

Final Product Delivery

Project Group 8: Yao-Fu Yang, Qiushi Li, and Rong Chang

Trello board: <https://trello.com/b/61ULBOVX/grocery-store-data-support>

Git directory: [cs5500_sum2020_group8/GrocerySprint3](https://github.com/cs5500_sum2020_group8/GrocerySprint3)

Features of Product

1. Presentation of Shopping Pattern

- a. The basic features of this product allow the user to observe (1) the customer visiting distribution over a day and over a week, and (2) the length of shopping time at different times on different days.
- b. Data was simulated in the consideration of all possible events that may affect change of grocery shopping (i.e., holiday, discount hours, lunch/dinner rush hours, and very bad/nice weather).
 - i. All these considerations were defined in the prefix of visiting ID.
 - ii. The users are able to identify how the special events and conditions affect the shopping traffic by modifying/removing the events per analysis needs.
- c. The users could modify/shift the event time or remove the event from a day and see the changes the event brings to the shopping traffic.
- d. Per request, this product currently contains two sets of data from May and June. The data is generated as a csv file for further analysis, and also stored in MongoDB for further modification and accessibility.


2. Accessibility of Database


- a. In addition to a csv file, the data is stored in MongoDB for further accessibility.
- b. The database could be accessed by GET queries with filtering and ordering options.
- c. The data could also be inserted or updated by POST queries.
- d. The backend's service uses Redis to store the date-to-holiday mapping.


3. Frontend UI

- a. The frontend UI allows both insert and present data more friendly.
- b. The form from <http://localhost:3000/new> allows the users to enter a new visiting data with two parameters - *holiday* and *day of week* automatically identified after the *visit date* is filled (see Figure 1).
- c. Data is presented at <http://localhost:3000/show>.
 - i. Users are able to browse all visit data stored.
 - ii. The page contains filtering and ordering options (see Figure 3)
 - iii. Users are able to visualize the distribution diagram of the selected visiting data.

Visit ID:

Visit Date:
 

Entry Time:
 

Leave Time:
 

holiday

dayOfWeek

Figure 1. Adding Single Data.

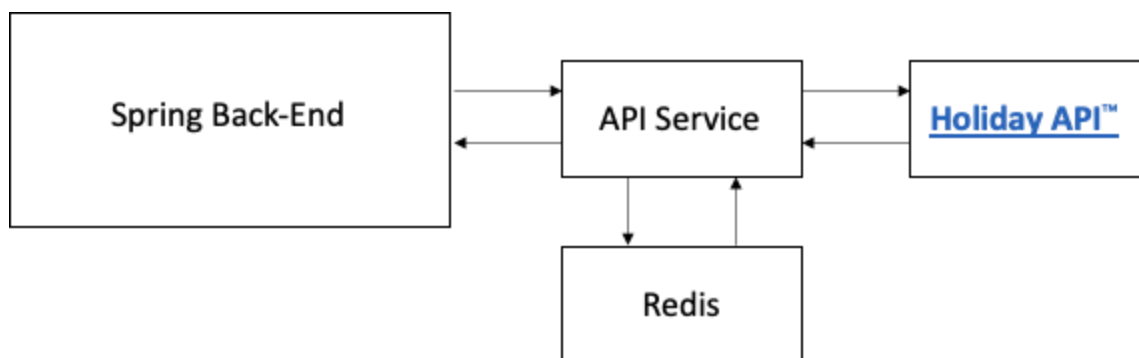


Figure 2. API Service

All Visits

Unsorted ▾	min duration	max duration	05/09/2020 📅	mm/dd/yyyy 📅	Apply
------------	--------------	--------------	--------------	--------------	-------

VISITID	ENTRYTIME	LEAVETIME	DURATION	HOLIDAY	DAYOFWEEK
N320001999	2020-05-09 06:02:00	2020-05-09 06:30:00	28	non-holiday	SATURDAY
N32000352	2020-05-09 06:05:00	2020-05-09 06:15:00	10	non-holiday	SATURDAY
N320001441	2020-05-09 06:06:00	2020-05-09 06:43:00	37	non-holiday	SATURDAY
N320002798	2020-05-09 06:06:00	2020-05-09 06:28:00	22	non-holiday	SATURDAY
N320001580	2020-05-09 06:08:00	2020-05-09 06:31:00	23	non-holiday	SATURDAY

Entry Number:	5 ▾
---------------	-----

Duration Distribution

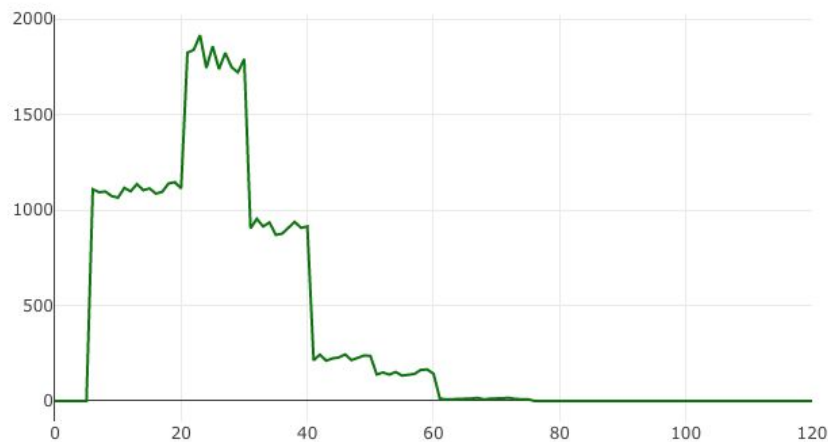


Figure 3. Show page.

Code Development & Test Metrics

Stage 1. Data simulation and generation

We have developed two phases for data generation.

1. The first part generates the initial visiting data for a normal day without any consideration of other factors.
2. The second part modifies the data (adds more visiting data or reduces customers) once a day is identified with any special events.
 - a. The customer visit data of every special event is defined separately.

- b. We defined methods such as `applyMealRush()`, `applySeniorDiscount()`, `applyHoliday()`, `applyDayBeforeHoliday()`, `applyNiceWeather()`, and `applyBadWeather()`.
- c. We borrowed Holiday API service (<https://holidayapi.com/>) to identify the holiday.
- d. Those functions allow the users to identify and create different patterns for special events/days, and allow these special patterns to be applied flexibly.

Code quality: After completing the classes in accordance with our design, we iteratively compared each stage of completeness with the new project requirements defined. At each stage, we also weighed the cost of our design choices in terms of extensibility, overhead, and time complexity. We primarily split our tests into black-box tests (unit testing) and white-box tests (integrated testing). We focused on unit testing for most of our model classes such as `Weather`, `Datetime`, `Visit`, `Day`, and `VisitParameters` classes, the ones used for simulating and generating the data, to ensure accurate behavior for the building blocks of our system. We also created white-box tests for our six static utility classes that operated mainly at the unit and integrated level to ensure that the arithmetic and logical aspects of our system were working properly. At this time, we have not yet tested our software at a system level (e.g. the `PilotSim` class) but have been using the outputs from our software and front UI to determine correctness (csv file, external data sets on MongoDB, and RESTful API).

Stage 2. Moving to database

1. We set up our environments to support MongoDB and Redis along with making the necessary configurations to transform our data from the csv files into the desired structure in the database.
2. Users and front-end are able to access the data via a series of *post* and *get* queries to the backend. SpringBoot back-end can interact with MongoDB for entries' CRUD.
3. In order to fetch specific dates' holiday information, we introduce an individual API service(as shown in Figure 2), which sends queries to an external API for responses of holiday information. Because HTTP requests and responses are expensive, we integrated this service with a Redis database memorizing all returned Date-Holiday mappings. With memorization, the number of queries can be significantly reduced and system's efficiency will be improved.

Code quality:

To assess our code metrics, we generated a CodeMR report for the new classes that were created inside the project `GrocerySprint3`, which implements the URL instructions and GET/POST requests. The `VisitController` class triggered a low-medium level indication for size, the

HolidayService class triggered a low-medium indication for complexity, and the Visit class triggered a low-medium indication for size and coupling.

We also generated a CodeMR report for our data generation project (GroceryStoryTraffic) since we made some modifications in sprint4. We found that the MetaDao class now had a medium-high level of coupling, which may be attributed to its function to populate the database. Therefore, we feel that this degree of coupling is within expectations. Similarly, the PilotSim class continued to have a medium-high level of coupling, which is within expectations and unchanged from before. All of the remaining classes were essentially unchanged and remained at low to low-medium indications across the board. Therefore, we are fairly confident that our code is still in good health by these metrics.

Stage 3. Frontend

1. UI was built in React components that could present all visit data stored in database in a spreadsheet as well as allow the users to add new single visit data in a form with two parameters - *holiday* and *day of week* automatically identified after the *visit date* is filled .
2. UI was connected to our backend through the urls of <http://localhost:3000/new> to add new data and <http://localhost:3000/show> to browse all data.
3. In the presentation of all the visit data, features of filtering and sorting the data by certain parameters are built.
4. We also add codes that produce distribution diagram of the selected visiting data.

Code quality: We did not create validation tests for this portion of the project, due to the difficulty in quantifying its product. We relied mainly on verification of the code functionality through the UI. We tested different filtering and ordering queries of the data presentation on each group member's console to assess functionality in different OS environments as well. We also tested the add queries by our team members.

Documentation of Final Product

Code location

- Main project directory: [cs5500_sum2020_group8](#)
- All sub-projects below are found directly inside the main project directory.
- Sub-project for data generation: [GroceryStoryTraffic](#)
- Sub-project for initializing RESTful API using Spring Boot: [GrocerySprint3](#)
- Sub-project for initializing alternative RESTful API using Node.js: [GroceryBack](#)
- Sub-project for launching interactive front-end UI using React: [grocery-front](#)

Procedure for building code

1. Download source code from the main project directory indicated above.

2. Initialize **IntelliJ** and open up the **GroceryStoryTraffic** sub-project.
 - a. Import/accept all changes/build prompted inside this project.
3. Switch to the remaining sub-projects (GrocerySprint3/GroceryBack, grocery-front) set up the environment.
 - a. Import all of the indicated changes prompted.
 - b. Build the backend either using subproject **GrocerySprint3** or **GroceryBack**.
 - c. Build the frontend subproject **grocery-front** with the command “npm start”.
4. See “Setup Phase” below for instructions to run each sub-project properly.

Product installation and setup

Downloads/Installations Phase

1. Download IntelliJ (recommend: Ultimate) for intended OS [here](#)
2. Download and install MongoDB (Community Ed.) for intended OS [here](#)
3. Download and install Postman for intended OS [here](#)
4. Download and install Node.js (LTS version) for intended OS [here](#)
5. Download and install Redis for intended OS [here](#)

Setup Phase

GroceryStoryTraffic Sub-Project

1. Import project build/changes with Gradle after opening project in IntelliJ
2. Install necessary plugins such as CodeMR
3. Run main() in the PilotSim class to generate data.

GrocerySprint3 Sub-Project

1. Import project build/changes with Maven after opening project in IntelliJ
2. Accept all install prompts for dependencies (this sets up the Spring environment)
3. Run main() inside GroceryVisitsApplication to launch the query server

Grocery-Front Sub-Project

1. Import project build/changes after opening project in IntelliJ
2. Through the terminal, type in the commands “npm install --save”
3. Through the terminal, type in “npm start” to launch the UI landing page.
4. Add /new to the landing page URL, which re-routes to another page to add a new visit.
5. Add /show to the landing page URL, which re-routes to the page visualizing all visits.

Known problems

GroceryStoryTraffic Sub-Project

1. Lacking a user-friendly UI

- a. Our current project defines some constants inside the PilotSim Class which allows the user to switch up which month they would like to generate data for. We feel that this allows the user too much access to the project and would like to create a user interface to take in these parameters to generate the data.
 - b. Currently our users cannot edit auto-generated entries but there are cases where some additional data need inserting and modifying. A User-Interface can help give them more control of generating the data they want.
 - c. Our program can write tons of data into CSV files or a database but there is a lack of intuition to how the data is like or visualization of simulation results for users.
2. Deprecated code inside PilotSim
- a. Our current project first generates the baseline visits under absolutely normal conditions. Afterwards, it will run through several checks and generate additional visits depending on modifying factors. However, when our current program generates the additional visits, we run through some duplicate steps to generate the parameters for each visit. For a large quantity of visits, this causes a lot of duplicate work that may be avoided if we improved the project design.
 - b. Conflicts between GroceryStoryTraffic and GrocerySprint3, different classes of the same entity have different attributes. Thus, new intermediate classes and conversion functions need introducing.

GrocerySprint3 Sub-Project

- 1. Lengthy method names. Because we are extending from the CRUD repository, the method declarations are restricted and can easily create long names. We found a solution to use an additional interface and abstract class to change the names, but we felt that it added additional complexity to this project without improving the functionality.
- 2. Lengthy URI pathways. Although the pathways are descriptive, we found that some parameters such as entry time and leave time are more complex and prone to type errors. Since our system does not handle query errors very well at this stage (see number 3), we feel that this is another issue we should aim to resolve.
- 3. Non-descriptive error messages for bad queries. Currently, our code does not generate descriptive error messages for most of our supported queries. Therefore, users who accidentally type a bad query will not know how to fix the issue unless they check the Javadoc inside the project method itself. We feel that incorporating descriptive error messages will help improve the user experience and avoid these situations. We also feel that including a printable help menu could be a potential solution.

Grocery-Front Sub-Project

- 1. Code has not quite been decoupled due to limited time and components(even though they are on the page) are clustering, which is not convenient to maintain and extend.

2. Resources' loading strategy can be changed, now it works as loading all resources from a single request which can be more flexible. For example, return at most 10000 entries for each request and then pages' loading time can be significantly reduced.
3. Lack of authentication and authorization. Managers and general users can be distinguished according to their privileges(e.g. some employees can edit but others can only review entries). In the future, Sign-in and Sign-up features can be introduced.

Other relevant Documents

1. Grocery store visiting pattern
 - a. <https://www.safegraph.com/dashboard/covid19-commerce-patterns>
 - b. paper reference: <https://www.nature.com/articles/s41597-020-0397-7#Sec14>
2. Holiday API service (<https://holidayapi.com/>)