POLITECNICO
MILANO 1863

Software Engineering 2
Politecnico di Milano, A.A. 2017/2018

Travlendar+
Design Document

Version: 1.0
Release date: 26/11/2017

Riccardo Novic (mat. 829184)
Lorenzo Pratissoli (mat. 893980)

1 Introduction

1.1 Purpose

This document is the Design Document for the *Travlendar+* platform. It provides an insight into the entities and components that form the *Travlendar+* application, with a section dedicated to Integration and Testing. The purpose of this document is to outline the platform's architecture structure and to provide detailed information about component interaction, along with explanations that justify the decisions that have been taken and an analysis of how said choices optimize the development process and the future scaling of the application.

1.2 Scope

The extent of the descriptions in this document concerns the component view, deployment view and runtime view of the system architecture, integrated with a section about algorithm design and a detailed implementation and testing plan.
Travlendar+ will provide a range of services that allow to manage:

- Users: *Travlendar+* will manage personal data of its registered users.
- Events: *Travlendar+* will give its users the ability to create and manage events.
- Calendars: *Travlendar+* will provide ways for its users to organize their events in different calendars, with specific customizable settings that are automatically applied to the event inside the calendar.
- Travel Instructions: *Travlendar+* will show to the users detailed travel information needed to reach his/her events, with step-by-step info.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Reverse proxy: a reverse proxy server retrieves resources on behalf of a client from one or more servers.
- Message broker: an intermediary platform when it comes to processing communication between two applications.
- NoSQL database: a NoSQL database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.

1.3.2 Acronyms

- RASD: Requirement Analysis and Specifications Document
- DD: Design Document
- API: Application Program Interface
- OS: Operative System
- REST: Representational State Transfer
- DB: Database
- JWT: JSON Web Tokens
- CI/CD: Continuous integration and Continuous delivery
- RDBMS: Relational Database Management System
- CQRS: Command Query Responsibility Segregation
- DNS: Domain Name System
- HTTP: HyperText Transfer Protocol
- TCP: Transmission Control Protocol
- IP: Internet Protocol

1.3.3 Abbreviations

- [Rn]: functional requirement number n.

1.4 Revision history

1.5 Reference Documents

- AA 2017-2018 Software Engineering 2—Mandatory Project goal, schedule, and rules

1.6 Document Structure

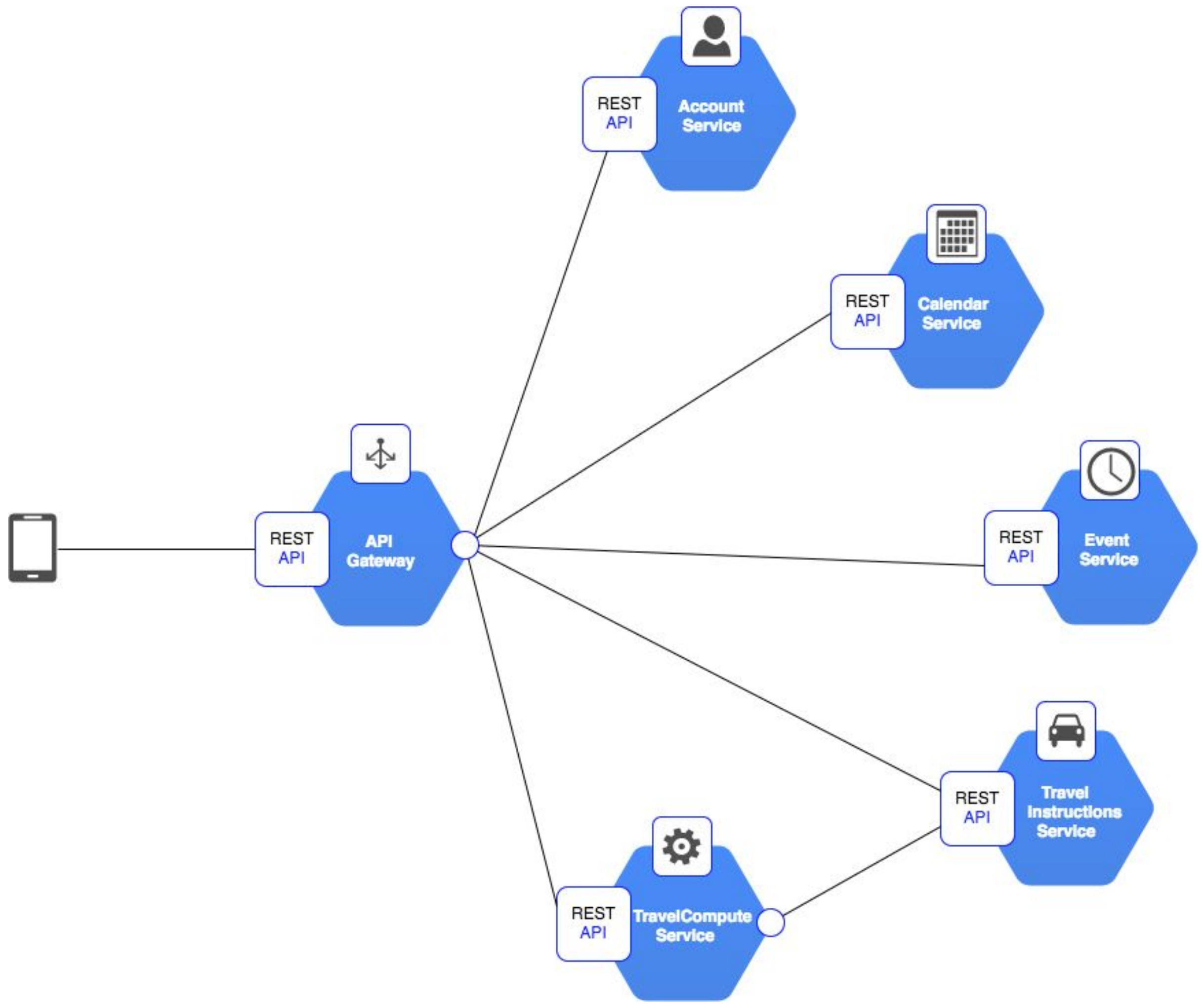
This document is structured in the following way:

- Section 2: this section describes the platform’s architecture structure and provides detailed information about component interaction.
- Section 3: this section includes an insight on the most relevant algorithmic aspects.
- Section 4: here it’s listed the mapping between the requirements in the RASD and the components defined in this document.
- Section 5: presents the way in which the implementation and the integration should be done and the aspects to focus the testing part on.

2. Architectural Design

2.1 Overview

The Travlendar+ platform is developed with a Microservices architecture:



As we can see in the diagram, the API Gateway is the single entry point of our application which is responsible to forward the request to the internal system, which is clustered in multiple Microservices in a modular way, where each service is responsible for a specific task.
The API Gateway behaves as a reverse proxy between the Client and the *Travlendar*+ backend, where the entry point is responsible for the 1:N mapping between the REST API exposed externally and the ones offered internally by each Microservice. For example, when the User requests his schedule to be shown on his Client, this request is routed by the API Gateway to three different internal calls:

- Calendar service, to get the user calendars.
- Event service, to get the events belonging to the User calendar.
- TravelInstructions service, to get the list of all the travel instructions saved to reach the future events.

As we can see from this flow of API calls, each service is responsible for a very specific task. This provides us with many advantages, such as:

- High level of modularity, which helps the developers to follow the CI/CD practices in an intuitive and fast manner.
- Possibility to develop service independently, with different deployment rates and the ability to include in each service the most updated technologies and framework without the need to take the whole system down.
- Improved scalability control, to satisfy capacity and availability constraints in an easy and immediate fashion.
- High supervising level over the Hardware used to best match the resource requirements for each service.

The data storage system is distributed across the multiple services, where this encapsulation ensures that the microservices are loosely coupled and can evolve independently one of another. By following this approach, we have possibility to use different kind of databases for different services (e.g. NoSQL implementation for the TravelInstructions service, a classic RDBMS one for the Calendar service), giving access to the so-called *polyglot-persistent architecture* approach.
The data partitioning is splitted as follows:

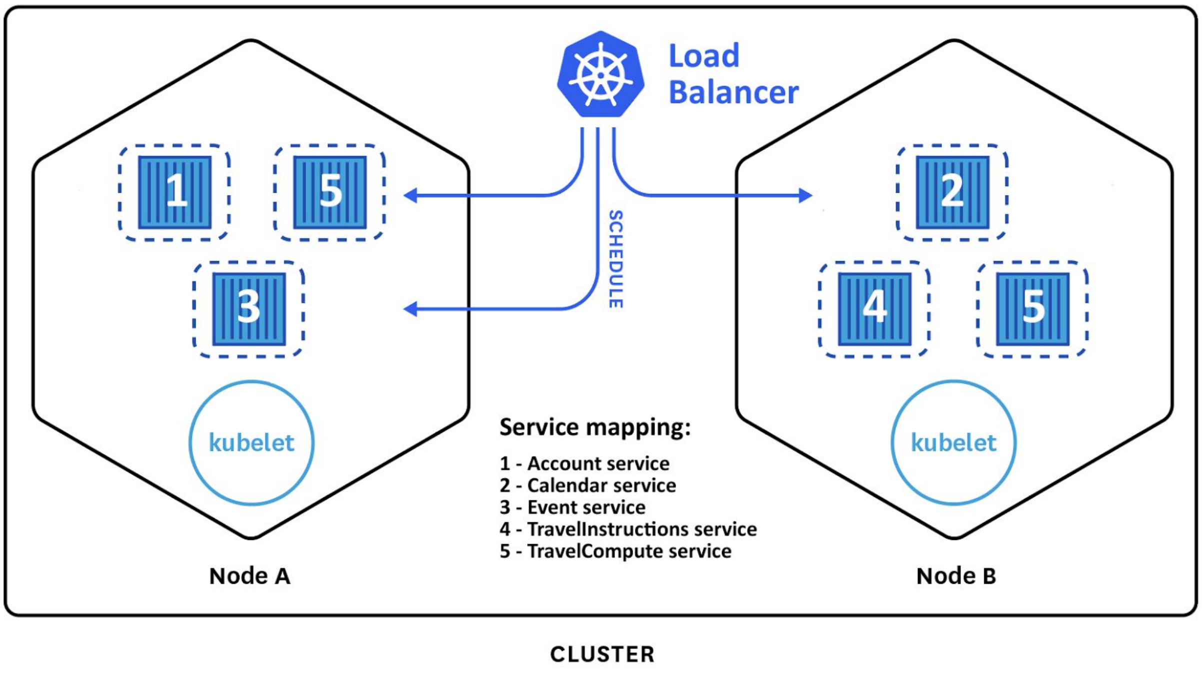
- Account related information are stored in the Account service.
- Calendar related information and User global preferences settings are stored in the Calendar service.
- Event related information are stored in the Event services, alongside with the related Recurrence options and exceptions, the latter responsible to store information when a single event that belongs in a recurrent context is deleted.
- Travel instructions information are stored in the TravelInstructions service.
- Cached travel calculations are stored in the TravelCompute service.

In order to maintain data consistency across different services, data management will be implemented in an event-driven architecture through the event sourcing pattern. Every service publishes an event as a part of an atomic operation, which is concluded after all the interested services subscribed finally consume that event. CQRS is used to implement the queries: as a result, the consistency is eventually guaranteed. Eventual consistency has to be taken into account by the developers for a proper implementation.

Docker is used as the packaging and deployment tool needed to develop the Microservices structure: every component shown in the Component diagram (section 2.2) will be deployed in a single ad-hoc container, built on a container image: a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it (code, runtime, system tools, system libraries, settings).
As a consequence, we have different containers to manage. In order to deploy, manage and scale this containerized application Google Container engine will be used, powered by the Kubernetes system.
Kubernetes orchestrates the containers using different levels of abstraction and provides us with a set of development and maintenance features, such as:

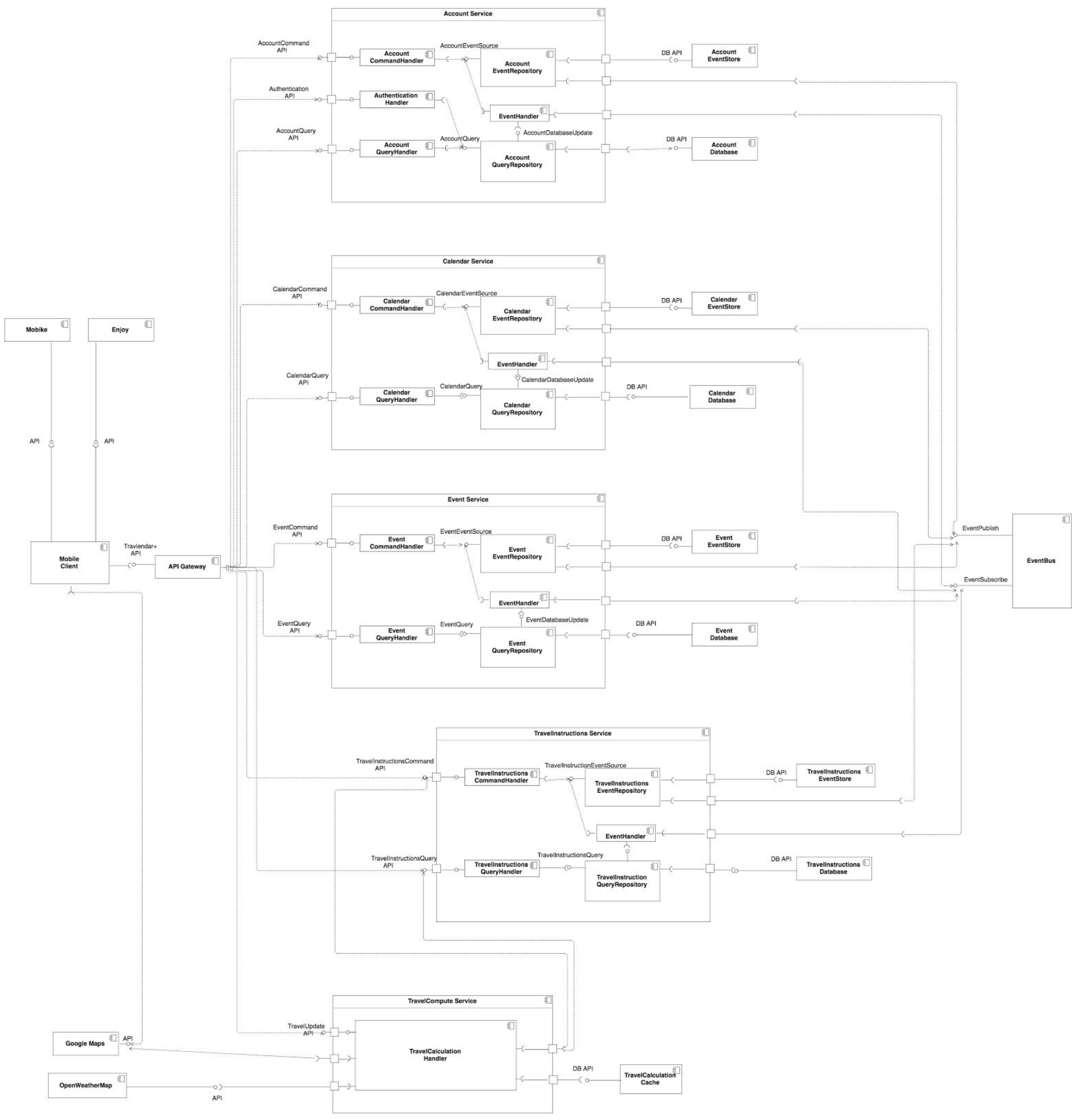
- **Automatic horizontal scaling**: this lets the system grow without having to manage manually hardware and software changes.
- **Service discovery and load balancing**: Kubernetes gives containers their own IP addresses and a single DNS name for a set of containers, and provides a load-balancing system that addresses both the external requests and the internal calls to the various services.
- **Automated rollouts and rollbacks**: Kubernetes progressively rolls out changes to your application or its configuration, while monitoring application health to ensure it doesn't kill all

your instances at the same time. If something goes wrong, Kubernetes will rollback the change for you. Take advantage of a growing ecosystem of deployment solutions.
- **Smart configuration management:** deploy and update application configurations without the need to rebuild docker images.
- **Self-healing and automatic logging**: with built-in integration of logging tools and monitoring systems, it is possible to manage containers accordingly to health-checks feedback and to have an insight of how the application is running, providing a stable service able to heal itself in order to run 24/7.

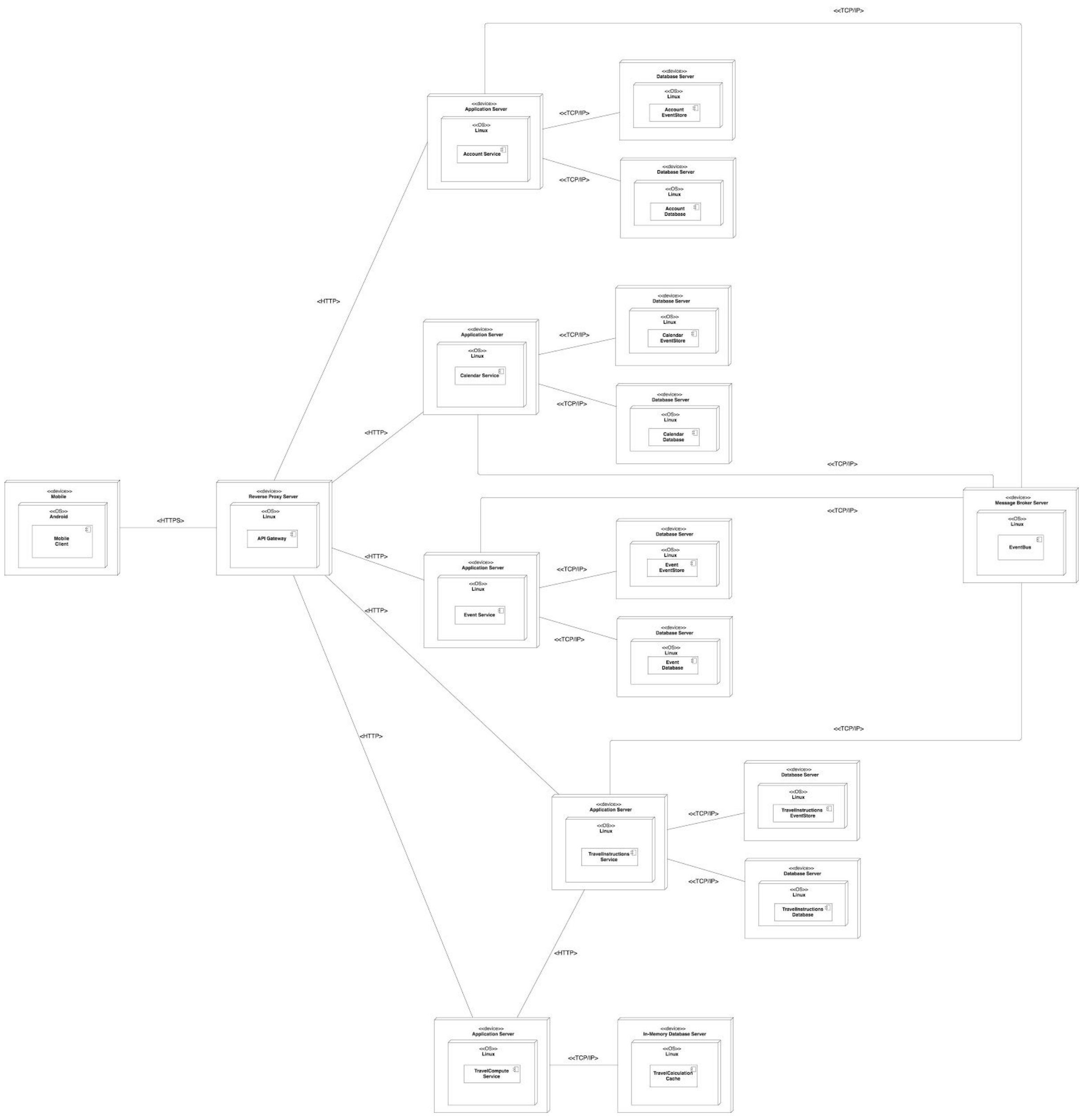


This diagram shows a possible initial configuration for the Travlendar+ cluster. In this case there are two nodes running, where the workload is handled by the Load Balancer. Every service has a single replica, except for the TravelCompute service which needs more computational power than the others. The kubelet is is responsible for the running state of each node (that is, ensuring that all containers on the node are healthy), taking care of starting, stopping, and maintaining application containers.

2.2 Component view



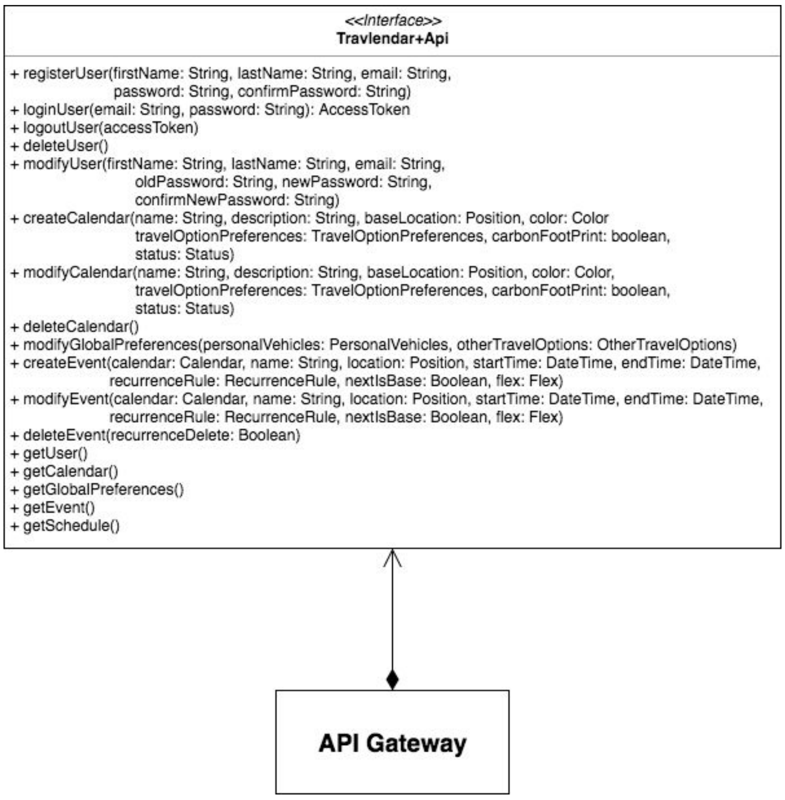
2.3 Deployment view



2.4 Component Interfaces

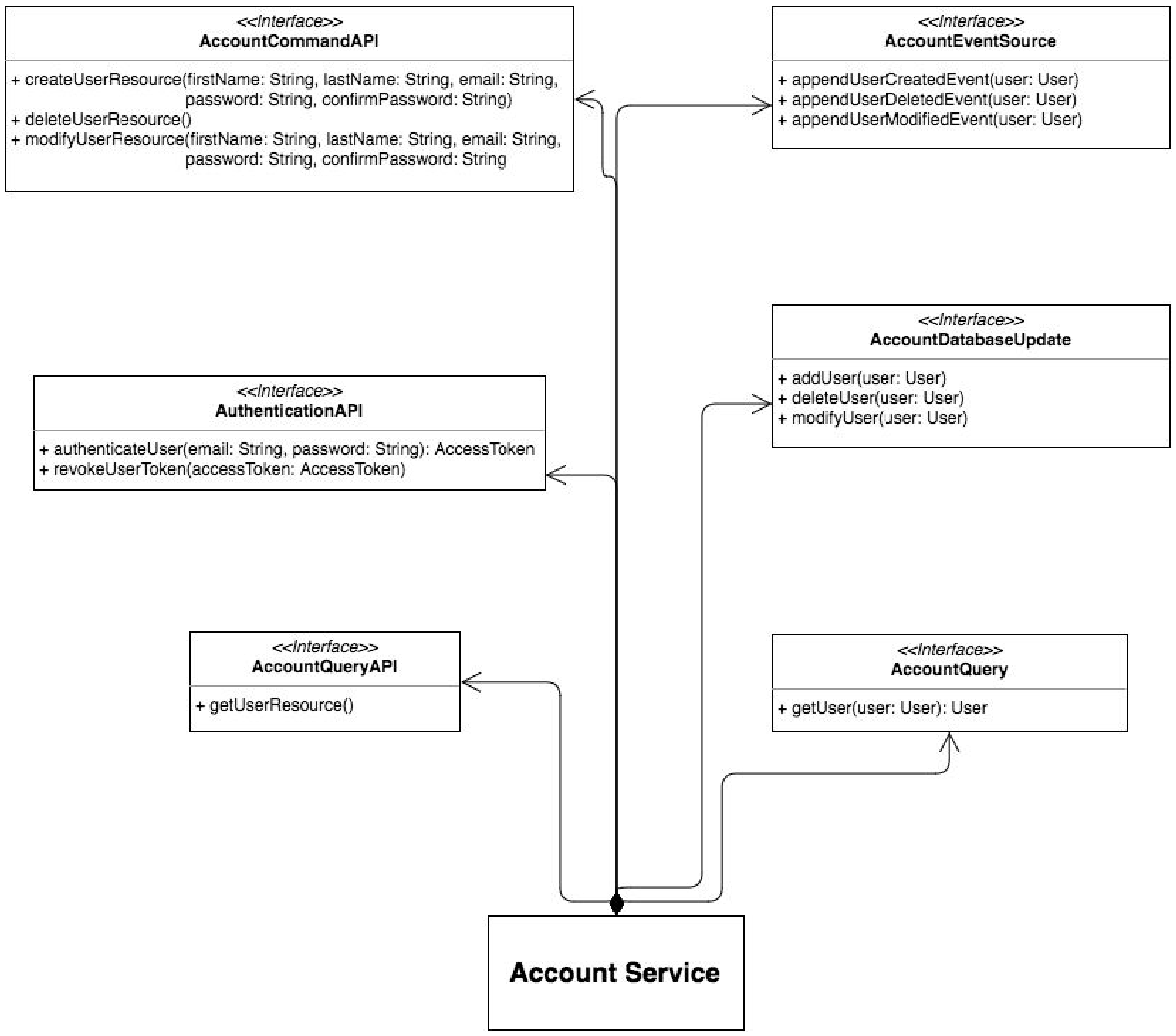
As mentioned in the overview every http communication between services is done in a RESTful way. Every method of interfaces exposed on a http/https communication link will be mapped in the implementation process to a proper http call. For example the getUser method exposed by the API gateway could be mapped to a GET /users/{id} where id is the id of the user to be retrieved. All the methods corresponding to GET calls do not have arguments because all the information to be sent to the backend must be already contained in the GET request (like the ID for the getUser). All the methods corresponding to PUT/POST/DELETE(like modifyCalendar) calls have the arguments corresponding to the request payload.
It is also worth noticing the accessToken has been put in the logoutUser method, even though it is not sent in the http body(an access token is always sent using headers), to emphasize the fact that it should be revoked in the backend.

API Gateway



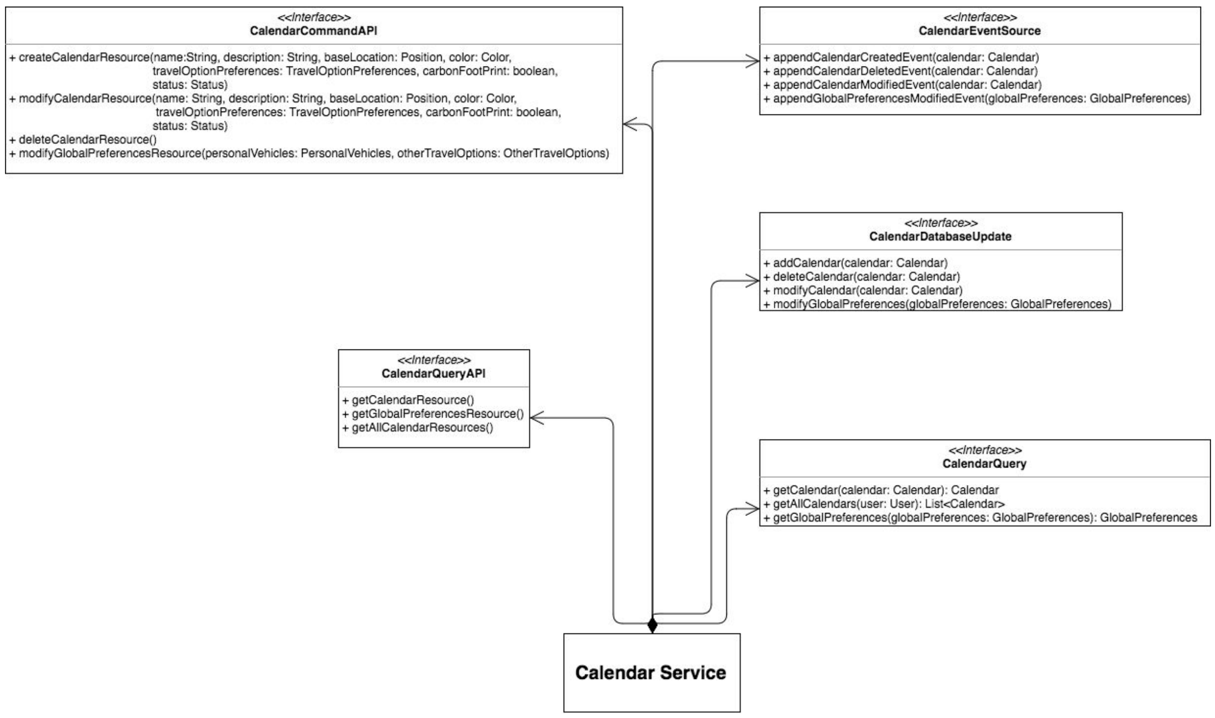
```
+ registerUser(firstName: String, lastName: String, email: String,
                password: String, confirmPassword: String)
+ loginUser(email: String, password: String): AccessToken
+ logoutUser(accessToken)
+ deleteUser()
+ modifyUser(firstName: String, lastName: String, email: String,
                oldPassword: String, newPassword: String,
                confirmNewPassword: String)
+ createCalendar(name: String, description: String, baseLocation: Position, color: Color
                travelOptionPreferences: TravelOptionPreferences, carbonFootPrint: boolean,
                status: Status)
+ modifyCalendar(name: String, description: String, baseLocation: Position, color: Color,
                travelOptionPreferences: TravelOptionPreferences, carbonFootPrint: boolean,
                status: Status)
+ deleteCalendar()
+ modifyGlobalPreferences(personalVehicles: PersonalVehicles, otherTravelOptions: OtherTravelOptions)
+ createEvent(calendar: Calendar, name: String, location: Position, startTime: DateTime, endTime: DateTime,
                recurrenceRule: RecurrenceRule, nextIsBase: Boolean, flex: Flex)
+ modifyEvent(calendar: Calendar, name: String, location: Position, startTime: DateTime, endTime: DateTime,
                recurrenceRule: RecurrenceRule, nextIsBase: Boolean, flex: Flex)
+ deleteEvent(recurrenceDelete: Boolean)
+ getUser()
+ getCalendar()
+ getGlobalPreferences()
+ getEvent()
+ getSchedule()
```

Account Service

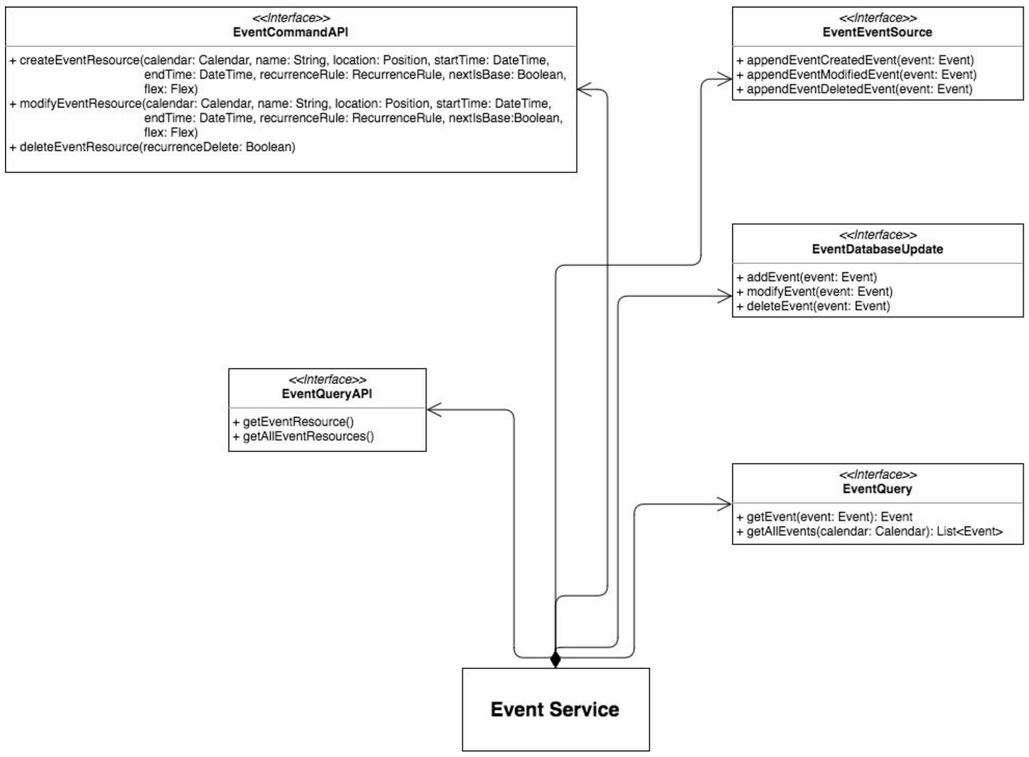


<<Interface>>
AccountCommandAPI
+ createUserResource(firstName: String, lastName: String, email: String, password: String, confirmPassword: String)
+ deleteUserResource()
+ modifyUserResource(firstName: String, lastName: String, email: String, password: String, confirmPassword: String
<<Interface>>
AccountEventSource
+ appendUserCreatedEvent(user: User)
+ appendUserDeletedEvent(user: User)
+ appendUserModifiedEvent(user: User)
<<Interface>>
AccountDatabaseUpdate
+ addUser(user: User)
+ deleteUser(user: User)
+ modifyUser(user: User)
<<Interface>>
AuthenticationAPI
+ authenticateUser(email: String, password: String): AccessToken
+ revokeUserToken(accessToken: AccessToken)
<<Interface>>
AccountQueryAPI
+ getUserResource()
<<Interface>>
AccountQuery
+ getUser(user: User): User
Account Service

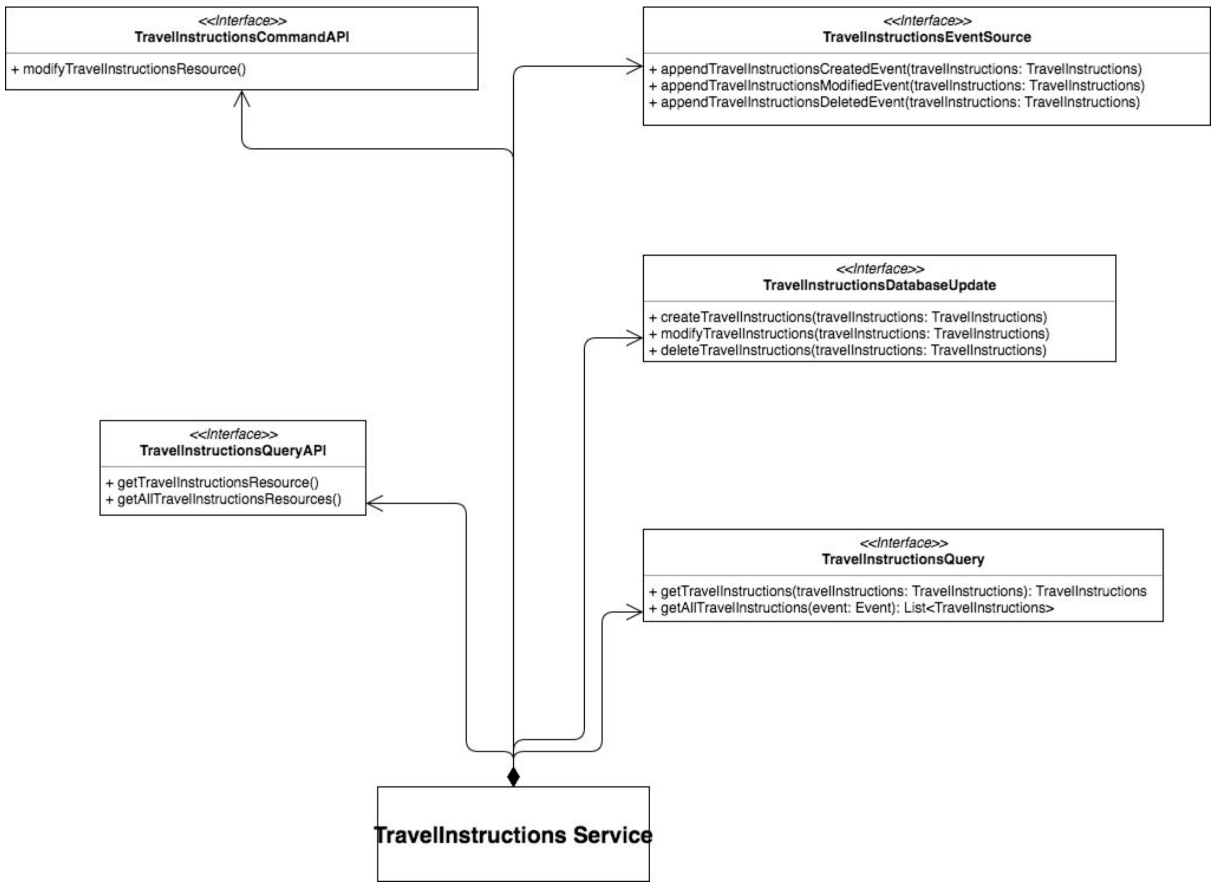
Calendar Service


<<Interface>>
CalendarCommandAPI
+ createCalendarResource(name:String, description: String, baseLocation: Position, color: Color, travelOptionPreferences: TravelOptionPreferences, carbonFootPrint: boolean, status: Status)
+ modifyCalendarResource(name: String, description: String, baseLocation: Position, color: Color, travelOptionPreferences: TravelOptionPreferences, carbonFootPrint: boolean, status: Status)
+ deleteCalendarResource()
+ modifyGlobalPreferencesResource(personalVehicles: PersonalVehicles, otherTravelOptions: OtherTravelOptions)
<<Interface>>
CalendarEventSource
+ appendCalendarCreatedEvent(calendar: Calendar)
+ appendCalendarDeletedEvent(calendar: Calendar)
+ appendCalendarModifiedEvent(calendar: Calendar)
+ appendGlobalPreferencesModifiedEvent(globalPreferences: GlobalPreferences)
<<Interface>>
CalendarDatabaseUpdate
+ addCalendar(calendar: Calendar)
+ deleteCalendar(calendar: Calendar)
+ modifyCalendar(calendar: Calendar)
+ modifyGlobalPreferences(globalPreferences: GlobalPreferences)
<<Interface>>
CalendarQueryAPI
+ getCalendarResource()
+ getGlobalPreferencesResource()
+ getAllCalendarResources()
<<Interface>>
CalendarQuery
+ getCalendar(calendar: Calendar): Calendar
+ getAllCalendars(user: User): List<Calendar>
+ getGlobalPreferences(globalPreferences: GlobalPreferences): GlobalPreferences
Calendar Service

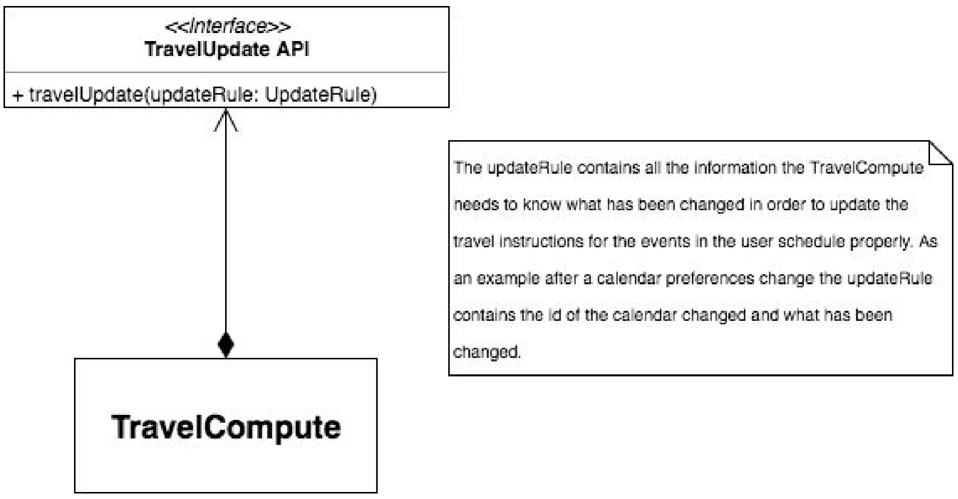
Event Service


<<Interface>>
EventCommandAPI
+ createEventResource(calendar: Calendar, name: String, location: Position, startTime: DateTime, endTime: DateTime, recurrenceRule: RecurrenceRule, nextIsBase: Boolean, flex: Flex)
+ modifyEventResource(calendar: Calendar, name: String, location: Position, startTime: DateTime, endTime: DateTime, recurrenceRule: RecurrenceRule, nextIsBase:Boolean, flex: Flex)
+ deleteEventResource(recurrenceDelete: Boolean)
<<Interface>>
EventEventSource
+ appendEventCreatedEvent(event: Event)
+ appendEventModifiedEvent(event: Event)
+ appendEventDeletedEvent(event: Event)
<<Interface>>
EventDatabaseUpdate
+ addEvent(event: Event)
+ modifyEvent(event: Event)
+ deleteEvent(event: Event)
<<Interface>>
EventQueryAPI
+ getEventResource()
+ getAllEventResources()
<<Interface>>
EventQuery
+ getEvent(event: Event): Event
+ getAllEvents(calendar: Calendar): List<Event>
Event Service

TravelInstructions Service

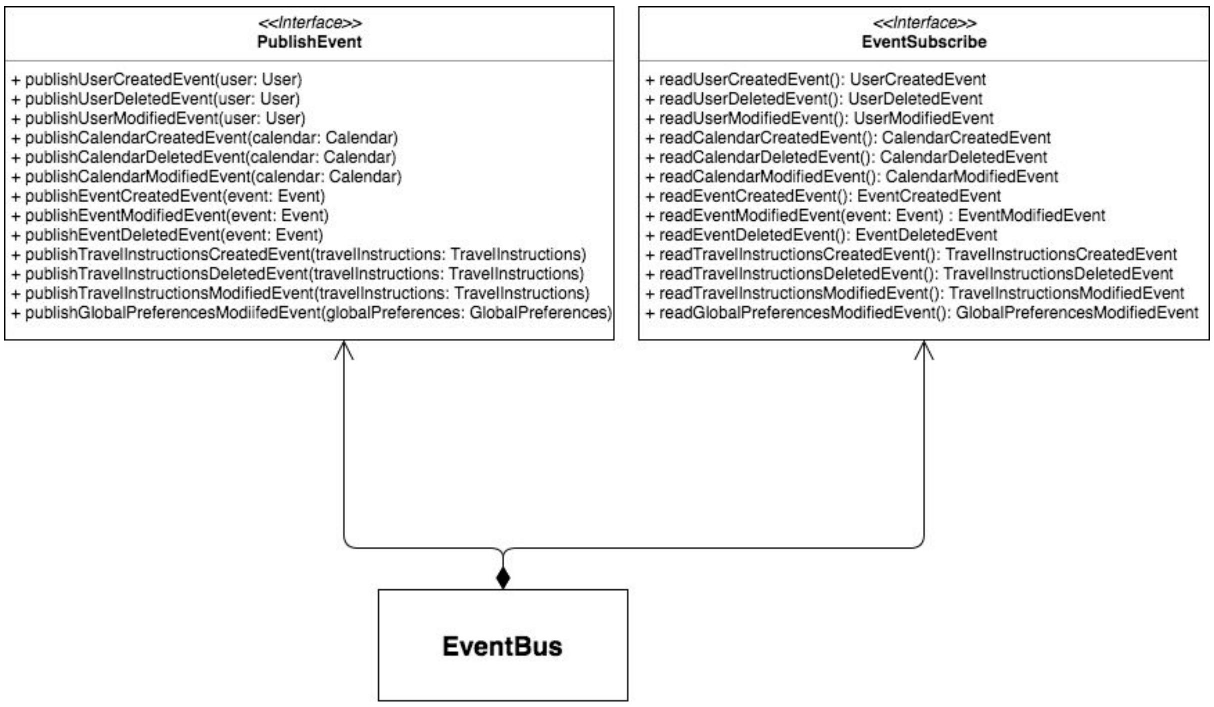

<<Interface>>
TravelInstructionsCommandAPI
+ modifyTravelInstructionsResource()
<<Interface>>
TravelInstructionsEventSource
+ appendTravelInstructionsCreatedEvent(travelInstructions: TravelInstructions)
+ appendTravelInstructionsModifiedEvent(travelInstructions: TravelInstructions)
+ appendTravelInstructionsDeletedEvent(travelInstructions: TravelInstructions)
<<Interface>>
TravelInstructionsDatabaseUpdate
+ createTravelInstructions(travelInstructions: TravelInstructions)
+ modifyTravelInstructions(travelInstructions: TravelInstructions)
+ deleteTravelInstructions(travelInstructions: TravelInstructions)
<<Interface>>
TravelInstructionsQueryAPI
+ getTravelInstructionsResource()
+ getAllTravelInstructionsResources()
<<Interface>>
TravelInstructionsQuery
+ getTravelInstructions(travelInstructions: TravelInstructions): TravelInstructions
+ getAllTravelInstructions(event: Event): List<TravelInstructions>
TravelInstructions Service

TravelCompute Service



The updateRule contains all the information the TravelCompute needs to know what has been changed in order to update the travel instructions for the events in the user schedule properly. As an example after a calendar preferences change the updateRule contains the id of the calendar changed and what has been changed.

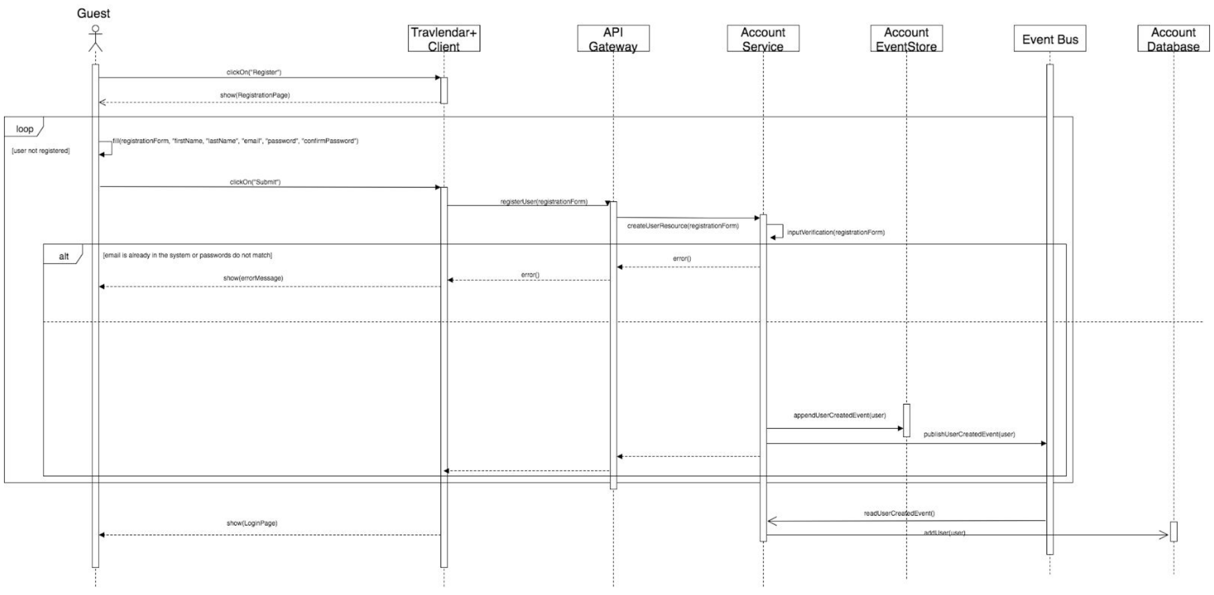
EventBus



<<Interface>>
PublishEvent
+ publishUserCreatedEvent(user: User)
+ publishUserDeletedEvent(user: User)
+ publishUserModifiedEvent(user: User)
+ publishCalendarCreatedEvent(calendar: Calendar)
+ publishCalendarDeletedEvent(calendar: Calendar)
+ publishCalendarModifiedEvent(calendar: Calendar)
+ publishEventCreatedEvent(event: Event)
+ publishEventModifiedEvent(event: Event)
+ publishEventDeletedEvent(event: Event)
+ publishTravelInstructionsCreatedEvent(travelInstructions: TravelInstructions)
+ publishTravelInstructionsDeletedEvent(travelInstructions: TravelInstructions)
+ publishTravelInstructionsModifiedEvent(travelInstructions: TravelInstructions)
+ publishGlobalPreferencesModiifedEvent(globalPreferences: GlobalPreferences)
<<Interface>>
EventSubscribe
+ readUserCreatedEvent(): UserCreatedEvent
+ readUserDeletedEvent(): UserDeletedEvent
+ readUserModifiedEvent(): UserModifiedEvent
+ readCalendarCreatedEvent(): CalendarCreatedEvent
+ readCalendarDeletedEvent(): CalendarDeletedEvent
+ readCalendarModifiedEvent(): CalendarModifiedEvent
+ readEventCreatedEvent(): EventCreatedEvent
+ readEventModifiedEvent(event: Event) : EventModifiedEvent
+ readEventDeletedEvent(): EventDeletedEvent
+ readTravelInstructionsCreatedEvent(): TravelInstructionsCreatedEvent
+ readTravelInstructionsDeletedEvent(): TravelInstructionsDeletedEvent
+ readTravelInstructionsModifiedEvent(): TravelInstructionsModifiedEvent
+ readGlobalPreferencesModifiedEvent(): GlobalPreferencesModifiedEvent
EventBus

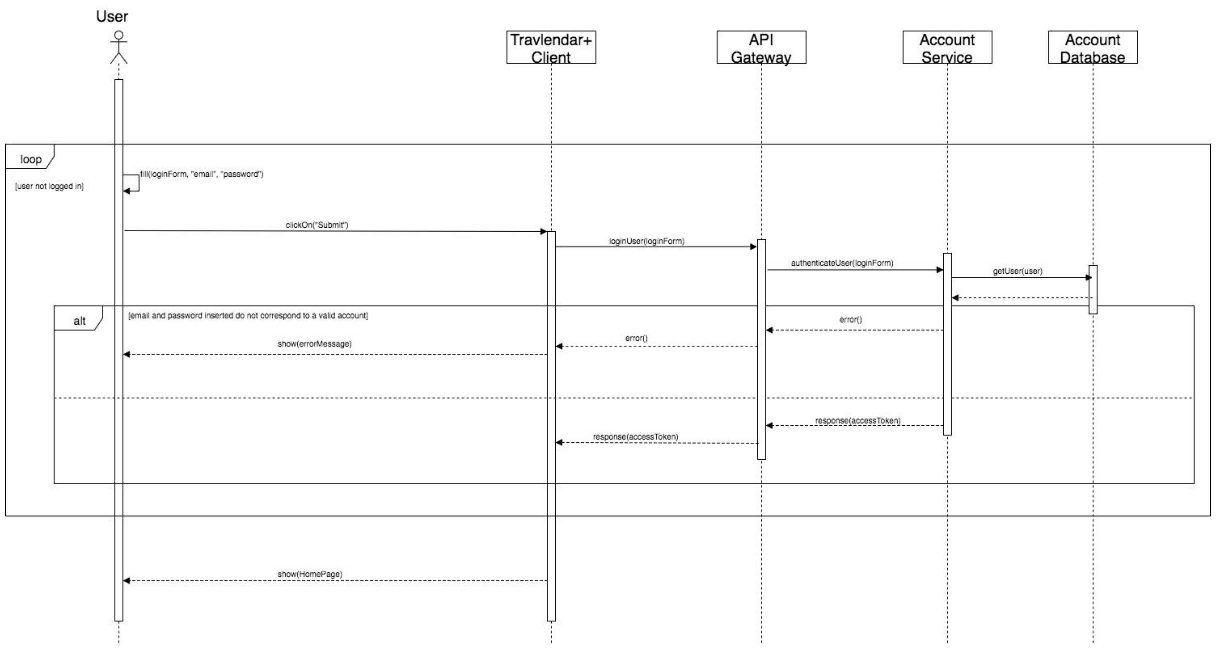
2.5 Runtime view

Guest Registers

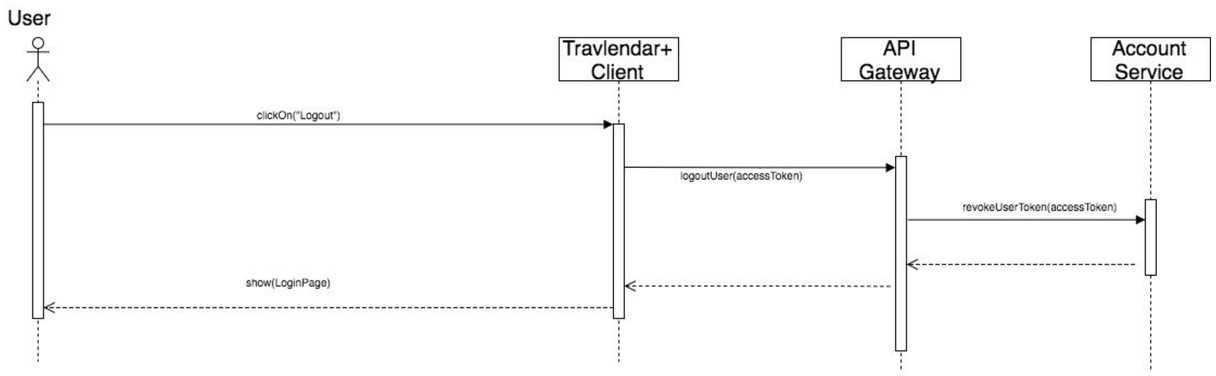


Guest
Travlendar+ Client
API Gateway
Account Service
Account EventStore
Event Bus
Account Database
clickOn("Register")
show(RegistrationPage)
loop
[user not registered]
fill(registrationForm, "firstName, "lastName", "email", "password", "confirmPassword")
clickOn("Submit")
registerUser(registrationForm)
createUserResource(registrationForm)
inputVerification(registrationForm)
alt
[email is already in the system or passwords do not match]
error()
error()
show(errorMessage)
appendUserCreatedEvent(user)
publishUserCreatedEvent(user)
readUserCreatedEvent()
addUser(user)
show(LoginPage)

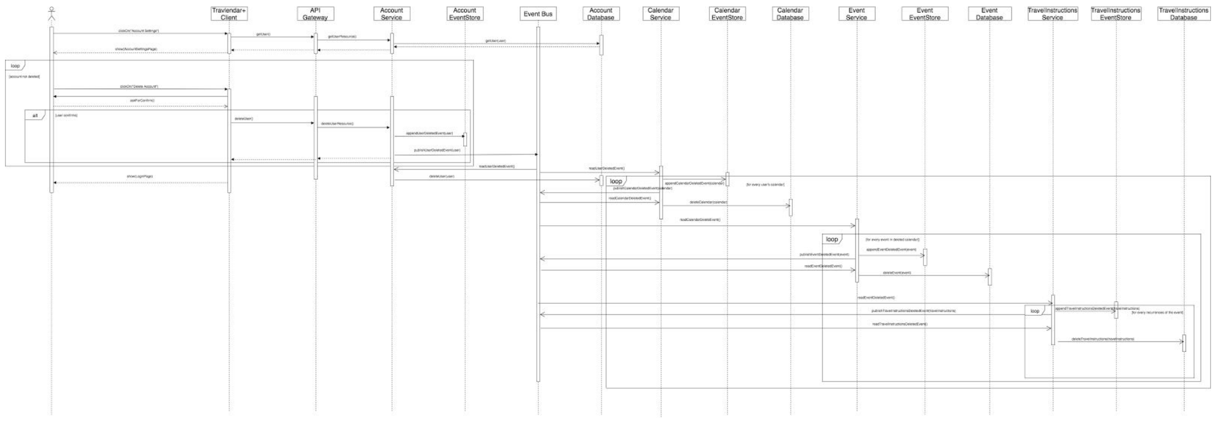
User logs in



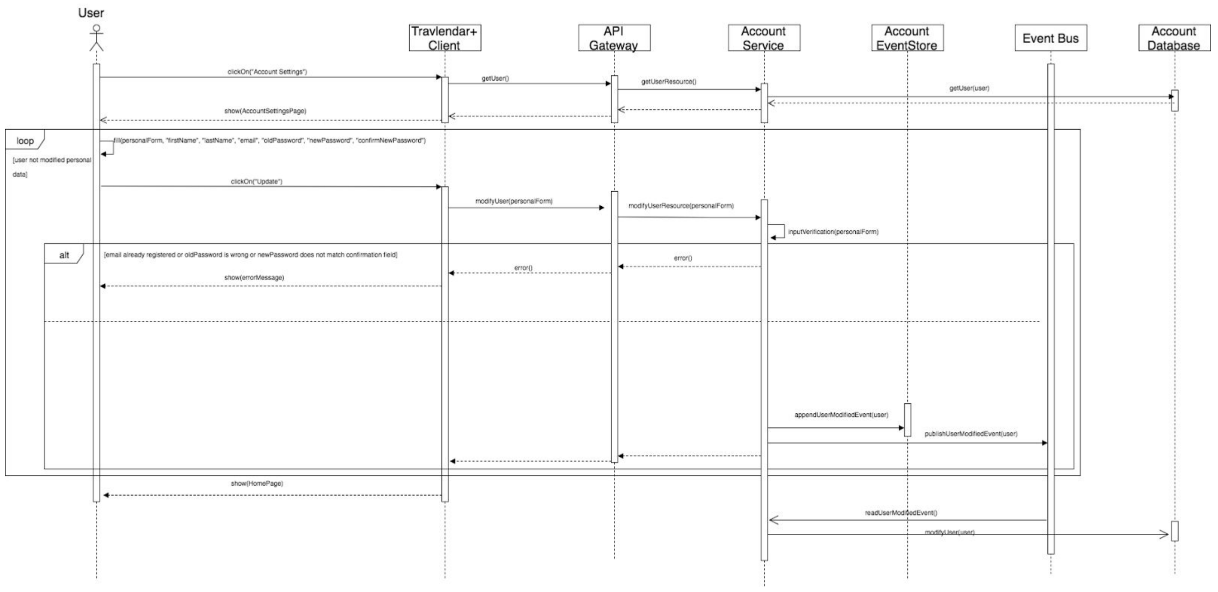
User logs out



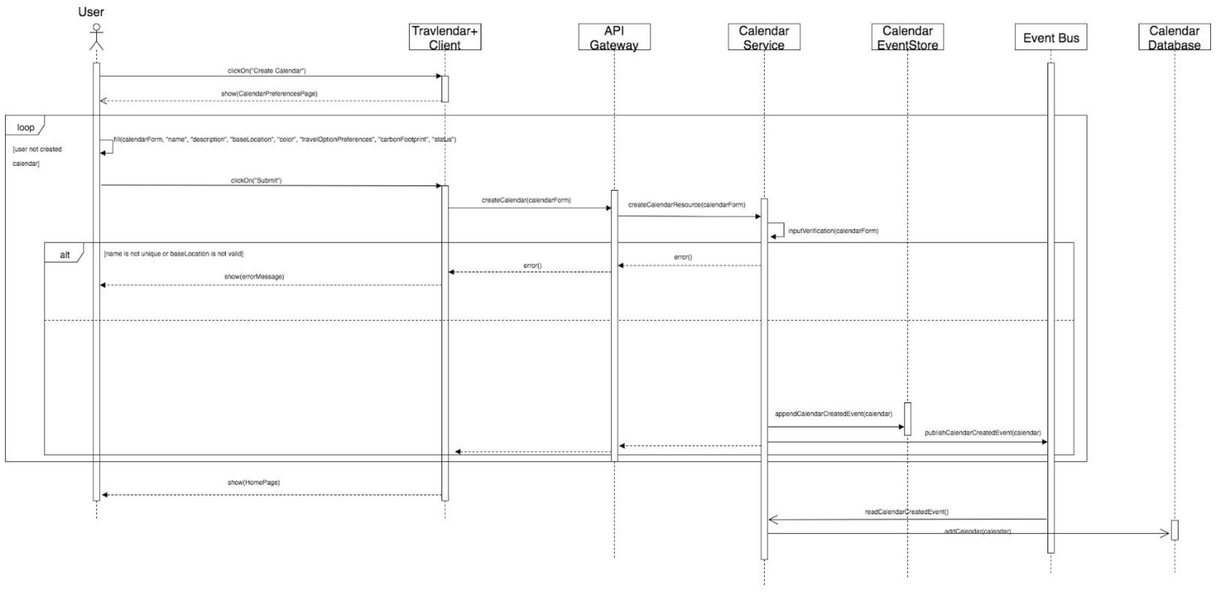
User deletes the account


Travlendar+ Client
API Gateway
Account Service
Account EventStore
Event Bus
Account Database
Calendar Service
Calendar EventStore
Calendar Database
Event Service
Event EventStore
Event Database
TravelInstructions Service
TravelInstructions EventStore
TravelInstructions Database
clickOn("Account Settings")
getUser()
getUserResource()
getUser(user)
show(AccountSettingsPage)
loop
[account not deleted]
clickOn("Delete Account")
askForConfirm()
alt
[user confirms]
deleteUser()
deleteUserResource()
appendUserDeletedEvent(user)
publishUserDeletedEvent(user)
readUserDeletedEvent()
deleteUser(user)
show(LoginPage)
loop
appendCalendarDeletedEvent(calendar)
[for every user's calendar]
publishCalendarDeletedEvent(calendar)
readCalendarDeletedEvent()
deleteCalendar(calendar)
readCalendarDeleteEvent()
loop
[for every event in deleted calendar]
appendEventDeletedEvent(event)
publishEventDeletedEvent(event)
readEventDeletedEvent()
deleteEvent(event)
readEventDeletedEvent()
loop
appendTravelInstructionsDeletedEvent(travelInstructions)
[for every recurrences of the event]
publishTravelInstructionsDeletedEvent(travelInstructions)
readTravelInstructionsDeletedEvent()
deleteTravelInstructions(travelInstructions)

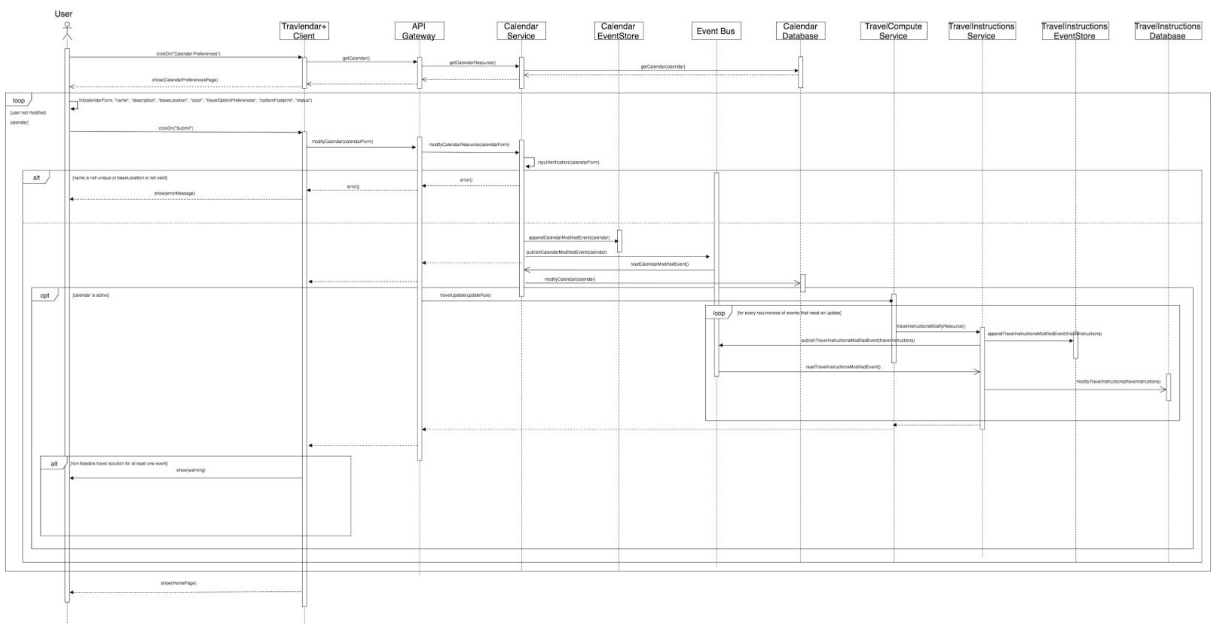
User modifies account personal data


User
Travlendar+ Client
API Gateway
Account Service
Account EventStore
Event Bus
Account Database
clickOn("Account Settings")
getUser()
getUserResource()
getUser(user)
show(AccountSettingsPage)
loop
[user not modified personal data]
fill(personalForm, "firstName", "lastName", "email", "oldPassword", "newPassword", "confirmNewPassword")
clickOn("Update")
modifyUser(personalForm)
modifyUserResource(personalForm)
inputVerification(personalForm)
alt
[email already registered or oldPassword is wrong or newPassword does not match confirmation field]
error()
error()
show(errorMessage)
appendUserModifiedEvent(user)
publishUserModifiedEvent(user)
show(HomePage)
readUserModifiedEvent()
modifyUser(user)

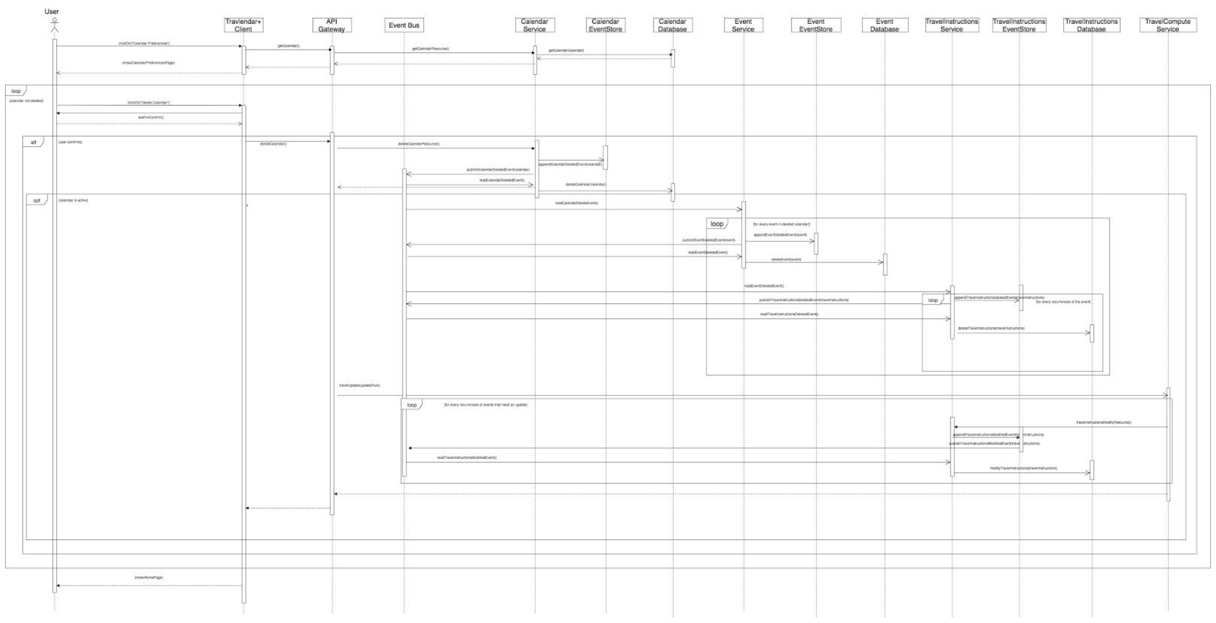
User creates a calendar


User
Travlendar+ Client
API Gateway
Calendar Service
Calendar EventStore
Event Bus
Calendar Database
clickOn("Create Calendar")
show(CalendarPreferencesPage)
loop
[user not created calendar]
fill(calendarForm, "name", "description", "baseLocation", "color", "travelOptionPreferences", "carbonFootprint", "status")
clickOn("Submit")
createCalendar(calendarForm)
createCalendarResource(calendarForm)
inputVerification(calendarForm)
alt
[name is not unique or baseLocation is not valid]
error()
error()
show(errorMessage)
appendCalendarCreatedEvent(calendar)
publishCalendarCreatedEvent(calendar)
show(HomePage)
readCalendarCreatedEvent()
addCalendar(calendar)

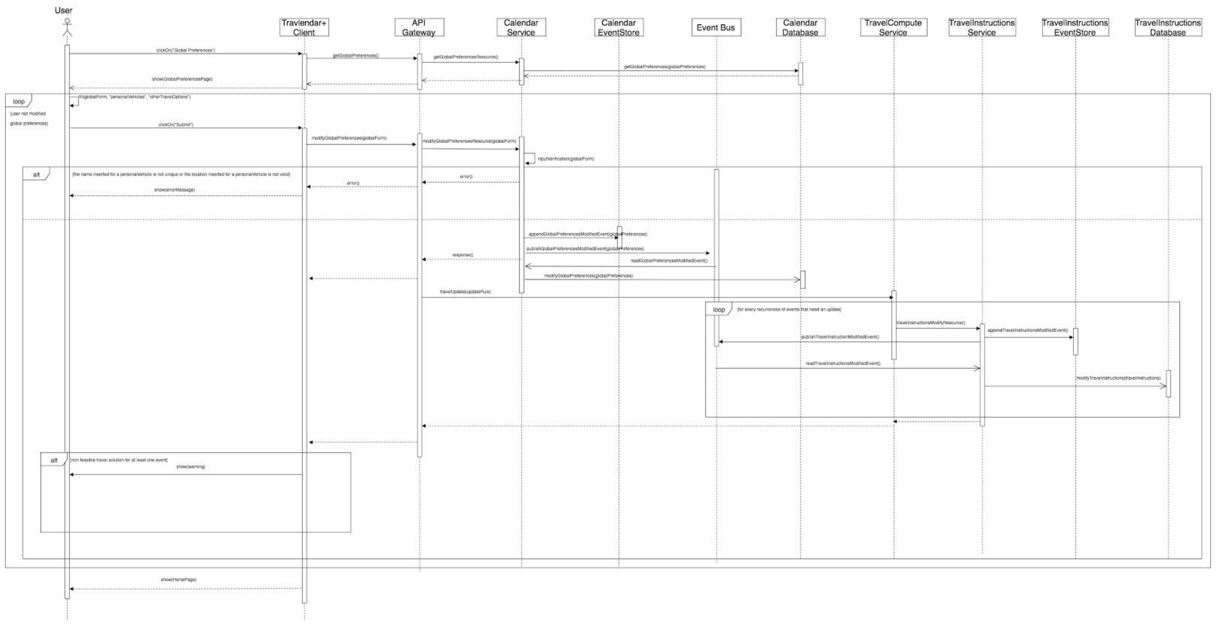
User changes calendar preferences


User
Travlendar+ Client
API Gateway
Calendar Service
Calendar EventStore
Event Bus
Calendar Database
TravelCompute Service
TravelInstructions Service
TravelInstructions EventStore
TravelInstructions Database
clickOn("Calendar Preferences")
getCalendar()
getCalendarResource()
getCalendar(calendar)
show(CalendarPreferencesPage)
loop
[user not modified calendar]
fill(calendarForm, "name", "description", "baseLocation", "color", "travelOptionPreferences", "carbonFootprint", "status")
clickOn("Submit")
modifyCalendar(calendarForm)
modifyCalendarResource(calendarForm)
inputVerification(calendarForm)
alt
[name is not unique or baseLocation is not valid]
error()
error()
show(errorMessage)
appendCalendarModifiedEvent(calendar)
publishCalendarModifiedEvent(calendar)
readCalendarModifiedEvent()
modifyCalendar(calendar)
opt
[calendar is active]
travelUpdate(updateRule)
loop
[for every recurrences of events that need an update]
travelInstructionsModifyResource()
appendTravelInstructionsModifiedEvent(travelInstructions)
publishTravelInstructionsModifiedEvent(travelInstructions)
readTravelInstructionsModifiedEvent()
modifyTravelInstructions(travelInstructions)
alt
[non feasible travel solution for at least one event]
show(warning)
show(HomePage)

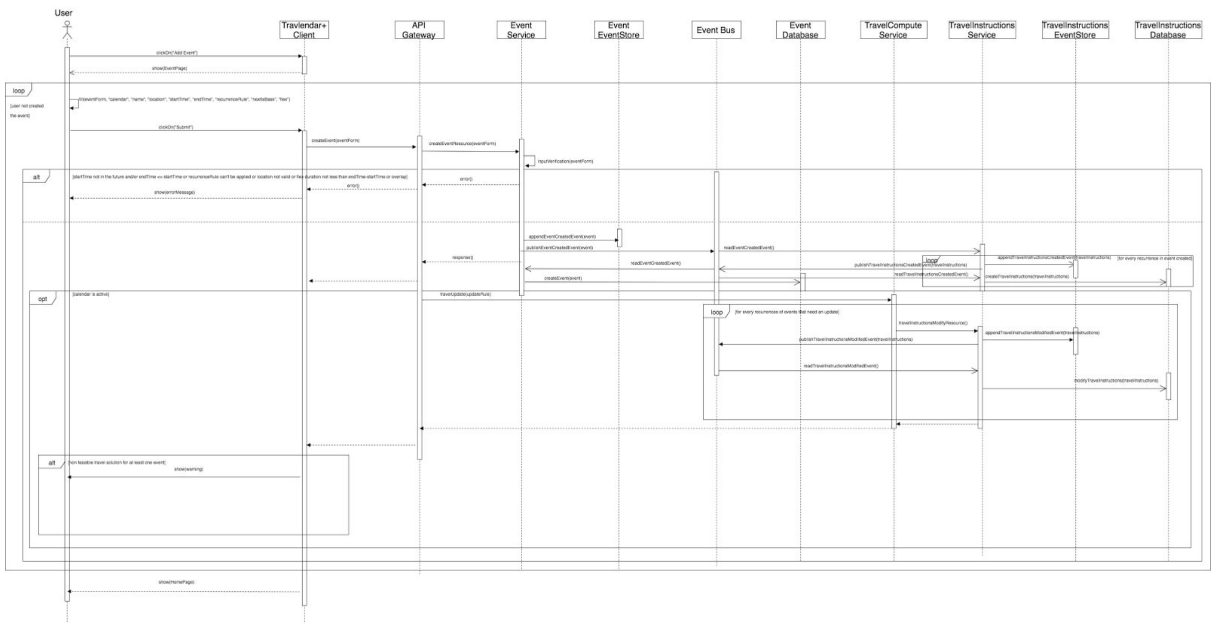
User deletes a calendar


User
Travlendar+ Client
API Gateway
Event Bus
Calendar Service
Calendar EventStore
Calendar Database
Event Service
Event EventStore
Event Database
TravelInstructions Service
TravelInstructions EventStore
TravelInstructions Database
TravelCompute Service
clickOn("Calendar Preferences")
getCalendar()
getCalendarResource()
getCalendar(calendar)
show(CalendarPreferencesPage)
loop
[calendar not deleted]
clickOn("Delete Calendar")
askForConfirm()
alt
[user confirms]
deleteCalendar()
deleteCalendarResource()
appendCalendarDeletedEvent(calendar)
publishCalendarDeletedEvent(calendar)
readCalendarDeletedEvent()
deleteCalendar(calendar)
opt
[calendar is active]
readCalendarDeletedEvent()
loop
[for every event in deleted calendar]
appendEventDeletedEvent(event)
publishEventDeletedEvent(event)
readEventDeletedEvent()
deleteEvent(event)
readEventDeletedEvent()
publishTravelInstructionsDeletedEvent(travelInstructions)
readTravelInstructionsDeletedEvent()
loop
appendTravelInstructionsDeletedEvent(travelInstructions)
[for every recurrences of the event]
deleteTravelInstructions(travelInstructions)
travelUpdate(updateRule)
loop
[for every recurrences of events that need an update]
travelInstructionsModifyResource()
appendTravelInstructionsModifiedEvent(travelInstructions)
publishTravelInstructionsModifiedEvent(travelInstructions)
readTravelInstructionsModifiedEvent()
modifyTravelInstructions(travelInstructions)
show(HomePage)

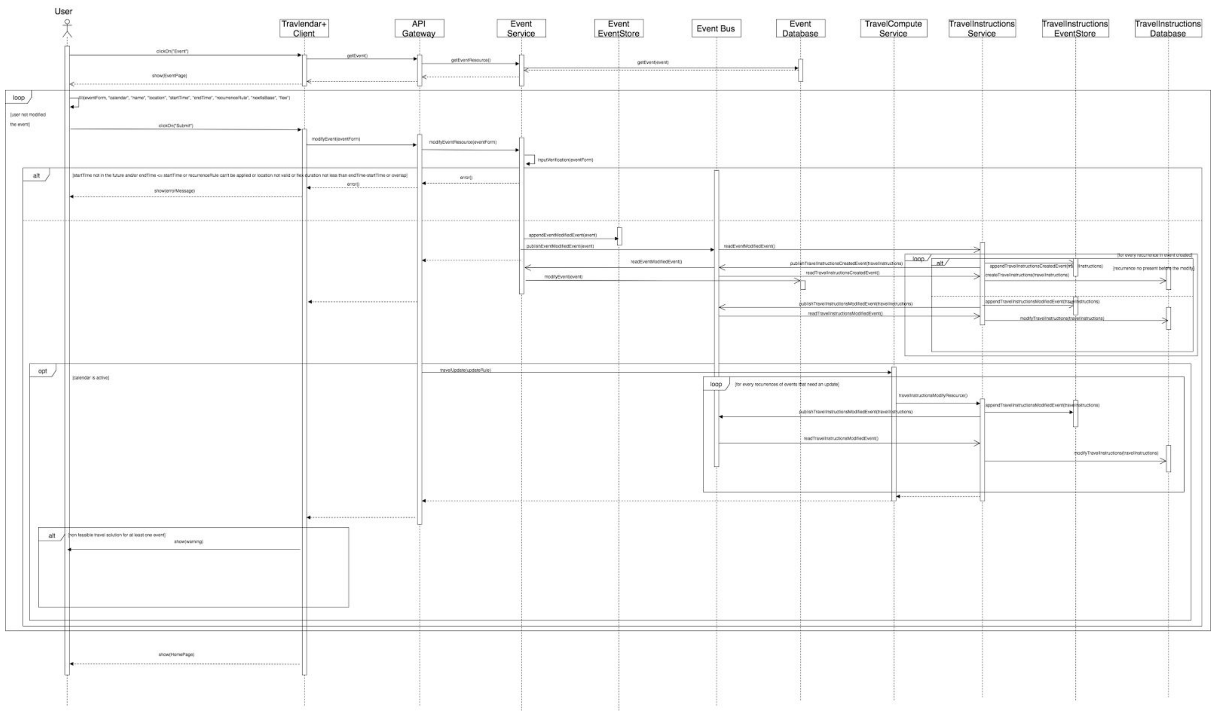
User changes global preferences


User
Travlendar+ Client
API Gateway
Calendar Service
Calendar EventStore
Event Bus
Calendar Database
TravelCompute Service
TravelInstructions Service
TravelInstructions EventStore
TravelInstructions Database
clickOn("Global Preferences")
getGlobalPreferences()
getGlobalPreferencesResource()
getGlobalPreferences(globalPreferences)
show(GlobalPreferencesPage)
loop
[user not modified global preferences]
fill(globalForm, "personalVehicles", "otherTravelOptions")
clickOn("Submit")
modifyGlobalPreferences(globalForm)
modifyGlobalPreferencesResource(globalForm)
inputVerification(globalForm)
alt
[the name inserted for a personalVehicle is not unique or the location inserted for a personalVehicle is not valid]
error()
error()
show(errorMessage)
appendGlobalPreferencesModifiedEvent(globalPreferences)
publishGlobalPreferencesModifiedEvent(globalPreferences)
response()
readGlobalPreferencesModifiedEvent()
modifyGlobalPreferences(globalPreferences)
travelUpdate(updateRule)
loop
[for every recurrences of events that need an update]
travelInstructionsModifyResource()
appendTravelInstructionsModifiedEvent()
publishTravelInstructionModifiedEvent()
readTravelInstructionsModifiedEvent()
modifyTravelInstructions(travelInstructions)
alt
[non feasible travel solution for at least one event]
show(warning)
show(HomePage)

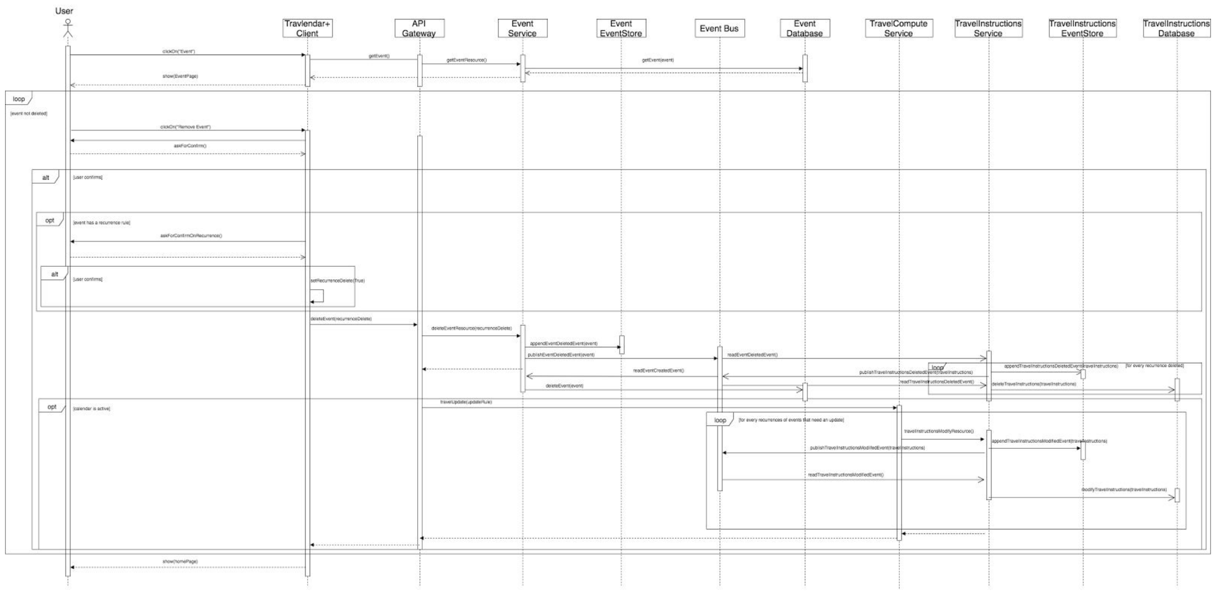
User adds new event in a calendar


User
Travlendar+ Client
API Gateway
Event Service
Event EventStore
Event Bus
Event Database
TravelCompute Service
TravelInstructions Service
TravelInstructions EventStore
TravelInstructions Database
clickOn("Add Event")
show(EventPage)
loop
[user not created the event]
fill(eventForm, "calendar", "name", "location", "startTime", "endTime", "recurrenceRule", "nextIsBase", "flex")
clickOn("Submit")
createEvent(eventForm)
createEventResource(eventForm)
inputVerification(eventForm)
alt
[startTime not in the future and/or endTime <= startTime or recurrenceRule can't be applied or location not valid or flex duration not less than endTime-startTime or overlap]
error()
error()
show(errorMessage)
appendEventCreatedEvent(event)
publishEventCreatedEvent(event)
readEventCreatedEvent()
response()
readEventCreatedEvent()
createEvent(event)
loop
[for every recurrence in event created]
appendTravelInstructionsCreatedEvent(travelInstructions)
publishTravelInstructionsCreatedEvent(travelInstructions)
readTravelInstructionsCreatedEvent()
createTravelInstructions(travelInstructions)
opt
[calendar is active]
travelUpdate(updateRule)
loop
[for every recurrences of events that need an update]
travelInstructionsModifyResource()
appendTravelInstructionsModifiedEvent(travelInstructions)
publishTravelInstructionsModifiedEvent(travelInstructions)
readTravelInstructionsModifiedEvent()
modifyTravelInstructions(travelInstructions)
alt
[non feasible travel solution for at least one event]
show(warning)
show(HomePage)

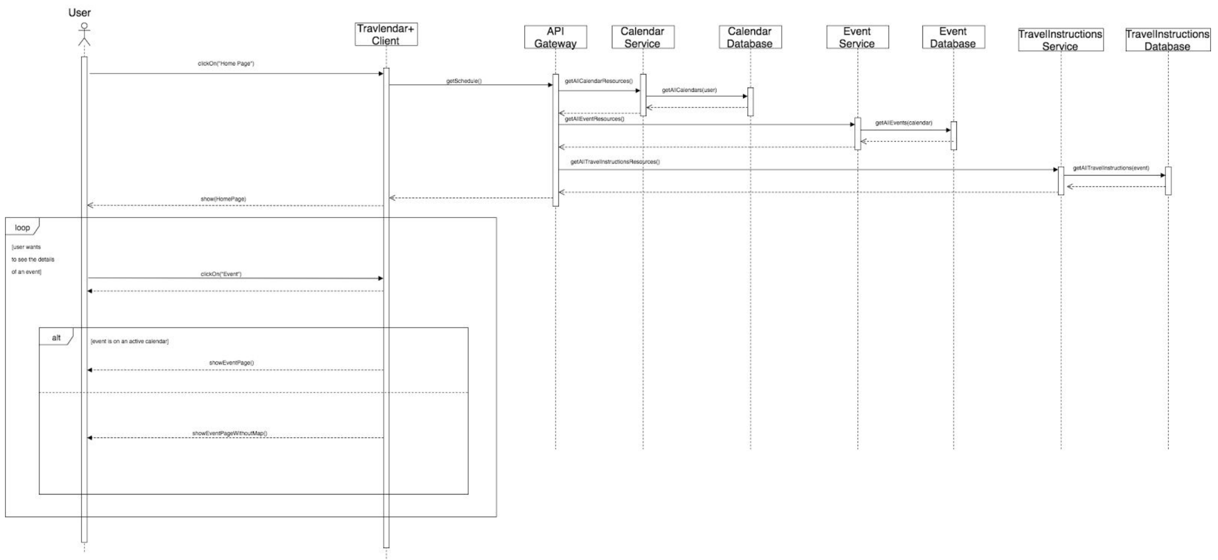
User modifies an event in a calendar


User
Travlendar+ Client
API Gateway
Event Service
Event EventStore
Event Bus
Event Database
TravelCompute Service
TravelInstructions Service
TravelInstructions EventStore
TravelInstructions Database
clickOn("Event")
getEvent()
getEventResource()
getEvent(event)
show(EventPage)
loop
[user not modified the event]
fill(eventForm, "calendar", "name", "location", "startTime", "endTime", "recurrenceRule", "nextIsBase", "flex")
clickOn("Submit")
modifyEvent(eventForm)
modifyEventResource(eventForm)
inputVerification(eventForm)
alt
[startTime not in the future and/or endTime <= startTime or recurrenceRule can't be applied or location not valid or flex duration not less than endTime-startTime or overlap]
error()
error()
show(errorMessage)
appendEventModifiedEvent(event)
publishEventModifiedEvent(event)
readEventModifiedEvent()
readEventModifiedEvent()
modifyEvent(event)
loop
alt
[for every recurrence in event created]
[recurrence no present before the modify]
publishTravelInstructionsCreatedEvent(travelInstructions)
readTravelInstructionsCreatedEvent()
appendTravelInstructionsCreatedEvent(travelInstructions)
createTravelInstructions(travelInstructions)
publishTravelInstructionsModifiedEvent(travelInstructions)
readTravelInstructionsModifiedEvent()
appendTravelInstructionsModifiedEvent(travelInstructions)
modifyTravelInstructions(travelInstructions)
opt
[calendar is active]
travelUpdate(updateRule)
loop
[for every recurrences of events that need an update]
travelInstructionsModifyResource()
appendTravelInstructionsModifiedEvent(travelInstructions)
publishTravelInstructionsModifiedEvent(travelInstructions)
readTravelInstructionsModifiedEvent()
modifyTravelInstructions(travelInstructions)
alt
[non feasible travel solution for at least one event]
show(warning)
show(HomePage)

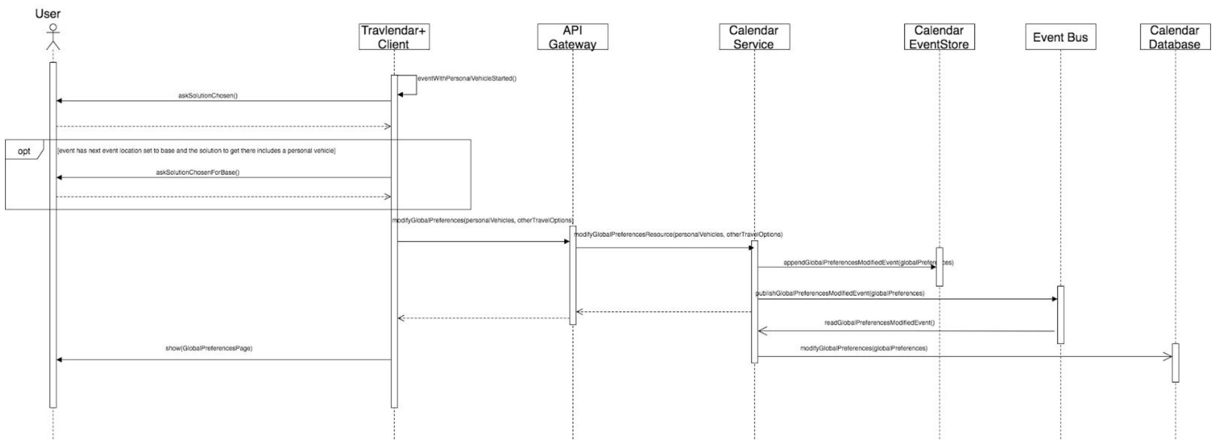
User deletes an event in a calendar


User
Travlendar+ Client
API Gateway
Event Service
Event EventStore
Event Bus
Event Database
TravelCompute Service
TravelInstructions Service
TravelInstructions EventStore
TravelInstructions Database
clickOn("Event")
getEvent()
getEventResource()
getEvent(event)
show(EventPage)
loop
[event not deleted]
clickOn("Remove Event")
askForConfirm()
alt
[user confirms]
opt
[event has a recurrence rule]
askForConfirmOnRecurrence()
alt
[user confirms]
setRecurrenceDelete(True)
deleteEvent(recurrenceDelete)
deleteEventResource(recurrenceDelete)
appendEventDeletedEvent(event)
publishEventDeletedEvent(event)
readEventDeletedEvent()
loop
appendTravelInstructionsDeletedEvent(travelInstructions)
[for every recurrence deleted]
publishTravelInstructionsDeletedEvent(travelInstructions)
readEventCreatedEvent()
readTravelInstructionsDeletedEvent()
deleteEvent(event)
deleteTravelInstructions(travelInstructions)
opt
[calendar is active]
travelUpdate(updateRule)
loop
[for every recurrences of events that need an update]
travelInstructionsModifyResource()
appendTravelInstructionsModifiedEvent(travelInstructions)
publishTravelInstructionsModifiedEvent(travelInstructions)
readTravelInstructionsModifiedEvent()
modifyTravelInstructions(travelInstructions)
show(homePage)

User sees all the events set


User
Travlendar+ Client
API Gateway
Calendar Service
Calendar Database
Event Service
Event Database
TravelInstructions Service
TravelInstructions Database
clickOn("Home Page")
getSchedule()
getAllCalendarResources()
getAllCalendars(user)
getAllEventResources()
getAllEvents(calendar)
getAllTravelInstructionsResources()
getAllTravelInstructions(event)
show(HomePage)
loop
[user wants to see the details of an event]
clickOn("Event")
alt
[event is on an active calendar]
showEventPage()
showEventPageWithoutMap()

User tells the system his/her personal vehicles movements


User
Travlendar+ Client
API Gateway
Calendar Service
Calendar EventStore
Event Bus
Calendar Database
eventWithPersonalVehicleStarted()
askSolutionChosen()
opt
[event has next event location set to base and the solution to get there includes a personal vehicle]
askSolutionChosenForBase()
modifyGlobalPreferences(personalVehicles, otherTravelOptions)
modifyGlobalPreferencesResource(personalVehicles, otherTravelOptions)
appendGlobalPreferencesModifiedEvent(globalPreferences)
publishGlobalPreferencesModifiedEvent(globalPreferences)
readGlobalPreferencesModifiedEvent()
modifyGlobalPreferences(globalPreferences)
show(GlobalPreferencesPage)

2.6 Selected architectural styles and patterns

2.6.1 Core pattern: Microservice Architecture

While deploying a server-side application, many factors need to be taken into account, especially when future development is considered. The architecture needs to scale quickly, handle HTTP(s) requests, expose APIs for different clients and separate the logic of its internal components in order to achieve maximum dynamism and flexibility.
While it might make more challenging to set up the application at the beginning, the Microservice Architecture pattern is an elegant and intuitive solution that offers all features said above, with several benefits such as:

- Small size of each microservice: this is tremendously helpful during development (easier for developers to understand), furthermore it significantly speeds up startup and deployment times.
- Isolation: the clustering obtained by this pattern enable developers to work independently on different services, where each one can be scaled and deployed at a different pace from the other. Moreover, this separation is a safe net in case of service failure, since other services won't be affected (on the other side, a memory leak in a monolithic application might take down the whole system).
- Elimination of long-term commitment to a technology stack: when deploying a new service there is no need to stick to old technologies present. This allows the application to scale easily and to implement new features with the latest framework and technologies present on the market, without any need to change the ones used on the other services.

2.6.2 Security pattern: Access Tokens

Exposing the API Gateway as the single entry point for client requests, the problem of communicating the identity of the requestor to the services arises. By implementing Access Tokens, the identity of the client is securely passed across the system and the services can easily identify the requestor.

2.6.3 Deployment pattern: Service instance per container

Implementing the Microservices Architecture pattern, we must ensure that services are packaged and deployed in an organized fashion. By using a container to wrap each service, this pattern ensures that isolation (fundamental in resource monitoring and constrainment) is preserved and makes easy to start, stop and scale up the containers, while keeping the deployment costs very low.

2.6.4 External API pattern: API Gateway

Since we need to expose application details via a REST API, which needs to fetch data that is spread across multiple services, the sequent problem arises: how do the clients of a Microservices-based application access the individual services? Implementing an API gateway as the single entry point for all the clients is a flexible solution to this issue: some requests are simply proxied/routed to the appropriate service, others are fanned out to multiple services. In addition, it can be adapted to expose different APIs to different clients if implemented with client-specific adapter code.

2.6.5 Data management pattern: Database per service

In order to have services that are loosely coupled and that can be developed, deployed and scaled independently we keep each Microservice's persistent data private to that service and accessible only via its API.

2.6.6 Data management pattern: Event-driven architecture through Event sourcing

Given that the database system is implemented through the Database per service pattern, the system cannot anymore guarantee the consistency between databases and operations atomicity.
To solve the consistency problem by staying compatible with the Microservices pattern, it is necessary to implement our services in an Event-driven way.
As long as the atomicity issue is concerned, Event Sourcing achieves atomicity without 2PC by using a radically different, event-centric approach to persisting business entities. Rather than store the current state of an entity, the application stores a sequence of state-changing events. The application reconstructs an entity's current state by replaying the events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic.
It's implemented with an event store, which provides an API to let the services subscribe to it in a way that they can be notified whenever an event that they are interested to is published.
Furthermore, since this pattern handles entities as a list of events instead of static tables, it provides a 100% reliable audit log of the changes made to a business entity, with possibilities to create snapshots of the application.
It is worth mentioning the fact that the consistency achieved is *eventual*, since there is an inherent mismatch between the start time (when the event is fired) and the consumption time on the query side in CQRS.

2.6.7 Data management pattern: Command Query Responsibility Segregation (CQRS)

With Event sourcing and the Microservices patterns implemented, the data is no longer easily queried. In order to solve this issue, we need to split the application into two parts: one command-side, which is responsible of creating, updating and deleting requests and emitting events when data changes; one query-side, that handles and executes queries against one or more materialized views that are kept up to date by subscribing to the stream of events emitted when data changes.

3 Algorithm Design

Quick reference to methods and function not explicitly explained here:

- `Schedule.getTravelInstructions(Event event)`: this method yields all future recurrences that are linked to this event. It performs a check on the RecurrenceExceptions Table to see if an edit made to this event affects its replicas or not (the latter happens if the user decides to delete a single event without making any changes to the future ones).
- `TravelInstruction.update(TravelInstruction previous)`: this method internally calls Google Maps API to fetch the travel instructions needed to reach the destination. This function is also responsible to handle the *Flex* feature, making sure that the user is able to attend the event in a flexible sliding window.

In our analysis the last point is the true bottleneck of our service (since external API calls are the most time consuming), so even the slightest optimization done in order to avoid a call to this method will result in huge speed and efficiency improvements.
A good solution is to utilize a cache system to avoid computing the same travel instructions over and over for recurrent events: if an event has a weekly recurrence rule and the previous and next events are consistent through the calendar, for instance, the travel solution in this situation can be computed just once and then cached for future replicas, resulting in a single Google Maps API call.

3.1 Event insertion

After the event is added to the database, we need to edit the travel instruction (there will be more if this event has a recurrence rule set) in the schedule that is (are) linked to this event. This is done by calling the `update` method, which requires as parameter the `TravelInstruction` object representing the instruction to reach the *previous* event. Finally, we just need to update the instruction related to the next event in schedule, since after insertion the starting point has changed.
Temporal complexity: **O(n)**, with n being the number of future recurrences this event has**.**

```
function updateAfterEventInsertion(Event event, Schedule schedule):
        foreach travelInstruction in schedule.getTravelInstructions(event):
                previousTravelInstruction =
schedule.getPreviousTravelInstruction(travelInstruction);
                nextTravelInstruction =
        schedule.getNextTravelInstruction(travelInstruction);
                travelInstruction.update(previousTravelInstruction);
                nextTravelInstruction.update(travelInstruction);
```

3.2 Event modification

Event modification is very similar to event insertion, since the only interested events are the same as the 3.1. An optimization can be made if the user updates only the starting time (in this case we need to update just the instructions to reach the event that has been modified) or the ending time (which will affect just the very next event). At this purpose a `requiresFullUpdate()` method is created, which returns True only if the conditions explained above are not satisfied.
Temporal complexity: **O(n),** with n being the number of future recurrences this event has.

```
function updateAfterEventModification(Event event, Schedule schedule):
        if event.requiresFullUpdate():
                updateAfterEventInsertion(event, schedule); // same as 3.1
        else:
                bool start = event.modifiedStart();
                foreach travelInstruction in schedule.getTravelInstructions(event):
                        if start: // we just need to update the instructions for this event
                                previousTravelInstruction =
                schedule.getPreviousTravelInstruction(travelInstruction);
                                travelInstruction.update(previousTravelInstruction);
                        else:  // just need to update instructions for very next event
                                nextTravelInstruction =
                        schedule.getNextTravelInstruction(travelInstruction);
                                nextTravelInstruction.update(travelInstruction);
```

3.3 Event delete

This affects only the consecutive event and we need to modify only that one. Same as the first two algorithms shown, this functions are called only after the event object has been successfully updated in the database, to maintain correct consistency between events and travel instructions.
Temporal complexity: **O(n),** with n being the number of future recurrences this event has.

```
function updateAfterEventDeletion(Event deletedEvent, Schedule schedule):
        foreach travelInstruction in schedule.getTravelInstructions(event):
                nextTravelInstruction = schedule.getNextTravelInstruction(event);
                nextTravelInstruction.update(travelInstruction);
```

3.4 Modify calendar settings

The modification of calendar settings forces us to check all future events and update all the travel instructions. The calls to the update() method can be avoided only if the user turned off one or more travel means from the settings, without doing anything else. In this case, an update must be called only if one of the travel means used in the travel instruction is not available anymore inside the calendar options.
Temporal complexity: **O(n),** with n being the number of future travel instructions found across all events that belong to this calendar.

```
function updateAfterCalendarPreferencesModifcation(Calendar calendar, Schedule schedule):
        foreach travelInstruction in schedule.getAllCalendarTravelInstructions(calendar):
                boolean skip = !calendar.needsFullUpdate();
                for travelMean in travelInstruction.getTravelMeans():
                        if travelMean not in calendar.getAvailableMeans():
                                skip = false;
                if not skip:
                        previousTravelInstruction =
        schedule.getPreviousTravelInstruction(travelInstruction);
                        travelInstruction.update(previousTravelInstruction);
```

3.5 Update global preferences

Here we just need to call the method defined at the point 3.4 on all calendars, since enabling / disabling a travel mean in the global preferences will simply reflect on all calendars.
Temporal complexity: **O(n),** with n being the number of future travel instructions found across all events that belong to this calendar.

```
function updateAfterGlobalPreferencesModification(List<Calendar> calendars, Schedule
schedule):
        foreach calendar in calendars:
                updateAfterCalendarPreferencesModification(calendar, schedule);
```

4 Requirements Traceability

The following table provides a mapping between the architectural component defined in this document and the requirements, defined in the RASD, they fulfill.

DD Component	Requirements
Account Service	[R1][R2][R3][R4][R5][R6][R7][R8][R9][R10][R11][R12][R13]
Calendar Service	[R14][R15][R16][R17][R18][R19][R20][R21][R22][R23][R24][R25][R26][R27][R28][R29][R30][R31][R32][R33][R34][R35][R36][R37][R38][R39][R40][R76][R77]
Event Service	[R41][R42][R43][R44][R45][R46][R47][R48][R49][R50][R51][R52][R53][R54][R55][R56][R57][R58][R78][R79]
TravelInstructions Service	[R61][R62][R68][R80][R81][R82][R83][R84]
TravelCompute Service	[R59][R60][R63][R64][R65][R66][R67][R69][R70][R71][R72][R73][R74][R75]

5 Implementation, Integration and Test Plan

5.1 Implementation plan

With the microservices system, various teams can work simultaneously on different services, with asynchronous deployment rates. Thus, the focus on the implementation part is just in the set-up phase:

- Setup the containerization platform (Docker)
    - Create a container for each service
    - Setup communication ports for each container
    - Decide API structure and usage

- Setup the deployment and management platform (Kubernetes)
    - Container managing: cluster system (configure pods, IP addresses, ..)
    - Auto-healing: monitoring and health checks
    - Communication: API gateway, Security protocols, Event sourcing

Once the steps stated above are completed, the teams can work independently on their service(s) without the need to follow a strict implementation order plan.

5.2 Integration and testing plan

The bottom up approach fits perfectly with the Microservices architectural pattern, since the starting point of the development are the services, that are finally linked together to form the larger clusterized system.

Going into the implementation detail, we have:

1. API Gateway
2. Account service (with parallel implementation of security and authentication protocols via JWTs)
3. Calendar service
4. Event service
5. TravelInstructions service
6. TravelCompute service

The order of the points above is hierarchical, and is the most intuitive sequence (even though with the Microservices pattern is perfectly reasonable to integrate the services in any order).

As long as the testing part is concerned, for each service we have two testing phases:

- **Internal testing**: this is very similar to the standard testing procedure in monolithic systems, where the inner components of the system are tested.
  Focus: code and branch coverage
- **External testing:** this phase consists of integrating the new functionalities added into the service with the publisher-subscribe system.
  Focus: API coverage, Event sourcing.

Some standard testing techniques can be applied as well, in particular regarding the cluster structure of the Microservices architecture:

- **Load testing:** a series of multiple requests is forwarded to the system's entry point, testing if the load balancing functionality provided by Kubernetes works properly and the various requests are distributed correctly across the pods present.
- **Stress testing:** testing Kubernetes reaction after single node and\or entire service failure.
- **Kubernetes performance testing:** testing the ability of Kubernetes to increase replicas of podes inside the cluster nodes accordingly to workload growth.

Going into further detail inside the Event Sourcing pattern mentioned above, which necessary to keep consistent the data across all services, it will be developed in a "step-by-step" fashion. Every time a service has an update that requires a new event to be added in the publisher subscribe schema, this will be integrated and tested. Thus, the event-driven architecture will expand as the functionalities implemented in the services increase.

6 References

- Google Cloud Kubernetes Engine documentation: https://cloud.google.com/kubernetes-engine/docs/
- Kubernetes documentation: https://kubernetes.io/docs/home/
- Docker documentation: https://docs.docker.com/reference/

7 Effort Spent

Riccardo Novic
14/11 - 4 hours
19/11 - 5 hours
21/11 - 2 hours
23/11 - 3 hours
24/11 - 4 hours
25/11 - 4 hours
26/11 - 2 hours

Lorenzo Pratissoli
13/11 - 2 hours
14/11 - 2 hours
15/11 - 2 hours
18/11 - 7 hours
19/11 - 7 hours
20/11 - 2 hours
21/11 - 2 hours
25/11 - 4 hours
26/11 - 2 hours