Project 7: Free Food Finder

Yifan (Catherine) Chen, Jett Crowson, & Michelle Tran

Final State of System Statement

What was implemented?
We were able to get full functionality of all planned components in our system including the ability to create food events, the ability to search and filter food events, the ability to display these food events on an interactive map, and the ability to report food events (as being empty). Each of these components had a corresponding API and service on the Spring Boot backend that assisted with interacting with the database and performing other functionality not appropriate for the frontend Angular.

What was not?
One of our stretch goals for the project was to implement a login and account system for users to post their food, view their events, and possibly edit or delete their events. Implementing the portions of our project that we did get around to implementing took a considerable amount of time, especially with learning new frameworks like Spring Boot. Login and account systems are not a light task, and so we decided against implementing it.

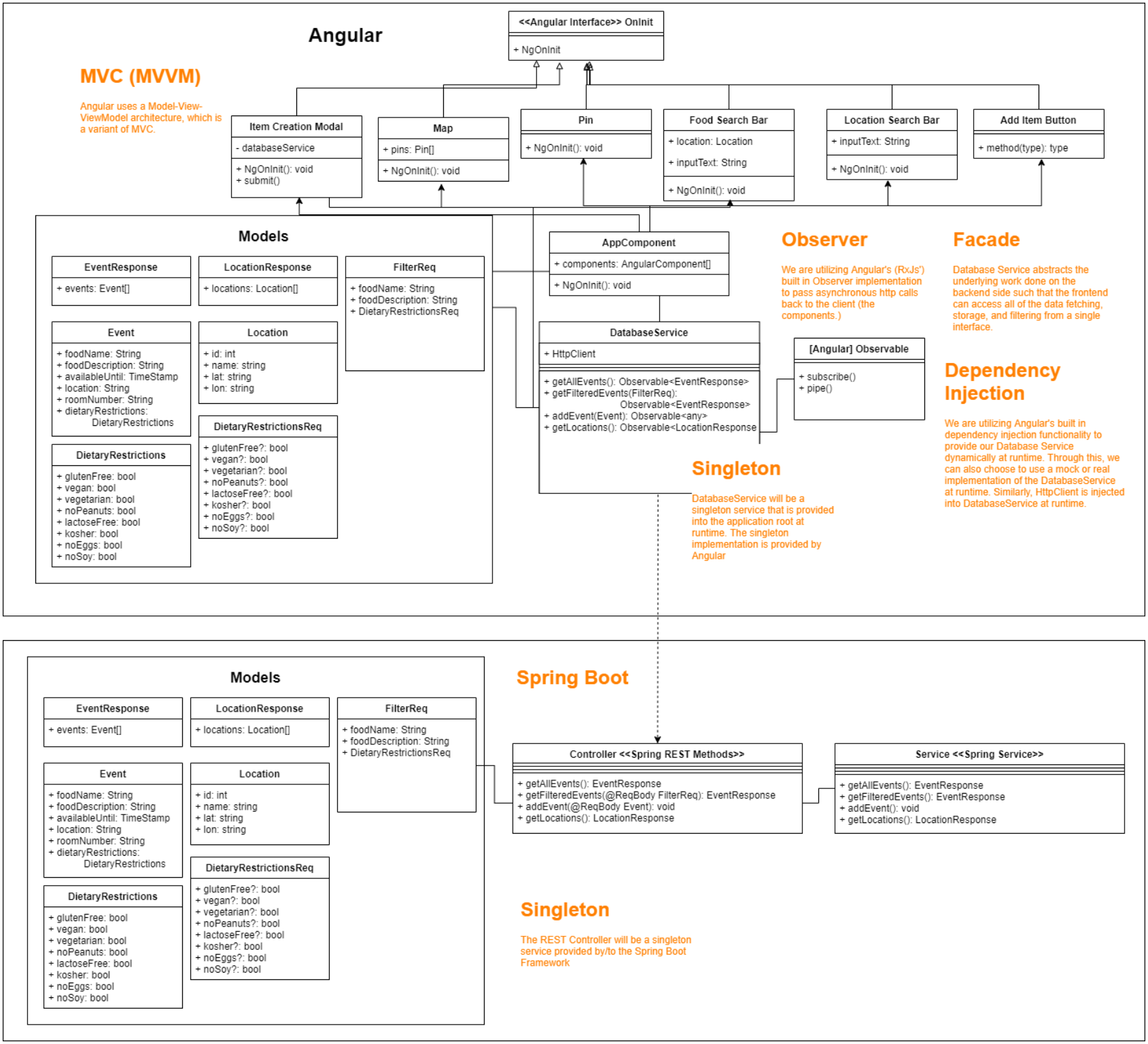
What changed?
We didn't have very many notable architecture changes from Projects 5 and 6 to Project 7, as our design worked out for our final implementation (aside from a few class/interface additions purely to appease frameworks). We did end up having time for one of our stretch goals, which was for users to be able to report events that were no longer available but were still listed on the map since it hasn't expired yet, so we did have to update our architecture for that.

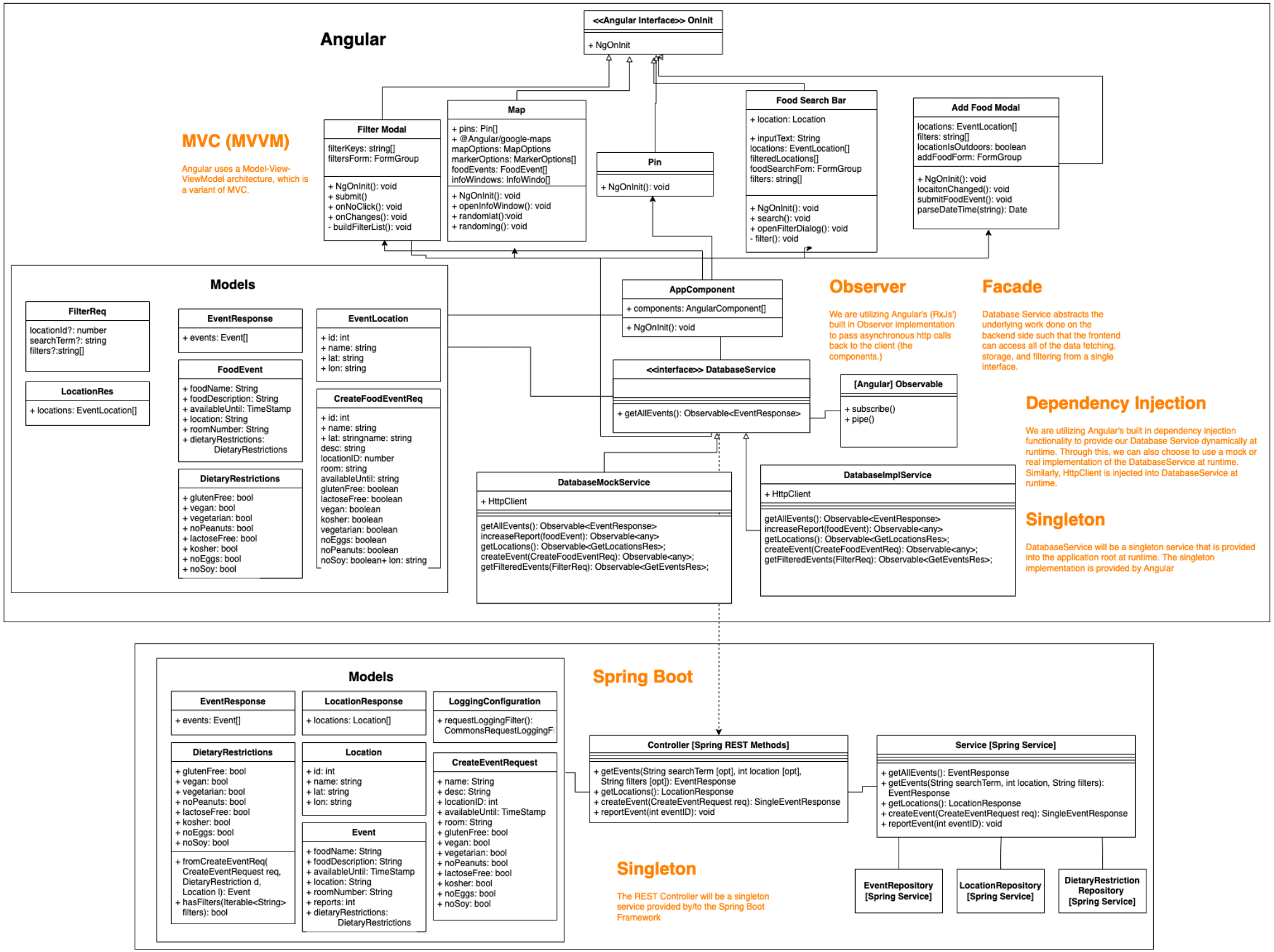
Class Diagram

Project 5 & 7 Diagrams are shown (in order) on the following 2 pages.

Evolution of Design

For the most part, our design was very stable throughout the process, and we only really added new functions or classes to modularize some helper functions or to appease the frameworks. Going into the project, some of the team members had experience with the frameworks that we intended to use, and were familiar with what the designs would roughly look like. This helped keep things relatively stable.


Angular
<<Angular Interface>> OnInit
+ NgOnInit
MVC (MVVM)
Angular uses a Model-View-ViewModel architecture, which is a variant of MVC.
Item Creation Modal
- databaseService
+ NgOnInit(): void
+ submit()
Map
+ pins: Pin[]
+ NgOnInit(): void
Pin
+ NgOnInit(): void
Food Search Bar
+ location: Location
+ inputText: String
+ NgOnInit(): void
Location Search Bar
+ inputText: String
+ NgOnInit(): void
Add Item Button
+ method(type): type
Models
EventResponse
+ events: Event[]
LocationResponse
+ locations: Location[]
FilterReq
+ foodName: String
+ foodDescription: String
+ DietaryRestrictionsReq
Event
+ foodName: String
+ foodDescription: String
+ availableUntil: TimeStamp
+ location: String
+ roomNumber: String
+ dietaryRestrictions: DietaryRestrictions
Location
+ id: int
+ name: string
+ lat: string
+ lon: string
DietaryRestrictionsReq
+ glutenFree?: bool
+ vegan?: bool
+ vegetarian?: bool
+ noPeanuts?: bool
+ lactoseFree?: bool
+ kosher?: bool
+ noEggs?: bool
+ noSoy?: bool
DietaryRestrictions
+ glutenFree: bool
+ vegan: bool
+ vegetarian: bool
+ noPeanuts: bool
+ lactoseFree: bool
+ kosher: bool
+ noEggs: bool
+ noSoy: bool
AppComponent
+ components: AngularComponent[]
+ NgOnInit(): void
DatabaseService
+ HttpClient
+ getAllEvents(): Observable<EventResponse>
+ getFilteredEvents(FilterReq): Observable<EventResponse>
+ addEvent(Event): Observable<any>
+ getLocations(): Observable<LocationResponse
[Angular] Observable
+ subscribe()
+ pipe()
Observer
We are utilizing Angular's (RxJs') built in Observer implementation to pass asynchronous http calls back to the client (the components.)
Facade
Database Service abstracts the underlying work done on the backend side such that the frontend can access all of the data fetching, storage, and filtering from a single interface.
Dependency Injection
We are utilizing Angular's built in dependency injection functionality to provide our Database Service dynamically at runtime. Through this, we can also choose to use a mock or real implementation of the DatabaseService at runtime. Similarly, HttpClient is injected into DatabaseService at runtime.
Singleton
DatabaseService will be a singleton service that is provided into the application root at runtime. The singleton implementation is provided by Angular
Models
EventResponse
+ events: Event[]
LocationResponse
+ locations: Location[]
FilterReq
+ foodName: String
+ foodDescription: String
+ DietaryRestrictionsReq
Event
+ foodName: String
+ foodDescription: String
+ availableUntil: TimeStamp
+ location: String
+ roomNumber: String
+ dietaryRestrictions: DietaryRestrictions
Location
+ id: int
+ name: string
+ lat: string
+ lon: string
DietaryRestrictionsReq
+ glutenFree?: bool
+ vegan?: bool
+ vegetarian?: bool
+ noPeanuts?: bool
+ lactoseFree?: bool
+ kosher?: bool
+ noEggs?: bool
+ noSoy?: bool
DietaryRestrictions
+ glutenFree: bool
+ vegan: bool
+ vegetarian: bool
+ noPeanuts: bool
+ lactoseFree: bool
+ kosher: bool
+ noEggs: bool
+ noSoy: bool
Spring Boot
Controller <<Spring REST Methods>>
+ getAllEvents(): EventResponse
+ getFilteredEvents(@ReqBody FilterReq): EventResponse
+ addEvent(@ReqBody Event): void
+ getLocations(): LocationResponse
Service <<Spring Service>>
+ getAllEvents(): EventResponse
+ getFilteredEvents(): EventResponse
+ addEvent(): void
+ getLocations(): LocationResponse
Singleton
The REST Controller will be a singleton service provided by/to the Spring Boot Framework


Angular
<<Angular Interface>> OnInit
+ NgOnInit
MVC (MVVM)
Angular uses a Model-View-ViewModel architecture, which is a variant of MVC.
Filter Modal
filterKeys: string[]
filtersForm: FormGroup
+ NgOnInit(): void
+ submit()
+ onNoClick(): void
+ onChanges(): void
- buildFilterList(): void
Map
+ pins: Pin[]
+ @Angular/google-maps mapOptions: MapOptions markerOptions: MarkerOptions[]
foodEvents: FoodEvent[]
infoWindows: InfoWindo[]
+ NgOnInit(): void
+ openInfoWindow(): void
+ randomlat():void
+ randomlng(): void
Pin
+ NgOnInit(): void
Food Search Bar
+ location: Location
+ inputText: String
locations: EventLocation[]
filteredLocations[]
foodSearchFom: FormGroup
filters: string[]
+ NgOnInit(): void
+ search(): void
+ openFilterDialog(): void
- filter(): void
Add Food Modal
locations: EventLocation[]
filters: string[]
locationIsOutdoors: boolean
addFoodForm: FormGroup
+ NgOnInit(): void
locaitonChanged(): void
submitFoodEvent(): void
parseDateTime(string): Date
Models
FilterReq
locationId?: number
searchTerm?: string
filters?:string[]
LocationRes
+ locations: EventLocation[]
EventResponse
+ events: Event[]
FoodEvent
+ foodName: String
+ foodDescription: String
+ availableUntil: TimeStamp
+ location: String
+ roomNumber: String
+ dietaryRestrictions: DietaryRestrictions
DietaryRestrictions
+ glutenFree: bool
+ vegan: bool
+ vegetarian: bool
+ noPeanuts: bool
+ lactoseFree: bool
+ kosher: bool
+ noEggs: bool
+ noSoy: bool
EventLocation
+ id: int
+ name: string
+ lat: string
+ lon: string
CreateFoodEventReq
+ id: int
+ name: string
+ lat: stringname: string
desc: string
locationID: number
room: string
availableUntil: string
glutenFree: boolean
lactoseFree: boolean
vegan: boolean
kosher: boolean
vegetarian: boolean
noEggs: boolean
noPeanuts: boolean
noSoy: boolean+ lon: string
AppComponent
+ components: AngularComponent[]
+ NgOnInit(): void
<<interface>> DatabaseService
+ getAllEvents(): Observable<EventResponse>
[Angular] Observable
+ subscribe()
+ pipe()
Observer
We are utilizing Angular's (RxJs') built in Observer implementation to pass asynchronous http calls back to the client (the components.)
Facade
Database Service abstracts the underlying work done on the backend side such that the frontend can access all of the data fetching, storage, and filtering from a single interface.
Dependency Injection
We are utilizing Angular's built in dependency injection functionality to provide our Database Service dynamically at runtime. Through this, we can also choose to use a mock or real implementation of the DatabaseService at runtime. Similarly, HttpClient is injected into DatabaseService at runtime.
Singleton
DatabaseService will be a singleton service that is provided into the application root at runtime. The singleton implementation is provided by Angular
DatabaseMockService
+ HttpClient
getAllEvents(): Observable<EventResponse>
increaseReport(foodEvent): Observable<any>
getLocations(): Observable<GetLocationsRes>;
createEvent(CreateFoodEventReq): Observable<any>;
getFilteredEvents(FilterReq): Observable<GetEventsRes>;
DatabaseImplService
+ HttpClient
getAllEvents(): Observable<EventResponse>
increaseReport(foodEvent): Observable<any>
getLocations(): Observable<GetLocationsRes>;
createEvent(CreateFoodEventReq): Observable<any>;
getFilteredEvents(FilterReq): Observable<GetEventsRes>;
Spring Boot
Models
EventResponse
+ events: Event[]
LocationResponse
+ locations: Location[]
LoggingConfiguration
+ requestLoggingFilter(): CommonsRequestLoggingF
DietaryRestrictions
+ glutenFree: bool
+ vegan: bool
+ vegetarian: bool
+ noPeanuts: bool
+ lactoseFree: bool
+ kosher: bool
+ noEggs: bool
+ noSoy: bool
+ fromCreateEventReq(CreateEventRequest req, DietaryRestriction d, Location l): Event
+ hasFilters(Iterable<String> filters): bool
Location
+ id: int
+ name: string
+ lat: string
+ lon: string
Event
+ foodName: String
+ foodDescription: String
+ availableUntil: TimeStamp
+ location: String
+ roomNumber: String
+ reports: int
+ dietaryRestrictions: DietaryRestrictions
CreateEventRequest
+ name: String
+ desc: String
+ locationID: int
+ availableUntil: TimeStamp
+ room: String
+ glutenFree: bool
+ vegan: bool
+ vegetarian: bool
+ noPeanuts: bool
+ lactoseFree: bool
+ kosher: bool
+ noEggs: bool
+ noSoy: bool
Controller [Spring REST Methods]
+ getEvents(String searchTerm [opt], int location [opt], String filters [opt]): EventResponse
+ getLocations(): LocationResponse
+ createEvent(CreateEventRequest req): SingleEventResponse
+ reportEvent(int eventID): void
Service [Spring Service]
+ getAllEvents(): EventResponse
+ getEvents(String searchTerm, int location, String filters): EventResponse
+ getLocations(): LocationResponse
+ createEvent(CreateEventRequest req): SingleEventResponse
+ reportEvent(int eventID): void
Singleton
The REST Controller will be a singleton service provided by/to the Spring Boot Framework
EventRepository [Spring Service]
LocationRepository [Spring Service]
DietaryRestriction Repository [Spring Service]

Third-Party Code

For specific code snippets/error debugging citations, see our Github repositories -- we've added links to each code location where they are used. In general, much of the infrastructure/plumbing code for both the Angular and Spring components were generated by their respective project generators, and our contributions were made (primarily) in their `src` folders.

General References and Tutorials

- Angular
    - Framework for frontend
    - https://angular.io/
        - The Frontend Framework
    - https://angular.io/tutorial
        - Useful for showing unfamiliar members the basics of Angular
    - https://angular.io/docs
        - Extremely helpful documentation that was utilized extensively throughout development.
- RxJs
    - Used for asynchronous operations in Angular
    - https://rxjs.dev/
- Google Maps API + Angular component
    - https://github.com/angular/components/tree/master/src/google-maps
        - Official Google Map component from Google for Angular
    - https://timdeschryver.dev/blog/google-maps-as-an-angular-component
        - Tutorial on how to use the google maps component
    - https://developers.google.com/maps/gmp-get-started
        - Google maps api tutorial
    - https://developers.google.com/maps/documentation/javascript/infowindows
        - Creating pop up windows for the pins
    - https://material.angular.io/components/button/overview
        - Creating button in popup window
- Spring
    - Generating Spring project: https://start.spring.io/
    - Connecting Spring backend to MySQL DB tutorial: https://spring.io/guides/gs/accessing-data-mysql/
    - Spring CRUD Repositories: https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html
- MySQL
    - DDL generation using MySQL Workbench
    - Getting MySQL dev environment setup: https://dev.mysql.com/doc/mysql-getting-started/en/

Statement on OOAD Process

1. [GOOD] Us defining our data/json models and API endpoints together in the UML and in project planning documents was very useful. Agreeing on them synchronously and then working separately to build them async worked very well. This is because we could bring our pieces together and they would work fairly seamlessly, and all parties knew the exact contract to program and design to.
2. [GOOD] Creating use cases, gathering and recording requirements, and architectural diagramming at the start gave us a clear understanding/goal of what we expected the final product to look like. This also guided all of our task divvying, goal setting, and software implementations, from before we even wrote a single line of code. Even with proper OOP implementations, change is expensive, so these helped to keep things consistent and everyone on the same page.
3. [NEUTRAL] We noticed that having prior knowledge about our intended frameworks/technical stack heavily influenced all aspects of the design process in Project 5, from our UML diagrams to state diagrams and everything in between. Had we not had someone with expertise about some of our anticipated frameworks, our designs from Project 5 probably would not have held up nearly as well for our final implementation.