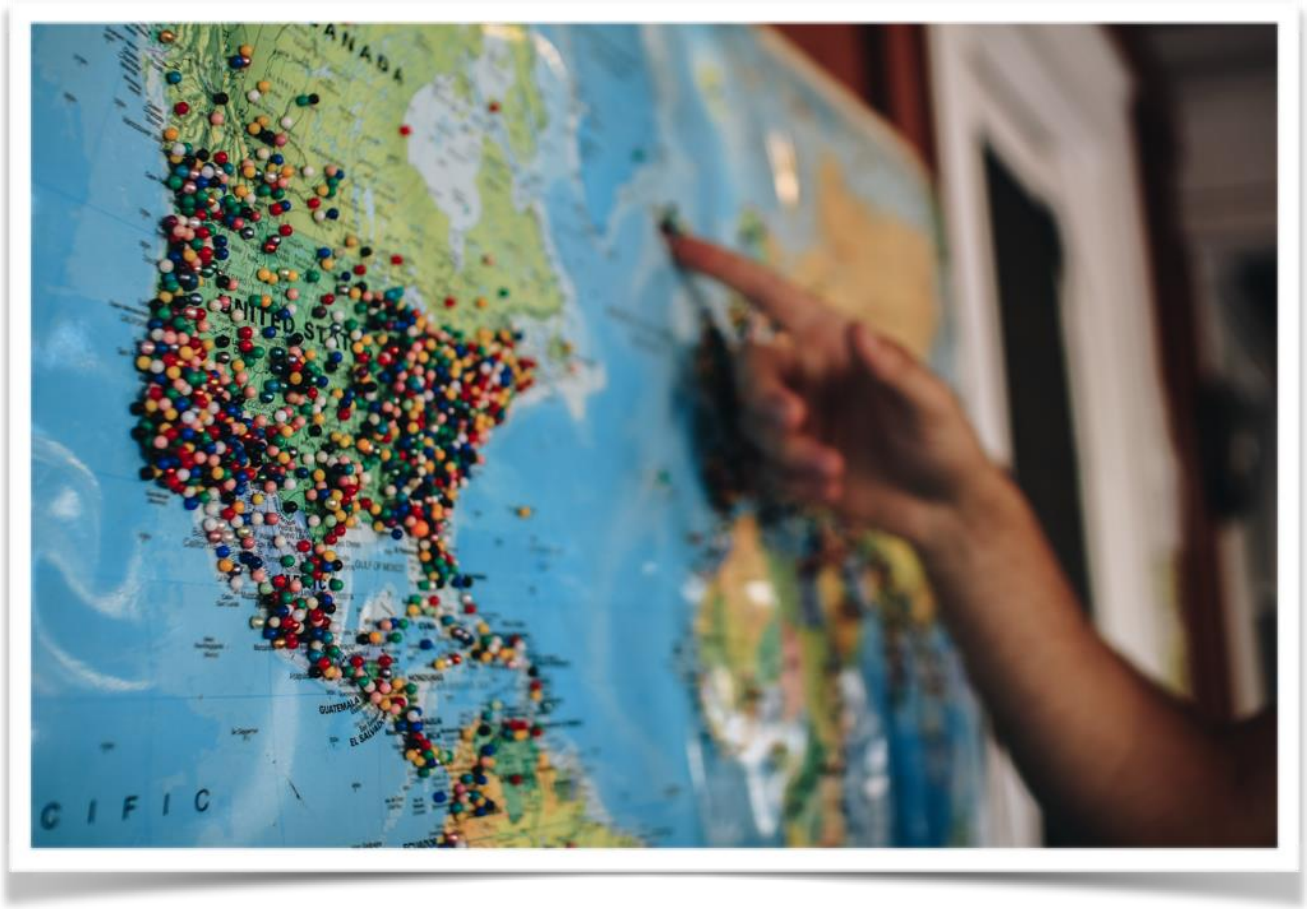
TruthTree



Developer Documentation

Contents

1. Preface

1.1 Intended Audience

This document is meant for the readers with basic web development background. Although we have used latest technologies for every component, we have tried to cover basics so a novice reader can also start with the development.

1.2 Related Documentation

Before going ahead with this document, it is good to go through our release notes to see what Truthtree has to offer. We also advise to have a quick look on our User guide to understand use cases better.

2. Objective and Scope

2.1 Objective:

The purpose of this documentation is to cover overall technical architecture of Truthtree application. Not only we explain the existing development of Truthtree release 1, through this document but also, we aim to educate developers to go ahead with further development of this product.

2.2 Structure:

- Internally, Truthtree is divided into 3 major layers – Data Layer, Service Layer, Client Layer. We have briefly covered these 3 layers in our document.
- **Working with Data Layer:**
  - How we used US Census data provided in excel files to process and store in a refined, optimized database format
  - How we analyzed information and leveraged machine learning algorithm to find groups of locations sharing same progress over the years
- **Working with Service Layer:** We explain how internally we create standard REST APIs for different sources at Data Layer and how to add new APIs in future
- **Working with Client Layer:** This explains our Front-end architecture and development in React-Redux framework

3. Architecture of Truthtree

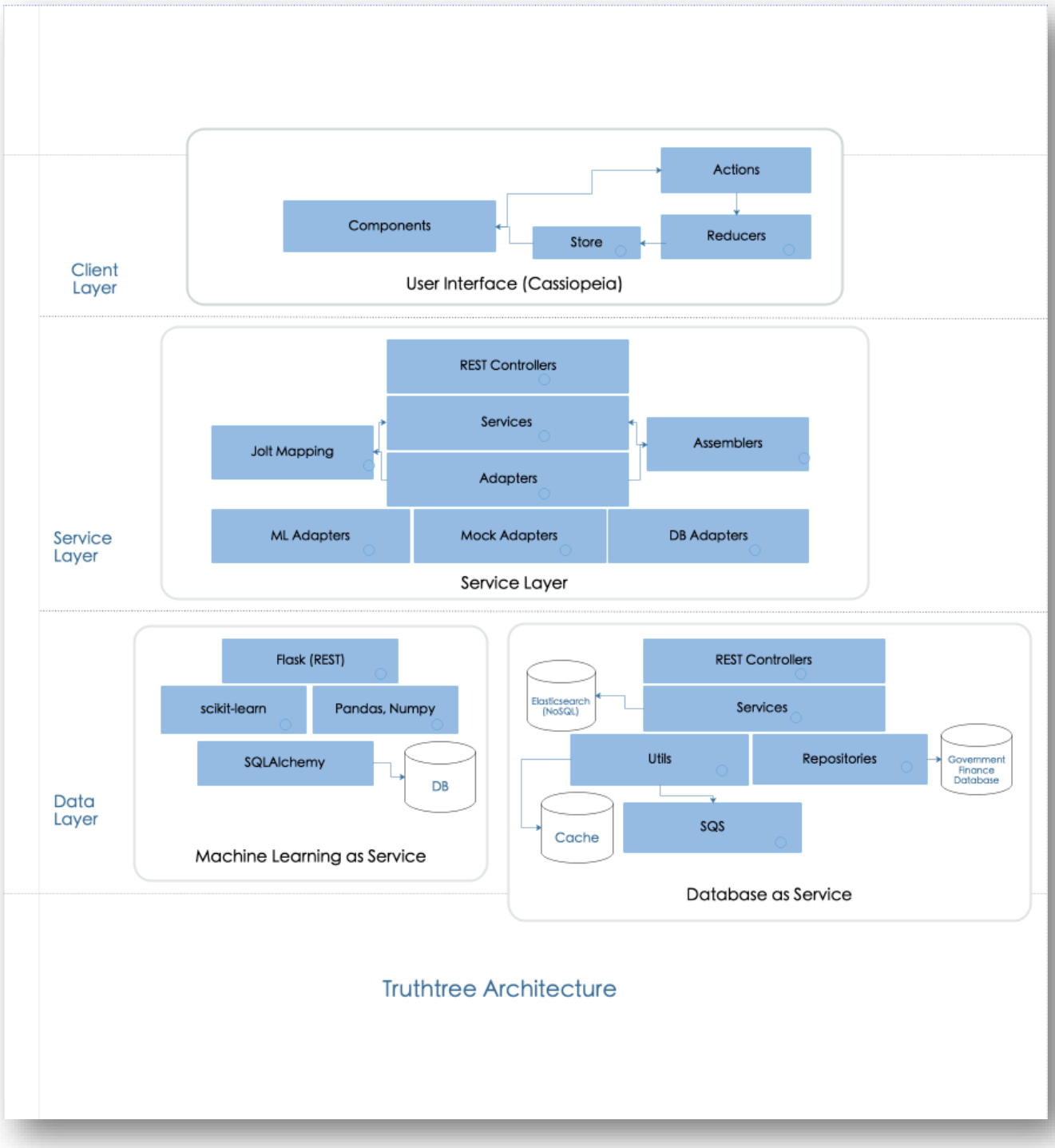


Fig 3.1 Truthtree Architecture

Truthtree follows a pretty straightforward, robust architecture which helps us to support extensibility in the application. As mentioned before, we have three major components – Data Layer, Service Layer, Client Layer. Let's see what each layer does in a Top-down fashion (Refer fig. 3.1)

Client Layer: Entitled for generating HTTP request from Truthtree web pages in React to the service layer and displaying the response in a creative manner.

Service Layer: Responsible for delegating request taken from client layer to the data layer and giving back a consolidated response generated from Data layer to the client. This layer also contains business logic of the application.

Data Layer: Responsible for handling any format of data. We divide data layer into two major components – Database as service and Machine Learning as service. Detailed explanation for each of them has been provided in the next part of this document.

4. Data Layer

4.1 Let's talk about the data

Before starting with anything else, let's see what we information we had in csv files and how we restructured the entire data for easier and faster access.

- We were provided excel files of size close to 3 GB which contained location data for 50 states, 3061 counties, and 20,050 cities in the United States for about 579 attributes over the period of 50 years (1967 – 2016).
- We spent just few days in analyzing the database design which could serve an easy fetching from enormous data and at the same time be robust enough to scale whenever required.

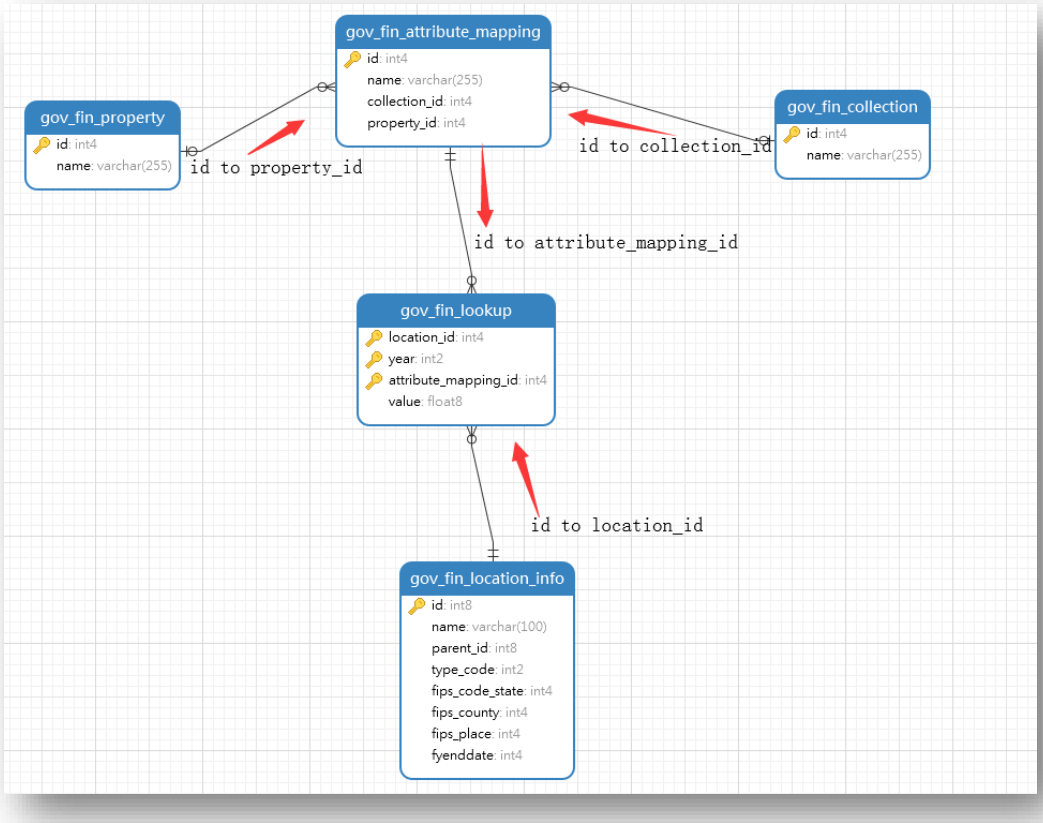


Figure 4.1 Truthtree Database schema

- For database storage, we divided identified Collection and Properties for each row in the excel column to come up with a term called 'attribute'. For instance, for any location, if we have Elementary Education as collection and its property as

Total Expenditure combining them would give us the attribute - Elementary Education Total Expenditure.

- We organized them into different tables – **gov-fin-collection** to store collections, **gov-fin-property** to store properties and consequently mapping between the two in **gov-fin-attribute-mapping** as attributes
- We also created location table (gov-fin-location-info) which stores location information such as name and type
- Now each of this attribute has some value over a period of 50 years. We have a generic lookup table **gov_fin_lookup** which stores mapping of these location, attribute and its value for each year. This table has over 50k records.

We used these two storage technologies:

- **PostgreSQL database** to store prepopulated Government finance data taken from http://willamette.edu/mba/research-impact/public-datasets/ and is being stored in a specific format where data is migrated (Section 4.4(g) for data overview and 4.4 (h) for data migration) from a csv sheet to PostgreSQL tables (Figure (b) for the design of database).

- **Elasticsearch database** which stores the "Stories" populated by Users. (Section 4.1.(b) for Elasticsearch db mapping). A Story is a blog post which any user can create to share images, links, and facts that they have found through TruthTree. This interactive post can be searched based on content, tags, title, author. There are features like upvoting, downvoting a story. Each story once uploaded has to be approved by an administrator in the backend. These features are supported by Elasticsearch for all stories.

4.2 Architecture of Database as Service

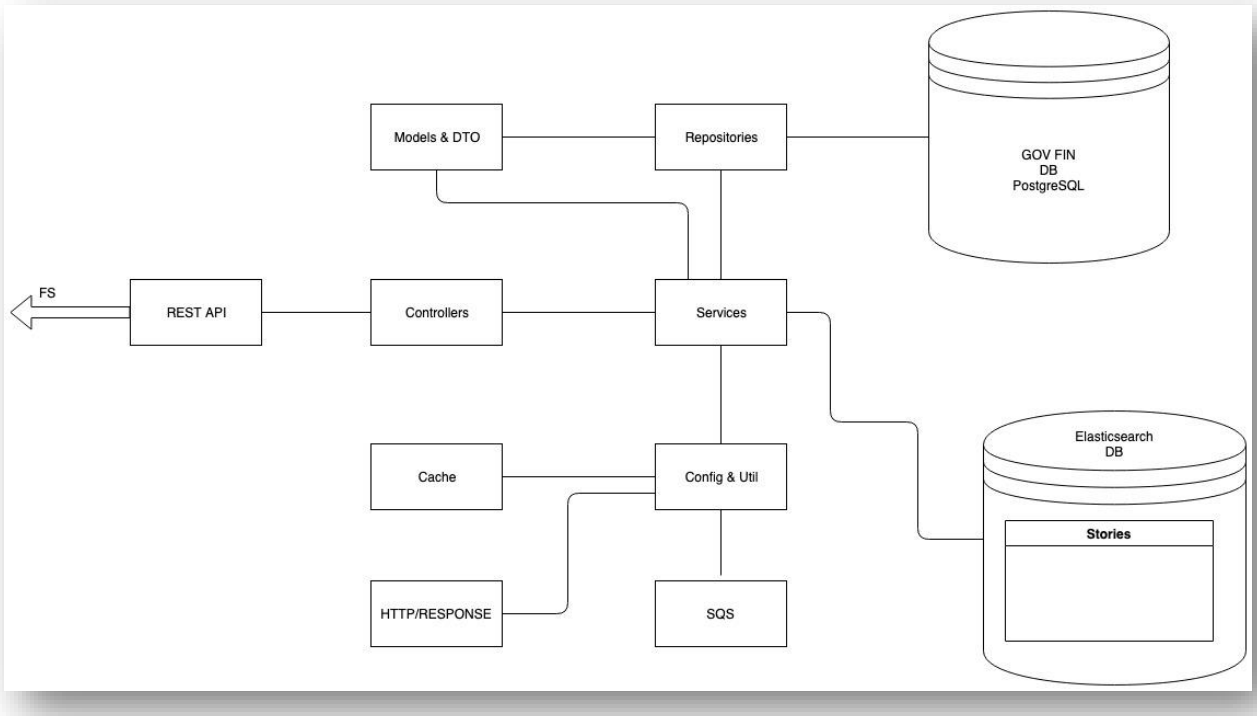


Figure 4.2 Architecture of Database as Service

The data from these two data sources are being accessed by repositories and services via a Spring Boot application. Section 4.2 gives a quick look at the technologies being used. These repositories and services use specific configurations and utilities such as Cache to store page information for specific query conditions and indexing over columns in tables in PostgreSQL database (Please refer section 4.4 for more details on configurations and specific utilities and their implementations)

To make the data organized as per the data sources, we have data models of each object, as well as their data transfer objects (DTOs) to map database models. We have specific controllers such as StoryController which enables us to do operations for Story object only. Similarly, we have multiple controllers such LocationController, AttributeController etc., which are specified controllers for each object needed. These controllers provide RESTful APIs to the Truthtree Service Layer.

We have created Response object and Page object to unify the response format. And we use Advice Controller to catch runtime exception globally. So, whoever calls our APIs will know whether exceptions occurred or there is no data qualified. We use filter

to log the request information (time, IP parameters, etc.) So, we can locate issues and bugs quickly.

The list of RESTful APIs is provided in Section 4.3. Section 4.3 also has a SWAGGER link for these APIs which enables you to have a better look at these APIs and can be tried out as well. Swagger link provides all the endpoints for all APIs as per each controller and also enables to test parameters and result formats.

For future developments, the process is straightforward as we are following Spring boot application development for providing RESTFUL APIs. (Section 4.5 for future developments)

- **Database:**
  GOV FIN DB PostgreSQL:
- **Elasticsearch Mapping:**
  https://search-gov-fin-es-3y3ydkxijtitmqim7xxyhtczeq.us-west-1.es.amazonaws.com/stories

4.3 Technology used

a. Framework: Spring Boot and Hibernate
b. Database: PostgreSQL(HikariCP) and Elasticsearch(jest)
c. Caching: EHcache
d. Testing:
   i. Unit test: junit
   ii. Performance test: jmeter
e. API documentation: Swagger
f. Others:
   i. Objects deep copy: dozer
   ii. Serialization: Gson

4.4 REST APIs and purpose

- **API for Location:**
  i. to return all locations
  ii. to return all states
  iii. to return all cities
  iv. to return all counties
  v. to return locations by type code, parent location id or location id

- **API for Collections:**
  i. to return all collections

ii. to return specific collections
iii. to return collections based on time range for given location and for specific attributes.
iv. to return all collections based on specific attributes by specific collection and property pairs

- **APIs for Properties:**
  i. to return all properties
  ii. to return property by id

- **APIs for Attributes:**
  i. to return all attributes
  ii. to return attributes with values for specified attributes
  iii. to return all attributes with values for a specific collection or list of collections
  iv. to return all attributes with values for a specific collection/collections and specific property/properties
  v. to return all attributes with values for a specific collection/collections and specific property/properties for specific states
  vi. to return attributes with values for specified attributes within a year range
  vii. to return attributes with values for specified attributes for a specific year list

- **APIs for Stories:**
  i. CRUD operations over a story
  ii. to return all stories
  iii. to upvote/downvote/approve a story
  iv. to fuzzy search a story
  v. to sort stories as per timestamp, upvotes, downvotes, freq

Note: All APIs can tried be here:

http://54.153.74.217:8080/swagger-ui.html

4.5 Implementation details

a) Data Overview:

The data has basic information about the location along with 579 attributes. These attributes have been divided into 85 collections and 186 properties. For example, attribute 'State_IGR_Gen_Sup' has collection 'Intergovernmental Revenue State' and property 'Gen Sup'.

- **Adding a new location**:
  Any new location needs to be added to gov_fin_location_info table. Id should be the same as provided in the csv. Type Code is 0 for state, 1 for county and 2 for city.
- **Adding a new attribute:**
  Identify the category and the property of the attribute according to the appendix provided with csv. If the category and property do not already exist, they need to be added to gov_fin_collection & gov_fin_property tables respectively. The new attribute mapping is to be added to gov_fin_attribute_mapping table and the collection_id and property_id can be checked from gov_fin_collection & gov_fin_property tables.

b) Data Migrations:

Two data transfer method

- **Import csv directly:** when the data set is large, we recommend use this method. Here is the guide:http://www.postgresqltutorial.com/import-csv-file-into-posgresql-table/
- **Using data transfer spring job:** a bit slower but easy to use. https://github.com/TruthTreeASD/data-transfer

c) Page information caching:

To improve performance, we have cached the page information for requests with same query conditions for five minutes.

For example, when user wants to query p1=a, p2=b, p3=3, our API will return a list of qualified data and additional page information, like Total number of qualified data, total page according to current page size. Ideally, we should not count the qualified records for same business parameters many times. So we implemented EHcache to store the Total number of qualified data for given business parameters and keep them for a period of time (default five minutes, configurable at ehcache.xml). Total elements in memory (default 500) and memory store eviction policy (default LRU) are also configurable at ehcache.xml. Please carefully change the arguments respect to current environment.

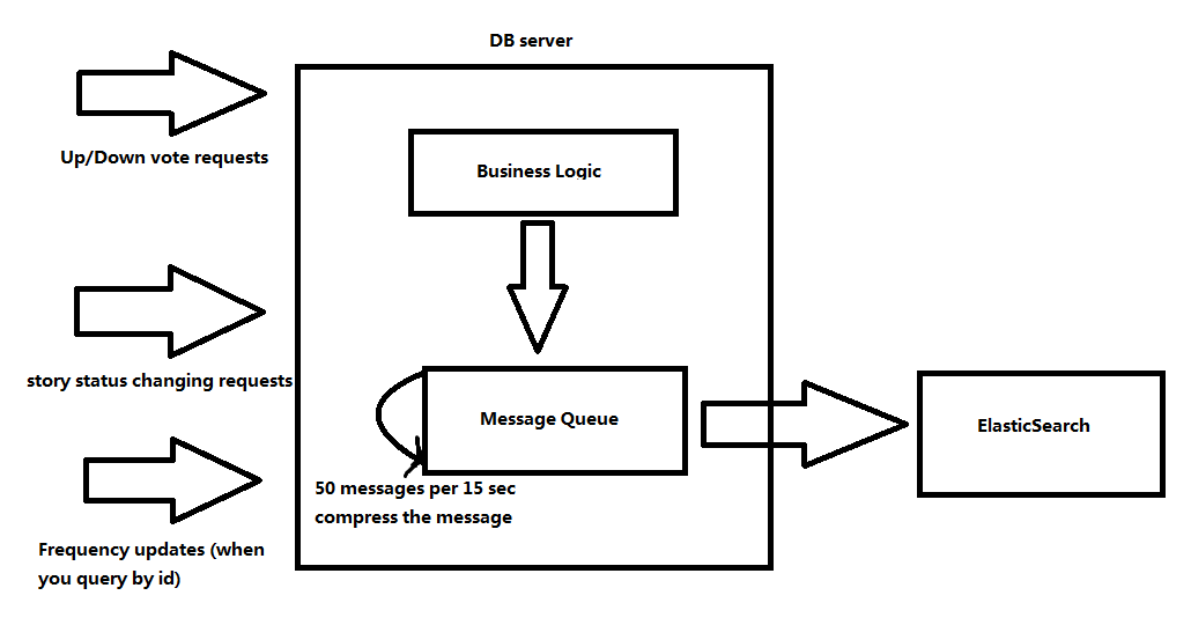
d) Indexing:

Indexing on tables are trivial except gov_fin_lookup, which contains a 500 million lines of data.

i. Indexed on location_id
ii. Indexed on attribute_mapping_id and location_id
iii. Indexed on attribute_mapping_id and year
iv. Indexed on val
v. Indexed on year and location_id

e) Message Queue:

We implemented message queue to solve the consistency issue in Elasticsearch (updating freq/upvote/downvote values.) When we received the request of updating the fields we mentioned above, the request will be inserted into message queue and program will extract 50 messages from the queue for every 15 sec. Then we will compress the messages with same story id to reduce the time waste on connecting Elasticsearch endpoint and waiting time.



f) Implementation:

i. Indexing
ii. Cache
iii. Materialized view: please refresh materialized view after inserting new data

4.6 Future development

In case you want to add a new feature, simple steps to do that would be -

- **Adding APIs:**

i. Follow Spring boot development guide to add new APIs. The flow of code can be understood from the architecture diagram.

ii. Please use response object to wrap your result!

- **Adding Data to SQL DB:**

i. Please refer to Data Migration which provides two methods to add data to PostgreSQL DB.

5. Machine Learning as a Service

5.1 Architectural Overview

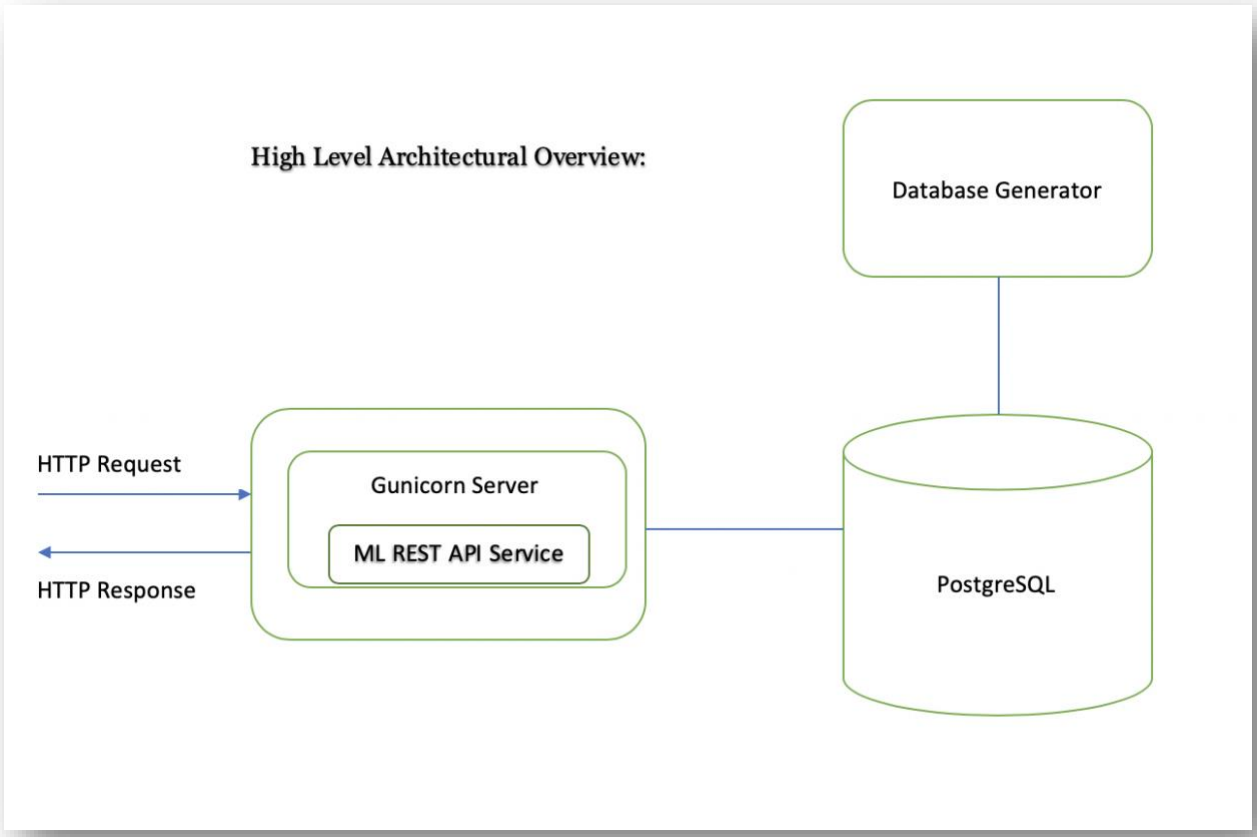


Figure 5.1 Architecture of Machine Learning as Service

- **Database Generator:**

i. Used to populate the database with the required tables.
ii. To populate the data run 'python3.6 app/db.py -p <path-to-csv>'

NOTE: The script db.py will create a table for each file(.csv) present inside the path specified. If you want to populate the database with new data or you want new tables to be created in the database, copy the datafile as .csv to app/data and run the script again.

- **PostgreSQL:**

This database contains 4 tables mentioned below,

i. state - table containing the state data
ii. county - table containing the county data
iii. city - table containing the city/municipality data

iv. attr_id_map - table containing mapping between attribute id and attribute name

- **ML REST API Service:**

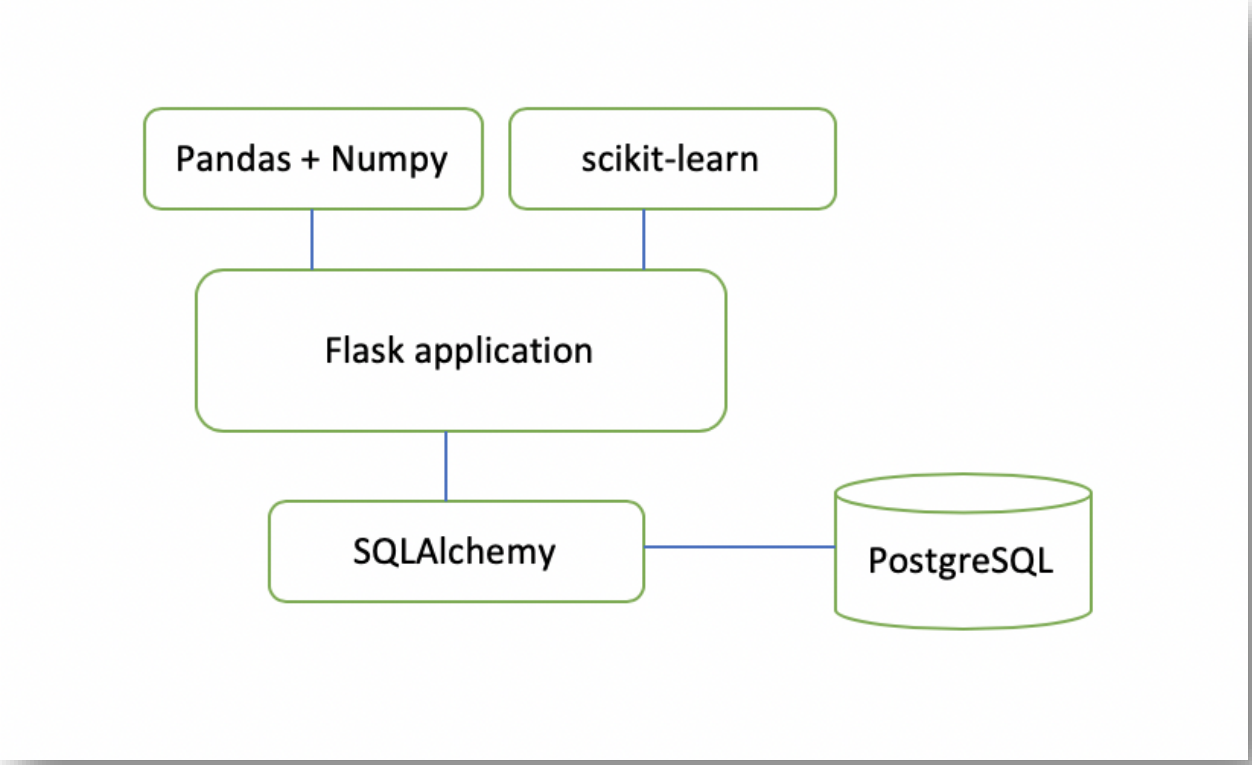


Figure 5.2 Machine Learning as Service – Technology stack

- **Technology used:**
  i. Flask - a micro web framework which allows building REST APIs for services written in Python
  ii. SQLAlchemy - Used to read data from databases such as PostgreSQL
  iii. Pandas + Numpy - Used for data handling and data reformatting
  iv. Scikit-learn – Library which provides different Machine learning classifiers to generate predictions used to find similar places using K-Nearest Neighbor algorithm
  v. Gunicorn - a python Web Server Gateway Interface HTTP server.

- **ML Database:**

We use PostgreSQL to store data of State, County and Cities. Data for each of these is stored in separate tables as for each request we need data from only one of these tables and also it's easier to load.

5.2 REST APIs

5.2.1 Supported Attributes:

- /api/similar/supported
  Returns a list of attribute IDs and corresponding display names supported for finding similar states, counties or cities.
- /api/similar/supported/{place_type}
  Returns a list of attribute IDs and corresponding display names for state/county/city depending on the value of place_type(0 - State, 1 - County, 2 - City).

5.2.2 Similar Places (Single Attribute and Multiple Years)

- /api/similar/single
  For a given place and attribute, returns a list of similar places having same pattern (similarities) over the range of years specified.
- We can compare the normalized value by providing the normalization attribute.
- Returns a list of Place IDs and Similarity Score for the closest 'n' places

5.2.3 Similar Places (Multiple Attributes and Single Year)

- /api/similar/multi
  Provides a list of similar places to a given place having closest values for all the given attributes in a particular year.
- We can compare the normalized values by providing the normalization attribute.
- Returns a list of Place IDs and Similarity Score for the closest 'n' places

5.3 Implementation details

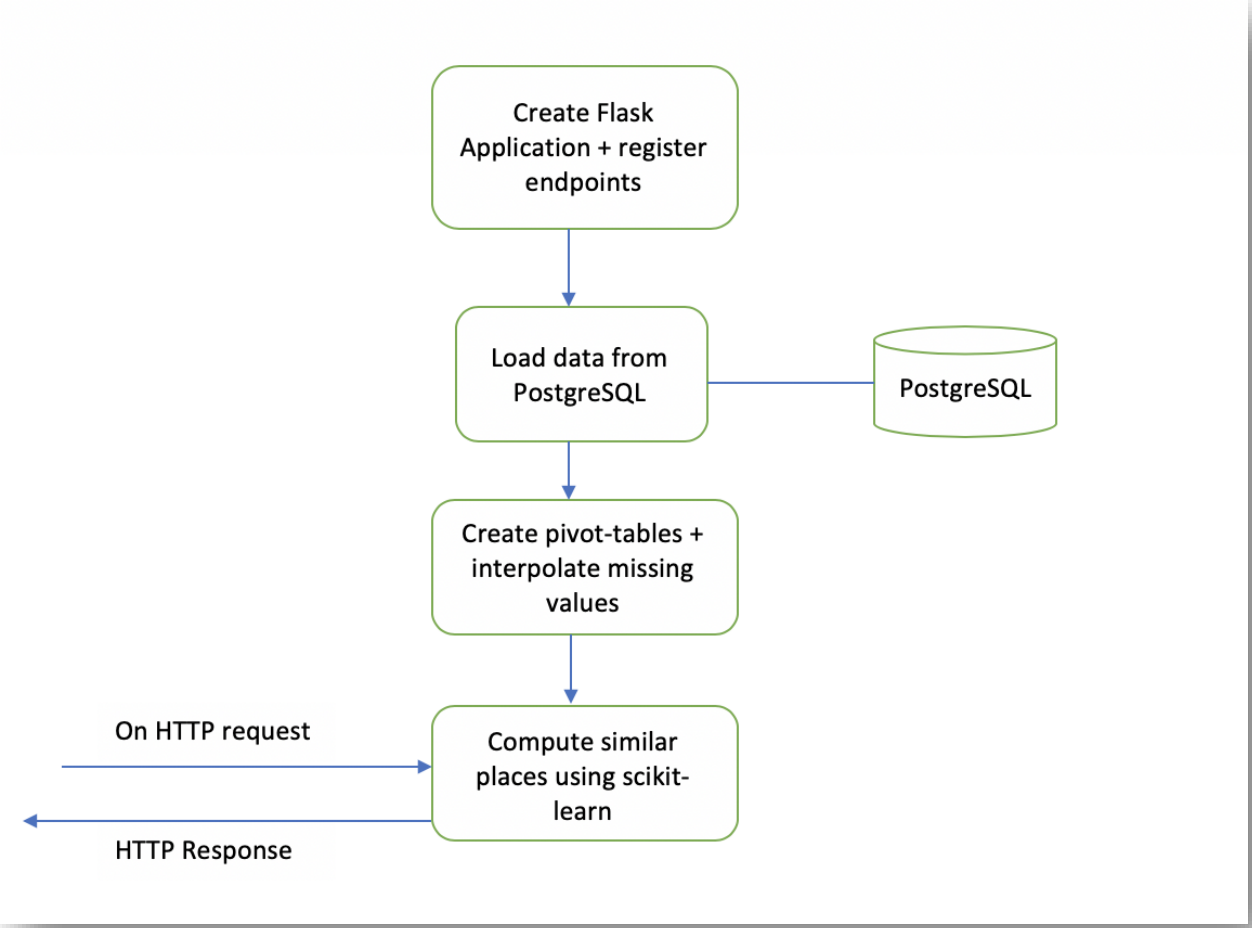


Figure 5.3 Sample workflow for finding similar places

- After loading the files, we first create a multi-level column pivot-table for each file.
- Then we interpolate the missing values along the years using linear interpolation provided by pandas.
- At runtime, we normalize the selected part of the data (if required) and then re-interpolate to avoid infinite or NaN values due to normalization.
- We apply KNN on this data and convert the distance into similarity scores.

Data Cleaning:

- Most of the data cleaning has been done manually either by visualizing the distribution of the data or with the help of 'The Government Finance Database.pdf'

- You can find our initial findings in the following link [These findings were on State data, but are applicable to County and Municipalities as well]

Finding similar locations:

- Data is processed using 'pandas' API
- We have used Sklearn's K-Nearest Neighbors algorithm (with k provided by user) to find the similar places
- KNN is relatively a suitable algorithm to form groups on the basis of features (in this context, attribute values)
- To form groups of similar locations, we have used Minkowski distance with p=1 (Manhattan)
- Results were confirmed by plotting time series graph between similar places.

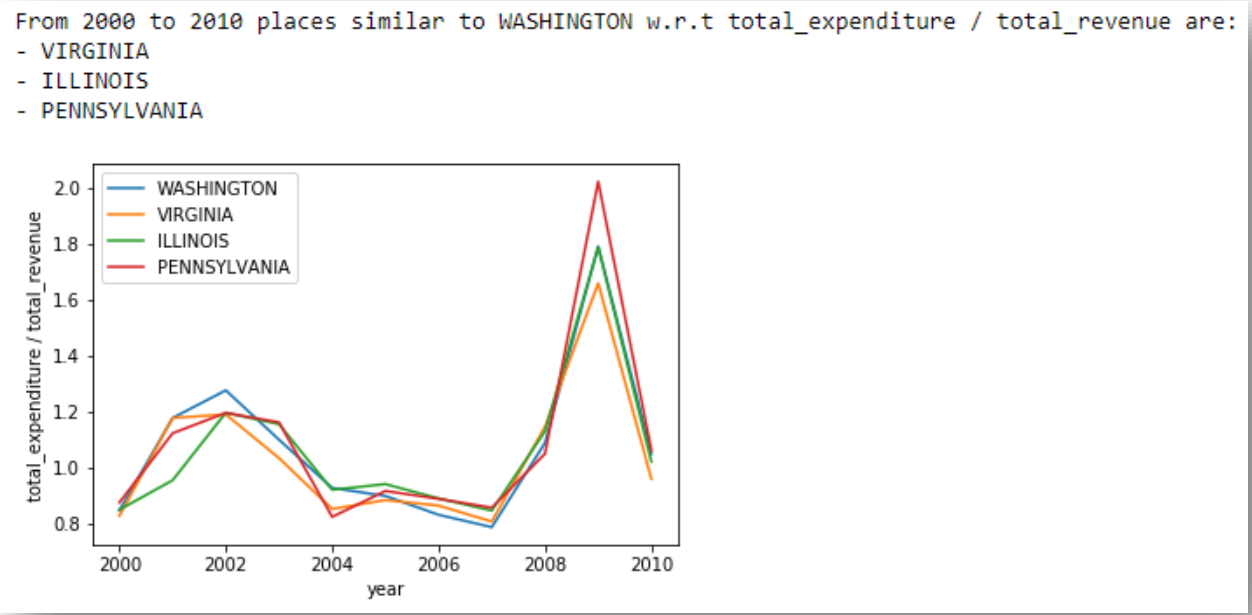


Figure 5.4 Plotting of Similar Places (Washington, Virginia, Illinois, Pennsylvania) having same path from 2000-2010

Optimizations

- We convert data into multi-level columns at the time of data loading to avoid loading the pivot-table for every request.
- Interpolating each attribute in all the three files took more than 20 minutes. We took benefit of the internal structure of pivot table and the working of Pandas linear interpolation, to interpolate an entire row (including all the attributes) at once. This reduced the total time to less than 5 minutes.

Future development

1. Optimizations
    A. Further optimization of the data loading part can be done by finding a way to save/load multi-level pivot tables directly.
    B. Another way is to convert the data back into original format after interpolating and save interpolated values. (This way we can skip the interpolation part at data loading step)
2. Features
    A. We can create cluster of places and show clusters with least inter-cluster variance.
    B. Apply hierarchical clustering on the attributes to find the relation between different attributes.

6. Service Layer

6.1 Architectural Overview

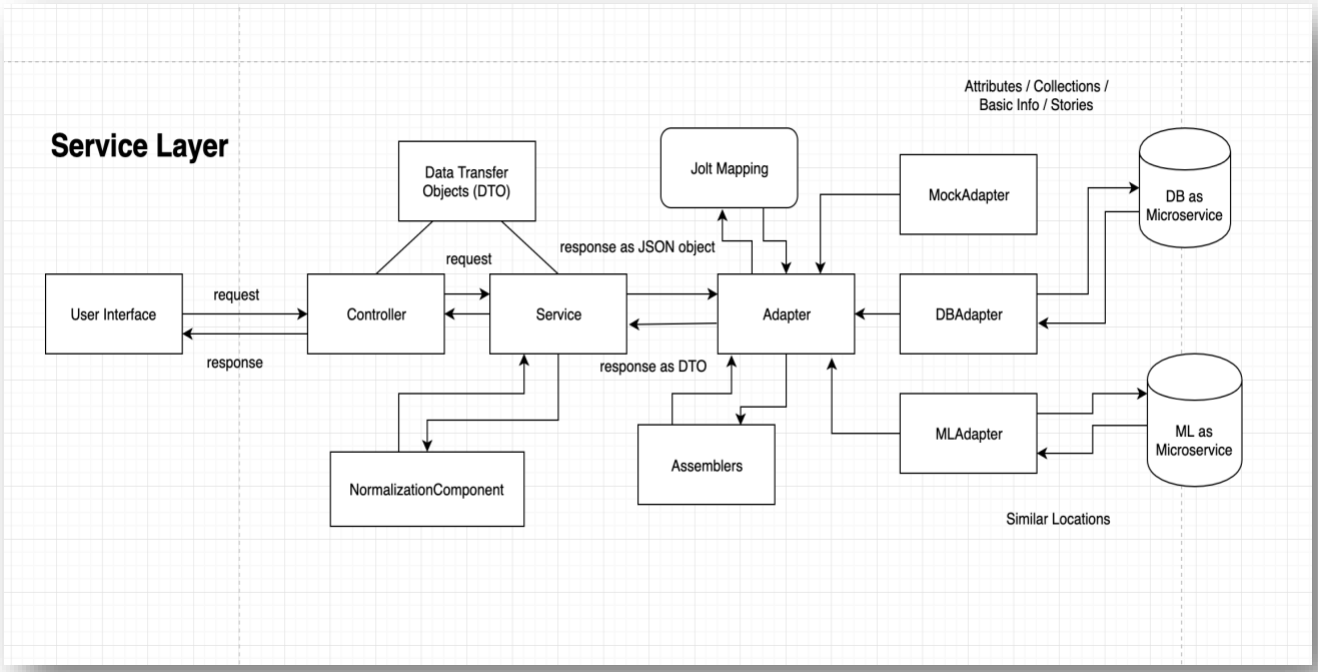


Figure 6.1 Architecture of Service Layer

Service layer provides REST endpoints to the client layer which can request and gather response for any sort of information. Internally, Service layer is a sophisticated 3 layered framework which can be explained as follows:

Controller: This is essentially Spring Boot REST controller which accepts HTTP request and further passes any information to the next component.

Service: This component controls flow and business logic of the application by checking if a valid admin user is requesting sensitive story related information. It also applies Per capita or Revenue Normalization logic to the attribute values.

Adapter: This component talks to the external endpoints which act as datasource. Depending upon the type of datasource, different types of adapters have been created which may either talk to ML or DB services.

6.2 Technology used

- Spring Boot Framework in Java
- Apache Maven for dependency management
- Jolt Mapping for converting one JSON format to other
- Jackson Mapper for converting JSON to Java Objects and vice versa
- Mockito for Junit testing
- TestNG for Integration testing

6.3 Sample workflow

Story Creation By User

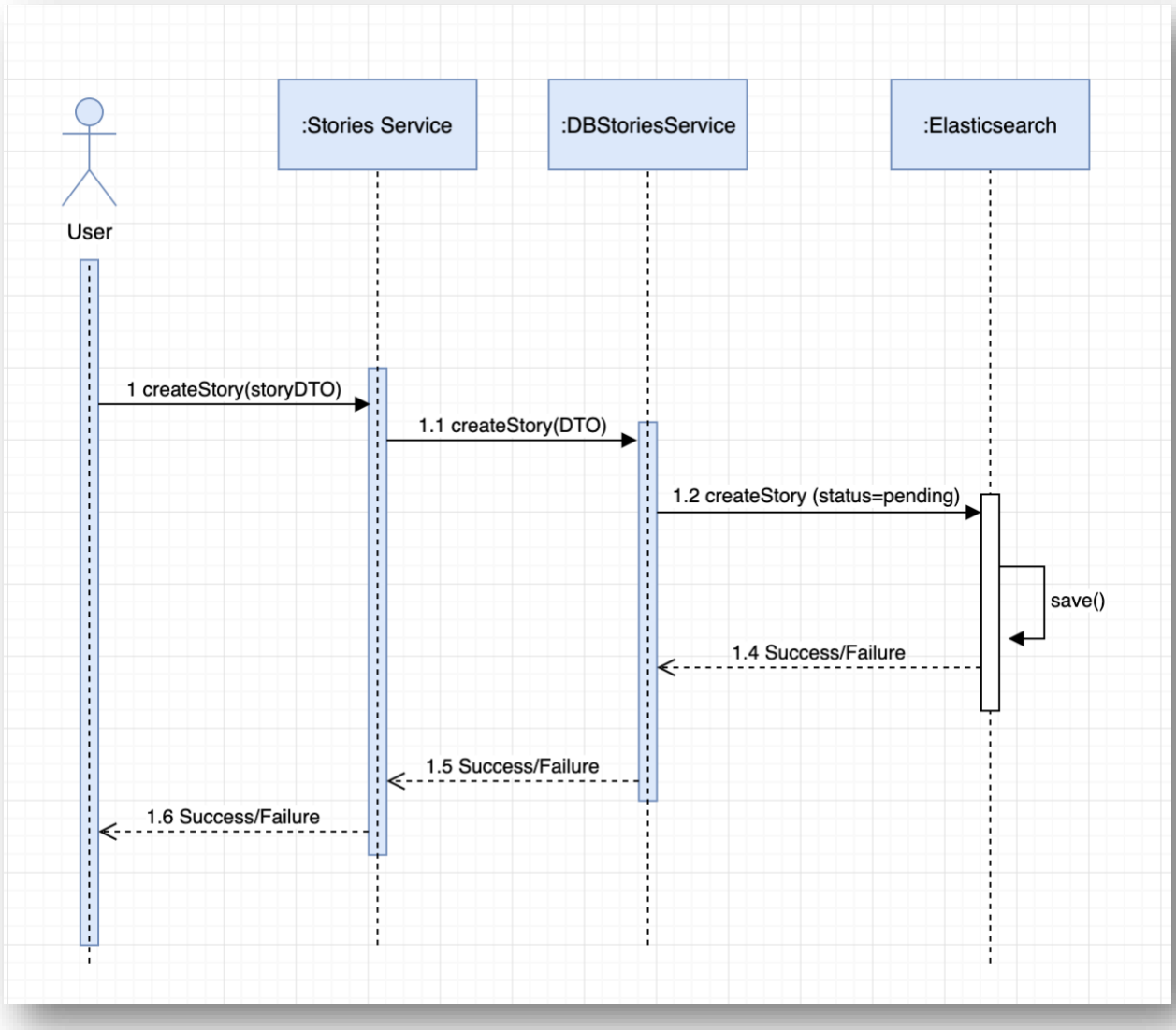


Figure 6.2 Sequence diagram for Story creation by user

- User submits a story which first contacts Stories Service at Service Layer. Since at this moment, submitted story requires admin moderation, it is further passed to Stories Service at Data Layer which saves the story as PENDING status
- This story is saved in Elasticsearch database and user gets notified that story is pending for approval

Story approval by Admin

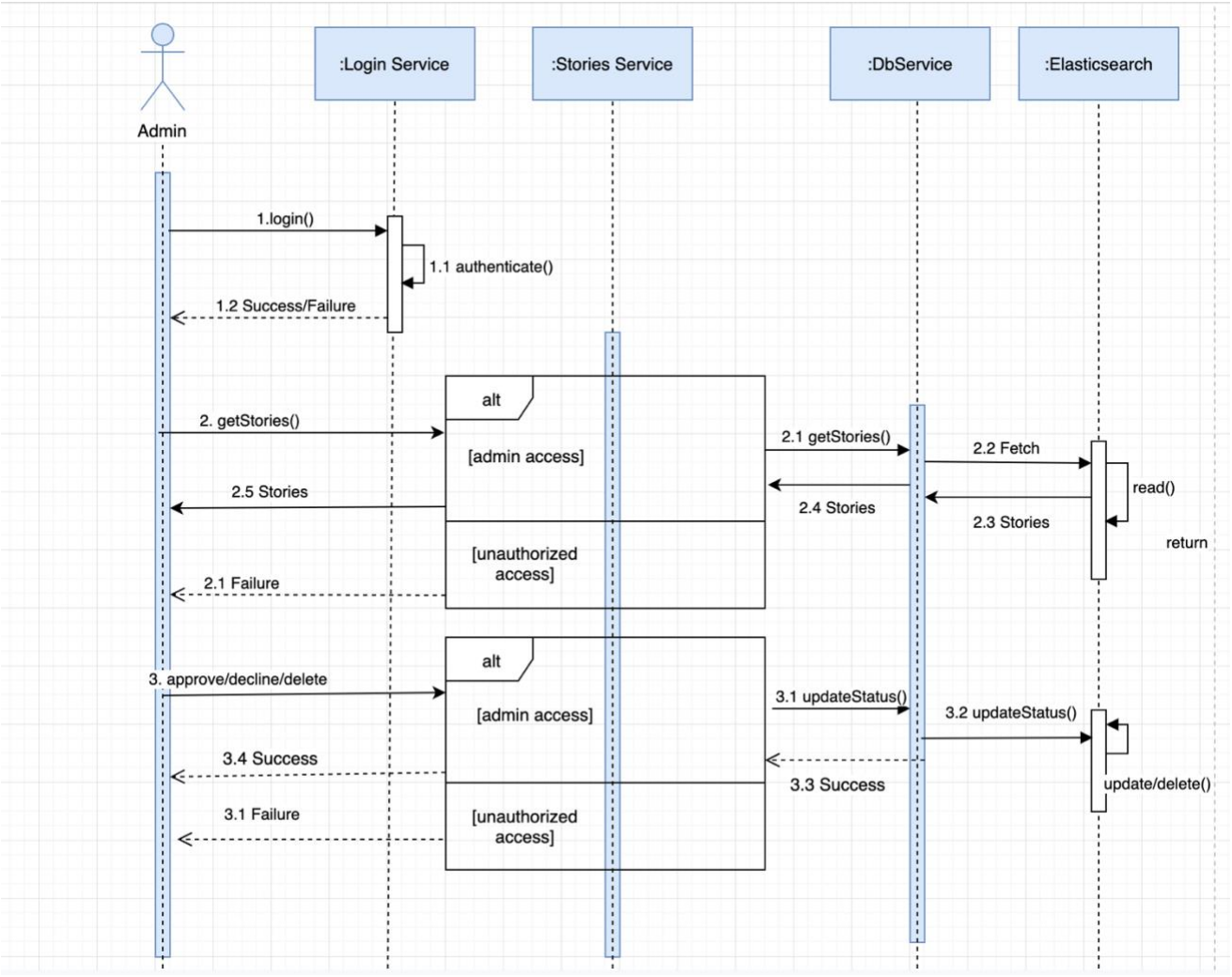


Figure 6.3 Sequence diagram for story approval by admin

- When admin needs to approve a story, he/she will have to login, which happens in Login Service at Service Layer
- Once admin is authenticated, he/she can request to see all the stories including stories yet to be approved with PENDING status
- Since HTTP requests are stateless, for each request, we bind session which contains user information. We leverage this fact and check in the subsequent call

to Service Layer to see if the user logged in has "admin" access or not. If that is true, we further send this request to Data Layer, otherwise throw an authorization exception

- Once admin gets all the stories in the previous calls, he/she can either approve, decline or delete that story. For any of these actions, the request will still be checked for admin access at service layer, since we do not want any other user, without proper admin access to approve this story
- If this access check is successful, we further send update request to Data Layer which updates the status of the story depending upon the request type. If admin wants to decline a story, we do soft delete which means we necessarily store the unapproved stories for future analysis. If admin wants to delete, we do not see it useful and remove this story permanently from our database.

6.4 REST APIs

We have categorized our REST APIs into following resources:

Collections: Represents different categories of financial data we hold for each location. For instance, Public Parks, Libraries or Highways. Our collections endpoint retrieves all the collections we have in Truthtree database.

Attributes: For each of the collection, we define property such as "Total Expenditure" or "Total Revenue". We combine the two make attribute such as "Public Parks Total Expenditure". In Attributes endpoint, we fetch attributes values for different query type such as collection, year, name etc

Basic Information: Retrieves cities, countries and state information

Stories: Performs creation, retrieval, deletion and update (approval) of stories.

Similar Places: Finds similar places corresponding to the single or multiple attributes over the year.

Login: Performs login and authentication for admin

6.5 Implementation Details

How to add a new REST API call

- **Adding REST controllers:** You will need to add a new Spring Boot REST controller implementation class along with its interface which gives the capability of accepting REST calls by default. All controller classes can be found in `edu.northeastern.truthtree.controller` package.
- **Adding Services:** They are created as Spring Boot component in `edu.northeastern.truthtree.service` package. If you are adding a new service class, make sure to add an interface for it. This ensures that we can include several different types of implementation which abide by the specification mentioned in the interface.
- **Adapter Implementation:** As mentioned before, there are 3 different types of implementation currently. If you want to make a call to Data Layer APIs - either DB or Machine Learning, you can add a new call in <feature>DBAdapter or <feature>MLAdapter. In adapter, you can use URLUtil class which has boilerplate code

for GET, PUT, POST, DELETE calls. This class is present in `edu.northeastern.truthtree.adapter.utilities` package.

- **Assembling Request/Response format to POJO:** Once you get a response from URLUtil class, you can simply use Object Mapper (provided by Jackson library) to map response JSON to the java data transfer object.

CORS configuration

- CORS stands for Cross-Origins Request Sharing. Following this configuration provides a specification to allow any request that comes from different client origins, which otherwise is restricted at REST controller level.
- We can configure CORS with the Spring Boot annotation `@CrossOrigin(origins = "*", maxAge = 3600, allowCredentials = "true")` by adding it at the start of REST Controller.

  Currently, we are allowing client request from all the origins by adding "*" as value. If you want to configure specific clients, you can add them as comma separated values. For instance you want to restrict clients to only 2 domains - https://abc.com and https://xyz.com, you will add something like this - `@CrossOrigin(origins = "https://abc.com, https://xyz.com", maxAge = 3600, allowCredentials = "true")`

Admin login and Session Handling

- For Admin Login service, we are accepting password as string in the request (It is advisable to use Https request calls only since the layer would be secured and encrypted)
- Once Login service receives a password, it checks whether the provided password matches with the correct. If that is true, we set an admin session parameter in HttpSession's attributes HashMap.
- Since all HTTP REST calls are stateless in nature (server doesn't know the client, just serves what is request), all HTTP calls which needs to have session/login as pre-requisite have to explicitly handle the client information in session object. At REST controller, we accept this Session Object to see if it contains the admin credentials in session attributes HashMap. If that is true, we allow further processing of request, otherwise we send back as Authentication failure.

Adding a new Normalization parameter

- If in future you want to add a new Normalization parameter (Currently, we support 3 – GROSS, PER-CAPITA, and BY REVENUE, you can simply add a new Enumeration parameter in NormalizationType Enum present in edu.northeastern.truthtree.enums

- We are following Strategy pattern to implement Normalization in Truthtree which helps us to easily scale without touching the base code for normalization. Normalization is essentially achieved by dividing a Gross/real value by the parameter value. For instance, dividing by population value for PER CAPITA normalization. If you want to add a new parameter, you can simply add a new Strategy class with the name <Type>NormalizationStrategy which extends AbstractNormalizationStrategy present in edu.northeastern.truthtree.common.normalization Package. You need to implement two

methods – setParameter() which essentially sets what is the normalization parameter value and apply() which takes gross response and applies normalization. Since both of them are abstracted, you do not have to write the logic everytime!

7. Front End - Cassiopeia

7.1 Architectural Overview

Cassiopeia is a React/Redux application with various 3rd-party libraries to help enhance user experience. It is bootstrapped with create-react-app to avoid initial complication with webpack. It also comes with development server with proxy capability so that developers can change which api to point to.

7.2 Technology Used

React: For easy and quick development of front-end components

Redux: For state management. React provides one-way data-binding (associating data fetched from server to the html components)

Node: We use node.js as our server in development background to compile React JSX code to Javascript.

SCSS: Stands for Sassy CSS. We use a novel way of managing styling of our application which has better way of managing CSS properties through inheritance

Bootstrap: For quick development of UI components with styling

7.3 Environment setup

Required: node 10+, npm 6+ (Might work with lower versions, but it's not guaranteed)

1. Clone this repo: https://github.com/TruthTreeASD/cassiopeia.git
2. Install dependencies with npm install
3. Run the project using npm start, this would automatically open a browser tab with localhost:3000
4. As you make changes to the code, the browser refreshes itself to reflect new changes

7.4 Overview of existing components

UI components are located under the `components` folder. We try to organize them on page-basis. For example, the Explore folder is the home page, so everything shows on home page will live in Explore folder. Some notable components are:

Header: main navigation menu which logo on the left and links to all the pages on the right.

SideMenu: this component slide in/out from the right, it currently houses StoryCreationForm so that user can create a story from anywhere in the app.

Explore (aka home page): this is the first page user see when visiting truthtree.wiki, it contains the map and search box where user can search for a location.

LocationSearchBox: for location searching. It equipped with typeahead feature powered by our backend ElasticSearch service.

Map: the map displayed on the front page, fully interactive with drag and zoom control. As user type in the search box, pins are dropped according to the suggestions list.

LeftSideBar: display a list of selectable attributes for a particular location.

AttributeRange: used for adjusting population range.

AttributeDeselector: display selected attributes, with the ability to click to remove.

DisplayComponent (could be renamed later): show the table with data for each locations and attributes

TimeSeriesGrid: chart view for the data, this show multiple charts, with each has the ability to expand to larger single char view

TimeSeriesChart: larger single chart view for 1 attribute

Stories (aka stories page): display a list of approved stories, user can click on each story to have them loaded on the right.

TrendingStories: list of approved story

TrendingStoryDetail: this component is updated when user choose different stories in the TrendingStories component

StoriesCreationForm: this form is used to compose new stories, has captcha check and field validation. HTML editor is powered by react-quill library.

Advance (aka advanced search page): this component contains advanced search form and a map that display closest matched locations based on calls to our machine learning service.

SimilarPlacesResponse: the map and list of similar locations

SimilarPlacesSearch: the advanced search form

Admin (aka admin page): this page allows admin to log in and approved submitted stories before they can show up in stories page.

PendingApprovalStories: display a list of unapproved stories, after approval these will show up in the stories page.

StoriesDetails: similar to TrendingStoryDetail, it displays the content of a story within the admin view, we should probably consolidate these two components into one in the future.

About: information about TruthTree project.

7.5 Simple Steps to add a new feature/component

- **Components**: place new component under the respective page it belongs to, inside the components folder with a sensible name. Each component can have both local state and redux state. If component is a new page, with its own endpoint, don't forget to update App.js as well
- **Reducers:** new reducer should be placed inside the reducer folder, following the naming convention of previous reducer.
- **Actions:** should be placed inside actions folder, with name closely matched its reducer counterpart
- **Styles:** component can have its style in a separate css or inline styles. In addition, cassiopeia supports scss, just rename css file with scss extension and everything should just work.