Navigator Core Architecture

Table Of Content

Overview

Navigator solution is expansive and supports an array of capabilities that can be leveraged by partners. This document covers the Navigator platform architecture detail.

Navigator Overview

Navigator supports teachers, and with the right tools, to enable them advance student outcomes and foster an innovative community that supports the success of every student. At the core of Navigator is the huge catalog of open education resources that are highly curated and metadata enhanced. This helps teachers pull together relevant content items for students. As students interact with the content items, the generated data is analyzed in various dimensions. The rich metadata at content helps Navigator track learner proficiency at various concepts.

Navigator Platform Components

Navigator platform comprises of various task specific microservices and orchestrators that can broadly be categorized into:

- **User Identity and Authorization**: Handles user identity management, authorization checks, auth token issue, and Single-Sign On (SSO) support.
- **Competency model**: Handles taxonomy model, framework setup and crosswalk support
- **Catalog services**: Handles content Create, Read, Update & Delete (CRUD) services, content index and content search
- **Suggest engine**: Handles suggestions based on learner identity (profile), learning maps (catalog) and principles of learning
- **Data write and read services**: Handles data logging, data reports, and data postback
- **Datascope** (Data stream processing pipeline)
- **Learner profile services**: Handles learner profile compute and read services
- **Navigator application**: Orchestrates all above to pull together various learner flows

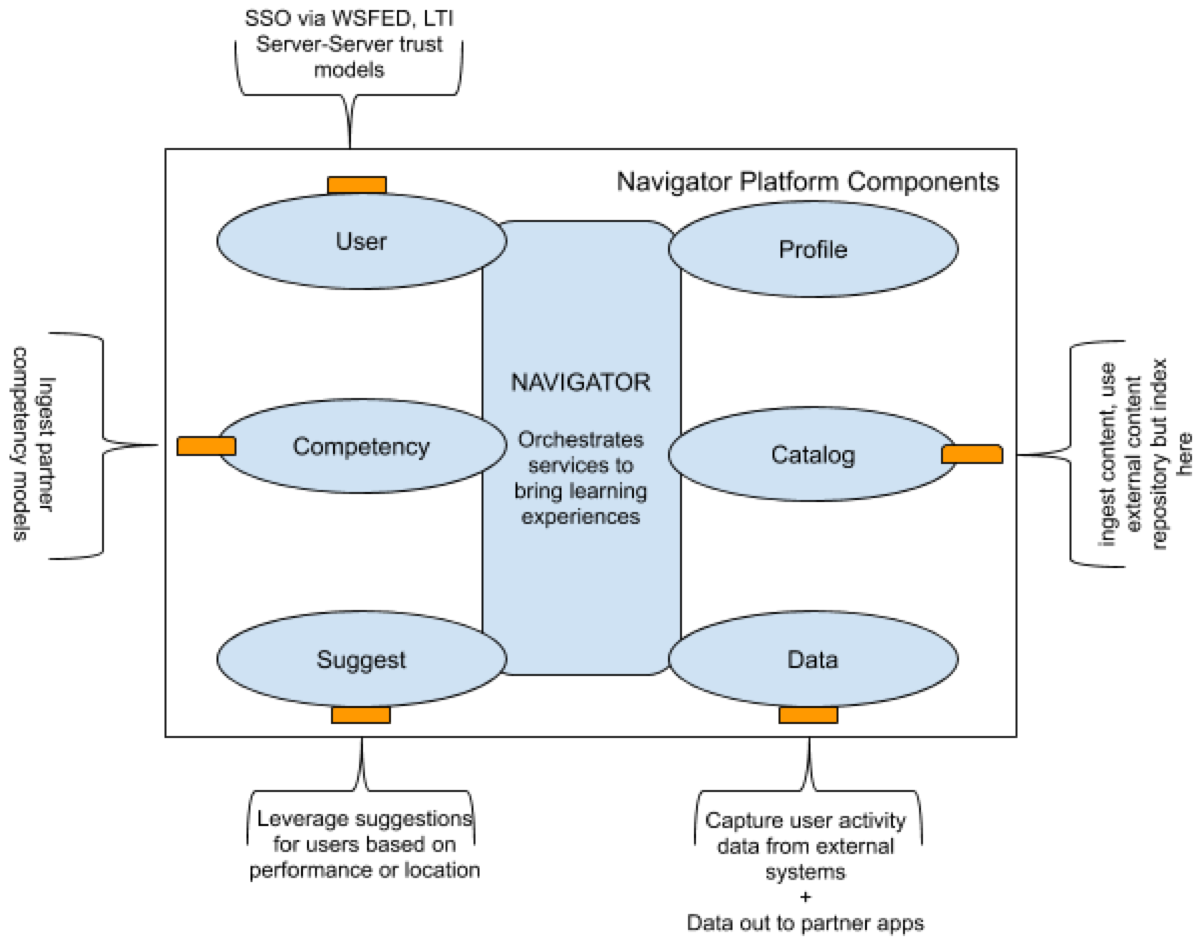
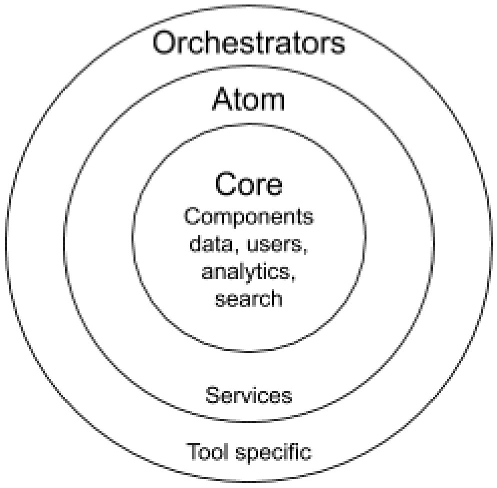

SSO via WSFED, LTI Server-Server trust models
Navigator Platform Components
User
Profile
NAVIGATOR
Orchestrates services to bring learning experiences
Ingest partner competency models
Competency
Catalog
ingest content, use external content repository but index here
Suggest
Data
Leverage suggestions for users based on performance or location
Capture user activity data from external systems + Data out to partner apps

Figure: Navigator Platform Components & Integration Points

Navigator Core Design Principle

Core layer is responsible for data management, provides data level CRUD capability as API, manages user, content, class and usage data. Atoms and Orchestrators built around these Core capabilities extend to specific needs is the preferred model. This helps us manage extensions consistently and without a lot of code bloat.



Essentially, our design would follow the below structure:

- Different data sources core to our systems will be central and abstracted to external layers
- Core services expose the data sources to achieve one level of abstraction to avoid systems tied directly to data sources.
- Extension APIs work on top of Core APIs and will be a mechanism to feedback data / information into the data sources which will be assimilated and used
- Orchestration APIs enable different kinds of responses for custom usage of tools. This could be either single orchestration API, which could be reused, or sub system level orchestration layer, which talks to core APIs to assemble the required responses.
- Subsystems and Tools layer which will provide the experience are built on top of this
- Message bus will enable different types of communication between different nodes of platform

Further, each Core component and services comply with the following guidelines:

- RESTful interface exposes core features, JSON responses only
- Controller-service separation of concerns
- Cacheable and layered architecture
- API versioning followed when upgrading the same
- Event based update of other subsystems like search / analytics
- All API protected with authentication key to avoid denial of service (DOS) attacks
- Role based access control for authorization of API

The basic constraint behind the architectural choices is governed by scalability. The system is designed to be horizontally scalable. To achieve this most core components are written using async frameworks. Using async frameworks enables the blocking operations to happen on worker pool threads while main threads are available for servicing user requests. Note that because of JDBC layer, the architecture is not async end to end.

Technology Stack

Platform	Purpose
Ubuntu 16.04 or above	OS recommended; but you can use other variants as well
Java 8	All backend microservices are implemented in Java
Kafka 0.10.1.1	Messaging system for cross-service communications
Zookeeper 3.4.10	Distributed coordination of kafka nodes
Cassandra 3.0	Data store for raw/unprocessed usage events
Elasticsearch 5.6.0	Content index and search engine backbone
Postgres DB 10.3	Database to store all relational and transactional data
Redis 4.0	Cache storage for frequently accessed critical data
Memcached	Cache storage used at real-time subsystem
HAProxy 1.6 or above	To manage routing rules to various services / sub-systems as well as load balance traffic; other alternatives like Application Load Balancer can also be used for the same purpose
Nginx 1.14.0 or above	To manage multiple front-end sites; other alternatives can also be used for the same purpose
Gradle 2.7 & 4.4	Build tool for most of the back-end microservices; some services need higher version of gradle as noted at build.gradle files in respective service repository
NodeJs 4.4.*	For frontend build as well as to run specific identity provisioning service component
Tomcat 8.0 Maven 3.3.9	While most of the component redesign has moved away from Maven build and managed under Tomcat server, there is one server (search and suggest service) that still is not migrated away. Hence this is needed today but may move away from the requirement at a later time.
Ember-Cli, Bower, PhantomJs, Grunt, Stubby	Build environment for front-end application

Architecture Overview

Component Architecture

At the heart of the Navigator are the Core services. These are designed as microservices created using VertX. API access requires an auth token which is validated at every API entry point. The tokens are managed at Redis cache and expiry is extended every time an API is invoked. Postgres database is the key data store for all core entities - user, content, classes, competencies. Messaging across clustered sub-systems is managed via Kafka message bus. Search and Suggest subsystem uses ElasticSearch. All activity data is stored at Cassandra while processed user activity data required for explicit reporting purposes is captured at analytics Postgres database. All services are exposed over Restful API and responses are always in JSON format. Frontend is developed in EmberJS and orchestrates the various services per application flow needs.

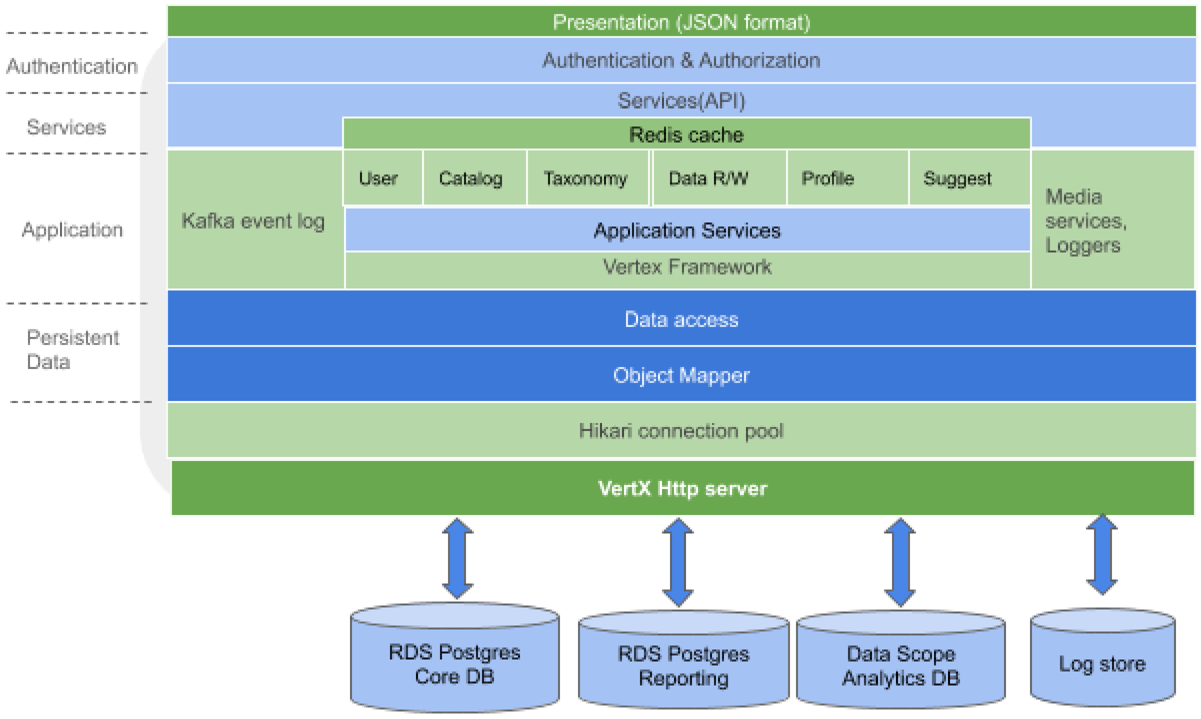


Figure: Core Architecture

Key Service Components are as below:

- Core Services: Content, Class, Metadata and Taxonomy

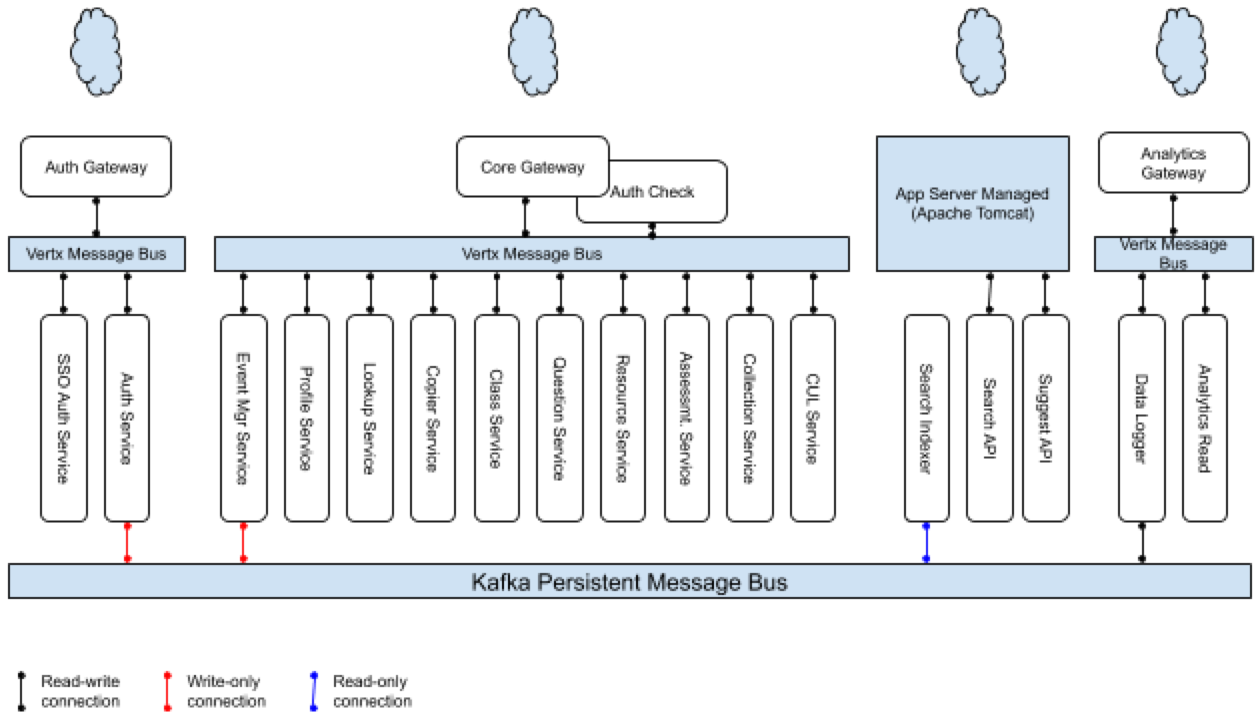
- Auth Services: Users, Authorization, Tenant, Partner, App
- Study Suggestions
- Class and User Reports
- Realtime
- User Interface: Navigator, Widgets
- Partner Integration Components:
    - SSO support for service providers LTI, GMail, SAML, WSFed, Shibboleth
    - Data In APIs
    - Data Out APIs / Post-back
    - Roster Sync

Clusters

Each component is a collection of one or more processes running. With such a view each component forms a cluster. In a cluster, there is at least one cluster member which is responsible for handling incoming requests. These requests are mostly HTTP based and thus the cluster member handling them is the API gateway. The gateway is aware of its members and what requests each member can process. Its primary task is to act as a web server listening for requests. Each cluster generally contains one cluster member which is responsible for auth checks. API gateway will bind this cluster member for all APIs to make sure that valid requests are reaching the downstream handlers.

Auth check handlers are responsible to verify the access token provided in request with respect to token store present at Redis. In case of auth failures, the response is returned to caller. If auth is successful, then request is dispatched to the next handler.

The communication between cluster members happen using TCP with a request response model which is asynchronous. The cluster management happens using Hazelcast where in a new cluster member is automatically detected and is made part of cluster.



Component Architecture

When it comes to horizontal scaling, the architecture provides multiple avenues to scale, e.g.

- One gateway cluster member backed by multiple instances of processing handlers. The load balancing in this model happens in round robin fashion.
- The cluster itself is replicated and load balancing is managed at the level of entry point of deployment infra (e.g. haproxy, nginx etc)
- These two approaches can be mixed as well. E.g. if there is load on only single handler for a cluster, only one cluster can be maintained and handler can with multiple instances.

There is another level wherein scalability can be dealt with. Since the clusters are implemented to be asynchronous, there is a concept of event loop threads and worker threads. Their count is configuration governed and in case where scaling needs change, the configuration can be tweaked to provide better scalability without additional hardware.

While the intracluster communication happens with the TCP bus which is non persistent and local to the cluster, the inter cluster communication happens using different methodologies. Two of which are HTTP calls and Kafka. Kafka communication is used where the communication does not need to be request/response type. However, if there is a need for request/response type of communication across the cluster, then HTTP based communication is used.

Environments

There are different environments for different purposes. Stage, Beta and Production are the most important ones from release readiness perspective. Development and QA environments are important from an engineering perspective for feature creation.

Each environment is completely isolated from other environments to provide for better security. This is done using Virtual Private Clouds specific to that environment. This makes sure that no member from one environment is able to access any member from other environments. The communication, if it has to happen, needs to happen using authenticated APIs.

Data In & Data Out

At a broad level, event data coming in is streamed to Kafka message bus which keeps accumulating events while ensuring all consumers in the system get a copy of the message. There are task/metric specific consumers that pick up events from Kafka, process and store them as needed. All events are always stored at Cassandra DB for any future reference. Class and other learner specific events data is processed for reporting needs and processed report data is stored at PostgreSQL reports database for better query ability. Additional aggregates computed for purposes of Learner Profile are stored at Postgres database.

Data In & Data Out components expose REST api for data logging with XAPI compliance on request payloads related to learner study activities.

Navigator collects data related to Content, Learner Study activities, Search, Social and other User events, whether originating within Navigator system or due to ingestion by partner systems.

The activity data is stored in a manner to enable filter on multiple dimensions: time, user, action, and content. Data would be used to generate Aggregated Reports, Learner Profile, Learner Performance report, Admin Reports, depict User Journeys, and more.

Data In

Navigator collects data as users interact with the system. There are a variety of events that generate activity data. Some are events explicitly logged by application / components (client-side events); and some are generated as a side-effect of certain user actions (server-side events). Further, there are scenarios where external entities log data to Navigator via an exposed API.

Event Types

Navigator application is one of the primary event generators for the learner activities. Events are generated for learners':

- Independent Learning Activities where learner self-studies an available course
- In-Class Activities where learner is assisted by teacher but learning is self-paced
- Daily Class Activities where teacher focuses entire class on a specific day activity
- Content Activities where users create and update content, collaboration, reuse, view
- Other Activities related to user sign-up, login, class join, follow & other social interactions

Typically, for a learner, the progress, performance and proficiency are key performance indicators (KPIs) captured. So, Score, Time spent and reactions for the given context of learning are logged in the system, as learners work with the Navigator application. There are additional events captured related to teacher grading of free response questions, teacher override of score, etc that impact the above KPIs.

Additionally we also plan to associate Mindset Vectors, Community Vectors, Skills vectors to describe the Learner's profile and to help the system locate the learner more precisely and to enhance search and suggest for the learner. Some of the examples of these vectors are - Self Confidence, Grit, Perseverance, Engagement, Citizenship, Authority etc.

Note that using the Navigator application triggers the data collection automatically. However, the models where partner users are not using Navigator application is also supported. In this model, Partners and System Integrators send discrete events to Navigator. Partners may send events in the standard XAPI-formatted payload or a variant based upon mutually agreed API contracts.

Different CRUD operations on the Navigator Core entities like Class Create, Course Copy, Add Content are captured as internal server-generated events. Internal events are also generated for other activities like User Sign-in, Assigning Collaborators, Student joining Classes etc.

Event Logger

The events generated are stored as Raw Events at Cassandra and are also further processed by the Event Logger Component. The events are also stored as commit log in Kafka for a predefined time.

The Event Logger is setup as a scalable and a high availability (HA) Cluster. The requests coming to the Event Logger are HTTP which is handled by an API Gateway. Primarily the API Gateway is a Web Server capable of handling high volume requests.

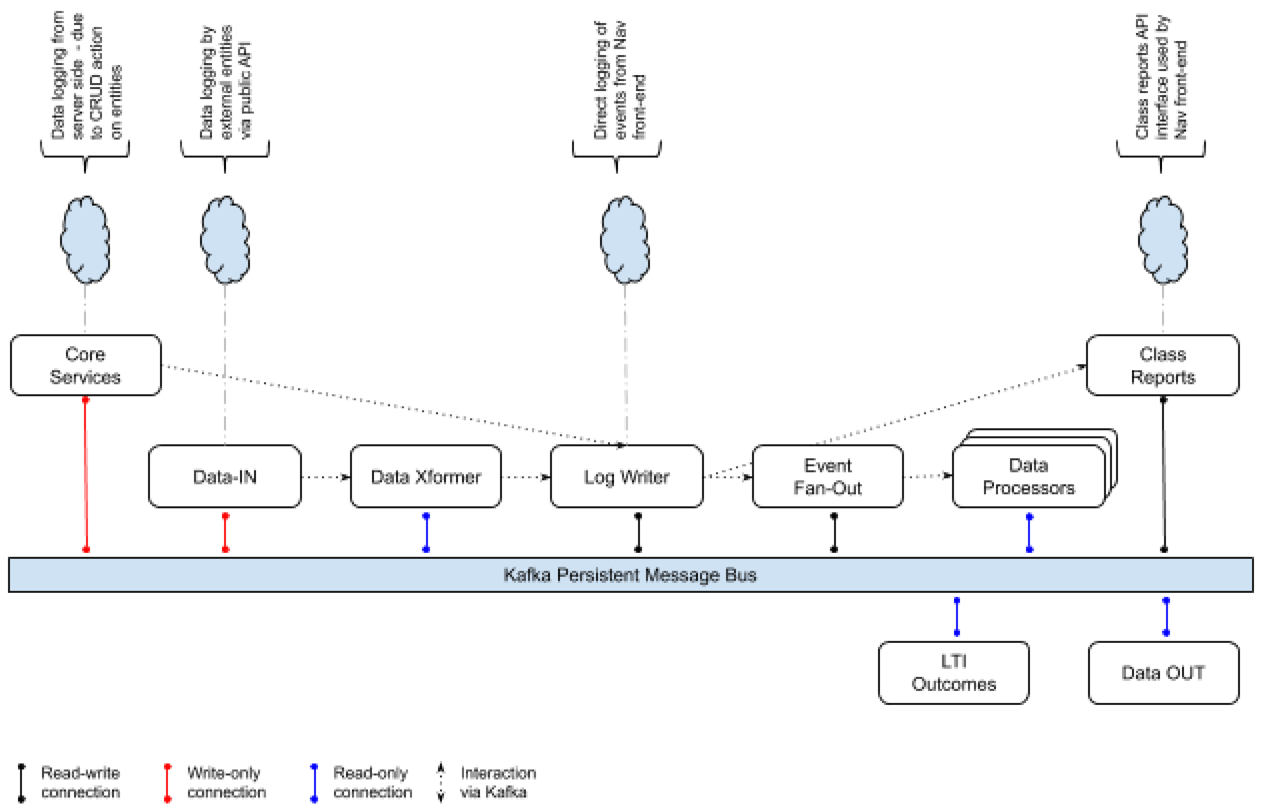
The gateway relays the requests (events) to the downstream processor(s) via Kafka. The downstream event processor is responsible to process every event and store the data into the underlying data store, PostgreSQL database.

Event Transformer

Partners & system Integrators can send data to Gooru using the shared Navigator APIs. Partners can send data to Navigator as a JSON Payload that is xAPI Standards compliant or any mutually agreed upon custom format. The Event Transformer component will transform the Partner messages into Navigator format and relay them down to other event processors and data stores.

Event Processing

All events are collected at data-in / event logger component which stores all raw events at Cassandra while propagating the events further for additional custom processing. The event fan-out component is responsible for generating necessary events and push through the message bus. Specific data processors then consume the messages and compute the specific metric that each one of the processors is responsible for. Based on the event type, the Data Out and outcome post-back components do the necessary additional transformation and relay of the same to the partner ecosystem.


Data logging from server side - due to CRUD action on entities
Data logging by external entities via public API
Direct logging of events from Nav front-end
Class reports API interface used by Nav front-end
Core Services
Class Reports
Data-IN
Data Xformer
Log Writer
Event Fan-Out
Data Processors
Kafka Persistent Message Bus
LTI Outcomes
Data OUT
Read-write connection
Write-only connection
Read-only connection
Interaction via Kafka

Event Flow Design

Global Event Processors

The events flowing in from the outside world are mostly stateless atomic events and they may not carry a lot of context information with them. Note that context implies the association or references to the metrics associated with the events and not necessarily imply the context of event attributes. For example, the content usage events are atomic and only project information related to the score of a particular question identified in that Events or time spent by the user on that particular resource. However it is always desirable to know the cumulative time spent for the entire collection or the cumulative score in assessment.

This needs further processing of the events by correlating it to the current context in the system. Global Event Processors (GEP) enables this post-processing of events. There maybe one or more global event processors in the system.

Event Fan-Out

The GEP events are sourced to the Event Fan-Out / Demultiplexer component. The primary purpose of Event Demux is to segregate the factual information provided in the GEP events, create specific internal events per measure (metric) and emit these Discrete Events downstream to the Discrete Event Processors. For example, the key measures (metrics) associated with an Assessment are time spent, score and reaction. Once the assessment is completed an event containing information of the Overall time spent, score and reaction along with the dimension information (User, Course, Unit, Lesson, etc.) is sent to the Demux. Demux will create three discrete events for the three measures (including the event attributes/dimensions) and send it downstream for further processing.

Discrete Event Processors

The discrete event processor (DEP) is responsible to process every event and store data into the underlying data store. Based on the events being consumed by the DEP's, they are also chained, i.e. one DEP can act as an event producer for another DEP.

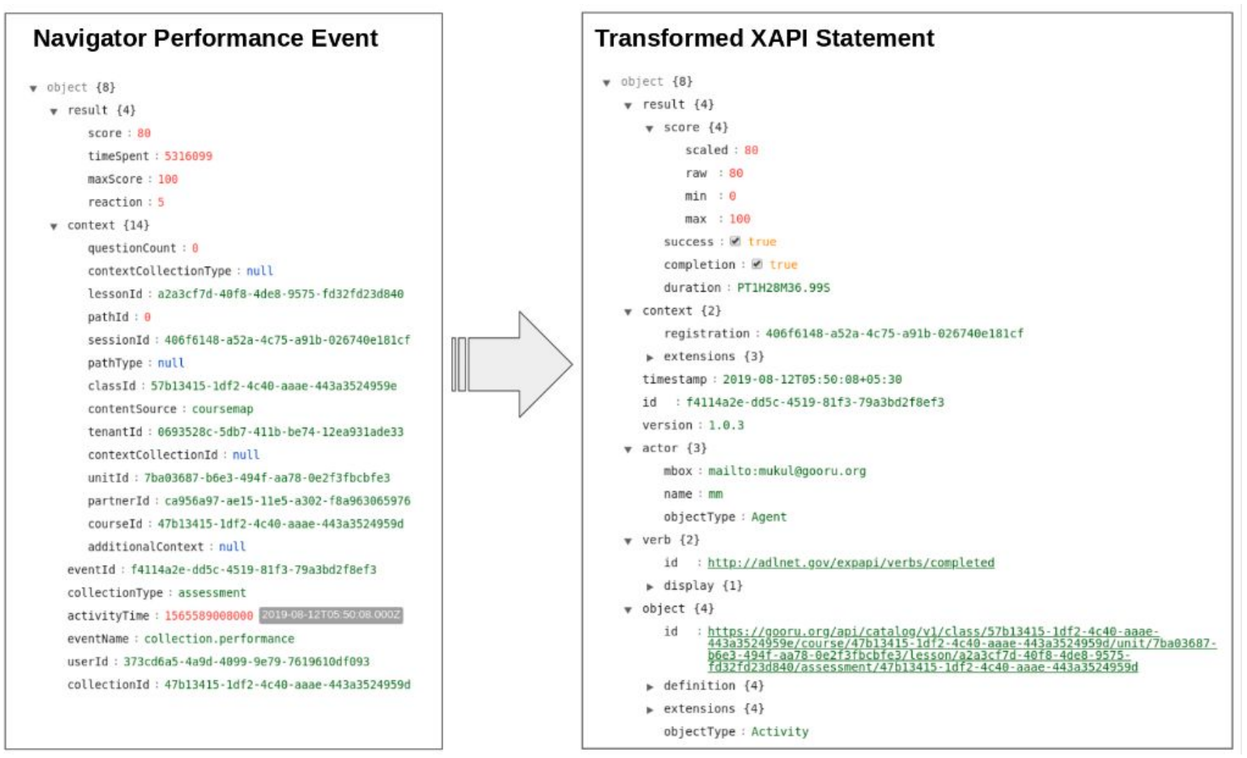
The processed Data from the DEP's will serve the Partner Data Out components and the Data Read API's.

Navigator XPI Transformer

xAPI is an interoperability specification that allows learning technologies to talk to one another. Integrating two xAPI conformant systems can be considerably faster and cheaper. This includes migrating data from old learning systems to new ones.It is possible to collect data about the wide range of experiences a Learner has (online and offline). xAPI captures data in a consistent format about a person or group's activities from many technologies. Heterogeneous systems are able to securely communicate by capturing and sharing this stream of activities using xAPI's simple vocabulary.

The Navigator Event Transformer will transform the relevant native Navigator Events into XAPI Statements. These XAPI Statements will be stored into Learning Record Store (LRS)

Sample Navigator Event Transformation to XAPI Statement



Navigator Performance Event

```
▼ object {8}
  ▼ result {4}
      score : 80
      timeSpent : 5316099
      maxScore : 100
      reaction : 5
  ▼ context {14}
      questionCount : 0
      contextCollectionType : null
      lessonId : a2a3cf7d-40f8-4de8-9575-fd32fd23d840
      pathId : 0
      sessionId : 406f6148-a52a-4c75-a91b-026740e181cf
      pathType : null
      classId : 57b13415-1df2-4c40-aaae-443a3524959e
      contentSource : coursemap
      tenantId : 0693528c-5db7-411b-be74-12ea931ade33
      contextCollectionId : null
      unitId : 7ba03687-b6e3-494f-aa78-0e2f3fbcbfe3
      partnerId : ca956a97-ae15-11e5-a302-f8a963065976
      courseId : 47b13415-1df2-4c40-aaae-443a3524959d
      additionalContext : null
    eventId : f4114a2e-dd5c-4519-81f3-79a3bd2f8ef3
    collectionType : assessment
    activityTime : 1565589008000 2019-08-12T05:50:08.000Z
    eventName : collection.performance
    userId : 373cd6a5-4a9d-4099-9e79-7619610df093
    collectionId : 47b13415-1df2-4c40-aaae-443a3524959d
```

Transformed XAPI Statement

```
▼ object {8}
  ▼ result {4}
    ▼ score {4}
        scaled : 80
        raw  : 80
        min  : 0
        max  : 100
      success : ✓ true
      completion : ✓ true
      duration : PT1H28M36.99S
  ▼ context {2}
      registration : 406f6148-a52a-4c75-a91b-026740e181cf
    ▶ extensions {3}
    timestamp : 2019-08-12T05:50:08+05:30
    id   : f4114a2e-dd5c-4519-81f3-79a3bd2f8ef3
    version : 1.0.3
  ▼ actor {3}
      mbox : mailto:mukul@gooru.org
      name : mm
      objectType : Agent
  ▼ verb {2}
      id   : http://adlnet.gov/expapi/verbs/completed
    ▶ display {1}
  ▼ object {4}
      id   : https://gooru.org/api/catalog/v1/class/57b13415-1df2-4c40-aaae-
             443a3524959e/course/47b13415-1df2-4c40-aaae-443a3524959d/unit/7ba03687-
             b6e3-494f-aa78-0e2f3fbcbfe3/lesson/a2a3cf7d-40f8-4de8-9575-
             fd32fd23d840/assessment/47b13415-1df2-4c40-aaae-443a3524959d
    ▶ definition {4}
    ▶ extensions {4}
      objectType : Activity
```

Data Out

Generically, data is served to the external world using the API Infrastructure. Navigator frontend application uses the available API for various in-product reporting purposes. Data can be pulled by the partners using the REST APIs. These data read / reporting APIs are tuned to report data at a specific context level - class level of learner activity and/or independent learner activity related reports.

Additional Data Out components optimized for Partners' specialized data requests are also created. Data Out component can be configured to handle a data pull request by the partners or data can be pushed to the partners at specified triggers. The trigger can be an incoming event or a specific time interval. Data Out also includes the IMS LTI outcomes service that allows Gooru to send information like score, time spent on content to its consumers (typically the Learning Management Systems).

For Interoperability with other Learning Technology Platforms/products Navigator Activity data is stored into Learning Record Store (LRS) as XAPI Statements. RESTful APIs are exposed to GET/POST XAPI Statements into this LRS. Apart from the Navigator Activity events, events from external learning systems can be stored and fetched from this LRS.

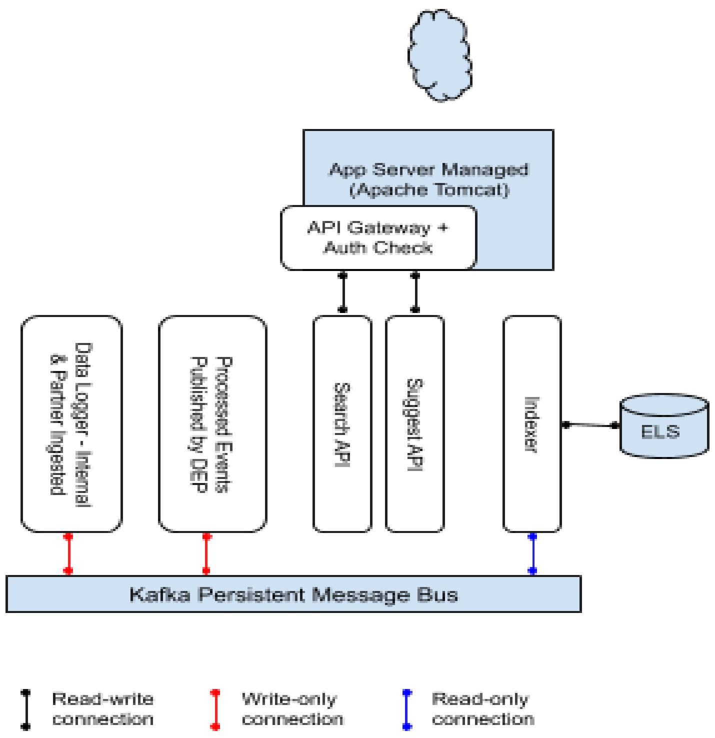
Search & Suggest

Navigator uses ElasticSearch (ES) at its core for Search and Suggest services. All content catalog items, along with its rich metadata, are stored at index as terms for faster search experience. There are multiple indexes in use that are used for specific purpose either standalone or in conjunction with the other: resource index, collection index, course index, unit index, lesson index, content publishers index and crosswalk index.

Architecture

Search infrastructure is distributed in nature and is designed to be horizontally scalable. Key components of the infrastructure include

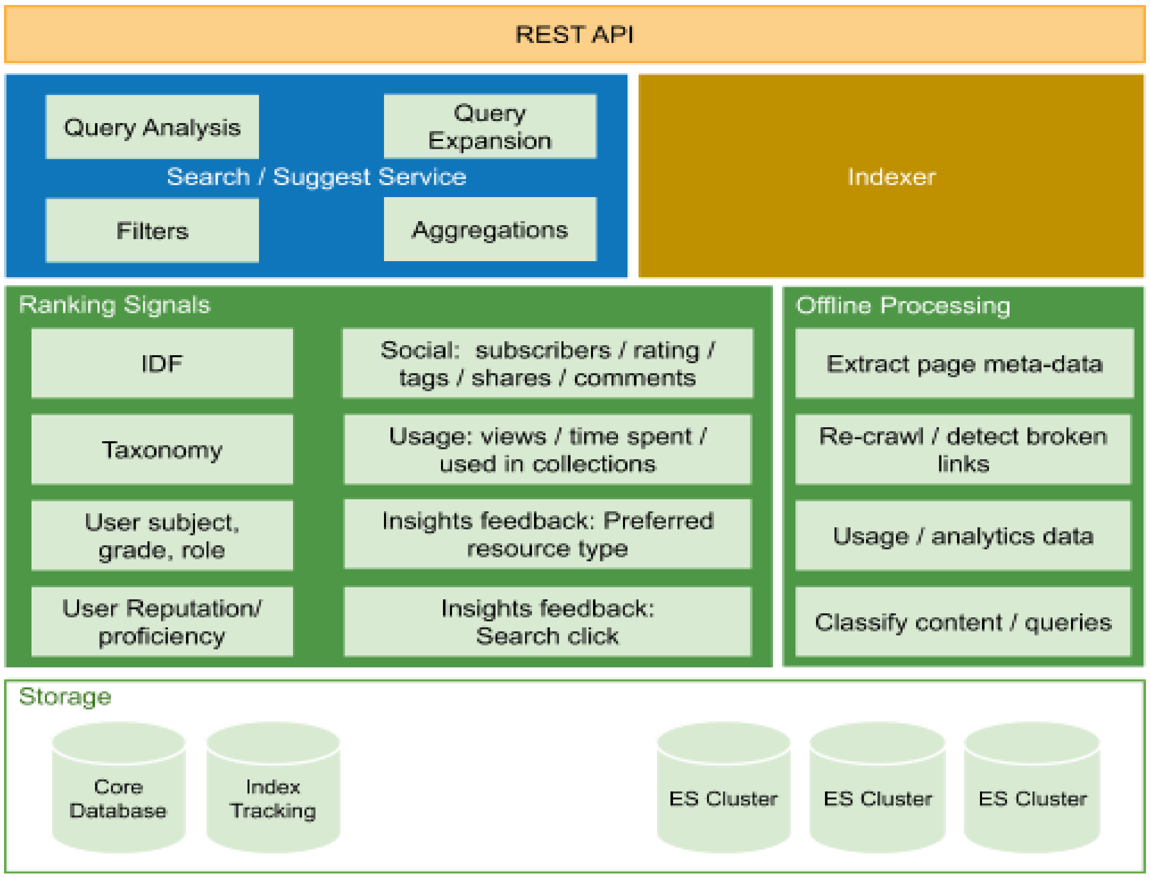
- Indexer adds / updates content and metadata into the search index.
- API layer handles search queries and is the user facing part of search & suggest system.


App Server Managed (Apache Tomcat)
API Gateway + Auth Check
Data Logger - Internal & Partner Ingested
Processed Events Published by DEP
Search API
Suggest API
Indexer
ELS
Kafka Persistent Message Bus
Read-write connection
Write-only connection
Read-only connection

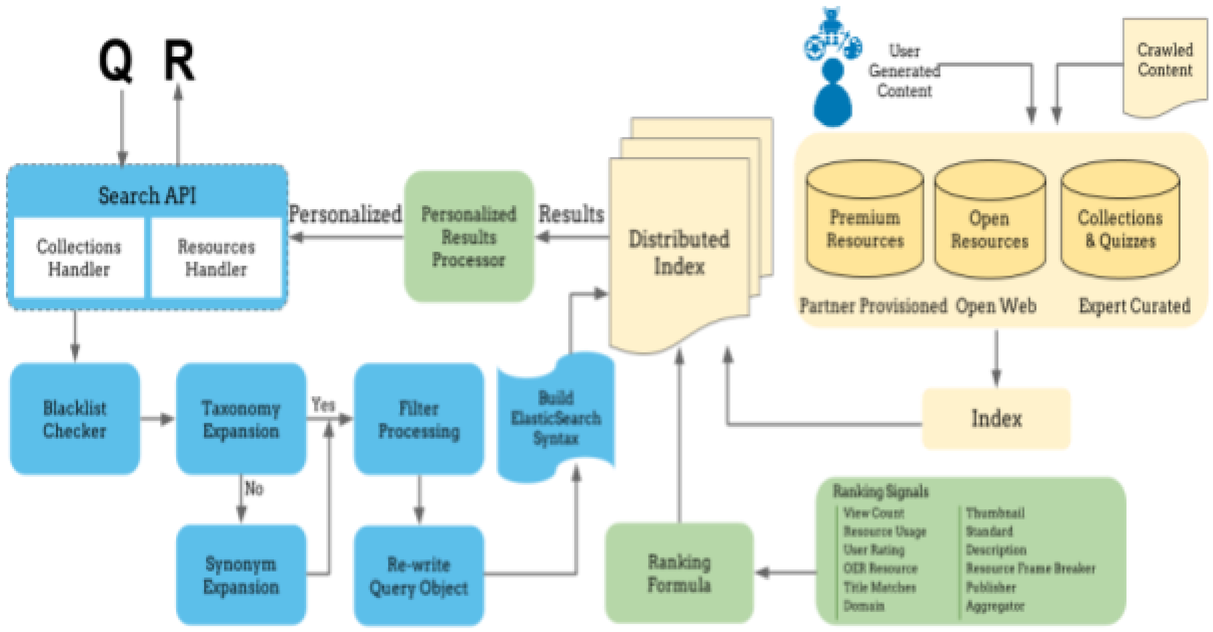
As content gets created or updated, Navigator core services publish this change data / events into the Kafka message bus. These messages are consumed by indexer component which then refreshes the documents that have changed and updates an index attributes across ES indexes. Index updates are in real-time for user updates & in batch mode for bulk content ingestion done via tools. 4-gram index is maintained currently based on usage trend analysis.

Search and Suggest services utilize various data points - user inputs, usage data gathered, content extracted metadata and other contextual information to serve response for that context.

Suggest, unlike Search, has no real user input query term but the same is assumed based on the context of invocation. The requests go through a processing pipeline to classify, expand, rewrite, apply necessary filters, search, rank using signals, and gets personalized.


REST API
Query Analysis
Query Expansion
Search / Suggest Service
Filters
Aggregations
Indexer
Ranking Signals
IDF
Social: subscribers / rating / tags / shares / comments
Taxonomy
Usage: views / time spent / used in collections
User subject, grade, role
Insights feedback: Preferred resource type
User Reputation/ proficiency
Insights feedback: Search click
Offline Processing
Extract page meta-data
Re-crawl / detect broken links
Usage / analytics data
Classify content / queries
Storage
Core Database
Index Tracking
ES Cluster
ES Cluster
ES Cluster

Relevance score of content is computed based on Term Frequency (TF) and Inverse Document Frequency (IDF) and adjusted based on usage signals collected by Navigator, with TF/IDF weighted at 60% and usage signals weighted at 40%.



Indexing Processing & Query Flow

REST APIs are exposed for search across catalog with a variety of filters, related content suggestions, performance based suggestions, crosswalk, and index export.

Key learning path suggestions are offered to student are as below:

a. Profile based route suggestions
b. Profile based rescope suggestions
c. Performance based reroute suggestions
d. Teacher influenced/discretionary reroute suggestions

Profile based Route Suggestion

At every student journey, based on learner profile, more specifically learner current location as reflected at competency skyline, learners are offered a route that supports student in successful completion of a chosen target journey.

Route suggestions are always made in the context of a course and at the entry point into course as a way to better prepare students for what's coming ahead in the course. These suggestions are made only once at the start of the journey; and are not offered after users start the journey. But learners can go back anytime to review previous suggestions and accept the same route.

Route suggestion is a preparatory course, compiled as an array of units. Depending on the domains across which the competency gaps are spread, this preparatory course may contain multiple units, one Unit per domain, and multiple lessons per unit, one Lesson per competency. The route suggestion intends to help learner cover the competency gaps prior to starting the destination journey.

Profile based Rescope Suggestion

When students choose a destination Navigator course for study, Rescope of selected destination course happens to focus learner on competencies learner is yet to master. The rescope computation identifies the content at destination course that the learner can skip and focus on the content that will enhance the learner mastery.

Rescope suggestions are always made in the context of a course and at the entry point into course as a way to focus the student on competencies not yet mastered at the course.

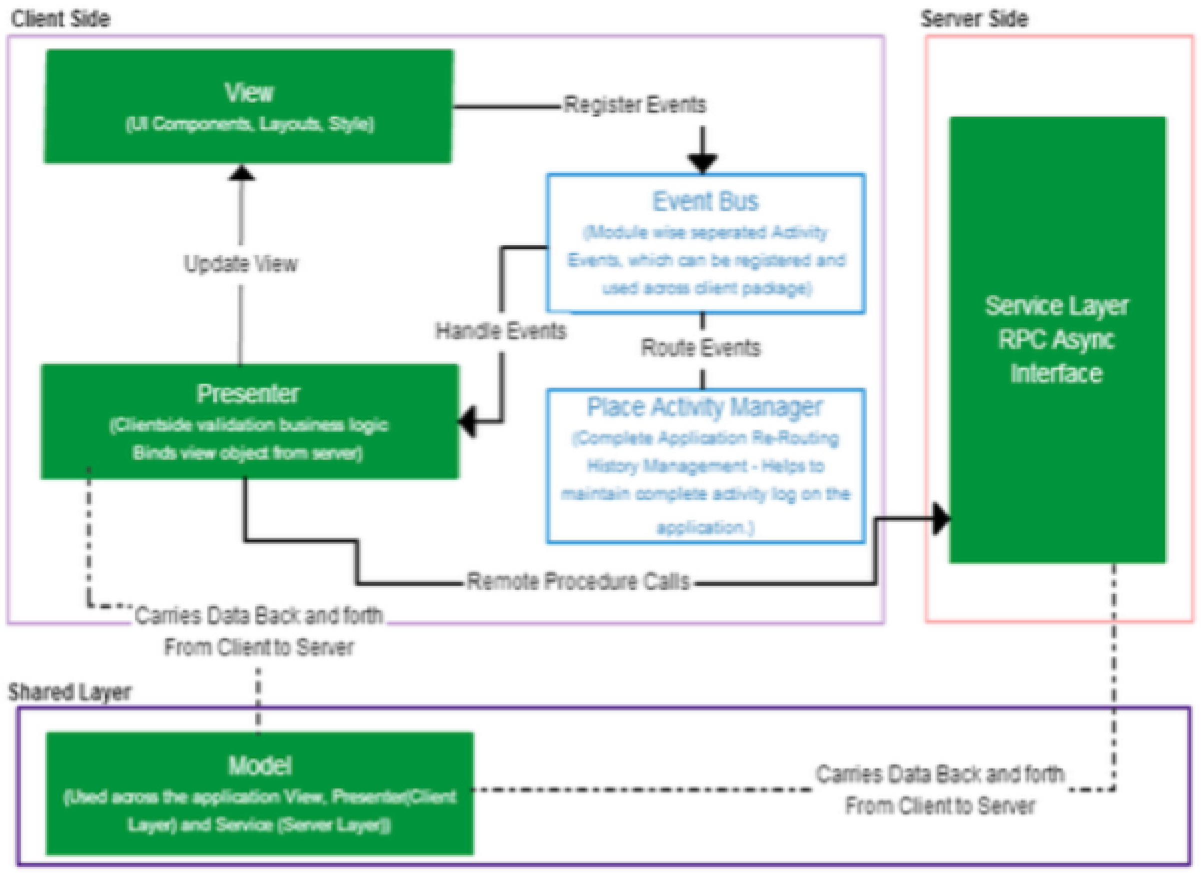
Performance based Reroute Suggestion

As students commit to a given journey and progress down a learning path, Navigator tracks a student activities and performance at every competency. Reroute always happens in the context of an assessment. Based on student performance, at the end of every assessment, a suggestion that can result in rerouting is made:

- If the student performs poorly (<80%) at the competencies covered by the assessment and if competency is not completed or mastered previously, an appropriate signature collection is suggested as a backfill.
- If the student performs very well (>= 80%) at the competencies covered by the assessment and if there is no mastery data (system may have completion data), student is offered a signature assessment as suggestion to earn mastery. If this assessment is part of a Route or Teacher suggestion, then reroute logic is not triggered, implies signature assessments are not suggested.

Navigator Application Design

- EmberJS used for front-end application
- Reusable components developed as embeddable components
- Ember-Data layer for data encapsulation
- Bootstrap framework stitched together for responsive UI
- Localization managed via ember-i18n services
- (Limited) Feature configuration managed via external config files
- Headless testing of functionality using Ember Tests, PhantomJS & Stubby

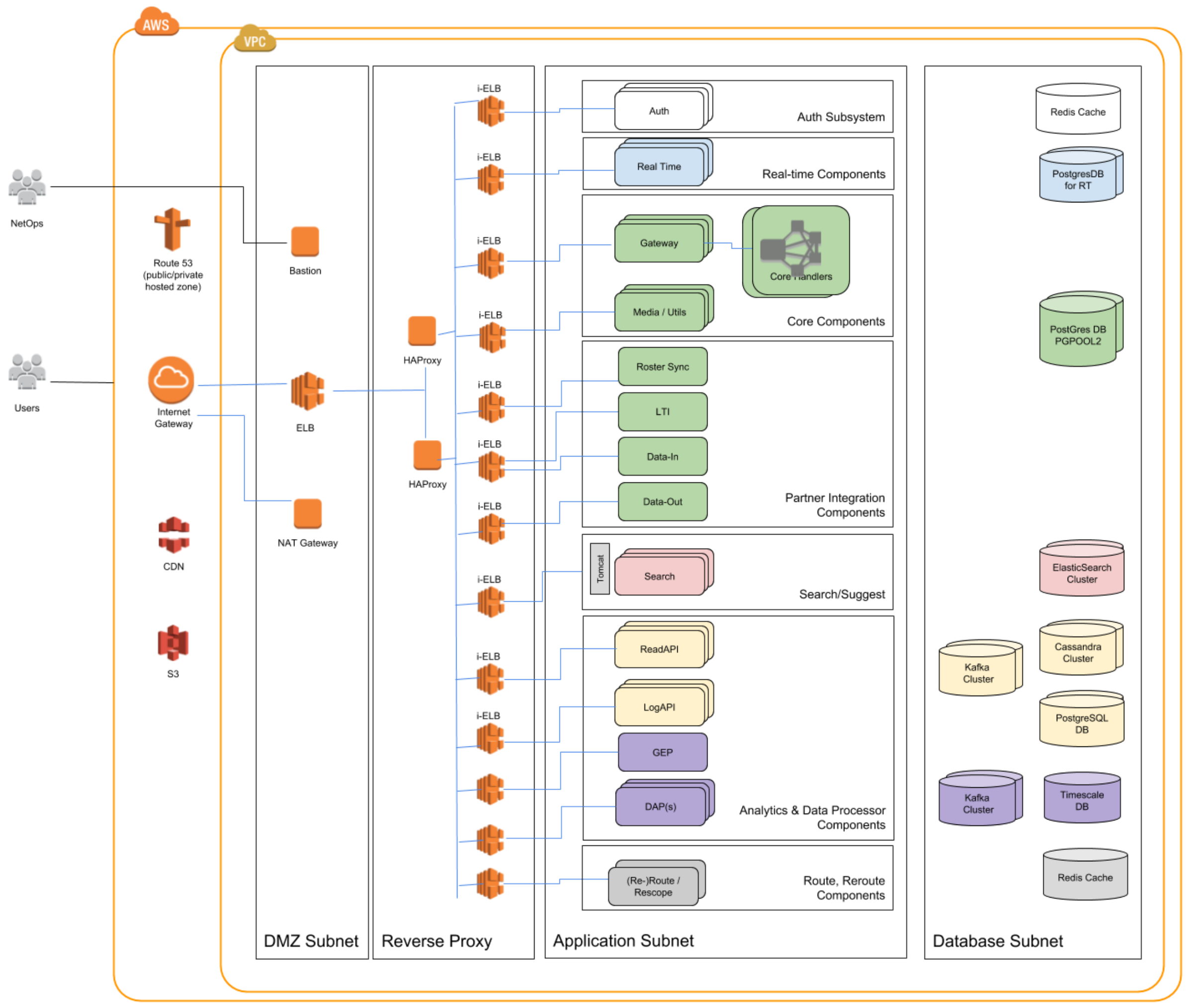

Client Side
Server Side
View
(UI Components, Layouts, Style)
Register Events
Event Bus
(Module wise seperated Activity Events, which can be registered and used across client package)
Update View
Handle Events
Route Events
Presenter
(Clientside validation business logic Binds view object from server)
Place Activity Manager
(Complete Application Re-Routing History Management - Helps to maintain complete activity log on the application.)
Service Layer RPC Async Interface
Remote Procedure Calls
Carries Data Back and forth From Client to Server
Shared Layer
Model
(Used across the application View, Presenter(Client Layer) and Service (Server Layer))
Carries Data Back and forth From Client to Server

Front-end Architecture

Deploy Architecture

Navigator sources are spread across Github repositories with build and deploy process managed at Bamboo-based Continuous Integration / Continuous Deployment (CI/CD) implementation and via AWS Code Deploy agents. For detail on manual build process, refer to this document. For access to repositories and other infra, do contact us at partners@gooru.org.

Navigator is deployed at AWS in a high-availability mode. As such, at each layer there is redundancy setup so as not to impact application & services availability. Bamboo-based CI/CD implementation is in place to help with automated build and deploy of application components.



The first component which is outside world facing from infra perspective is Elastic Load Balancer (ELB). ELB serves purpose of:

- Termination of SSL
- End point of domains
- Load balancing between different haproxy(s)

Once the request is forwarded from ELB, it ends up on one of two parallel setups which are frontended by haproxy. These two setups are identical and they belong to same VPC. Haproxy rules are responsible to identify the cluster which is meant to serve the incoming request.

Most of the clusters only have specific ports opened to haproxy. Once the request is dispatched to cluster, the cluster determines its processing methodology. For haproxy the request is synchronous, even though cluster may process it asynchronously.

The infrastructure setup is continuously monitored using a combination of strategies:

- Server monitoring is done using Vistara agents and AWS health check services
- App level monitoring using scripts extracting log data and verifying HTTP response codes. Failures are flagged to team for review and action.
- API response monitoring passively by extracting haproxy logs and raise alerts if time taken for API exceeds thresholds
- 24 x 7 infra monitoring of the system by dedicated team

For more detail on deploy, refer to the Navigator Deployment document.