POLITECNICO DI MILANO

TRAVLENDAR+

LIFE, SIMPLIFIED

SOFTWARE ENGINEERING 2

Design Document

Authors:
Alessandro ALFONSO
Fabrizio CARSENZUOLA
Filippo CREMONESE

Supervisor:
Elisabetta DI NITTO

November 26, 2017

Contents

1 Introduction

1.1 Revision History

Version	Date	Authors	Summary
1.0	-	Alessandro Alfonso, Fabrizio Carsenzuola, Filippo Cremonese	Living document

1.2 Purpose

This document describes the details of the proposed implementation of Travlendar+. The requirements for the system were described in the Requirement Analysis and Specifications Document which should be read before reading this document. Here we document and explain design and implementation choices, first in overview and then in detail. The target audience of this document are the future developers of the system, as well as the project owner, prof. Di Nitto.

1.3 Scope

Travlendar+ will be available on all the most used platforms (mobile and desktop). Other apps that partially cover Travlendar+ functionality already exist:

- calendar apps that provide weather information (e.g AccuWeather. See references 2.6.3)
- public transport assistants (e.g. Citymapper. See references 2.6.2)
- public transport ticketing apps (e.g. Trenord. See references 2.6.4)
- navigation apps (e.g. Google Maps. See references 2.6.1)
- taxi apps (e.g. MyTaxi, Uber. See references 2.6.9, 2.6.7)
- sharing mobility services applications (e.g. Enjoy, Mobike. See references 2.6.6, 2.6.8)

Altough applications such as Citymapper or Google Maps already offer an excellent user experience, none of those apps allow to seamlessly schedule an appointment while keeping track of weather and travel times. Moreover, ticketing apps for public transportation are often poorly built, which increases the need for the in-app ticket purchase functionality in Travlendar+. The existing sharing mobility services don't offer the possibility to automatically reserve the nearest vehicle at a certain date and time.

The system will need to interact with external entities and events beyond of our control (public transportation, sharing mobility services, weather) which can disrupt user experience. The specifications of the service are based upon the assumption that most of the time those external entities will behave as expected (documented in section 2.5 in the Requirement Analisys and Specifications Document). When one or some external services won't behave as expected, the application will work the same but in a degraded state. If it is possible the system should notify the user that the application will work in a limited way (and tells him which services are unaviable).

1.4 Reference documents

- RASD.pdf: Requirements Analysis and Specification Document
- "Mandatory Project Assignments.pdf": Project assignments

2 Architectural Design

2.1 Logical components overview

Travlendar+ will consist of several logical components, following a traditional multi-layer architecture:

- Clients
- Web layer
- Application logic layer
- Database layer
- External APIs and data sources

Clients

End users will be able to use Travlendar on their mobile devices and desktop computers. We will develop applications compatible with smartphones and tablets running the two most common operating systems, iOS and Android. A desktop web application will be developed using the React framework, and the same code will be reused to build a standalone desktop application with support for push notifications and better OS integration by leveraging the Electron framework.

Web layer

The web application will fetch resources (HTML, CSS, JS, images, ...) from a web server, which will implement no application logic, forwarding API requests to the appropriate backend service instead.

Application logic layer

Server side functionality will be implemented by services reachable as HTTP REST APIs. Each service will implement some specific functionality (see section 2.2.3 for more details on how the services will be split). Each service will depend only on the public APIs provided by other internal and external services, allowing for better reliability and availability. We will use Python and leverage the Flask framework.

Database layer

All the services will store and fetch data from a single logical relational database. We will use MariaDB as DBMS.

External APIs and data sources

Travlendar+ will rely on external services for:

- public transport information
- weather forecast
- map data

- traffic information
- find and reserve sharing mobility vehicles
- obtaining informations on public transportation tickets
- reserving taxis
- route planning
- notifications and emails
- processing payments

2.2 Component view

Each logical component is made of several subcomponents.

2.2.1 Clients

End users will be able to use Travlendar+ on

- smartphones and tablets running Android 4.4+
- iPhone and iPad running iOS 10+
- laptops/desktops browsers (Chrome, Firefox, Safari and Edge)
- desktop (Linux, macOS and Windows)

Android app The android app will be built using Kotlin, the new official language for building apps for the platform.

iOS App The iOS app will be built using Swift.

Desktop web and native apps The desktop web application will be built using the React framework. The desktop application will be built using Electron reusing most of the code of the web application, adding code to support native functionality such as OS integrated notifications.

2.2.2 Web layer

One or more Nginx load balancers will distribute requests to other Nginx web servers which will serve the web application resources (HTML, CSS, JS, ...) and proxy incoming requests to the Python HTTP APIs.

2.2.3 Application logic layer

- User management service
  - Authentication
  - Sign up
  - Password reset
  - User preferences
- Calendar management service
  - Manages calendars and events
- Travel planning service
  - Provides travel suggestions
  - Uses external API to get itineraries, maps, and travel means information
- Event notifications service
  - Generates notifications related to calendar events
- Travel notifications service
  - Generates travel-related notifications (e.g. if a travel needs to be rescheduled due to a strike)
  - Uses external APIs to get travel means informations and weather
- Notification delivery service
  - Pushes notifications to users via email or push
- Taxi reservation service
- Mobility sharing reservation service
  - Searches for a vehicle to reserve before the start of a trip
- Ticket shop service
  - Links our service, public transport ticketing service and payment processor

2.2.4 Database tier

We will use MariaDB as DBMS for storing data.

2.2.5 Component view diagrams

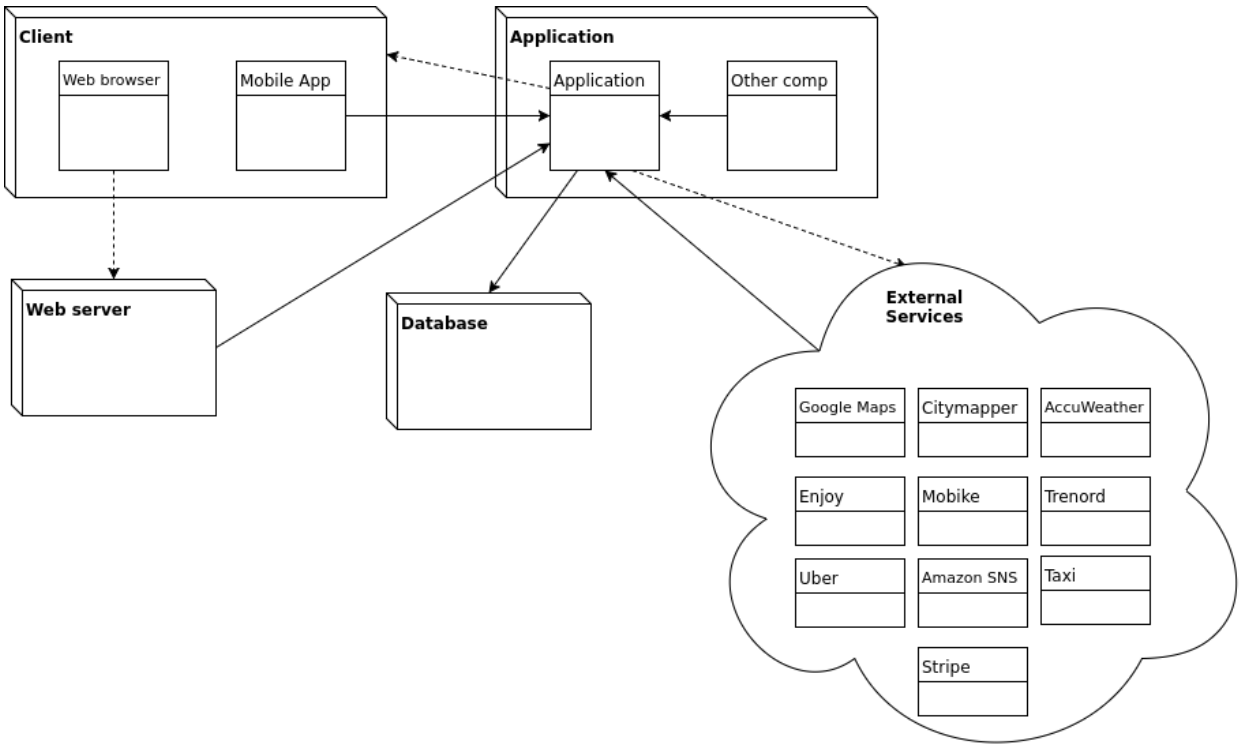


Figure 1: High Level Component View

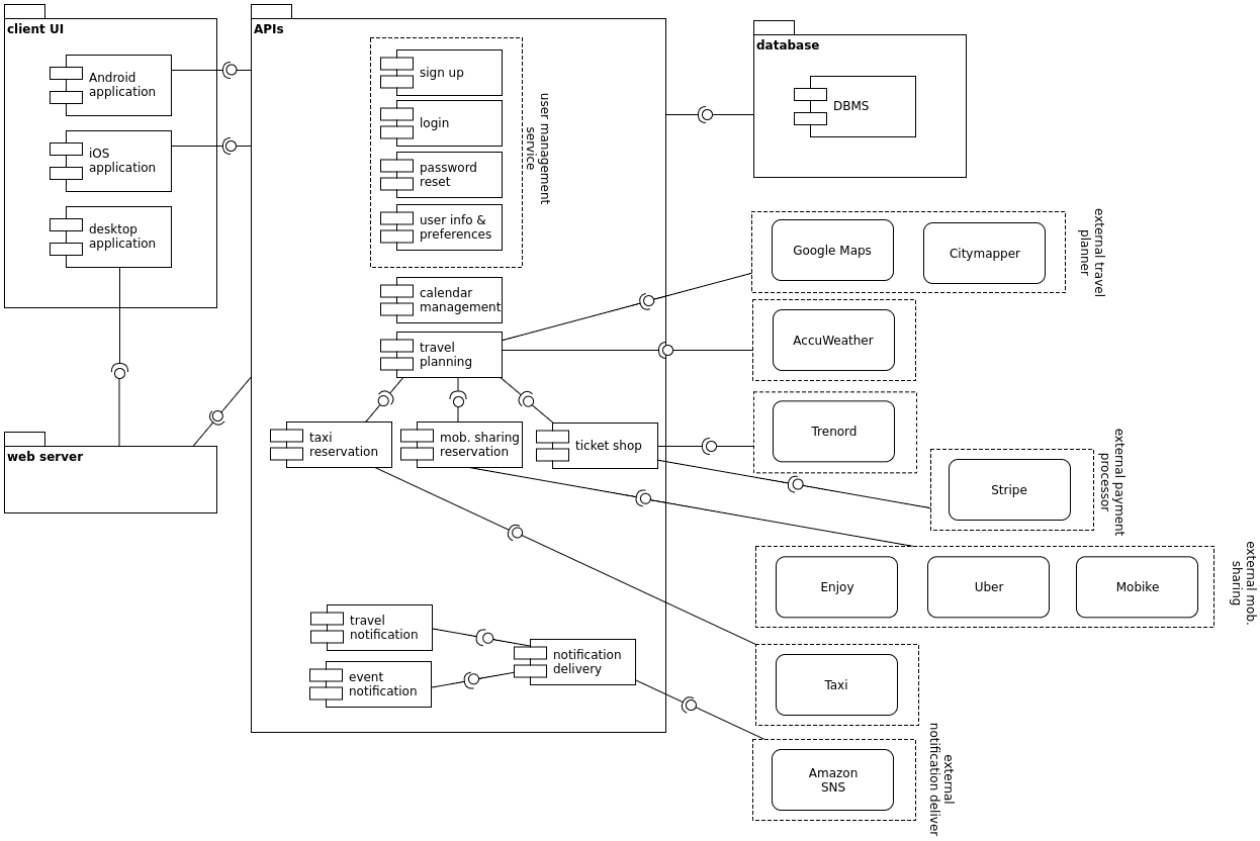


Figure 2: Component View

2.3 Deployment view

In this section we show how we plan to map the software components of our system on physical machines, planning for future scalability.

2.3.1 Web tier

In a first moment we will deploy a single Nginx web server and there will be no load balancer. One or more machines will be provisioned as load balancers when more capacity will be needed. Requests to the load balancers will be balanced via DNS, and the load balancers themselves will distribute load uniformly to the web and application servers in a way that a single user requests will always reach the same backend server (so to avoid the problem of distributing server-local storage).

2.3.2 Application logic tier

The various Python services implementing Travlendar+ APIs will initially run on the same machine hosting the Nginx web server, under the gunicorn application server. When the need will arise the most demanding

services will be moved to one or more dedicated machines and load will be split between them, allowing for a gradual transition to a distributed system.

2.3.3 Database tier

Initially we will deploy a single instance of the MariaDB RDBMS on a single machine. We expect to reach the machine limits in read performance as our user base grows, while we don't expect write performance to pose a problem in the near future. We will monitor load and deploy more machines as slave/read-only servers, and distribute reads between them. We anticipate the possibility of sharding the database on user id when the write performance will become insufficient, as different users interact sparely and mostly in read only way on other user's data.

2.3.4 Other notes

Initially there will be no dedicated firewall applicance. Depending on need (e.g. if the service will be subject of DDOS attacks) a dedicated appliance could be installed in front of the load balancers, or a service such as CloudFlare could be used to mitigate attacks.

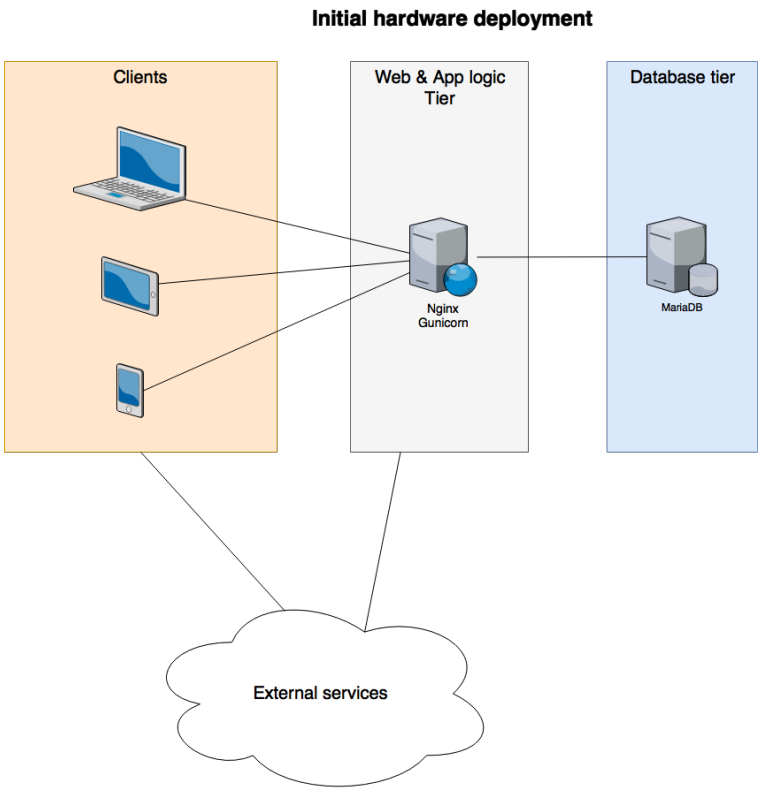


Figure 3: Initial Deployment View

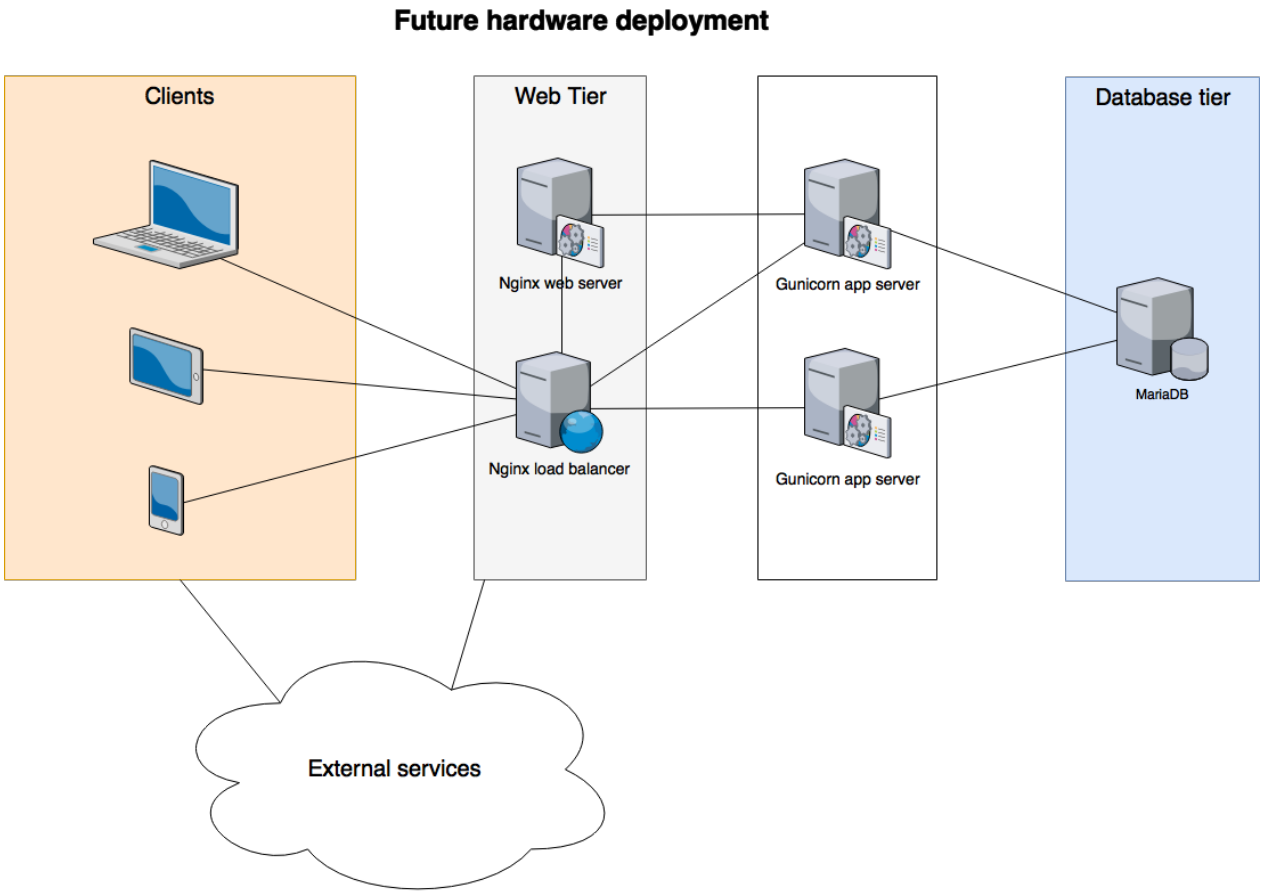
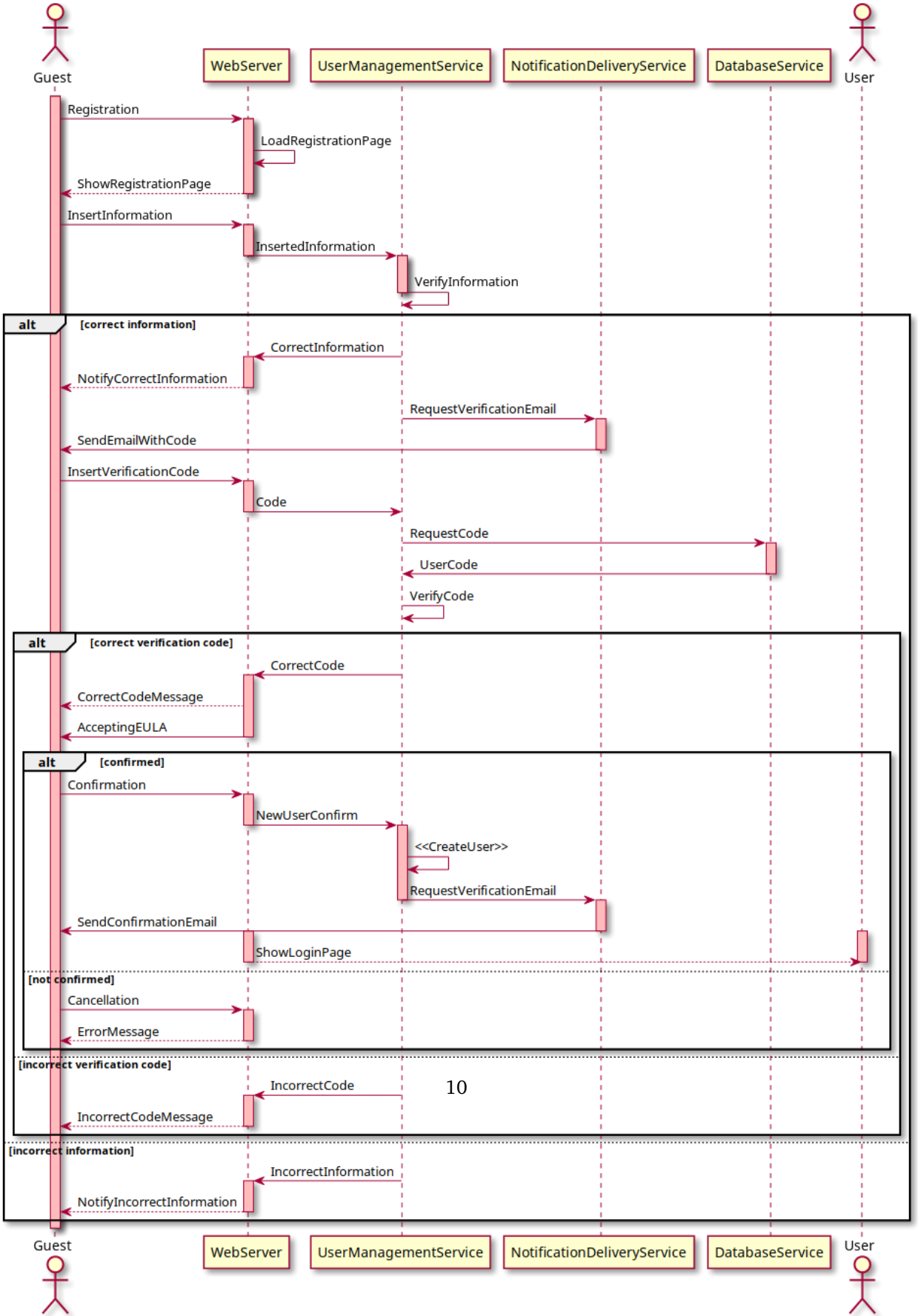


Figure 4: Future Deployment View

2.4 Runtime view

In this section we use sequence diagrams to explain some of the most important interactions between components and actors of the system. The actor is using the web application.

Signup Sequence Diagram



Billing Information After Signup Sequence Diagram

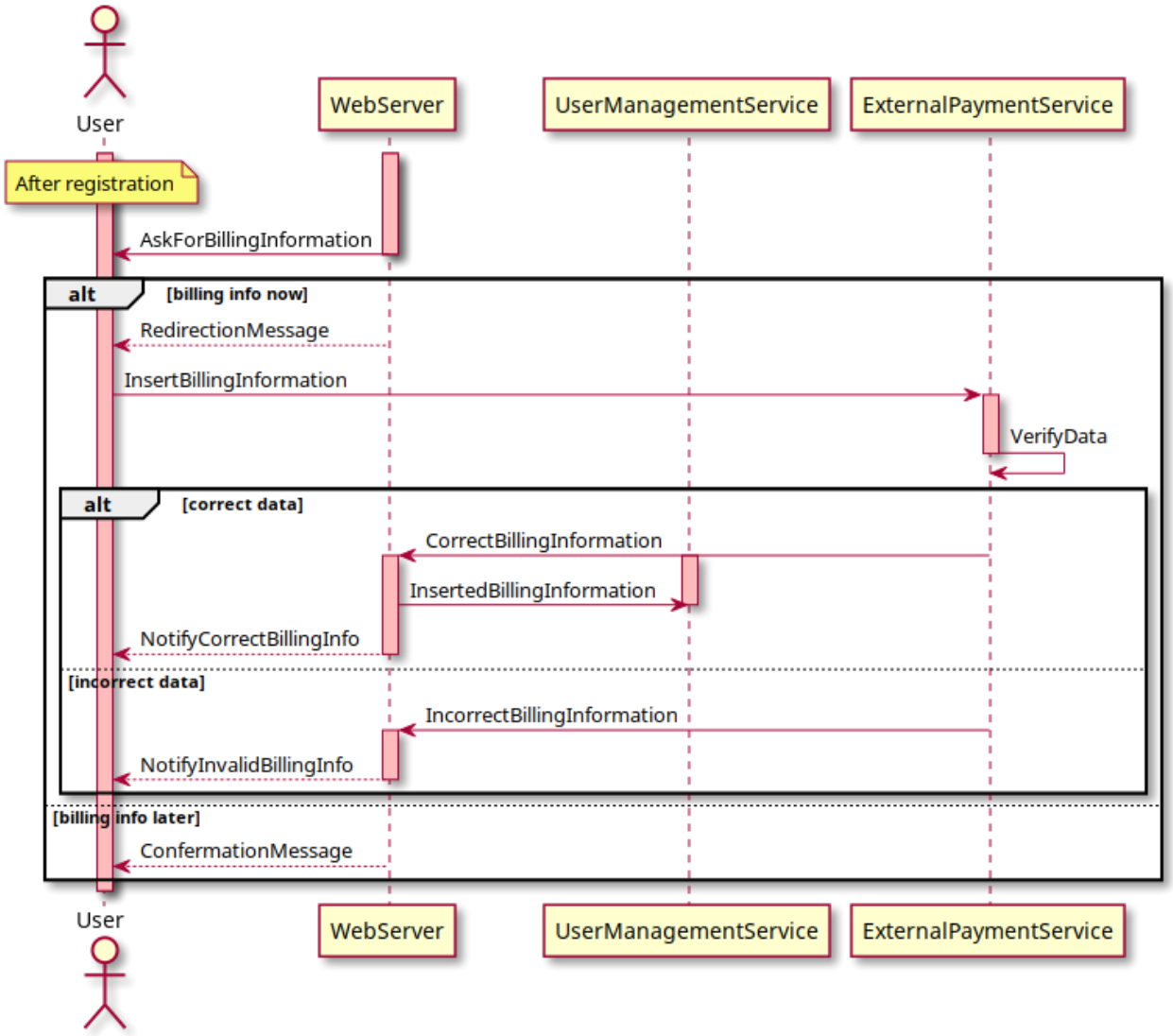


Figure 6: Billing Information After Signup Sequence Diagram

Login Sequence Diagram

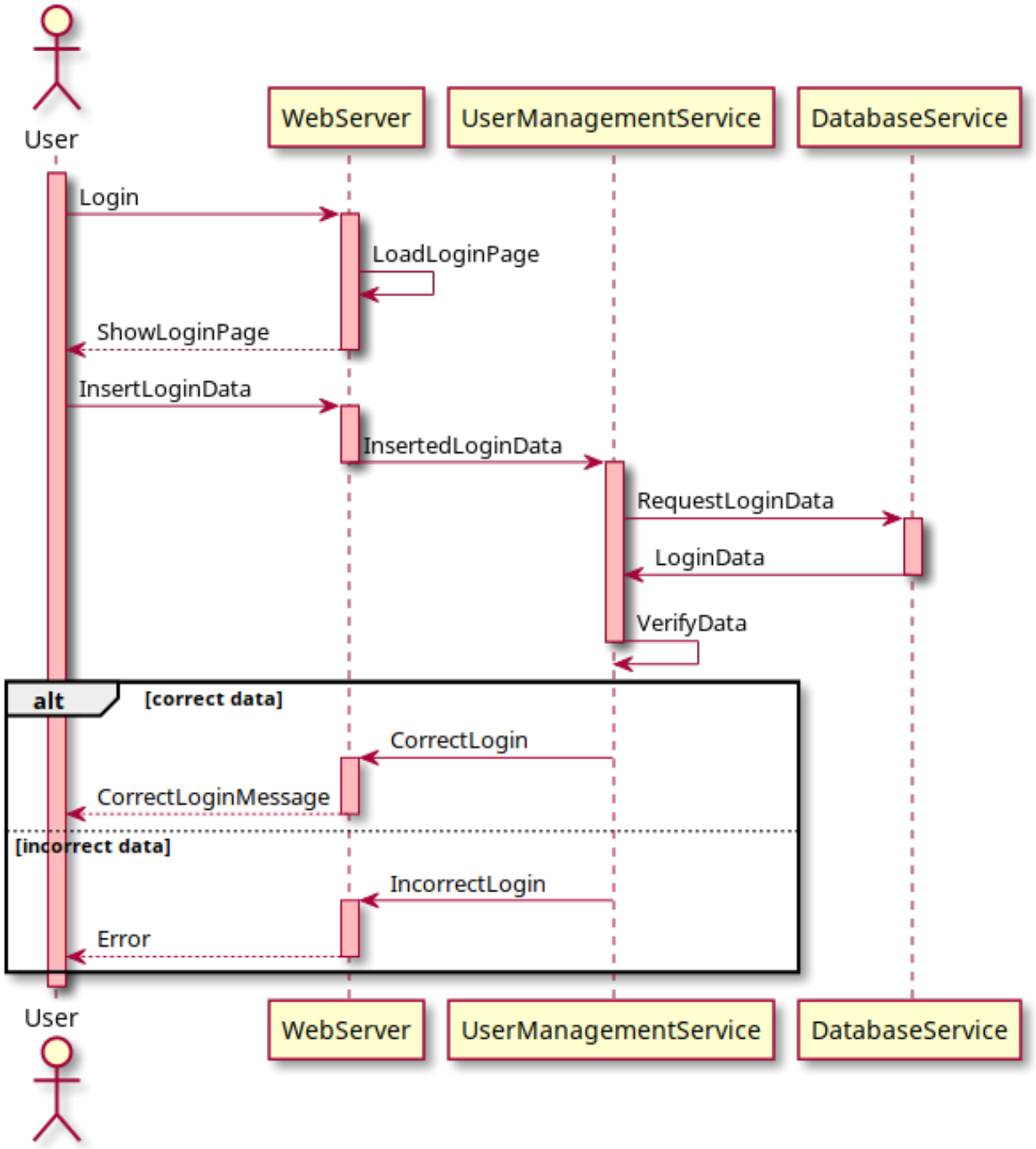


Figure 7: Login Sequence Diagram

Edit Personal Information Sequence Diagram

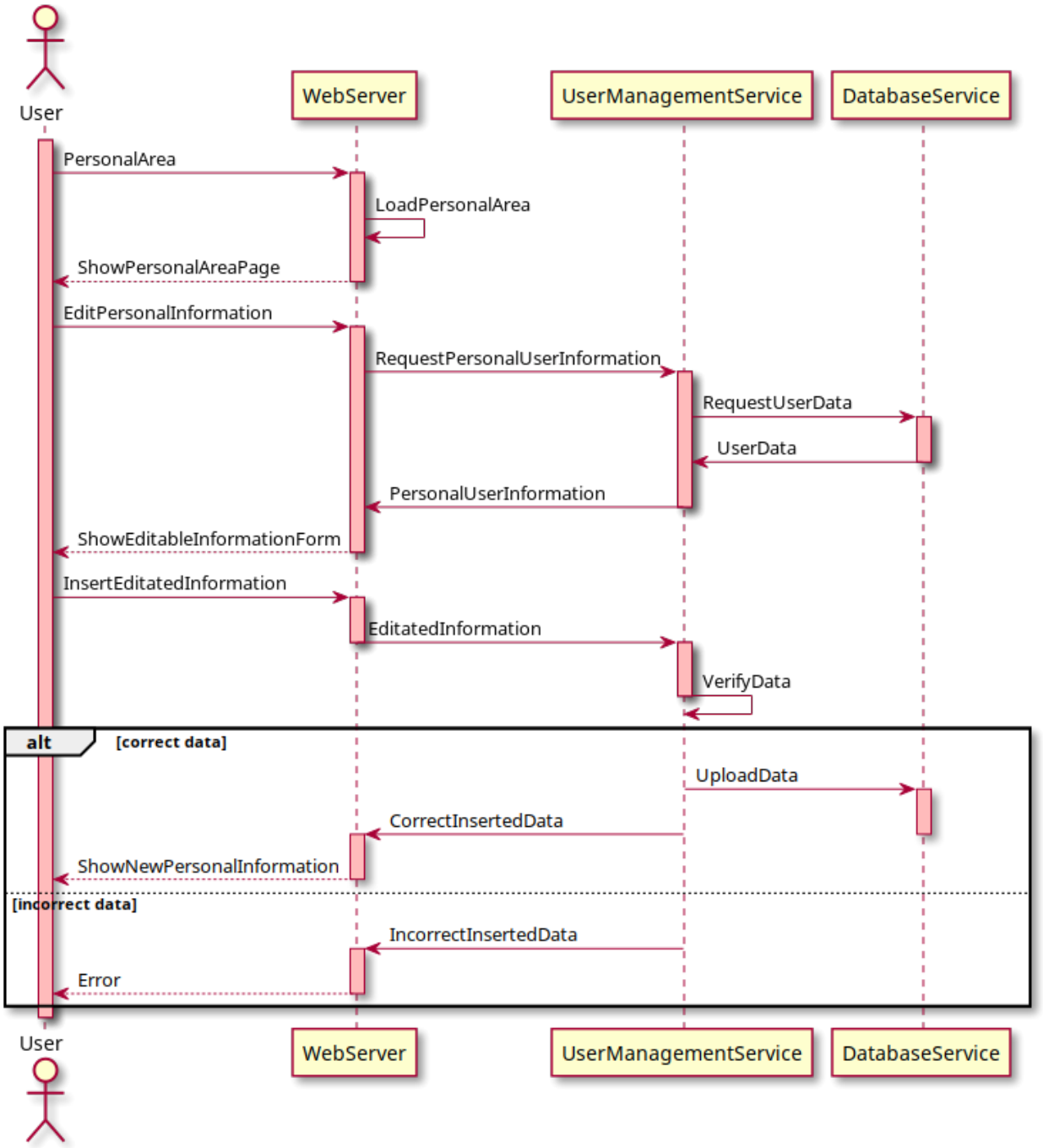


Figure 8: Edit Personal Information Sequence Diagram

Add/Edit Travel Preferences Sequence Diagram

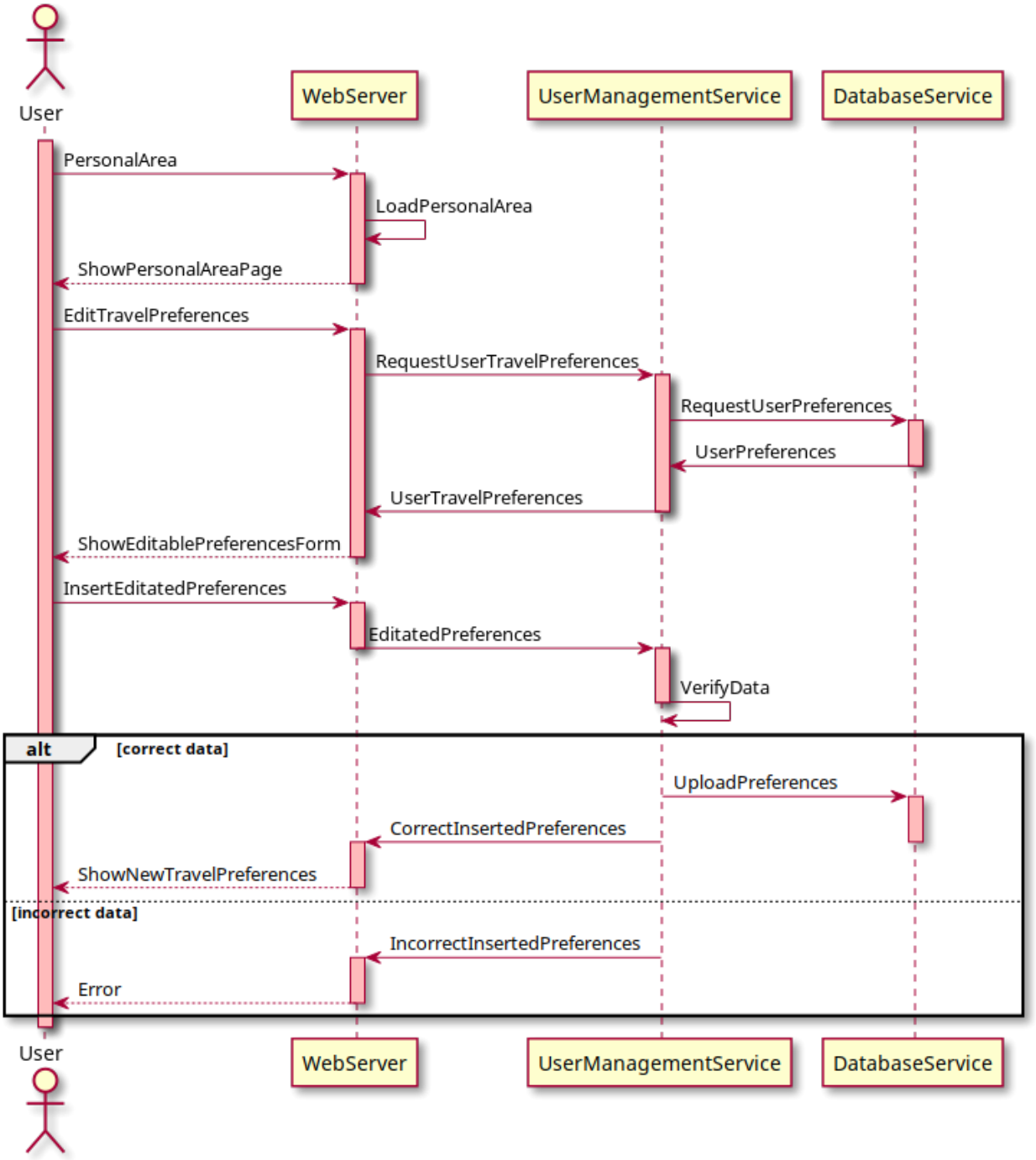


Figure 9: Add/Edit Travel Preferences Sequence Diagram

Edit Billing Information Sequence Diagram

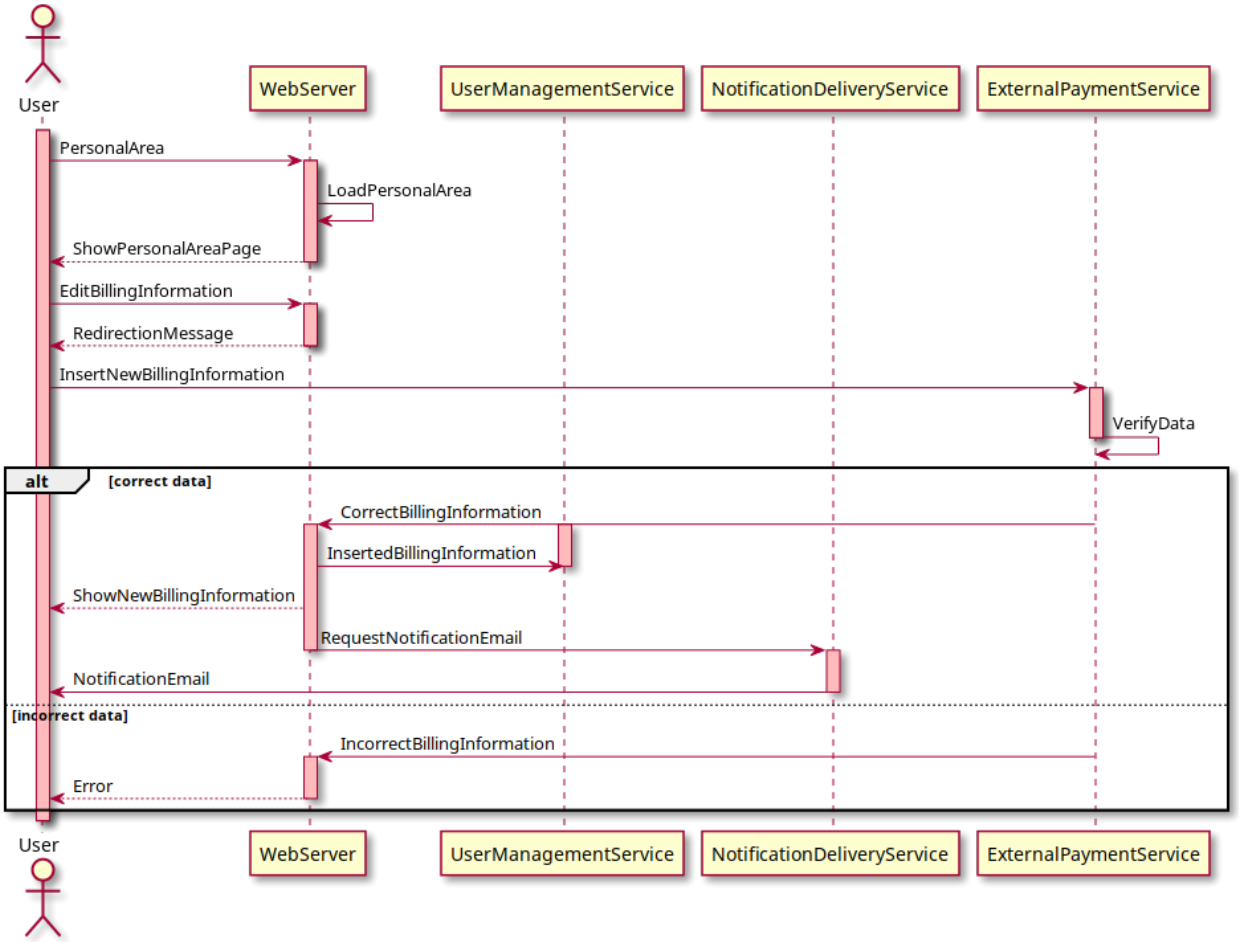


Figure 10: Edit Billing Information Sequence Diagram

Buy Ticket/Pass Sequence Diagram

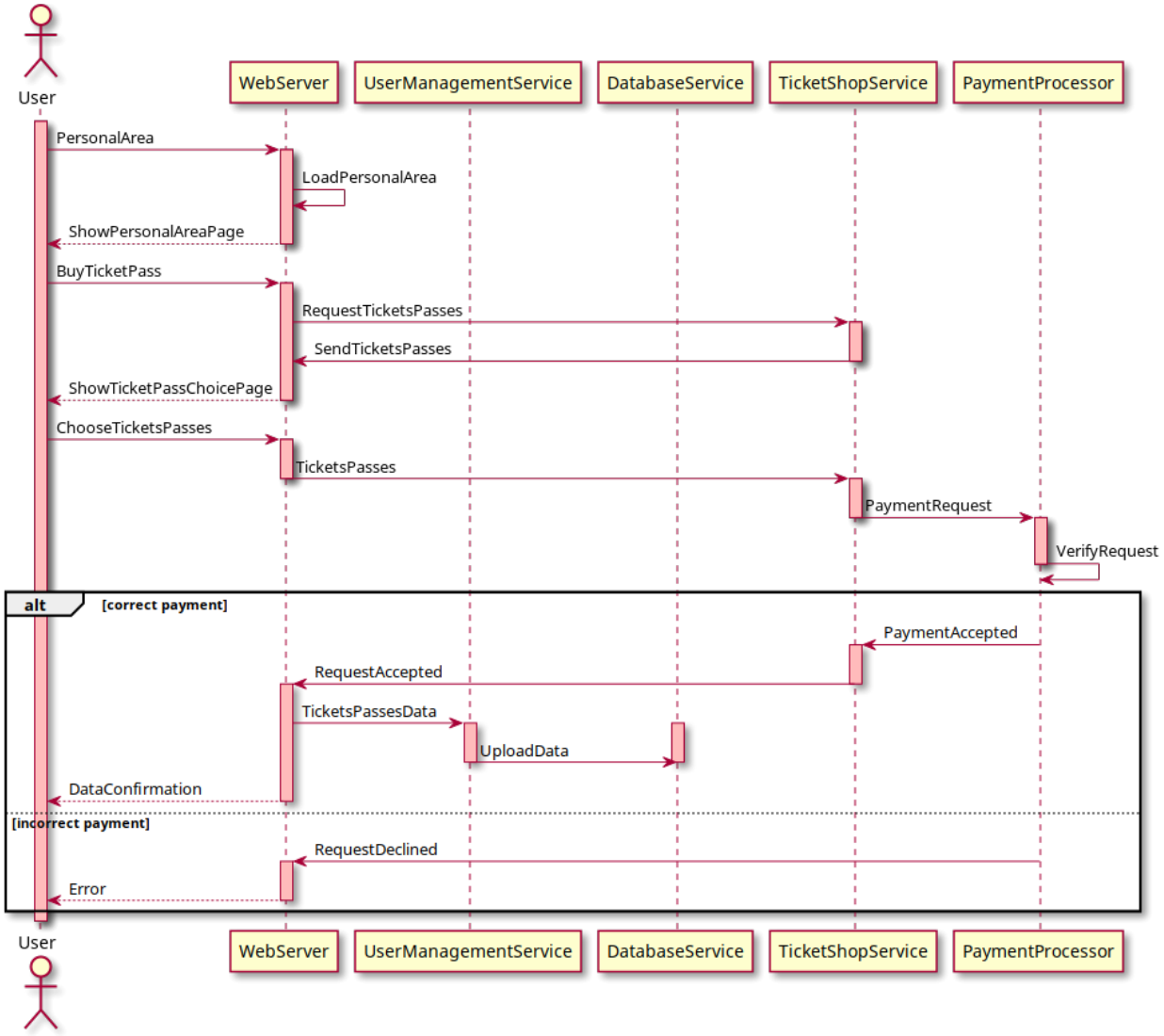


Figure 11: Buy Ticket/Pass Sequence Diagram

New Standard Event Sequence Diagram

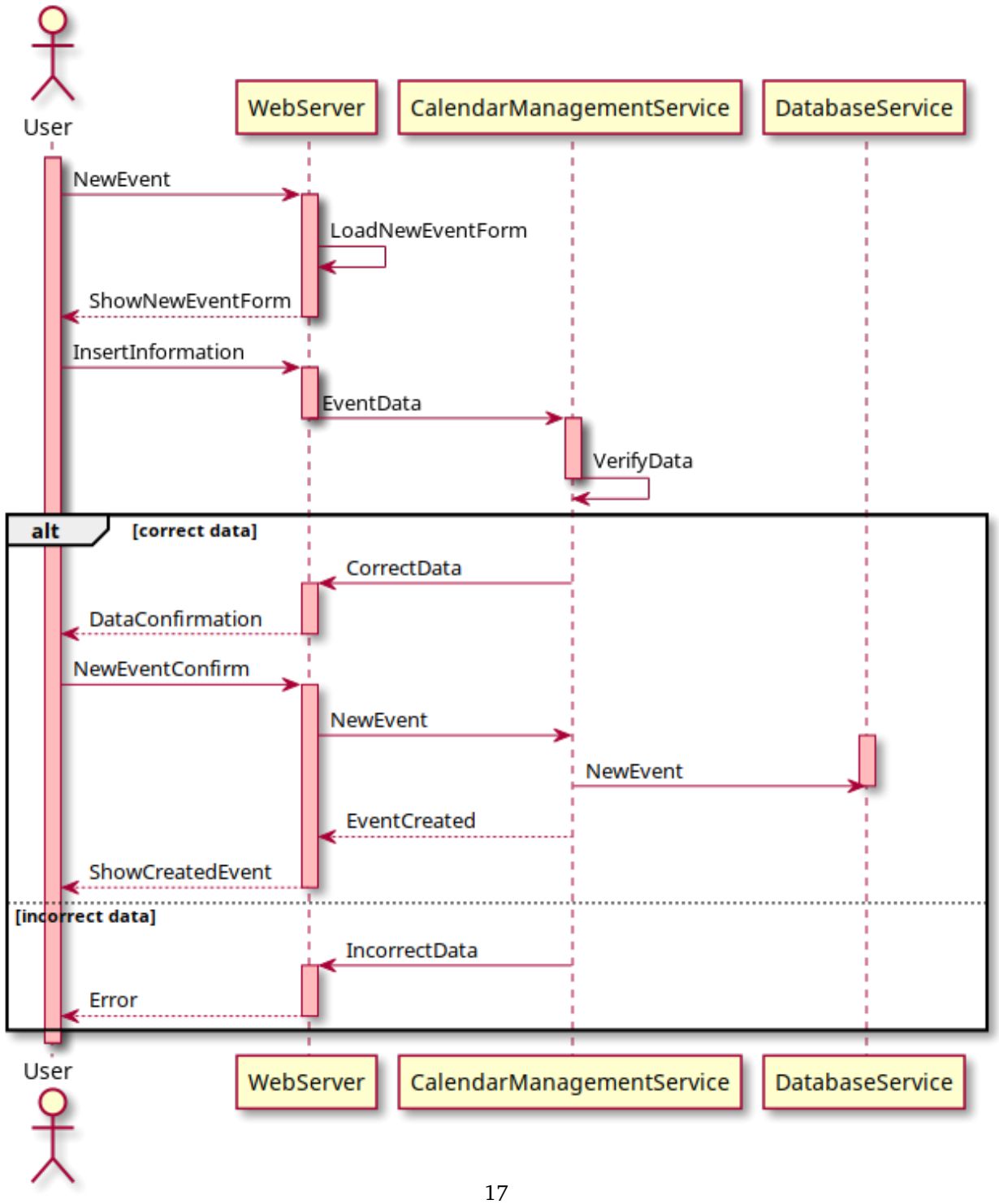


Figure 12: New Standard Event Sequence Diagram

New Event With Sharing Services Sequence Diagram

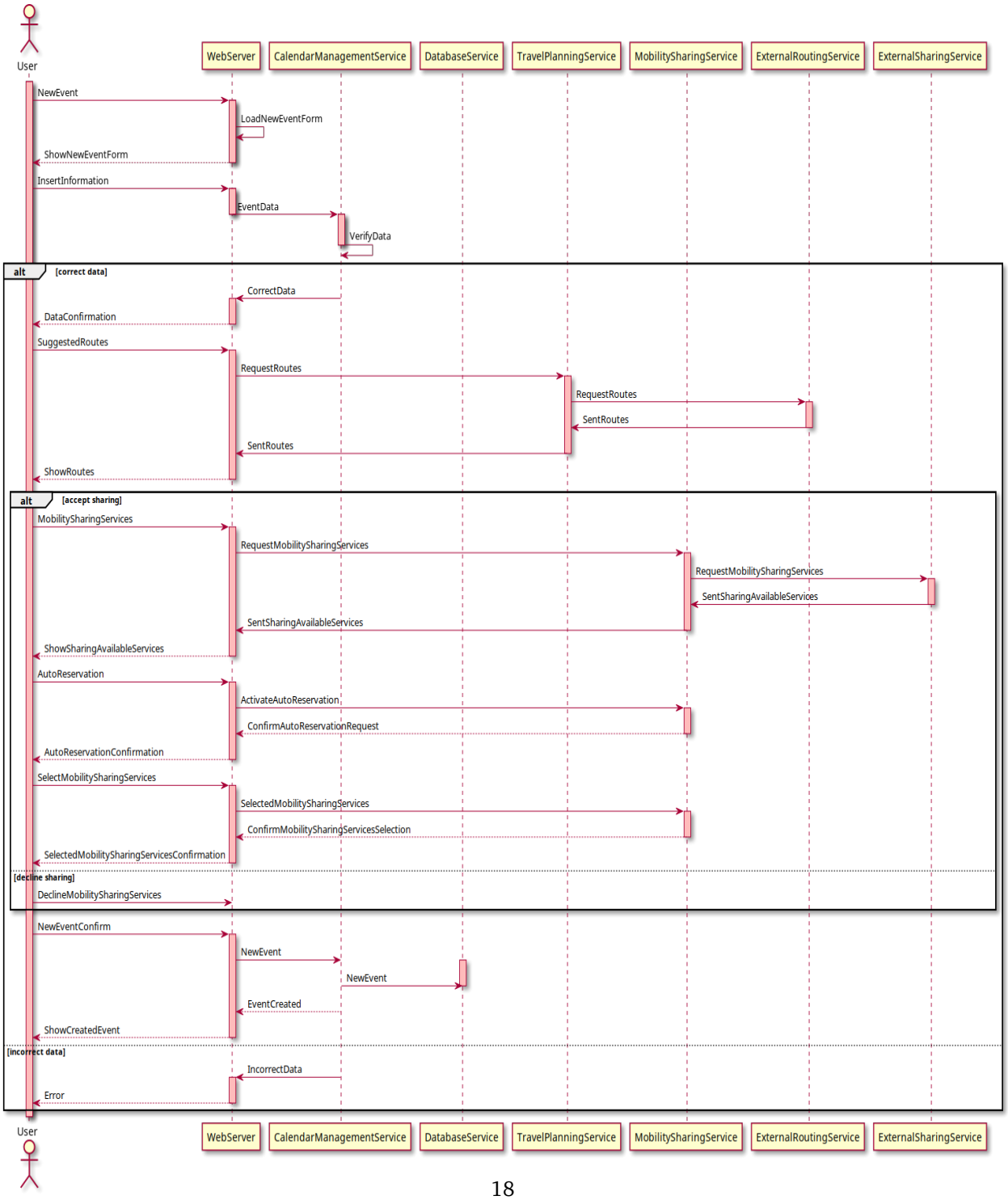


Figure 13: New Event With Sharing Services Sequence Diagram

New Event With Flexible Option Sequence Diagram

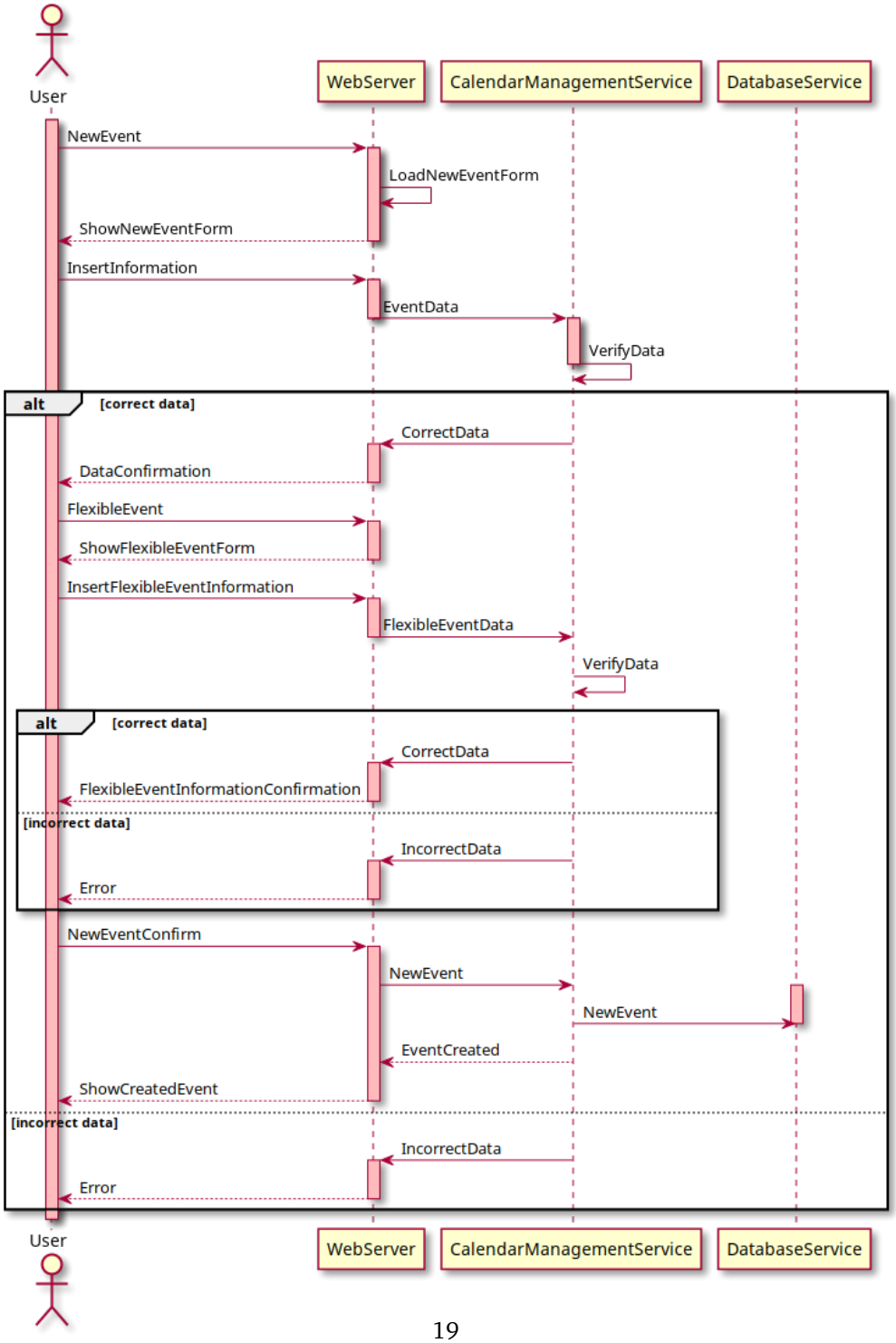


Figure 14: New Event With Flexible Option Sequence Diagram

New Shared Event With Invitation Sequence Diagram

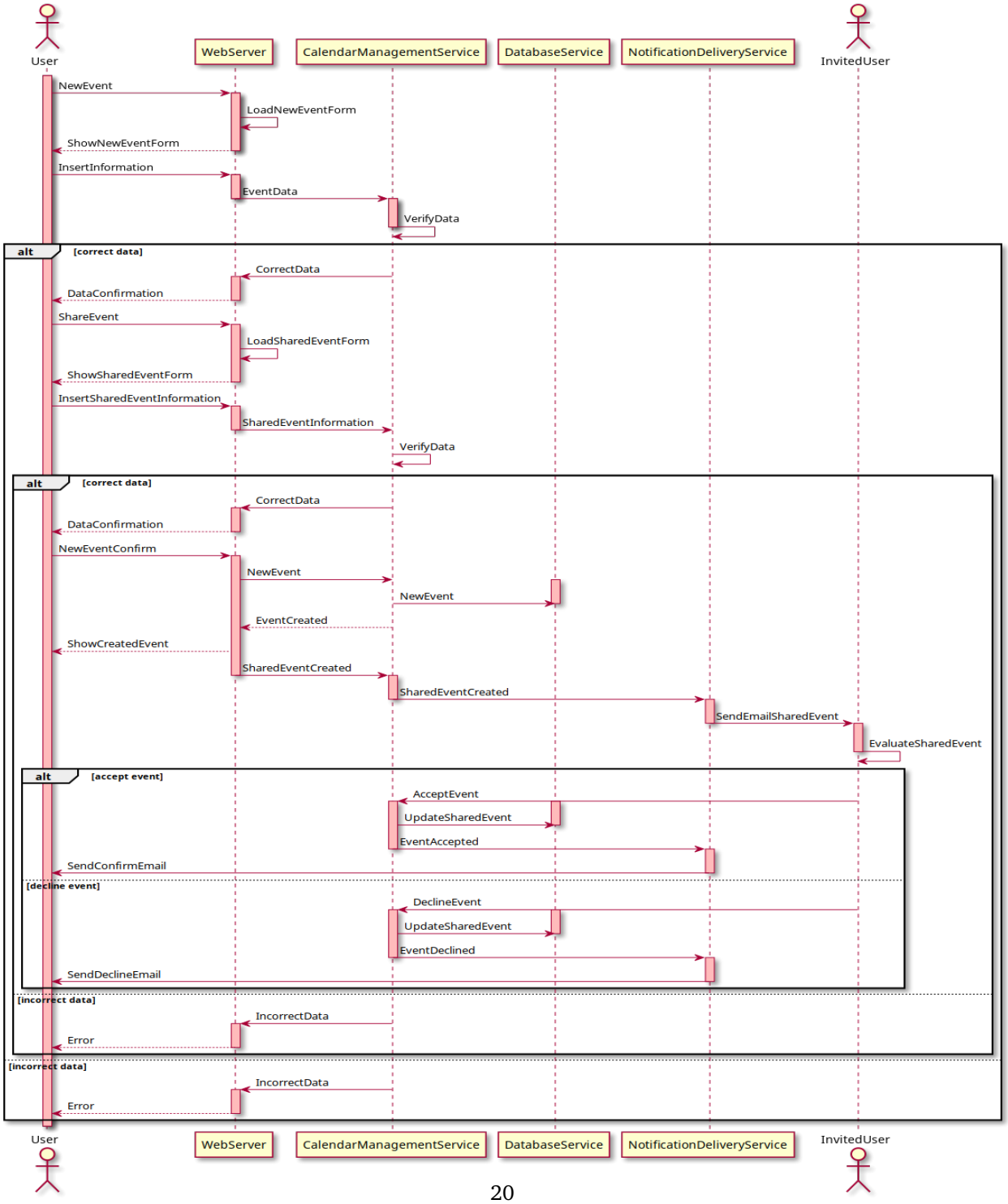


Figure 15: New Shared Event With Invitation Sequence Diagram

Edit Event Sequence Diagram

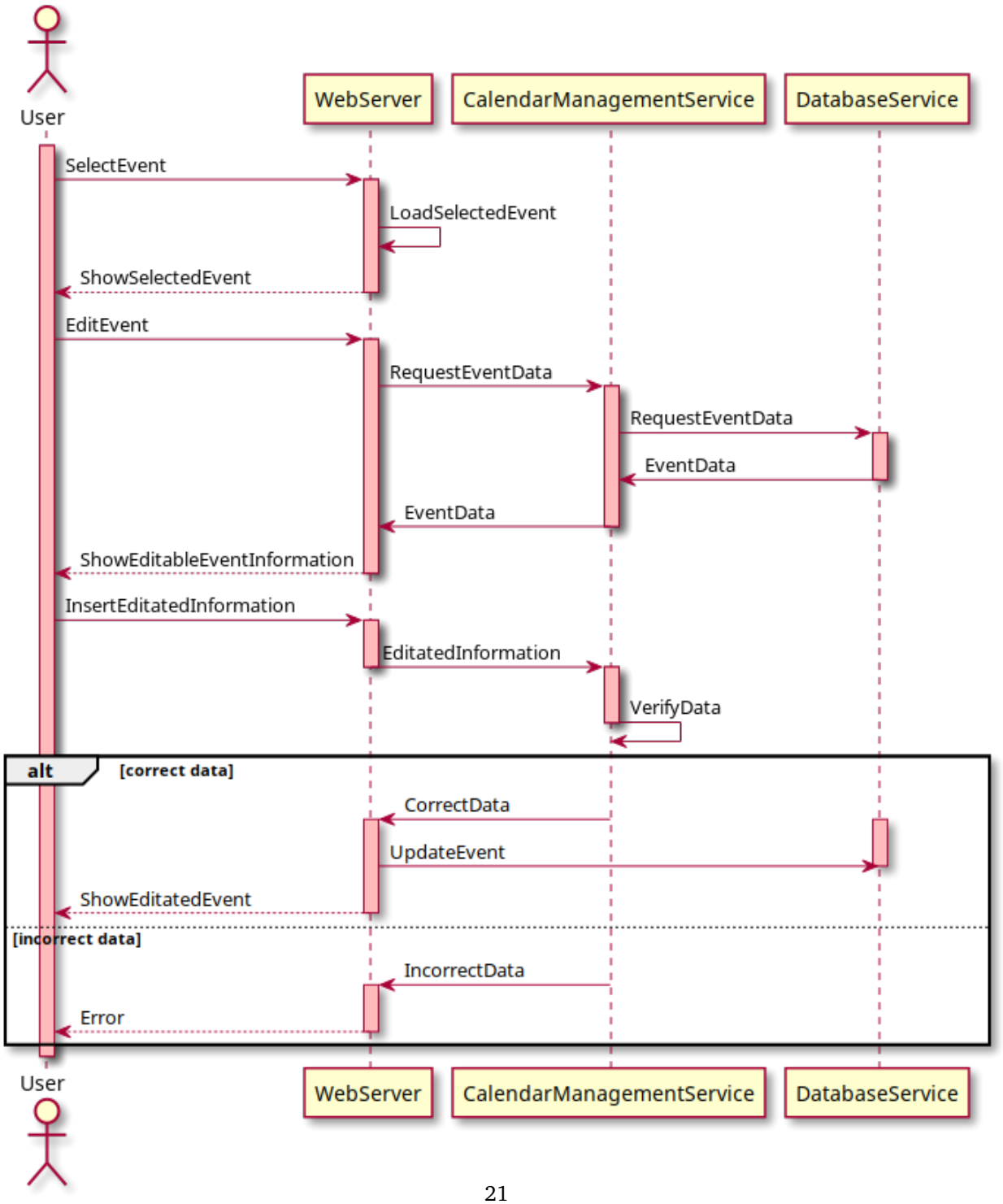


Figure 16: Edit Event Sequence Diagram

Delete Event Sequence Diagram

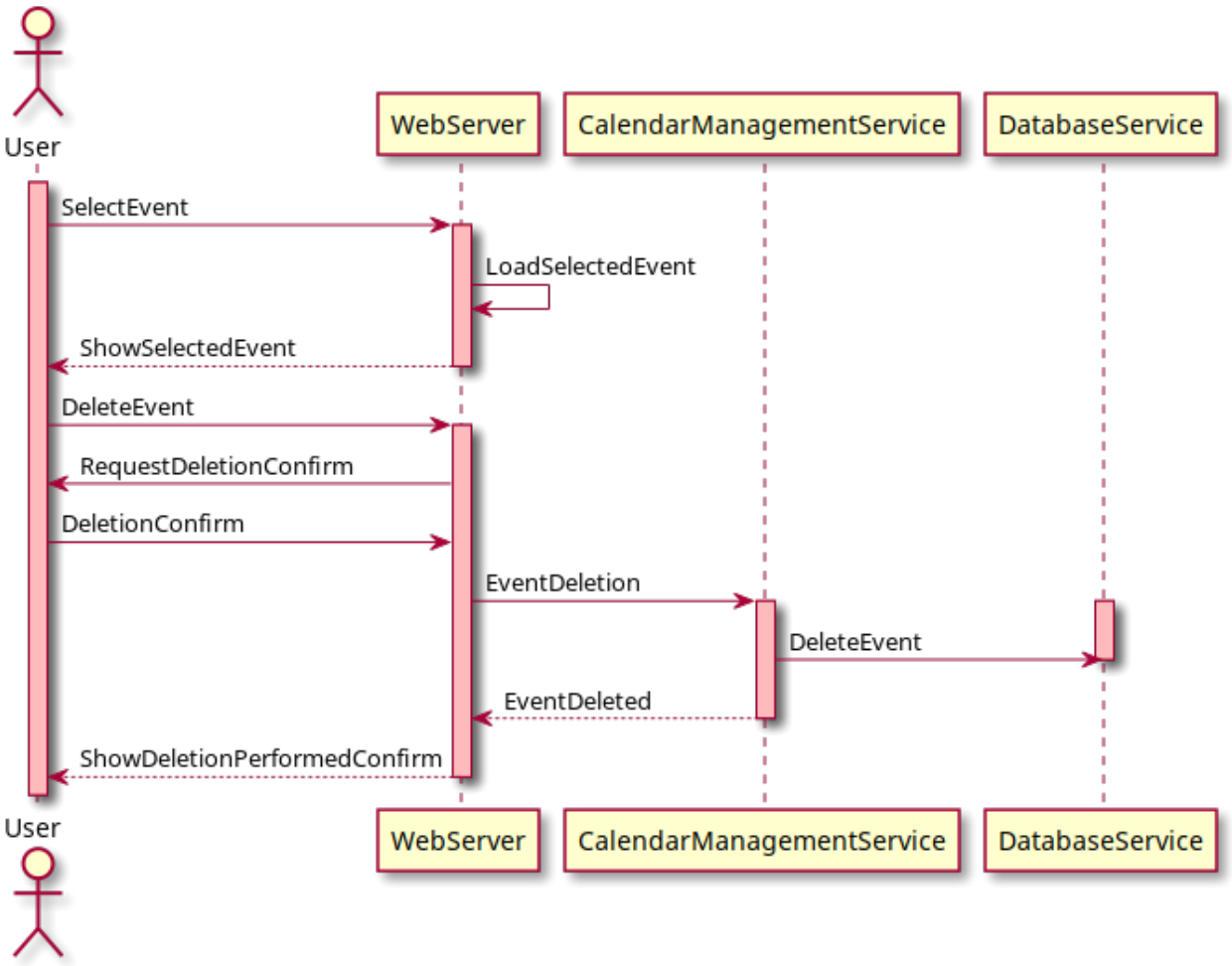


Figure 17: Delete Event Sequence Diagram

Contact Customer Service Sequence Diagram

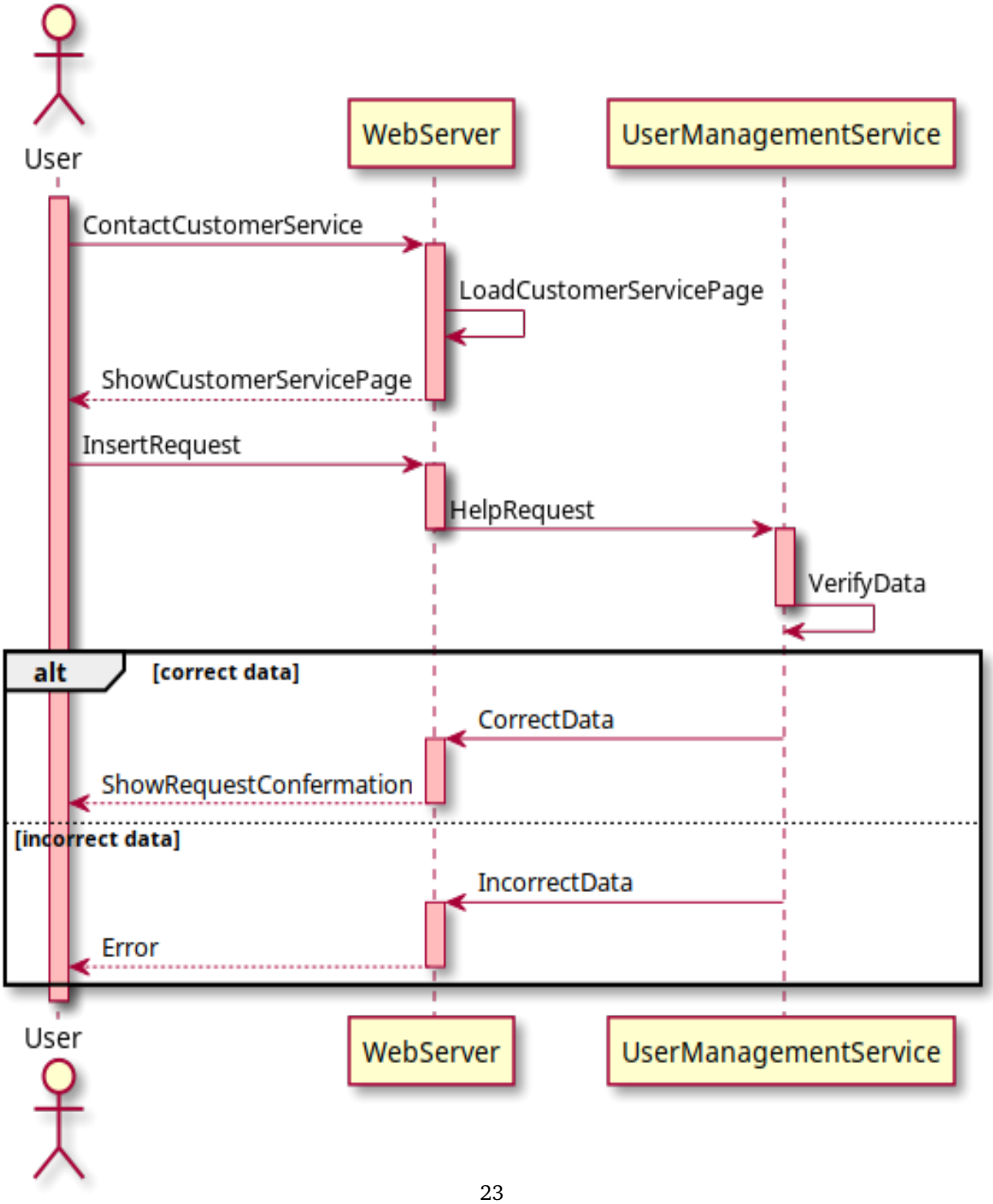


Figure 18: Contact Customer Service Sequence Diagram

2.5 Component interfaces

In this section we describe the the APIs provided by each component of the system. A mid-high level view of the main functionalities is presented, and other trivial API endpoints may and will be added during the implementation phase. Endpoints that require authentication are clearly marked at the beginning of the description.

User managment service

Sign up and account managment	
Endpoint	Description
initiate_signup(user_data)	Creates a new `unconfirmed user` in the system with the information provided during the registration phase and sends the verification email.verifies correctness and completeness of the datacreates user instance in the databasegenerates a secure random verification codeassociates the code with the user in the databaseuses the Notifications delivery service to send a verification email with the code to the user
complete_signup(verification_code)	Ends the user signup process by verifying the code sent by email to newly registered users. If the code is correct the corresponding user is marked as verified and will be able to use the service.
change_password(current_password, new_password)	Requires authentication. 1. Verifies that current_password matches the current user password. If it does not match, go to 5. 2. Verifies that new_password is different than the current one and satisfies password strength requirements 3. Stores the new password in the database 4. Uses the Notifications delivery service to inform the user that his password has changed. End. 5. Uses the Notifications delivery service to inform the user that an attempt to change his password is failed.

initiate_password_recovery(username)	Initiates password recovery procedure for a user. 1. Generates a secure random verification code 2. Associates the code with the username in the database 3. Uses the Notifications delivery service to send a verification email to the user containing a link with the verification code
recover_password(verification_code, new_password)	Allows to recover a user password. 1. Checks the verification_code against the database and gets which user it belongs to 2. Asserts that the new password satisfies password strength requirements 3. Changes the user password 4. Uses the Notifications delivery service to inform the user that his password has changed.

Login	
Endpoint	Description
login(user, password)	Verifies the supplied user credentials and, if correct, returns an authentication token which can be used (via an HTTP header or cookie) to access other restricted APIs.

Account managment	
Endpoint	Description
get_account_information()	Requires authentication. Returns user account informations and preferences.

update_account_information(user_data)	Requires authentication. Updates the account information. 1. Verifies that the user can update the data he has provided 2. Updates the user data

Calendar managment service

Endpoint	Description
create_calendar(calendar_name)	Requires authentication. Creates a new calendar for the authenticated user.
get_calendars()	Requires authentication. Returns the calendars accessible by the authenticated user.
delete_calendar(calendar_id)	Requires authentication. Deletes a calendar belonging to the authenticated user.
create_event(event_details, calendar_id)	Requires authentication. Adds an event to a calendar. 1. Verifies correctness and completness of the event details (e.g. the event must have a name, coherent start/end dates, ...) 2. Verifies that the calendar belongs to the authenticated user 3. Adds the event to the calendar, saving it in the database
get_events(query)	Requires authentication. Returns the events matching the given query.
delete_event(event_id)	Requires authentication. Deletes an event that belongs to the authenticated user.
update_event(event_id, event_details)	Requires authentication. Updates an event belonging to the authenticated user. 1. Verifies correctness of the updated event details 2. Verifies that the event belongs to the authenticated user 3. Updates the event details

Travel planning service

Endpoint	Description
get_travel_suggestions(start_location, end_location, travel_start_time[, travel_preferences])	Returns travel suggestions, combining results from external services.
schedule_shared_reservation(location, datetime, event_id, vehicle_type)	Schedules a reservation of a shared vehicle associated to an event
schedule_taxi_reservation(location, datetime, event_id)	Schedules the reservation of a taxi associated to an event

Notification delivery service

Endpoint	Description
send_push(username, notification)	Internal API callable only by other services. Sends a push notification to the user.
send_email(username, notification)	Internal API callable only by other services. Sends an email notification to the user.
send_notification(username, notification)	Internal API callable only by other services. Send a notification to the user, accounting for their preference when it comes to sending an email or a push or both.

Ticket shop service

Credit card information is processed through Stripe and associated with our users. Once a user has provided credit card information our service can bill them directly when they buy a ticket.

Endpoint	Description
get_lines()	Returns a list of the public transport lines.
get_tickets(line)	Returns a list of tickets eligible for a given transportation line
get_passes(line)	Returns a list of passes eligible for a given transportation line

purchase(ticket)	Requires authentication. Orders the purchase of a ticket for the authenticated user.

Event notifications service

When an event is inserted in a calendar one or more notifications might be scheduled. This service provides endpoints for scheduling notifications and is responsible for sending them to the users via the Notification delivery service.

Endpoint	Description
schedule_notification(event, datetime[, notification_preferences])	Schedules a notification
remove_notification(notification_id)	Deletes a scheduled notification

Travel notifications service

This service does not expose any endpoints.

Taxi reservations service

When a user schedules or cancels the reservation of a taxi this service is called. The request is forwarded to the taxi provider (uber/myTaxy).

Endpoint	Description
schedule_reservation(event, datetime[, reservation_preferences])	Schedules the reservation of a vehicle
remove_reservation(reservation_id)	Deletes a scheduled reservation

Mobility sharing reservation service

When a user requests the automatic reservation of a vehicle a reservation is added to this service. Thirty minutes before the planned start of the trip this service proceeds to apply the auto-reservation procedure described in section 3.1.

Endpoint	Description

schedule_reservation(event, datetime[, reservation_preferences])	Schedules a reservation for a scheduled vehicle
remove_reservation(reservation_id)	Deletes a scheduled reservation

2.6 External components

2.6.1 Google Maps

Travlendar+ will use Google Maps to calculate the distance between two locations, get travel times for driven trips and to show maps to the users. We chose to use an external service because:

- developing maps for each city is not a viable solution due to the tremendous effort required for coding and data collection
- Google Maps has almost perfect coverage of the cities we will initially target
- Google Maps offers APIs, enabling programmatic access to its features
- Google Maps interface is already familiar to many users

2.6.2 Citymapper

We will use Citymapper to schedule trips using public transportation means or walk. The reasons of this choice are the following:

- Citymapper works very well in large urban cities like the ones we will target (like Milan)
- Citymapper offers APIs enabling programmatic access to its services
- Citymapper estimates the calories spent for a walking or biking trip

2.6.3 AccuWeather

We will use AccuWeather as provider for real time weather forcast. It has APIs and has full coverage of the areas targeted by Travlendar+.

2.6.4 Trenord

The system queries Trenord APIs to know the cost of tickets and passes for railway transportation. A commercial agreement with the company was achieved in order to get access to APIs for buying tickets. Other public transportation companies will need to be integrated as Travlendar+ target area will expand.

2.6.5 Stripe

We will use Stripe as payment processor. It is easy to integrate, offers great documentation and client code for all the platform we will support. It is also secure and easy to use.

2.6.6 Enjoy

The system uses Enjoy to reserve a shared vehicle (car or scooter). The reason of this choice are the following

- Enjoy is the most widely used in Milan
- many users already know how Enjoy works and feel comfortable using it

2.6.7 Uber

The system will use Uber to reserve a car with driver.

2.6.8 Mobike

The system will use Mobike to reserve a shared bike. The reason of this choice is the following

- Enjoy is widely used in Milano and in every part of the city it is expected to be possible to find a bike

2.6.9 myTaxi

We reached a commercial agreement with myTaxi to provide taxi reservations. It has extensive coverage of many major cities and a well built technical infrastructure (compared to other taxi services).

2.6.10 Amazon SNS

The system uses Amazon SNS to deliver notifications. The reasons of this choice are the following:

- offers support for email, push and (if needed) SMS

2.7 Architectural styles and patterns

2.7.1 Architectural patterns

As described in this section, particularly in subsection 2.3, our system will initially adopt a simple 3 tier architecture, composed of the clients tier, the web and application logic tier, and the database tier. We will then gradually migrate to a more complex 4 tier architecture to handle more load as our user base will grow, splitting the web and application logi tier into a web/load balancing tier and a dedicated application logic tier. Each tier will first have one machine but more will be added to handle more traffic.

2.7.2 Design Patterns

The main design pattern used for building our clients will be the well known Model-View-Controller. We choose this design pattern because it is simple and effective, and allows to write functionality independently from the user interface.

2.8 Other design decisions

2.8.1 Storage of passwords

We will use the bcrypt algorithm with cost factor 12 to hash the passwords of our users.

3 Algorithm Design

In this section we will detail pseudocode for some of the crucial functionality of our system.

3.1 Automatic reservation algorithm

The `try_auto_reservation` function is called by the mobility sharing reservation service' scheduler when travel of an event is approaching and the user has requested automatic reservation of a shared vehicle.

It searches an available shared vehicle that matches the user's preferences within a specific time-dependent radius centered around the current position of the user.

If no vehicles are found and travel should start in less than 20 minutes the user is notified, and the search continues.

The user can halt the search and manually reschedule the trip or allow the search to continue.

```
def try_auto_reservation(event, user):
    # Time before the travel will start in minutes
    travel_starts_in = (event.travelStartTime - time.time()) / (1000 * 60)

    if travel_starts_in > 30:
        raise Exception("Automatic reservation procedure called too early")

    # Returns the radius of search in meters depending on time
    radius = get_search_radius(travel_starts_in)

    # Tries to find a suitable vehicle in the specified radius
    success = reserve_vehicle_if_available_in_radius(event, user, radius)

    if success:
        return True

    if travel_starts_in <= 20:
        notification_service.send("No suitable shared vehicle found! We will continue searching,
            but you could be required to reschedule your trip.")

    # Reschedule another try in 5 minutes
    shared_vehicle_reservation_service.schedule_auto_reservation_at(time.time() + 5 * 60 * 1000,
        event, user)

    return False


def get_search_radius(travel_starts_in):
    if travel_starts_in <= 20:
        return 2000
    elif travel_starts_in <= 25:
        return 1500
    else:
        return 1000

```

```
34
35  def reserve_vehicle_if_available_in_radius(event, user, radius):
36      allowed_vehicles = event.shared_vehicle_reservation.vehicle_kinds
37
38      for sharing_mobility_service in sharing_mobility_services.filter_on(allowed_vehicles):
39          success = sharing_mobility_service.try_reserving_in_radius(user.current_position, radius)
40          if success:
41              return True
42
43      return False
```

3.2 Flexible event verification algorithm

This function is called when a new fixed event is added to the calendar. It reschedules the flexible events that overlap with the fixed event or throws an exception if it is not possible to do so. The algorithm is simplified to account for a single calendar, but is trivial to extend to the multiple calendars scenario. The user calendar should be locked before calling this function to avoid race conditions.

```
def reschedule_flexible_events(event, user):
    # One way to see if flexible events can be rescheduled is to make a copy of the user calendar
        and act on it
    calendars = user.snapshot_calendar()
    # Add the event to the calendars
    calendar.add(event)

    overlapping_flexible_events = calendar.search_events(flexible=true, between=(event.start,
        event.end))
    for fevent in overlapping_flexible_events:
        success = calendars.try_rescheduling(fevent)
        if not success:
            raise CouldNotRescheduleFlexibleEventException(fevent)

    # If we came this far then it is possible to reschedule overlapping flexible events
    # Add the event to the real user calendars
    user.calendar.add(event)

    overlapping_flexible_events = user.calendar.search_events(flexible=true, between=(event.start,
        event.end))
    for fevent in overlapping_flexible_events:
        success = user.calendar.try_rescheduling(fevent)
        if not success:
            # This should never happen and would leave some flexible events incorrectly scheduled
            raise CouldNotRescheduleFlexibleEventException(fevent)
```

3.3 Event reachable verification algorithm

This function is called when the user adds a fixed event to his calendar. It checks if the event location is reachable in time from the previous event location. Does not check if there's another overlapping event.

```
def is_event_location_reachable_in_time (event, user):
    # The previous event is automatically suggested to the user when possible (if one and only one
        preceding event exists)
    # otherwise the user has to set it manually.
    if event.previous_event is None:
        return True
    travel_start = event.previous_event.end_time
    prev_location = event.previous_event.location

    # If we don't know the event location or previous location we assume the user knows what it's
        doing
    if event.location is None or prev_location is None:
        return True

    min_travel_time = travel_planning_service.get_shortest_travel_time(prev_location,
        event.location, user.travel_preferences)

    return (min_travel_time < event.start - travel_start)
```

3.4 CO2 emissions calculation algorithm

This code is used to give a rough estimate of the CO2 emissions for a travel. It loads a list of emission coefficients based on the travel distance. The rationale behind this is that as a travel gets longer generally more kilometers will be part of high speed roads, where emissions differ. From the list of (travel_length, co2_emissions_per_km) we find the two closest estimates, do a linear interpolation on the emission coefficient and estimate the CO2 emissions with the interpolated value.

```
def co2_estimation(travel_distance, transportation_mean):
   coefficients = load_co2_coefficients(transportation_mean)

   km1, c1, km2, c2 = find_two_nearest(travel_distance)

   c = (c2 - c1)/(km2 - km1)

   return travel_distance * c
```

3.5 Cost calculation algorithm

This function calculates the cost of a route with a specific transport mean. It uses external APIs (GoogleMaps for car, motorbike,... and Citymapper for public transportation means).
In case of personal motor transport means it is assumed that the total cost is the sum of a toll (if the route include a toll road) and the cost of the gasoline used by the car, motobike.
In case of public transport mean Citymapper gives the cost of the travel.
In case of personal free means (bike) or walk the cost will be 0.

```
def calculate_cost (start_position, end_position, transport_mean):
   if transport_mean == personal_motor_transport_mean:

      # Thanks to an external service (Google Maps) it receives the distance iand the distance of
          the route that provides for the payment of a toll
      distance, distance_toll = GoogleMaps.travelDistance(start_position, end_position,
          transport_mean)
      cost_coefficient = load_cost_coefficient(transport_mean)

      if distanceToll != 0:
         return distance * costCoefficient + calculateToll(distance_toll)
      return distance * costCoefficient


   if transport_mean == public_transport_mean:

      # Thanks to an external service (Citymapper) it receives the cost of the route
      mean_route_cost = Citymapper.route_cost(start_position, end_position, transport_mean)
      return mean_route_cost


   if transport_mean == (personal_free_transport_mean || walk)

      # The cost of a free transport mean (like bike or walk) is zero
      return 0
```

4 User Interface Design

We included our mockups for the user interface in the Requirement Analisys and Specifications Document.

5 Requirements Traceability

In the RASD document we have defined all the system's requirements: in this section we explaine how they map into the design elements defined in this document.

Component (DD)	**Requirements (RASD)**
User management service	3.1.1 User account management
Calendar management service	3.1.2 Calendar and event management
Travel planning service	3.1.3 Travel planning
Notification delivery service	3.1.1 User account management 3.1.2 Calendar and event management 3.1.3 Travel planning
Ticket shop service	3.1.3 Travel planning
Event notifications service	3.1.2 Calendar and events management
Travel notifications service	3.1.3 Travel planning
Taxi reservations service	3.1.3 Travel planning
Mobility sharing reservation service	3.1.3 Travel planning

6 Implementation, integration and and test plan

6.1 Suggested development methodology

Not knowing the development team we leave to them the final choice of which development methodology to use. We note that since the requirements of the project are well defined and not subject to big changes both classical and modern methodologies should work well. We suggest in the hypotesis of a small team of 4/5 people the use of an agile methodology inspired to Scrum. We would assume the role of project managers/product owners and help the scrum master create and schedule sprints for the the team, as well as attend to daily meetings and periodic sprint reviews, ideally together with the project commissioner (prof. Di Nitto).

6.2 Gantt chart

Here we propose a schedule for the development of the project. With a bigger team more tasks could be started in parallel, at the cost of spending more time for integration.

6.2.1 Tasks

Events	Duration
Inital meeting with Developers Team and Supervisor	2 days
Requirements analysis	
Q&A session	
Initial database schema development	5 days
Implementation and testing user management service	10 days
Implementation and testing calendar management service	10 days
Implementation and testing travel planning service	15 days
Implementation and testing taxi reservation service	10 days
Implementation and testing mobility sharing reservation service	10 days
Implementation and testing ticket shop service	10 days
Implementation and testing notifications delivery service	5 days
Implementation and testing event notifications service	5 days
Implementation and testing travel notifications service	5 days
Integration between services	10 days
Implementation and testing web application	20 days
Implementation and testing desktop interface	5 days
Implementation and testing Android and iOS apps	30 days
Final testing	15 days
Release & Deployment	3 days

6.2.2 Gantt diagram

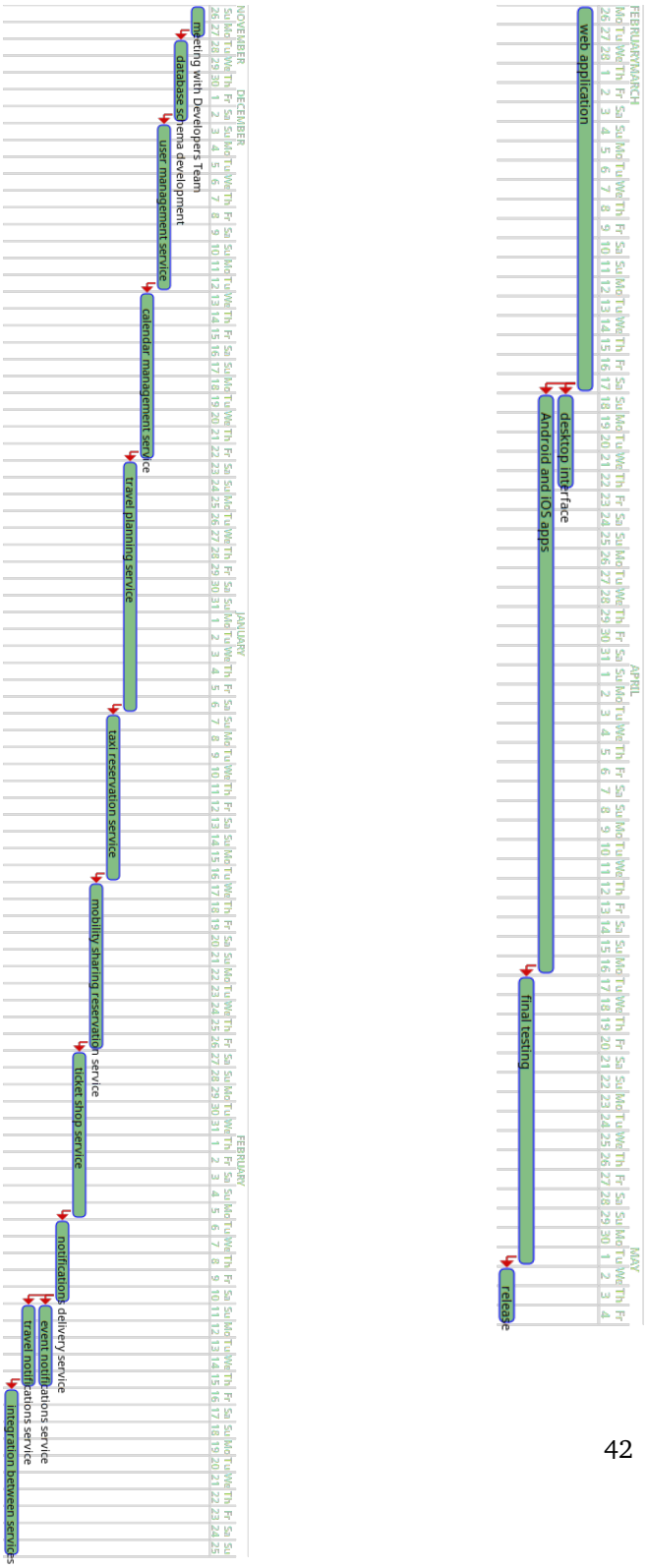


Figure 19: Gantt chart

7 Appendix

7.1 Tools used

We used the following tools to produce this document:

- LaTex as typesetting system to write this document
- LyX as editor
- plantuml and draw.io to draw all the diagrams

8 Effort spent

Data	Alfonso	Carsenzuola	Cremonese
08/11/2017	1.5	1.5	1.5
09/11/2017	2	2	2
13/11/2017	2	2	2
15/11/2017	3	3	3
18/11/2017	2	2	2
20/11/2017	5	5	5
21/11/2017	4.5	4.5	4.5
22/11/2017	3.5	3.5	3.5
23/11/2017	3	3	4
24/10/2017	5	5	5
25/11/2017	4.5	4.5	4.5
26/11/2017	4	4	4
Totale	40	40	41

9 References

9.1 External services

Google Maps	https://www.google.it/maps/
Citymapper	https://citymapper.com/
Uber	https://www.uber.com/
Enjoy	https://enjoy.eni.com/
Mobike	https://mobike.com/
Trenord	http://www.trenord.it/
AccuWeather	https://www.accuweather.com/
Stripe	https://stripe.com/
Amazon SNS	https://aws.amazon.com/sns/

9.2 Frameworks, languages and technologies

Python	https://www.python.org/
MariaDB	https://mariadb.org/
Flask	http://flask.pocoo.org/
Kotlin	http://kotlinlang.org/
Swift	https://www.swift.com/
React	https://reactjs.org/