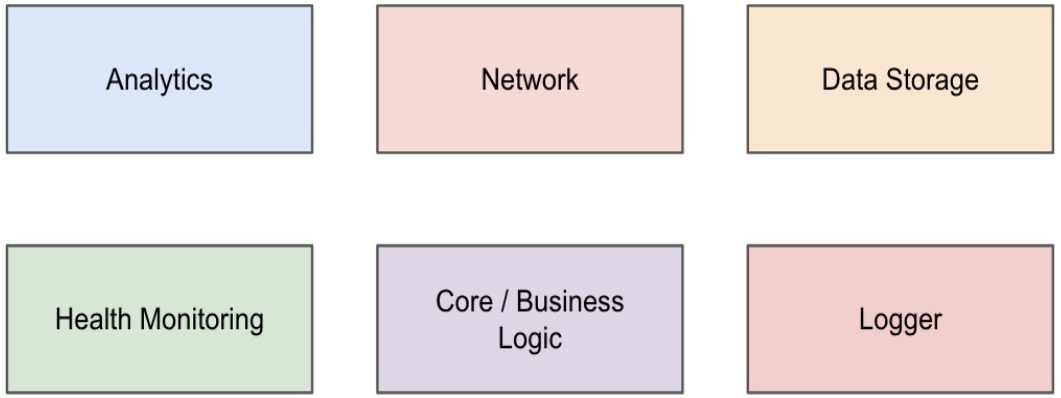




Tracking ***location info*** of our delivery partners (needed to assign trips, real-time tracking of an order, payment based on distance travelled etc.), ***order info*** (state of the order, item info, restaurant or store info, customer info etc.), ***earnings and incentives info*** etc. These components also interact with device resources such as GPS, network, bluetooth and other sensors for location tracking, proximity detection, distance travelled measurement, activity recognition. to keep things simple we decided to build only ***UI components in React native and long running, background intensive, data storage components in Native (Android/iOS)***

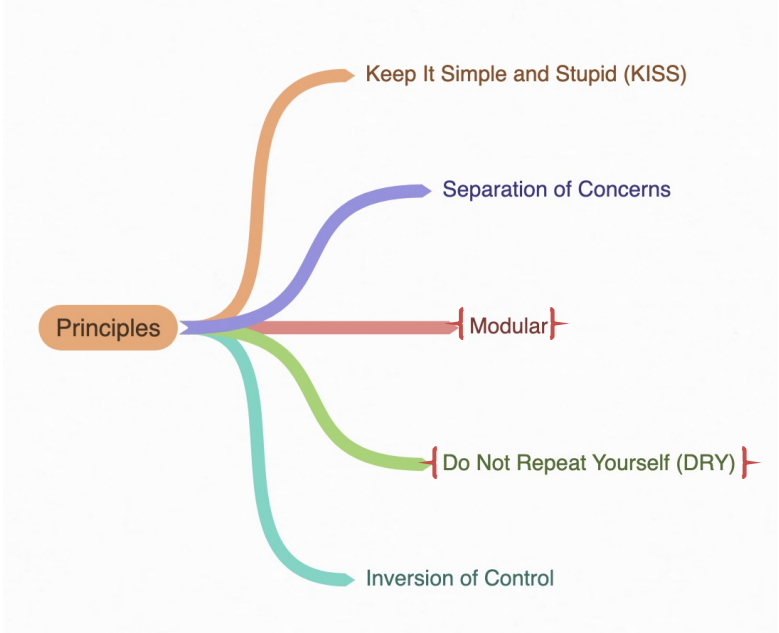
DESIGN PRINCIPLE- SEPARATION OF CONCERN

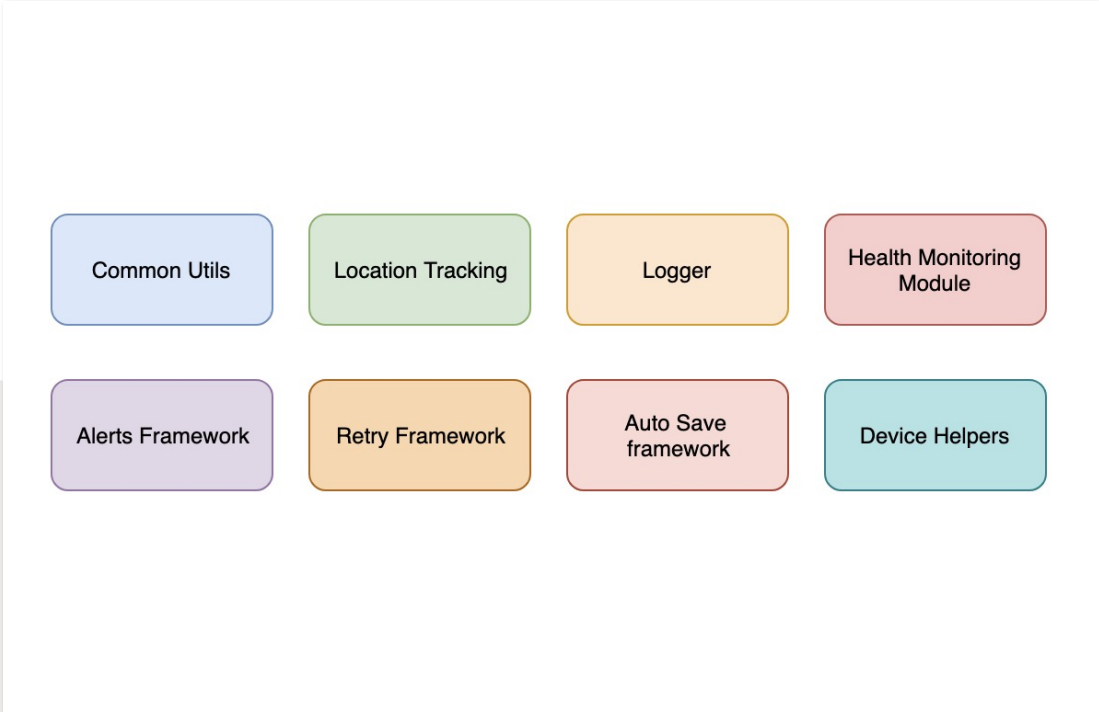




User interactions, business logic, data sync and storage, network transactions, instrumentation of data such as user journey, touchpoints, real-time health metrics, etc. to name a few concerns. Instead of sprinkling these concerns all over the code base which becomes a developer's nightmare to change/clean-up/revamp the code related to any concern, we separated these concerns either by creating a module or exposing a framework

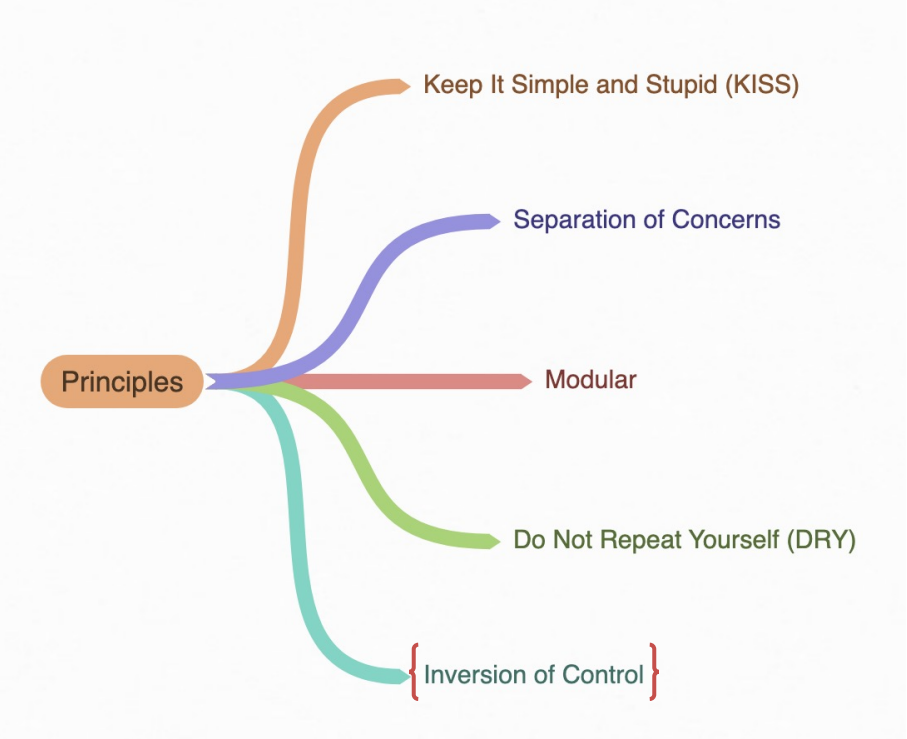
DESIGN PRINCIPLE – MODULAR (DRY)





components that are built can be shared across multiple apps (Consumer, Delivery, Vendor, Daily etc.). Repeating the same code often involves development cycle, QA cycle, maintenance and stability monitoring, etc. To avoid these cycles and fast track production releases, we create components (however small it is) as agnostic to any specific application.

DESIGN PRINCIPLE – INVERSION OF CONTROL



As we build more components in the future, the interaction between these components, dependency resolution and the flow of control will be cumbersome. So, we relied on this principle which is analogous to the Hollywood principle:

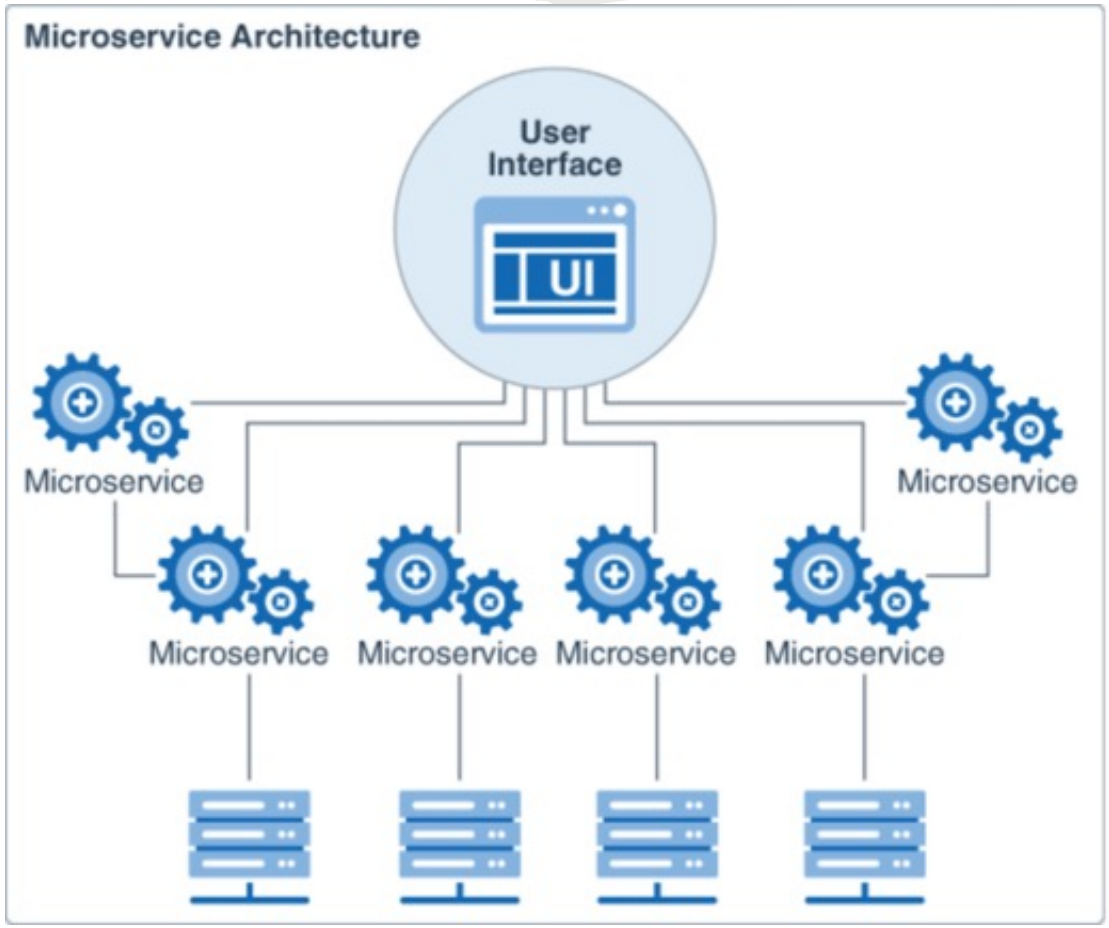
"Don't call us, we will call you".

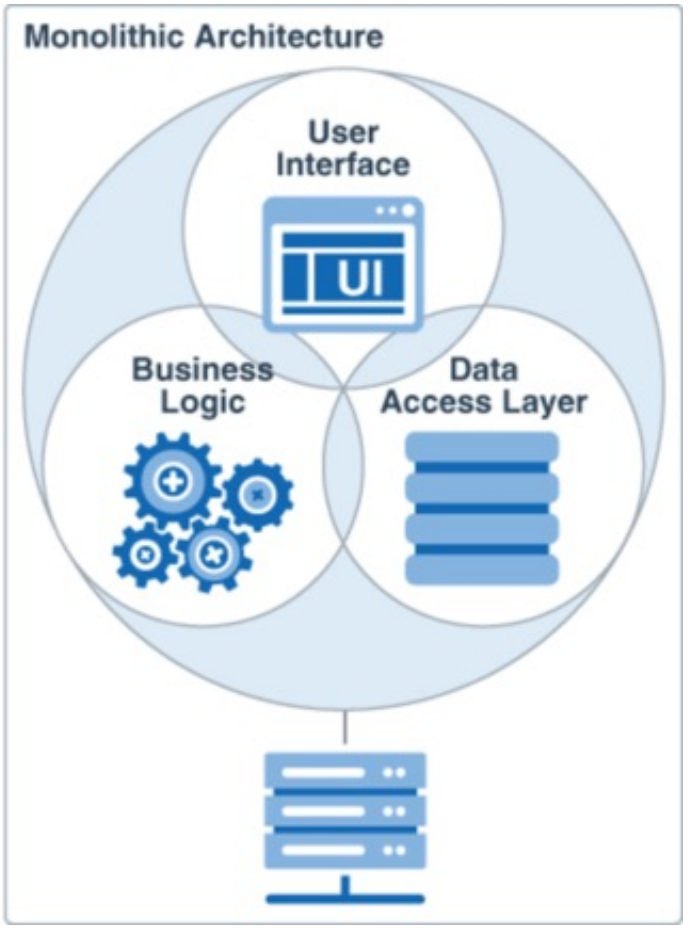
Imagine you are the CEO of a company and going to visit an important business partner. You can definitely drive a car yourself. In this case, you control the car. On the other hand, you can also hire a driver to drive the car for you instead. The control of the car is inverted from you to the driver now.

```kotlin
class Ceo {
  private val driver = Driver()

  fun visitBusinessPartner() {
    // 1. prepare presentation materials
    driver.goto("business partner's office")
    // 3. convince the partner to cooperate with our company
  }
}

class Driver {
  fun goto(destination: String) {
    // drive a car to destination
  }
}
```

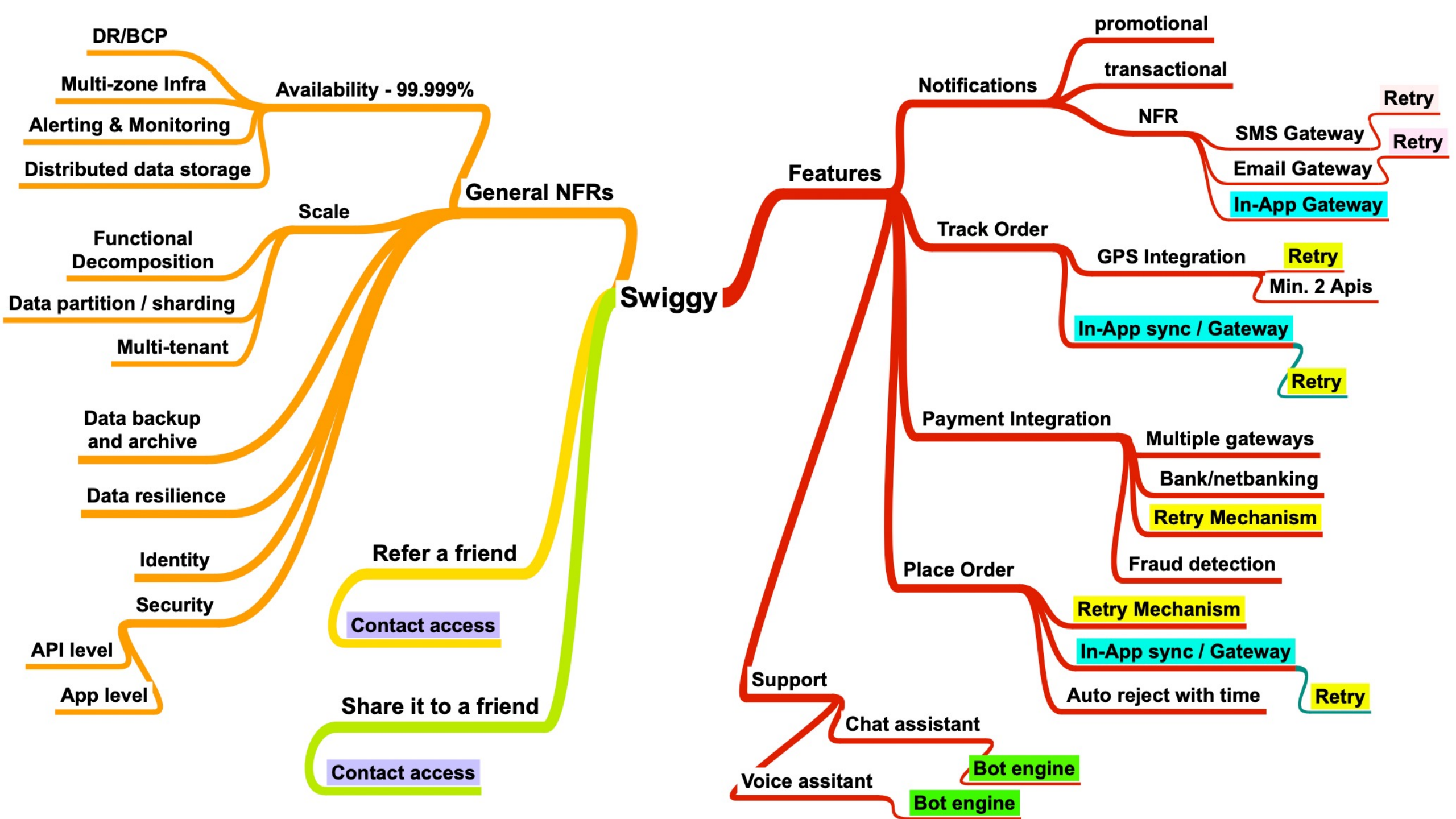
Ceo.kt hosted with ❤ by GitHub view raw

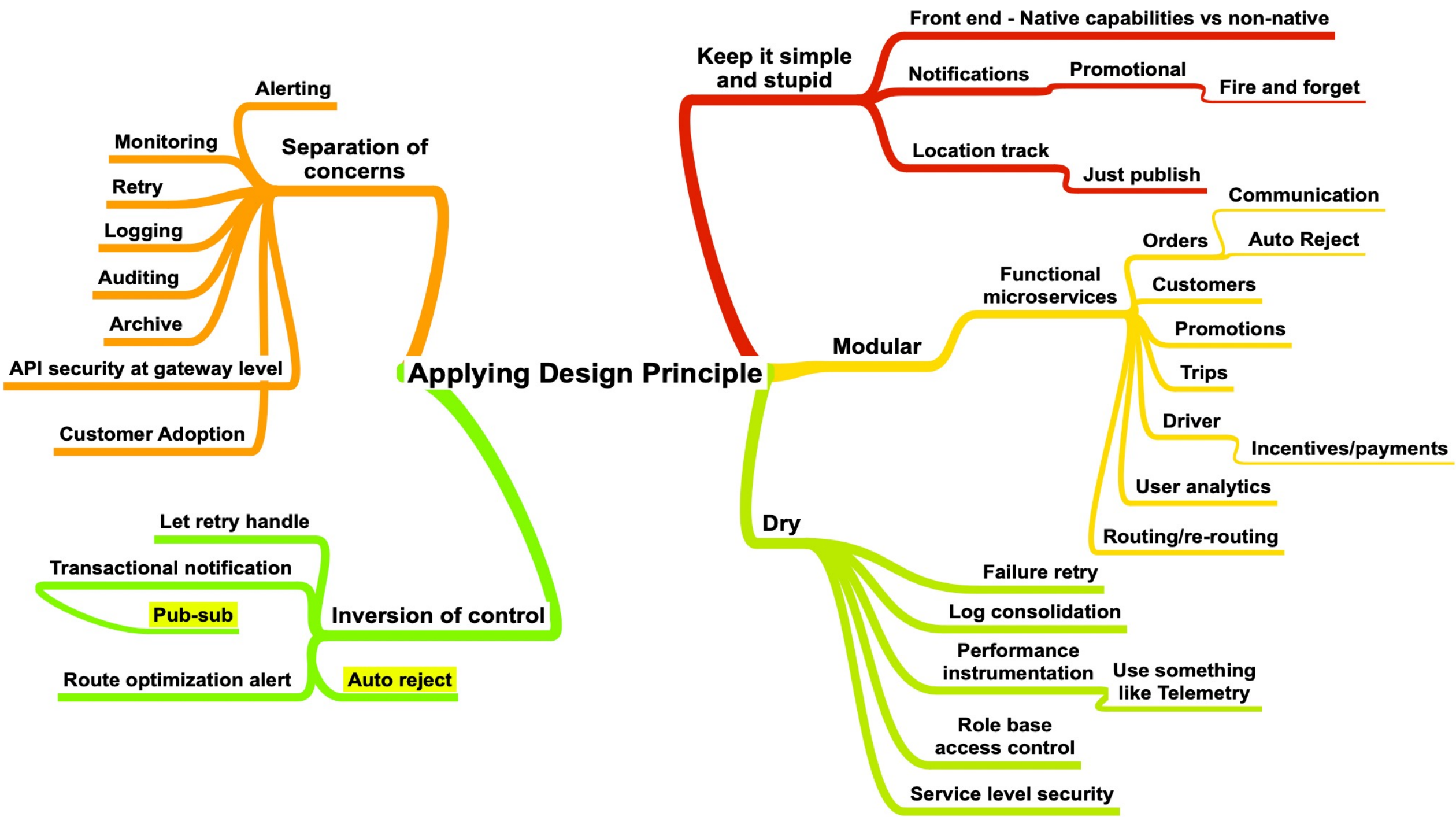
MONOLITHIC VS MICRO SERVICES

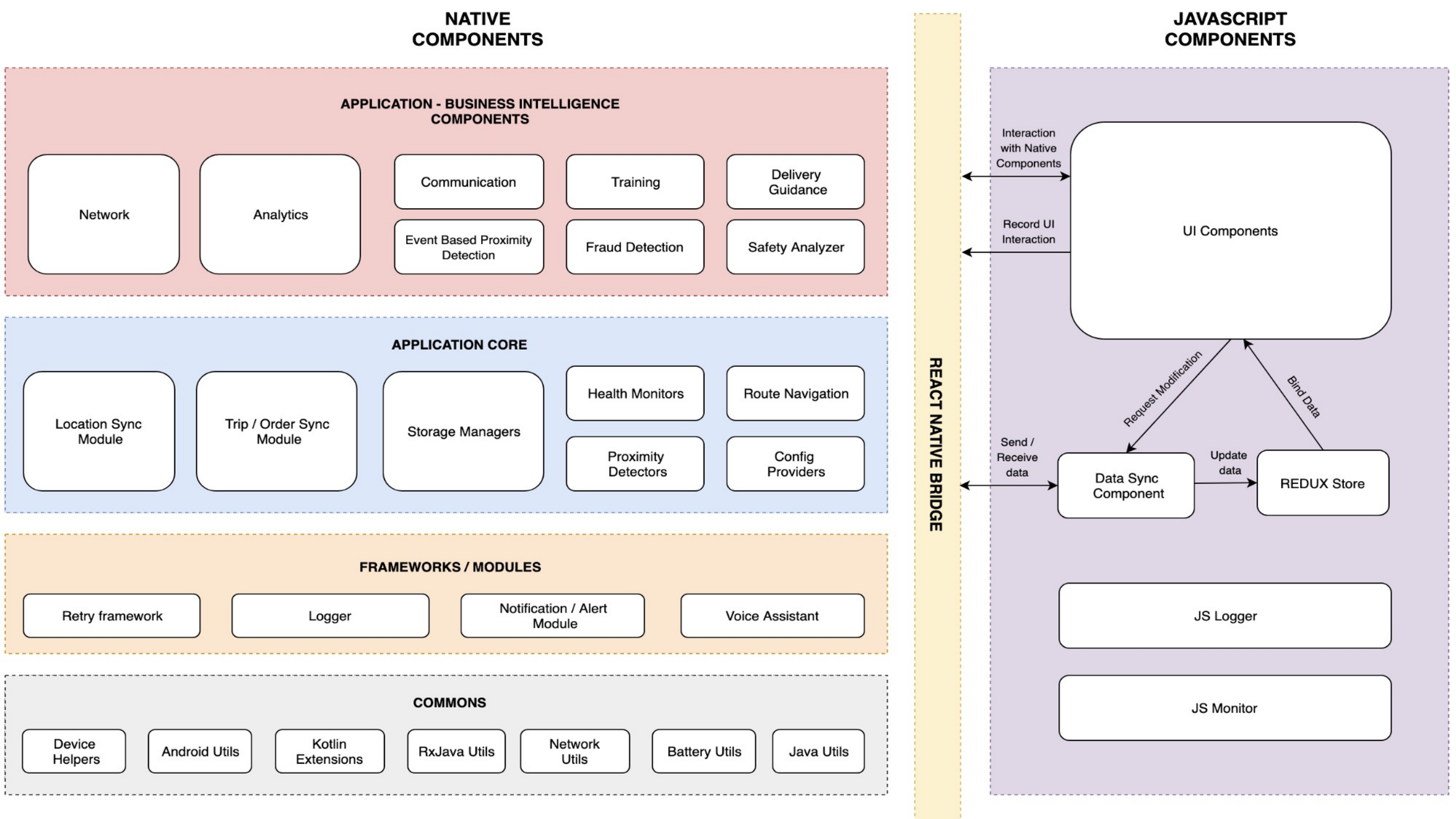




APPLYING DESIGN PRINCIPLE TO SWIGGY USE CASE


Swiggy
General NFRs
Availability - 99.999%
DR/BCP
Multi-zone Infra
Alerting & Monitoring
Distributed data storage
Scale
Functional Decomposition
Data partition / sharding
Multi-tenant
Data backup and archive
Data resilience
Identity
Security
API level
App level
Refer a friend
Contact access
Share it to a friend
Contact access
Features
Notifications
promotional
transactional
NFR
SMS Gateway
Retry
Email Gateway
Retry
In-App Gateway
Track Order
GPS Integration
Retry
Min. 2 Apis
In-App sync / Gateway
Retry
Payment Integration
Multiple gateways
Bank/netbanking
Retry Mechanism
Fraud detection
Place Order
Retry Mechanism
In-App sync / Gateway
Auto reject with time
Retry
Support
Chat assistant
Bot engine
Voice assitant
Bot engine


Applying Design Principle
Separation of concerns
Alerting
Monitoring
Retry
Logging
Auditing
Archive
API security at gateway level
Customer Adoption
Inversion of control
Let retry handle
Transactional notification
Pub-sub
Route optimization alert
Auto reject
Keep it simple and stupid
Front end - Native capabilities vs non-native
Notifications
Promotional
Fire and forget
Location track
Just publish
Modular
Functional microservices
Communication
Orders
Auto Reject
Customers
Promotions
Trips
Driver
Incentives/payments
User analytics
Routing/re-routing
Dry
Failure retry
Log consolidation
Performance instrumentation
Use something like Telemetry
Role base access control
Service level security


NATIVE COMPONENTS
JAVASCRIPT COMPONENTS
APPLICATION - BUSINESS INTELLIGENCE COMPONENTS
Network
Analytics
Communication
Training
Delivery Guidance
Event Based Proximity Detection
Fraud Detection
Safety Analyzer
APPLICATION CORE
Location Sync Module
Trip / Order Sync Module
Storage Managers
Health Monitors
Route Navigation
Proximity Detectors
Config Providers
FRAMEWORKS / MODULES
Retry framework
Logger
Notification / Alert Module
Voice Assistant
COMMONS
Device Helpers
Android Utils
Kotlin Extensions
RxJava Utils
Network Utils
Battery Utils
Java Utils
REACT NATIVE BRIDGE
Interaction with Native Components
Record UI Interaction
UI Components
Request Modification
Bind Data
Send / Receive data
Data Sync Component
Update data
REDUX Store
JS Logger
JS Monitor

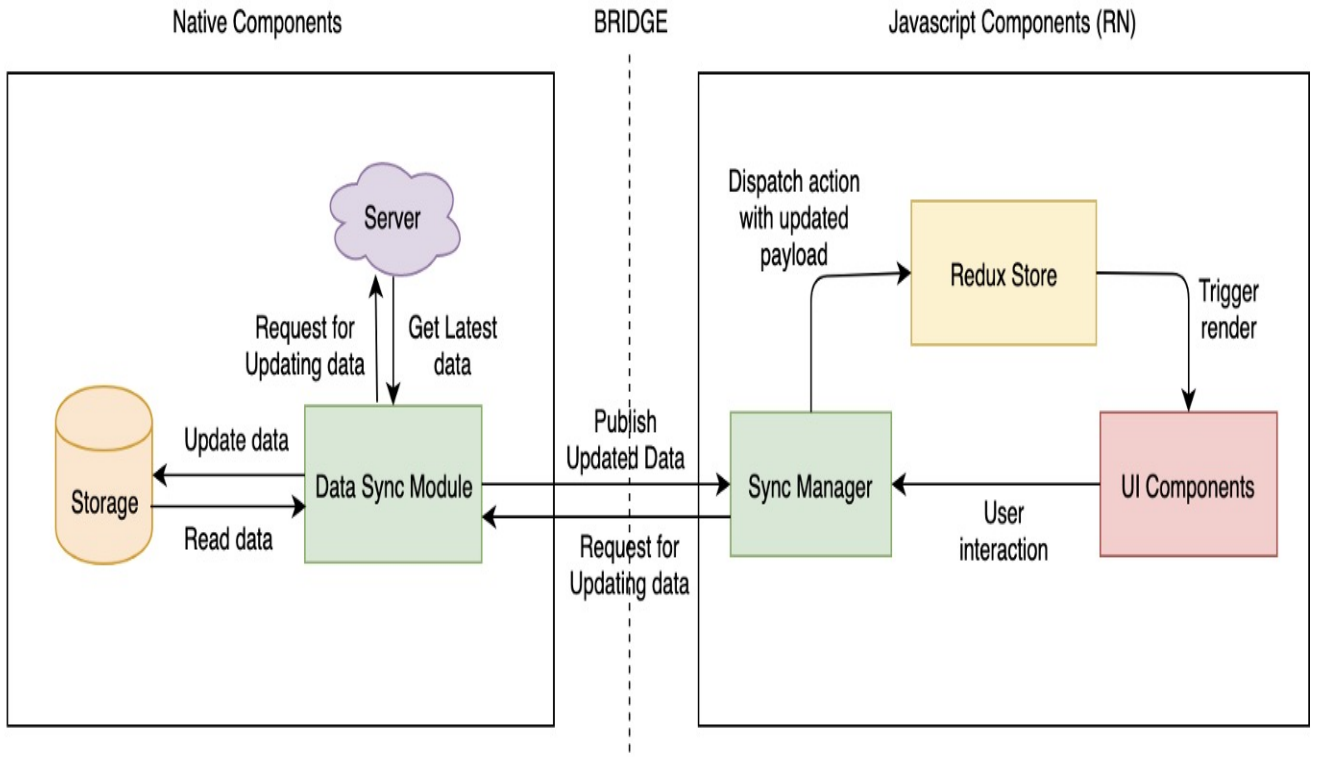
LETS DEEP DIVE

- Data Integrity or Sync Issue
- Distribution of responsibility - Inversion of control

DESIGN PATTERN – DATA INTEGRITY

Unidirectional Data Flow

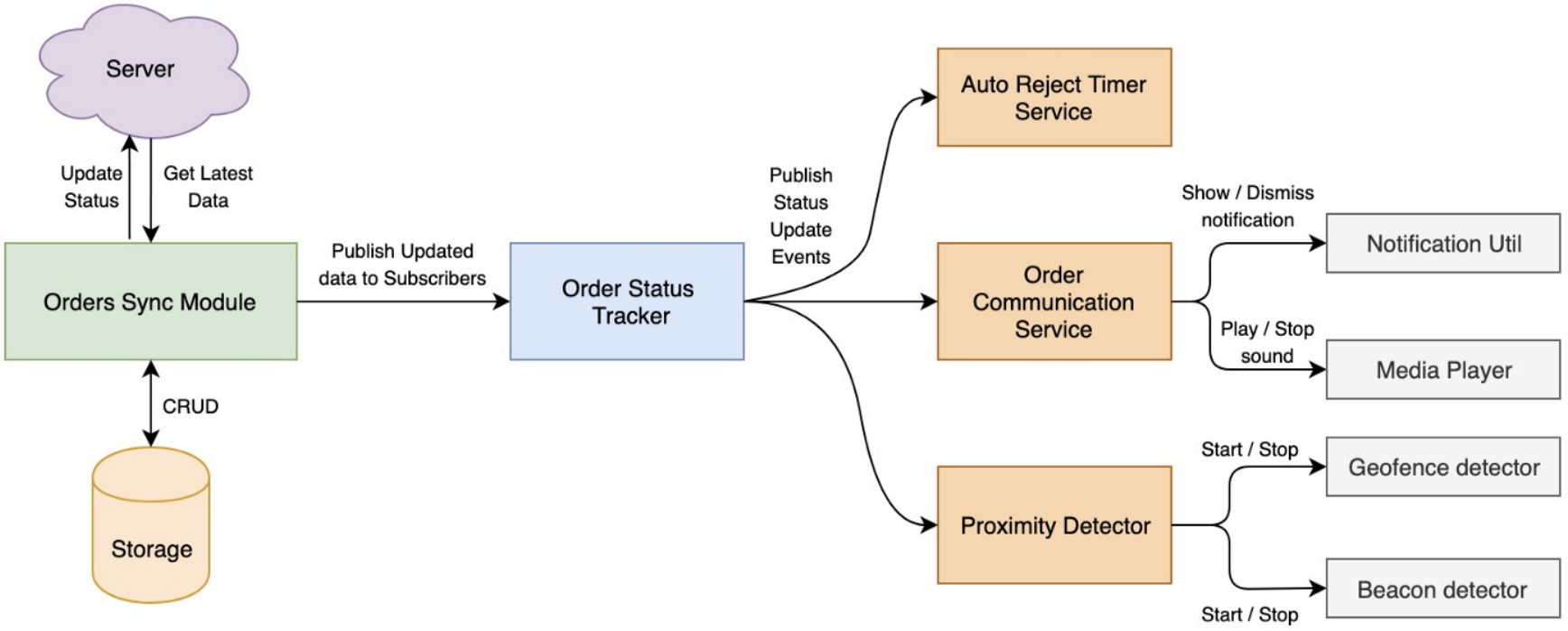
As we have decided to build our sync modules, storage components in Native and UI in React native (based on above principles) we also need to stick to a pattern in the way data should be flowing across these components. Taking inspiration from the Flux pattern and especially the way REDUX works, we designed below blueprint of the data flow(***server → storage → redux store → UI components***).



DESIGN PATTERN – INVERSION OF CONTROL

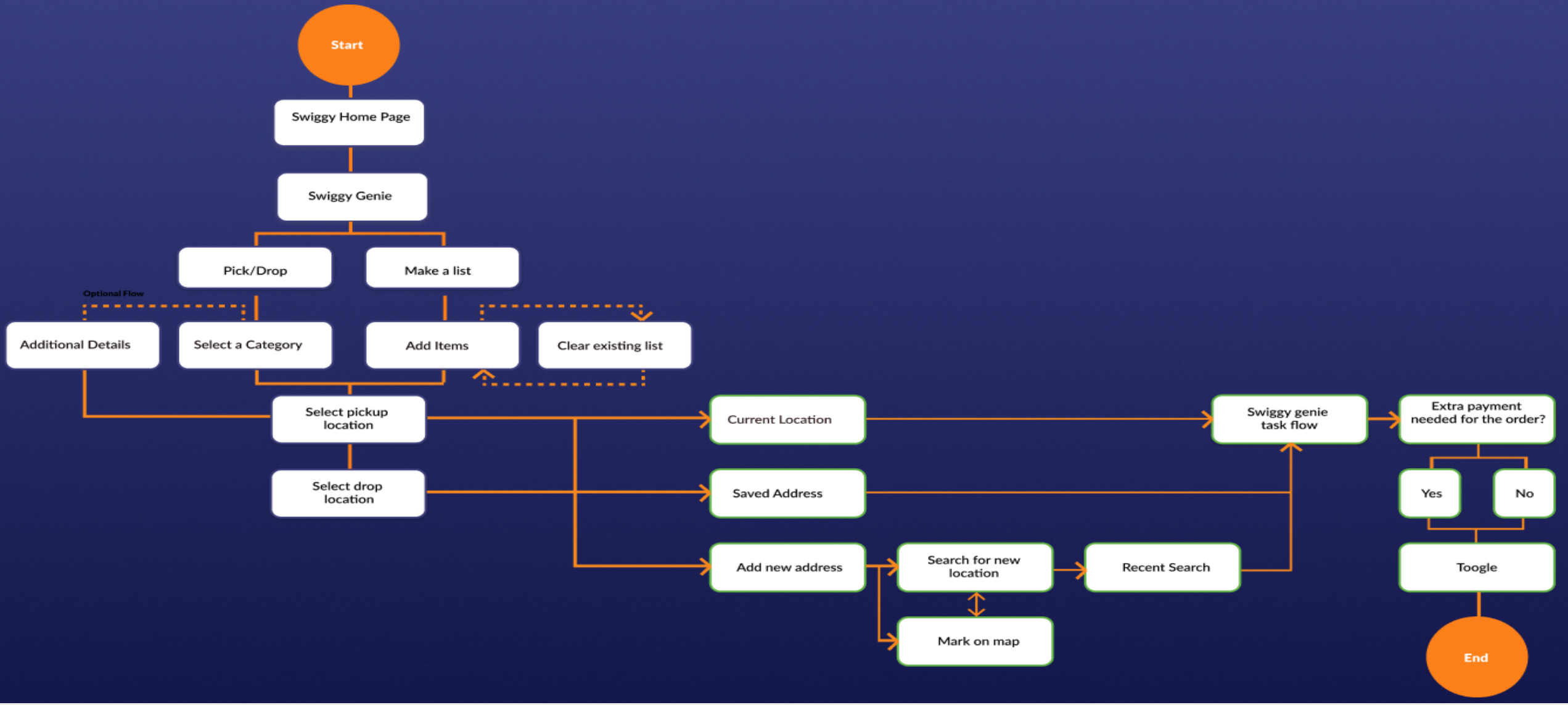
- ***Event driven and a finite state machine for each delivery flow***

Whenever any event or state transition occurs we might need to perform tasks like playing a sound and vibrating the phone as soon as an order is Assigned, showing a confirmation pop-up on the screen, uploading logs for real-time debugging, alerting partner when the battery level is critical or in poor network area, etc.



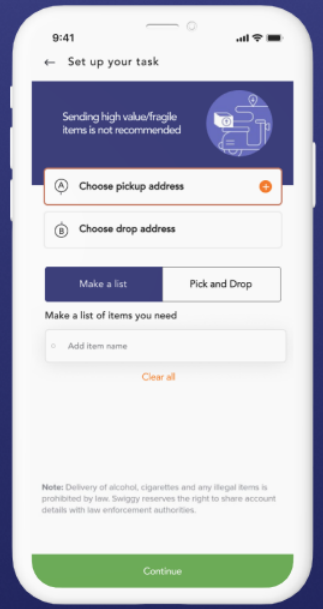
WHY USER FLOW IS IMPORTANT BEFORE WE BUILD THE UX

User Flow



SWIGGY GENIE

Make a list task flow

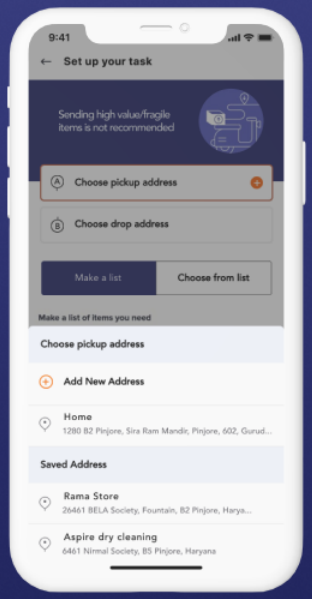


1

1. This is the home screen or the default screen of swiggy genie where the user wants to buy some groceries from the store.

2. The user will select the pick up address and the drop address from this home page.

3. The user will make a list of items to purchase from the store.

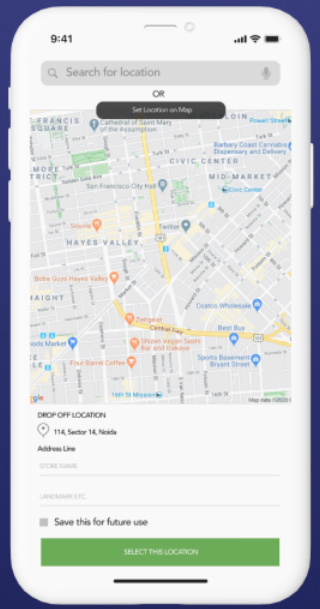


2

1. After clicking on the button of pickup/drop location this pop up appears.

2. The user can select the pickup/drop address from the saved address or from the home section.

3. When the user clicks on add new address this page is redirected to the map location.

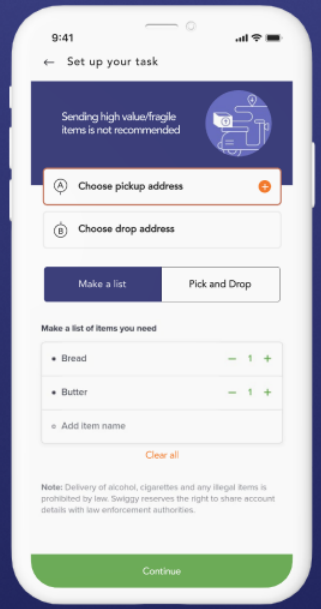


3

1. The user can select the address from the search bar.

2. The user can locate on the map directly by dragging it to the desired location.

3. The user can add the address manually by typing and can save it for further use.

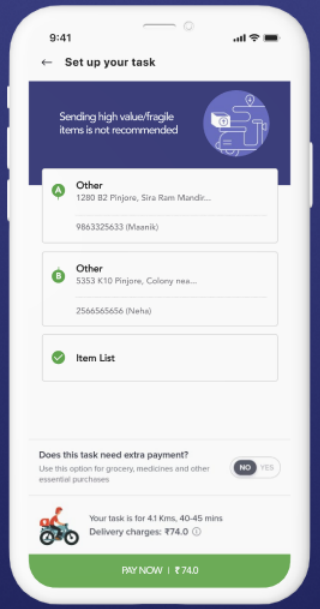


4

1. The user can add list items to be purchased and can aslo check their quantity.

2. The user can add maximum upto 10 items to purchase at once.

3. On clicking the clear all button the user can able to clear all the input items at once.



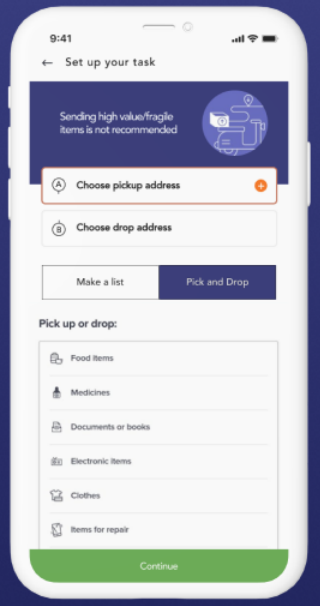
5

1. In this summary screen the user get the summary of pickup address and drop address.

2. Get the details of items which are added.

3. There is a toogle button where the user can add whether the task needs a extra payment.

Pick/drop task flow

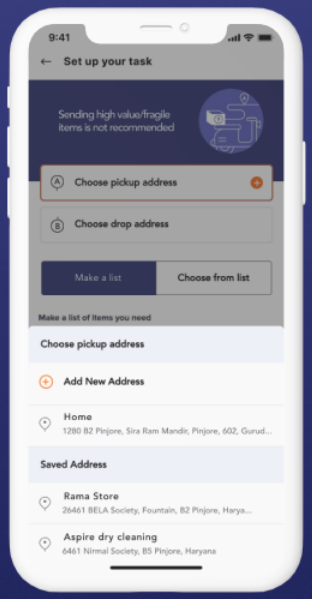


1

1. This is the home screen of pick and drop where the user can select from the list of categories mentioned below.

2. The user will select the pick up address and the drop address from this home page.

3. The user will make a list of items to purchase from the store.

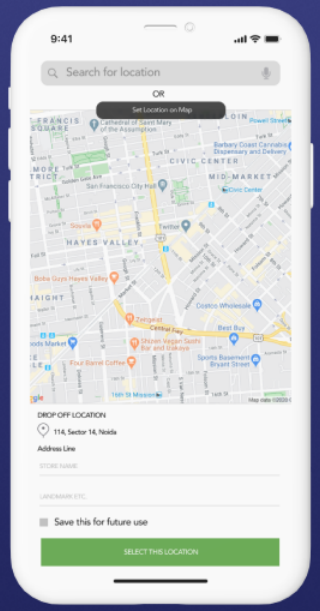


2

1. After clicking on the button of pickup/drop location this pop up appears.

2. The user can select the pickup/drop address from the saved address or from the home section.

3. When the user clicks on add new address this page is redirected to the map location.

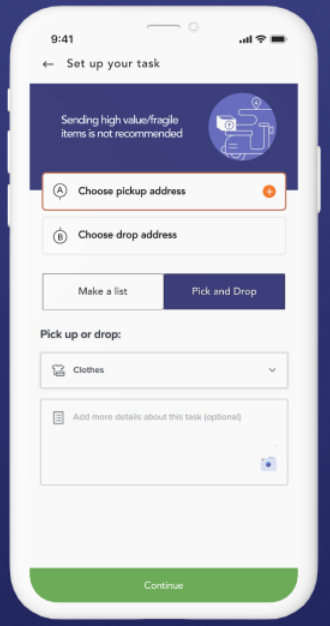


3

1. The user can select the address from the search bar.

2. The user can locate on the map directly by dragging it to the desired location.

3. The user can add the address manually by typing and can save it for further use.

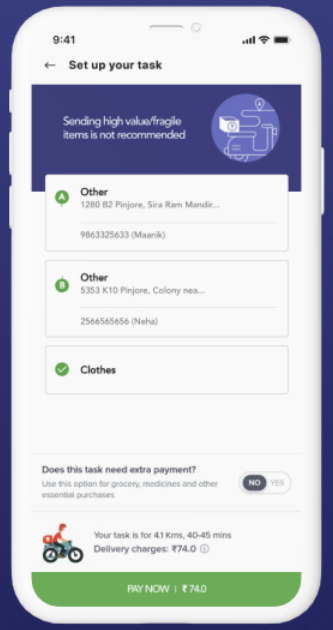


4

1. The user can add list items to be purchased and can aslo check their quantity.

2. The user can add image of the item which is to be picked up and also some description about it

3. The user can click on the dropdown button to select different options.



5

1. In this summary screen the user get the summary of pickup address and drop address.

2. Get the details of the task which is added.

3. There is a toogle button where the user can add whether the task needs a extra payment.