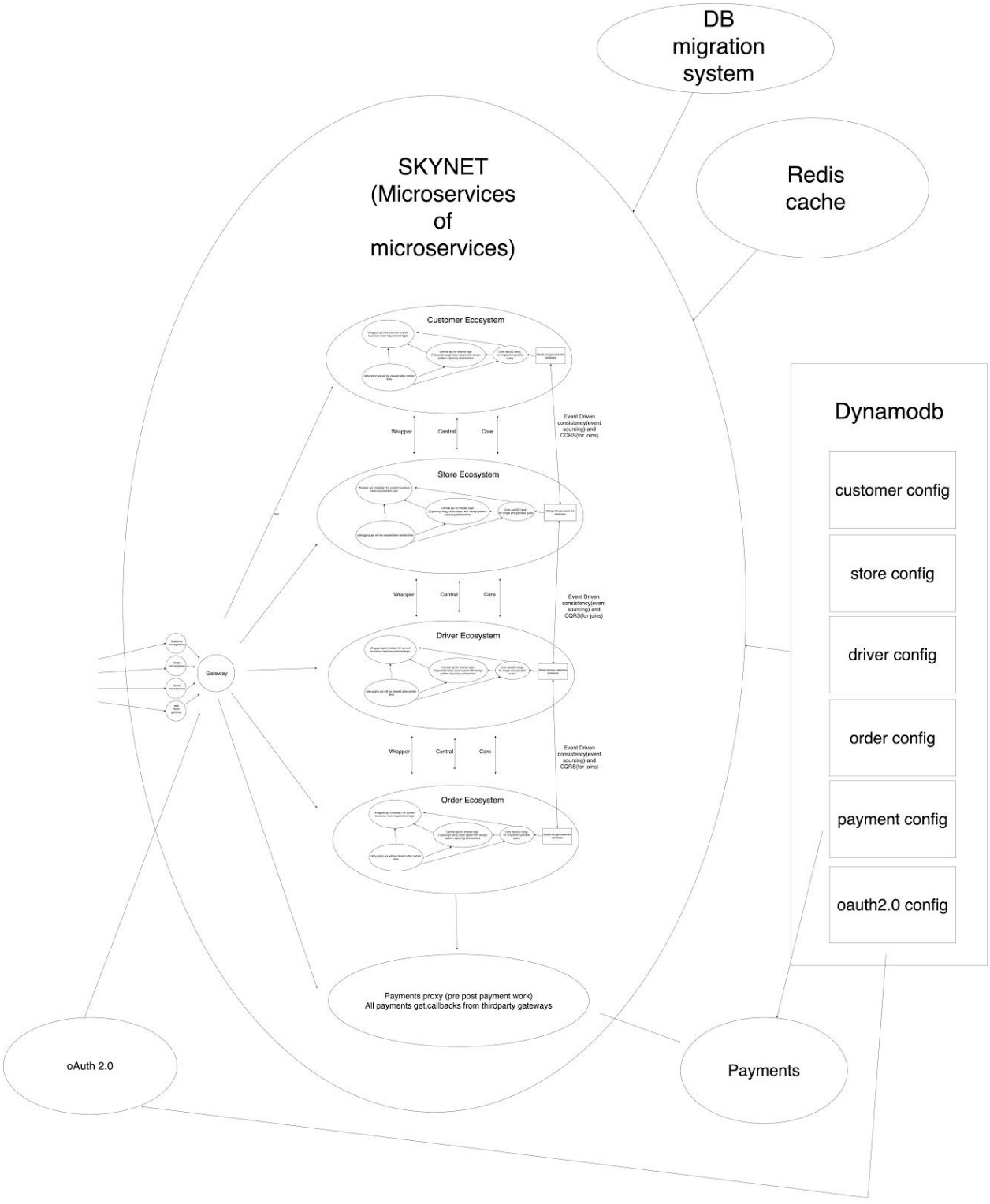
Next Generation Architecture(SKYNET)



High level design:

- Micro gateways : One gateway is a single point failure instead we can keep some small micro-gateways whose whole purpose is to minimize blast radius to its domain itself.
- Domain ecosystem : every domain eco system will have 4 independent microservices.
    - Core/db microservice -
        - For api directly accessing db we can again split into 2 services for read and write.
        - Api like to get order detail by id or get order details of array of id.
        - We will use golang for these services because go is better at machine level code so highly reliable with db connectivity.
        - Golang is best when it comes to concurrency so all parallel fetching and other parallel task we can push in go.
        - Only core api will have access to read and write to db with separate read write token available to only them.
        - Seperate db api are necessary in future if we want to change database it will act just as connection our behind logic wont change just changes in this connector.
    - Central microservice -
        - These service will directly consume coredb api and have all central shared logic of company which needs to be shared across all products
        - All critical api like order validate,coupon validate, credit validate where we don't need changes very often and if needed then it should be done by central team or core dev team.
        - Central api should not contain flow diverge for a specific brand,product,category it need to written in abstract form and pure function for single input it should provide same output every time.
        - Abstract logic because many product/brands will use these logic to implement there wrapper api because of them no changes should be pushed in central api.
        - Abstraction and modularity can be achieved with proper design patterns so all api will be written guided by some design pattern class based.
        - At last we can use Typescript because it all best part of javascript elimination bad part of javascript which is duck typing. So all type checking will done at module level.
        - Every module will test driven because type script support it natively.
        - Type script will provide better inheritance, interfaces backed by microsoft so not going anywhere plus increasing popularity every year.
        - Highly micro modular design with caching and auto logging in and out and documented so that new member can read and make changes.
        - Central api will provide instances of classes.
        - Critical api must be ensured with fallbacks even if some microservices fails basic functionality of order place should work.

        - Only central and core team will work on these taking responsibility of these and coredb api only.
    - Wrapper microservice:
        - These will consume central api and again micro modular design wrapper.
        - According to separate product, brand business different wrapper can be developed and they can be exposed.
        - Later some new requirement come and if needs changes in wrapper so wrapper or any api will not be patched with just some para for different behaviour new wrapper must be written.
        - Even with slightly different behaviour also different wrapper must be written.
        - Different business logic or brand or product different wrapper must be written to isolate logic.
        - Wrappers must be small highly modular documented in and out you can write big wrappers by consuming small wrappers.
        - All wrapper must be written with fallbacks and intelligent error code and response.
        - Contextual error message should get back to originator from where error happened flow should not brake.
        - With use of type script all wrappers will be strictly typed in terms of input and test driven.
- Separate payment microservice
    - for pre and post payment related work, after order validation with intiate payment call will comes down to this repo
    - It will ensure all pre work before hitting payment repo.
    - As payment repo will exists outside this arch alone stateless deployment will only go in that only if new payment mode is introduced rest all payment patch work integrations will at these proxy payment service and stateless payment service will only hit third party gateway no reverse callback can hit it.
    - All reverse callback will hit payment proxy first then will hit payment repo.
- Different dyanmo db for different ecosystem or microservices country wise env wise.
- oAuth with bearer token by client (IP+hardware id like macid)
    - Google oAuth2.0
    - 4 type of token
        - Customer specific token
        - Database read token
        - Database write token
        - General token
- Database
    - Domain driven database for each microservice
    - Authorization in database table
    - Strictly typed database table for mysql
    - Mysql only for relation heavy table

    - Mongo for dump data fetch data or mapping
    - Stored procedures in mysql
    - Casandra distributed database helps in syncing
    - Casandra with high write speed for saving event history
    - CQRS event sourcing for distributed database consistency
- Dev ops:
    - Health check dashboard for all services, which branch deployed ,status, trending exceptions, timeouts, fallbacks, retry
    - Canary release
    - Different dynamo db json for every eco system, one central dynamodb.
    - Only respective domain microservice can access its dynamodb otherwise use central
    - CICD pipeline for test cases , mobile team automate build share.
    - Blue green deployment
- Nano services
    - For single api hosted like npm package for exporting to any third party, driver app, iot devices
    - Chat bot, help support
- Kafka as backbone of communication
    - Backbone for every communication intra service call
    - Kaka/socket io for real time communication to client
    - Activity tracking(iot device noti)
    - Message queue
    - Logging
    - Streaming api
    - Decoupling dependencies
- One shared not microservice but shared module for all services having
    - Log module
    - Auth api
    - Central config
    - Request validator
    - Redis controller
    - Mapping module - all system wide mapping will be here no more client_os,client_source,etc etc
    - Shared constant
    - All service constant
    - Error code constant
    - Encryption module
- Cms feature toggle feature:
    - Feature toggle:
        - All development feature will done keeping in mind in real time this feature can be turned off just like payment mode with configuration in feature schema.

        - Feature schema : it will be a json with true and false hierarchy wise for every feature sub feature to toggle feature.
        - Using these schema a dashboard website which get auto generate dynamically with these schema json, changes will be two way.
        - Feature toggle will be used by stakeholders like po,leads.
- Module design:
    - All module will have middleware which will auto log in and out of call
    - All module should be documented when feature release
    - All module should be test driven
    - Naming convention of constants and modulename
    - Module caching with toggle in flow needs cached response or not keeping in mind dynamic programing approach.
    - All module should have intelligent error response
    - Standard error code which suits for scenario for error
    - Static fallback response if dependent or child module is failing
    - Source should get knowledge. It's a cached response.
    - Generator module for streaming files
    - One central module for calling a api with route url,para getting a promise in response module will print a curl response
- Broke microservice keeping in mind
    - 1.different types developer team different responsibilities
    - 2.deployment criticality
    - 3.what logic needs to be shared and what need to be isolated
- Debugging
    - Better dynamic logging module which needs to be injected at module middleware which will auto log every information in and out will have contextual information repo name,file name,module name, even line number
    - we have to make sure every log generated should be unique
    - Can collect information like function callee name in api flow like trace id and build graph with contextual graph for every api journey having nodes as module/function name.
    - Will use this graph trace id in api journey dashboard to get exact pin point location of issue.
- Client side library
    - Caching of certain api
    - Interceptor for parameter control standards of inputs to api.

Low level design:

Design patterns:

CREATIONAL PATTERN

CONSTRUCTOR PATTERN

function copies won't be created every time a new object gets created

saves object memory

FACTORY PATTERN

one method will created diff type of object as requirement,no need to import everything

MODULE PATTERN

avoid code duplication

avoid global space pollution

reusability

SINGLETON PATTERN

restricts multiple instances of a class to one

STRUCTURAL PATTERN

DECORATOR PATTERN

modify object dynamically in the flow

use composition instead of inheritance

COMPOSITE PATTERN

need to group classes

when dealing with hierarchies/complex or three structures of object,then use these patten to encapsulate everything

FACADE PATTERN

hide complexities

wrapper classes

A single method used for getting details of different complex entities we dont need to know each entities complex logic

BEHAVIOUR PATTERN

CHAIN OF RESPOSIBILITY

a group of object is trying to solve a problem if cant solve then pass problem to another in linked style

FLYWEIGHT PATTERN

any thing which can be shared or cache should be saved by dp using flyweight classes

increase memory efficiency

STRATEGY PATTERN

can change strategy dynamically from anywhere of given flow

can select which algo to run on run time

OBSERVER PATTERN

simple pub sub.

one class will trigger notification to respective or concerned classes

Basic idea of all patterns with implemented for our current architecture available at my github repo: https://github.com/saru998/javascript_design_patterns_es6