Ride Share Design Document

Written By: Students in CSE430, Section 1, Winter 2025

April 9, 2025

v1.0

Scope

Our project is a convenient ride-sharing app designed specifically for BYU-Idaho students. It connects students who need a ride from Rexburg to Utah with drivers who have available seats, making travel more accessible and affordable. Instead of developing our own, we will integrate third-party services, including PayPal for payments, a mapping system, and Firebase for a cloud-hosted database. These pre-built systems will be accessed via APIs, ensuring efficient implementation and scalability.

References

We are referencing PayPalAPI Documentation, Google Maps API Documentation and Amazon Web Services

Version History

Version	Update(s)	Date
v0.0	- First view component.	01/24/2025
v0.1	- Made corrections to the System View. - Corrected the view table format. - Added view Headings. - Expanded system descriptions. - Added detailed component breakdowns for UserInterface, AccountManager, Authentication Manager, RideManager, PaymentManager, DatabaseQuerier, Controller, PayPalAPI, and MapAPI. - Included references to Helfrich (2024), *Software Design (2nd ed.)*. - Structured viewpoints based on diagrams (Component Diagram, Data Flow Diagram, Flowchart).	02/08/2025
v0.2	- Changed view naming to use ":" instead of "()" (See 1.2.2.1). - Changed view numbering to include letters for different perspectives of the same thing (See 1.3.1.A). - Added ChatService (1.2.6) and CommunicationManager (1.2.6.1). - Integrated OrderConfirmation Service (1.1.1.5.1) to notify RideManager upon successful payment. - Expanded RideManager (1.2.3) to include trip matchmaking, fare calculations, and trip modification features. - Refactored PaymentService (1.1.1.2.1) to support Venmo payments alongside PayPal.	02/22/2025

	- Added Pseudocode and Structure Chart references in Viewpoints. - Fixed formatting inconsistencies in database querying descriptions. - Updated PayPalAPI interaction details in PaymentManager (1.2.4.3).	
v0.3	- Added Presentation UI for enhanced user display. - Implemented Communication Manager Gateway Controller. - Expanded Order Service with createOrder, confirmOrder, and orderDetails objects. - Integrated structured PayPal API JSON handling for orders, payments, and refunds. - Improved Ride Management with structured trip request handling. - Introduced Cloud Services (1.3) and Content Database. - Added Chat User Interface and Message Cache.	03/08/2025
v0.4	- Added detailed pseudocode for core functions in the Payment and Order Services (e.g., validatePayment, transformPayment, confirmOrder, displayReceipt). - Introduced structured JSON processing in the backend for payment and receipt data using PayPal API (e.g., ConfirmOrder, OrderDetails, CreateOrder). - Refined the OrderService logic to include modular functions like validateOrder, createOrder, confirmOrder, sendDetails, and orderDetails. - Enhanced PaymentScreen with functions for redirecting to PayPal (redirectToPayPal) and user receipt display (displayReceipt). - Integrated object transformation logic for receipts and payments into reusable functions (transformReceipt, sendReceipt). - Clarified function roles with consistent pseudocode blocks and improved class and data flow diagram references. - Standardized object and element naming conventions throughout the document to better align with system structure.	03/29/2025
v1.0	- Promoted the design document to version 1.0 after completing core system architecture. - Finalized full integration of Order and Payment Services with PayPal API, including createOrderCall and confirmOrderCall. - Expanded frontend services: completed PresentationUI, ChatUserInterface, and map integration.	04/09/2025

	- Implemented backend logic for user creation, authentication, and password recovery. - Cleaned up section numbering and Table of Contents for final structure alignment.	

Stakeholders

Sponsor – Brother Helfrich

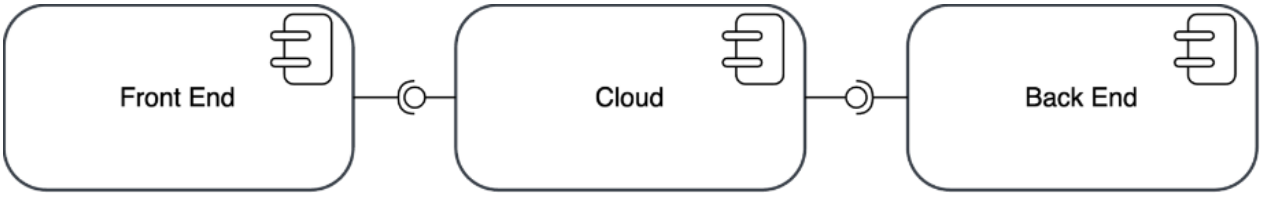
Company – CSE 430 Class

Viewpoints

Name	Source
Component Diagram	Section 4.0 of Helfrich (2024). Software Design (2nd ed.). Kendall Hunt Publishing. https://byui.vitalsource.com/books/9798385152506
Data Flow Diagram	Section 1.1 of Helfrich (2024). Software Design (2nd ed.). Kendall Hunt Publishing. https://byui.vitalsource.com/books/9798385152506
Design Information Table	Section 4.1.7 of Helfrich (2024). Software Design (2nd ed.). Kendall Hunt Publishing. https://byui.vitalsource.com/books/9798385152506
Flowchart Diagram	Section 0.1 of Helfrich (2024). Software Design (2nd ed.). Kendall Hunt Publishing. https://byui.vitalsource.com/books/9798385152506
Pseudocode	Section 0.2 of Helfrich (2024). Software Design (2nd ed.). Kendall Hunt Publishing. https://byui.vitalsource.com/books/9798385152506
Structure Chart	Section 1.0 of Helfrich (2024). Software Design (2nd ed.). Kendall Hunt Publishing. https://byui.vitalsource.com/books/9798385152506
Class Diagram	Section 3.0 of Helfrich (2024). Software Design (2nd ed.). Kendall Hunt Publishing. https://byui.vitalsource.com/books/9798385152506
JSON Structure	What Is JSON? Explained With JSON Examples https://codeblogmoney.com/what-is-json/

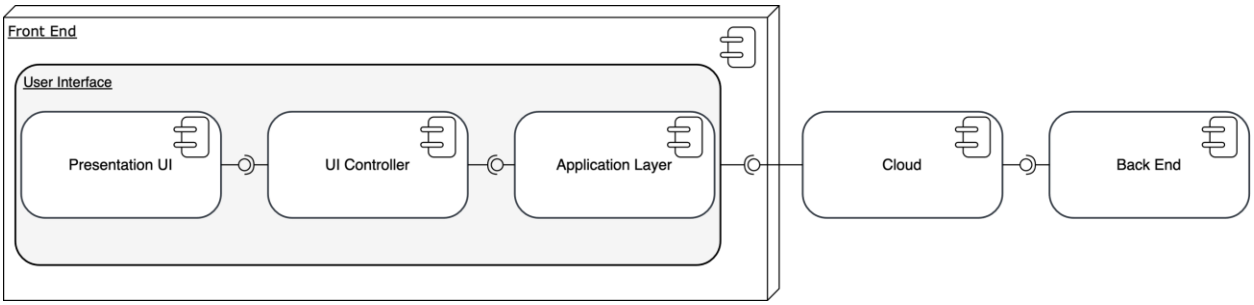
Table of Contents

1 System



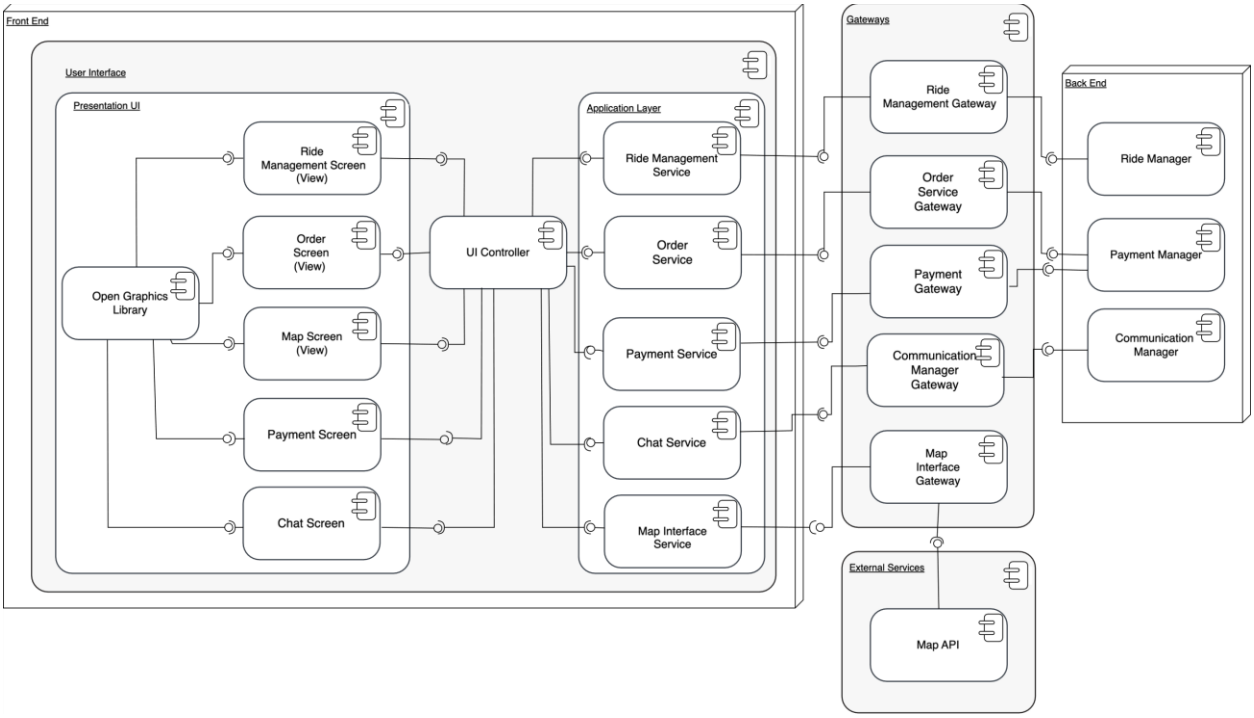
Name	1 System
Purpose	To demonstrate a top-level view of the system and how the frontend, backend, and cloud interact.
Description	A visual representation of the system's architecture, highlighting the interactions and relationships between the frontend and backend.
Requirements	1-22
Elements	**Frontend:** 1.1
	Backend: 1.2
	Cloud: 1.3
Referenced By	
Viewpoint	Component Diagram

1.1 Frontend



Name	1.1 Frontend
Purpose	This Component Diagram is intended to illustrate the relationships in the Frontend.
Description	The Frontend Component Diagram is designed to have a PresentationUI, UI Controller and an Application layer to provide a seamless and cohesive user experience.
Requirements	1-27
Elements	**UserInterface:** 1.1.1
	PresentationUI: 1.1.1
	UI Controller: 1.1.1.4
	Application Layer: This will contain all of the necessary applications for the user Interface to work
	Cloud: 1.3
	Back End: 1.2
Referenced By	1 System, 1.1.1 UserInterface, 1.1.1.2 PaymentUserInterface, 1.1.1.18 ChatUserInterface, 1.2.2.1 Authentication: New User, 1.2.2.2 Authentication: Return User, 1.3 Cloud
Viewpoint	Component Diagram

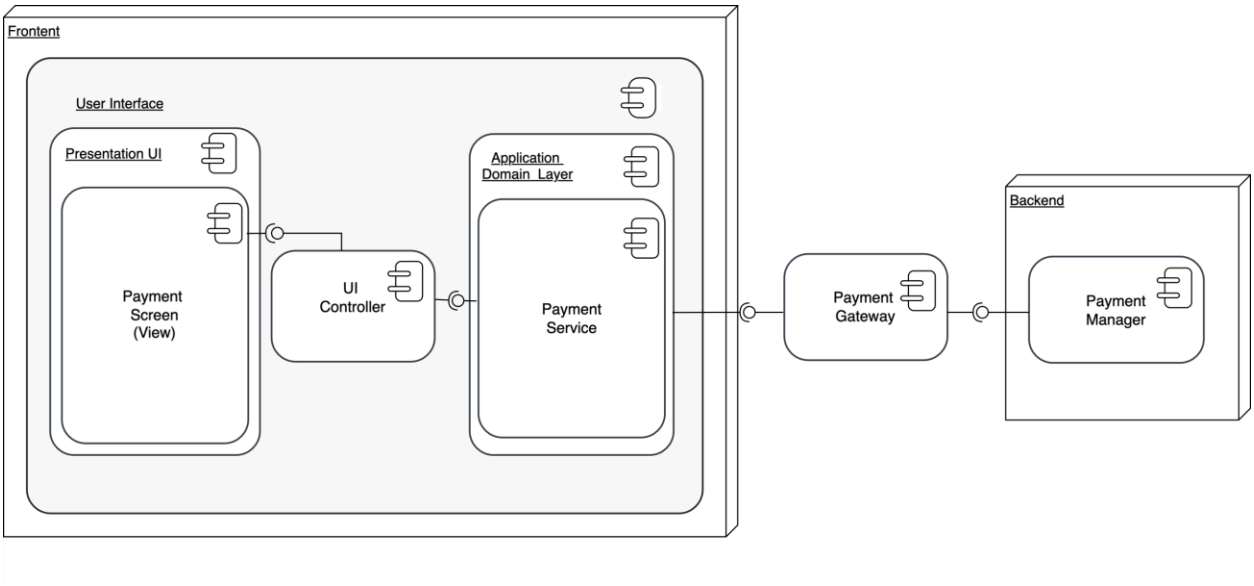
1.1.1 UserInterface



Name	1.1.1 UserInterface
Purpose	This Component Diagram is intended to illustrate the relationships between the distinct parts and components of the for the Ride Share app. It presents how the UserInterface interacts with Orders, Ride Management, and the Map Interface.
Description	The UserInterface Component Diagram is designed to depict the structured interaction between key UI Elementsof the Ride Share app. It demonstrates how graphics, user controls, ride management, orders, and map rendering integrate to provide a seamless user experience.
Requirements	1-20
Elements	**Open Graphics Lib**: This component is responsible for managing the images, text, and styles used within the app that are used for all pages or sections of the app.
	RideManagementScreen: 1.1.1.7
	OrderScreen: 1.1.1.14
	MapScreen: 1.1.1.8
	UI Controller: 1.1.1.4
	RideManagementService: 1.1.1.6
	OrderService: 1.1.1.5
	MapInterfaceService: 1.1.1.9

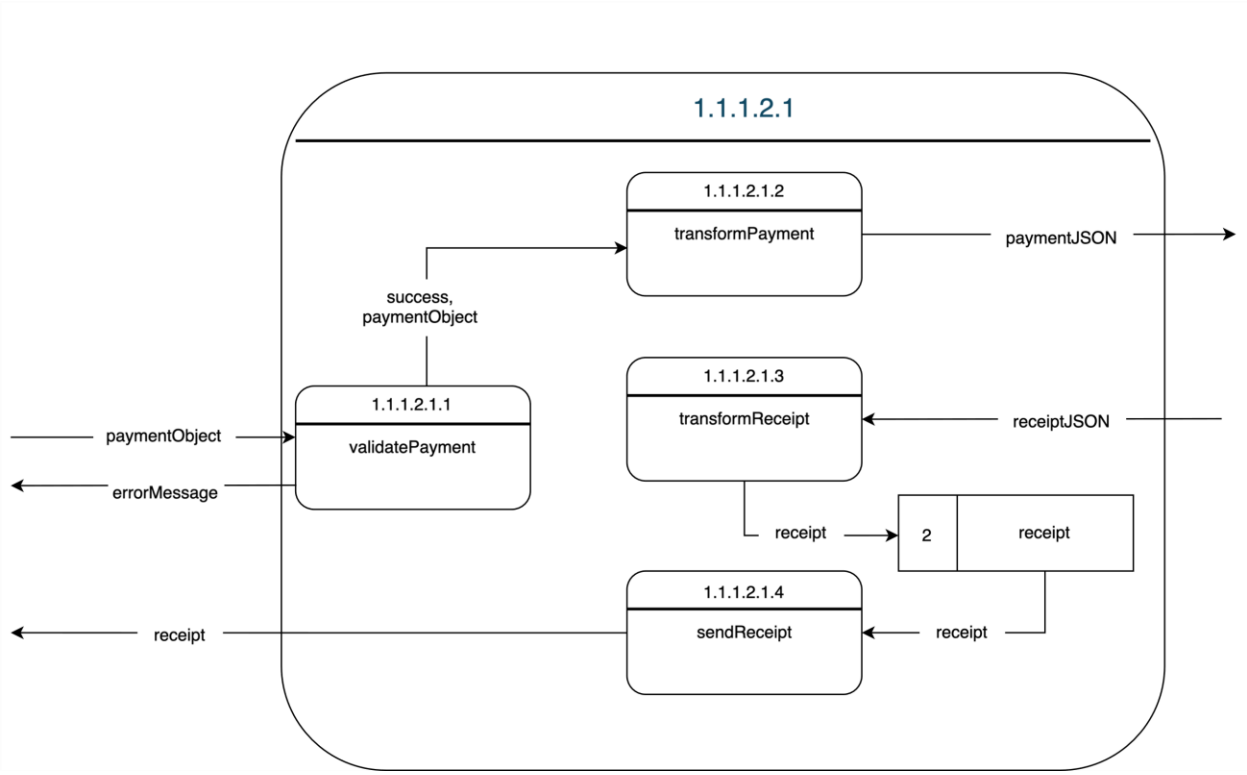
	Ride Management Gateway: This serves as a bridge between the Frontend and back end to communicate and share data for Ride Management.
	Order Service Gateway: This serves as a bridge between the Frontend and back end to communicate and share data for PaymentManager.
	CommunicationManager: 1.2.6
	CommunicationManager Gateway: 1.1.1.17
	ChatService: 1.1.1.16
	ChatScreen: 1.1.1.15
	Payment Gateway: 1.2.4.3
	PaymentService: 1.1.1.2.1
	PaymentScreen: 1.1.1.2.2
	MapAPI: 1.3.3
Referenced By	1 System, 1.1.1.2 PaymentUserInterface, 1.1.1.18 ChatUserInterface, 1.2.1.1.A EditAccount, 1.2.1.3.A Display, 1.2.2.3.A ForgotPassword
Viewpoint	Component Diagram

1.1.1.2 PaymentUserInterface



Name	1.1.1.2 PaymentUserInterface
Purpose	The subsystems, Interaction Layer and View within UI communicate with the PaymentManager in the backend.
Description	Basic UI to Backend architecture where the two subsystems communicate to show the payment data of the trip to the user before and after the payment has been processed.
Requirements	19.0-20.0
Elements	**UserInterface:** 1.1.1
	Frontend: 1.1
	PaymentManager: 1.2.4
	Application Domain Layer: Will service the backend for Payment Information
	PaymentService: 1.1.1.2.1
	PresentationUI: 1.1.1.13
	PaymentScreen View: Will display the checkout form and allow for input details from the user.
	UI Controller: 1.1.1.4
	Payment Gateway: Handles the external services that are in the backend and provide it to the Frontend.
	Backend: 1.2
Referenced By	1 System
Viewpoint	Component Diagram

1.1.1.2.1 PaymentService



Name	1.1.1.2.1 PaymentService
Purpose	The System can send Payments through Venmo
Description	Validates payment information, generates receipts, and sends receipts to the Payment Gateway for local database storage.
Requirements	8, 9
Elements	**paymentObject:** 1.2.4.5
	paymentJSON: 1.2.4.5.A
	validatePayment: Validates the information associated with the payment.
	errorMessage: Message containing error specification if a payment can't be validated.
	TransformPayment: 1.1.1.2.1.2
	receipt: Proof of payment includes information contained within the payment. It is a object of the JSON schema.
	transformReceipt: 1.1.1.2.1.3
	SendReceipt: 1.1.1.2.1.4
Referenced By	1.1.1.2 PaymentUserInterface, 1.1.1.2.2 PaymentScreen, 1.2.4.3 Payment Gateway Controller , 1.1.1.2.1.1 validatePayment
Viewpoint	Data Flow Diagram

1.1.1.2.1.1 validatePayment

```
validatePayment(fare, paymentObject)

   IF fare == paymentObject.total
      RETURN true, paymentObject

   THROW error <- "Error: fare does not match total"
```

Name	1.1.1.2.1.1 validatePayment
Purpose	Validates the information associated with a payment
Description	A Function that takes a fare and a paymentObject, then compares the total stored in the paymentObject with the fare. If they match, return a successful verification ("true" Boolean). If not, an error is thrown.
Requirements	8, 9
Elements	**fare**: The total cost of a ride.
	paymentObject: 1.2.4.5
	error: 1.2.4.1
Referenced By	1.1.1.2.1 PaymentService, 1.2.4.1 Payment Manager: Get Total
Viewpoint	Pseudocode

1.1.1.2.1.2 transformPayment

```
transformPayment(paymentObject)

   newJSON <- new paymentJSON
   newJSON["rideID"] <- paymentObject.rideID
   newJSON["userID"] <- paymentObject.userID
   newJSON["total"] <- paymentObject.total
   newJSON["initiationDate"] <- paymentObject.initiationDate
   newJSON["fulfillmentDate"] <- paymentObject.fulfillmentDate
   newJSON["isPaid"] <- paymentObject.isPaid

   RETURN newJSON
```

Name	**1.1.1.2.1.2 transformPayment**
Purpose	Convert a paymentObject into a paymentJSON
Description	A function that takes a paymentObject and moves the data into a paymentJSON instance.
Requirements	8, 9
Elements	**paymentObject**: 1.2.4.5
	newJSON: Local paymentJSON variable that holds the info contained in the passed paymentObject.
	paymentJSON: 1.2.4.5.A
Referenced By	1.1.1.2.1 Payment Service
Viewpoint	Pseudocode

1.1.1.2.1.3 transformReceipt

```
transformReceipt(receiptJSON)

   newReceipt <- new receiptObject

   newReceipt.amount <- receiptJSON["amount"]

   newReceipt.orderID <- receiptJSON["orderID"]

   newReceipt.date <- receiptJSON["date"]

   newReceipt.details <- receiptJSON["details"]

   Return newReceipt
```

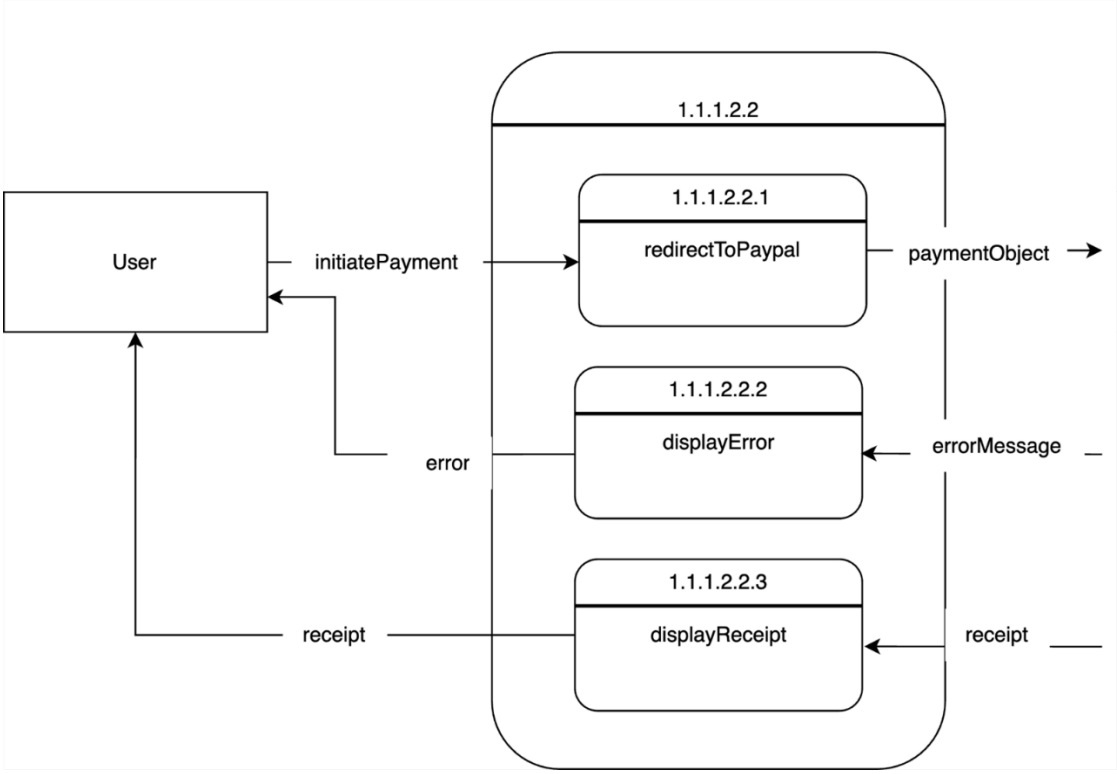
Name	1.1.1.2.1.3 transformReceipt
Purpose	Convert a receiptJSON into a receiptObject
Description	Function that makes a receiptObject and populates it with data from a receiptJSON instance.
Requirements	8, 9
Elements	**receiptJSON:** 1.3.2.1.4
	newReceipt: Local Receipt objectvariable that holds the info contained in the passed receiptObject.
	receiptObject: 1.2.4.2.A
Referenced By	1.1.1.2.1 Payment Service
Viewpoint	Pseudocode

1.1.1.2.1.4 sendReceipt

```
createReceiptJSON(receiptJSON)

   newObject <- new receiptObject

   newObject.orderID <- receiptJSON.ReceiptID

   newObject.amount <- receiptJSON.amount

   newObject.date <- receiptJSON.date

   newObject.details <- receiptJSON.details

   RETURN newObject
```

Name	1.1.1.2.1.4 sendReceipt
Purpose	To convert a receiptJSON to a receiptObject to display to a user.
Description	Accepts a receiptJSON as a parameter and returns a receiptObject containing the information stored within the passed receiptJSON.
Requirements	9
Elements	**newObject:** Local variable containing a receiptObject instance that holds the information held within a passed receiptJSON.
	receiptObject: 1.2.4.2.A
	receiptJSON: 1.3.2.1.4
Referenced By	1.1.1.2.1 Payment Service
Viewpoint	Pseudocode

1.1.1.2.2 PaymentScreen



Name	**1.1.1.2.2 PaymentScreen**
Purpose	The System can send Payments through Paypal
Description	Displays the checkout form and confirmation and allows input details from the user.
Requirements	8, 9
Elements	**user**: Person using the app
	initiatePayment: User response to pay for their order.
	redirectToPaypal: 1.1.1.2.2.1
	paymentObject: 1.1.1.2.1
	errorMessage: 1.1.1.2.1
	DisplayError: 1.1.1.2.2.2
	Error: Formatted error message that displays to the user.
	DisplayReceipt: 1.1.1.2.2.3
	Receipt: 1.1.1.2.1
Referenced By	1.1.1.2 Payment User Interface
Viewpoint	Data Flow Diagram

1.1.1.2.2.1 redirectToPayPal

```
redirectToPayPal(initiatePayment)

   orderInfo <- postRequest(success, initiatePayment)

   urlRedirect(orderInfo.approval_URL)
   urlRedirect(orderInfo.return_URL)
```

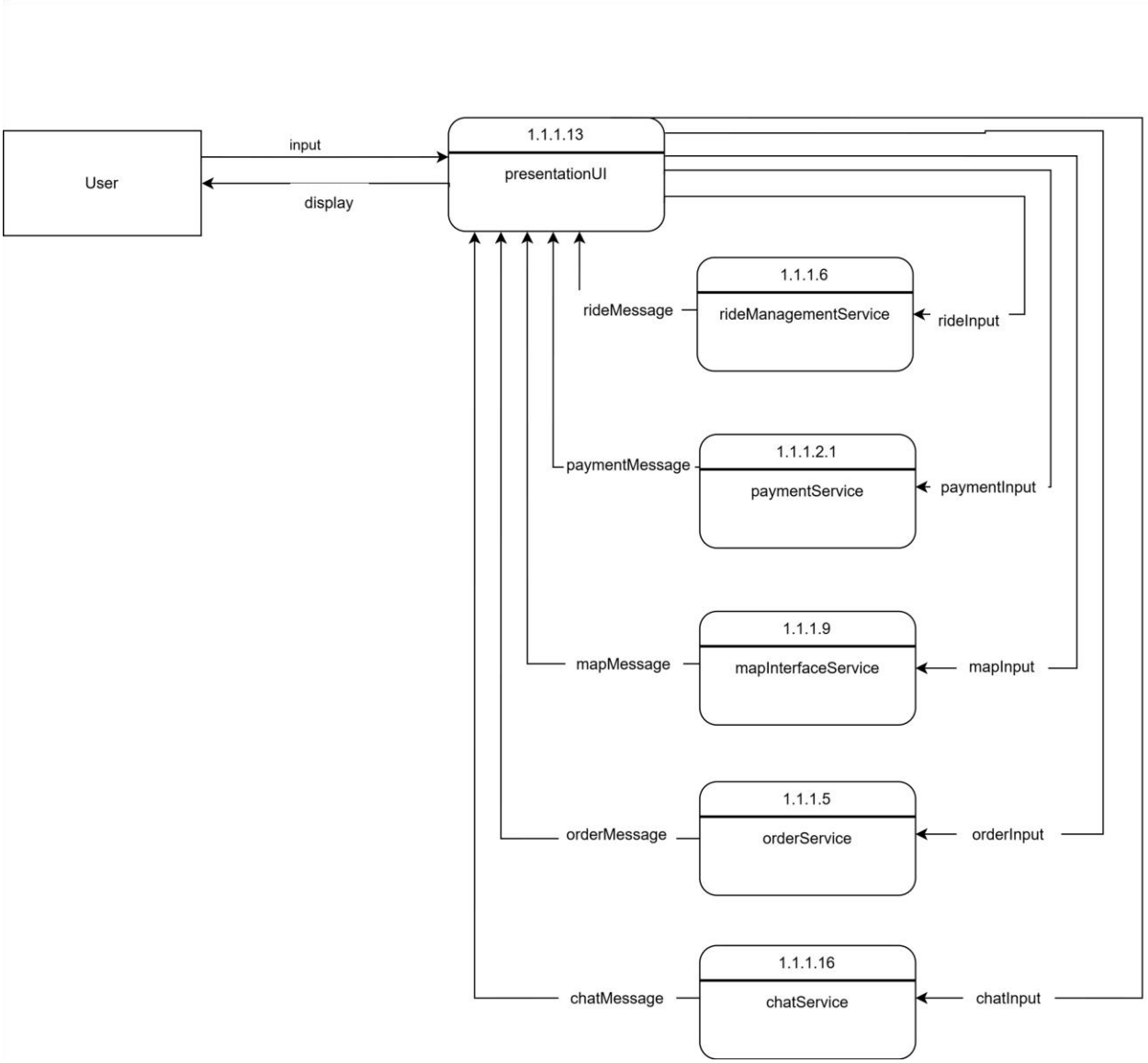
Name	1.1.1.2.2.1 redirectToPayPal
Purpose	Redirect the User to PayPal
Description	A payment is created for PayPal, confirmed by the user in PayPal urls, and then captured by PayPal, utilizing PayPal API endpoints
Requirements	9
Elements	**orderInfo:** Local map variable that contains an orderID, approval_URL, and a return_URL from PayPal.
	postRequest: 1.2.4.4.1
	urlRedirect: Redirects a user to the provided url.
Referenced By	1.1.1.2.2 Payment Screen
Viewpoint	Pseudocode

1.1.1.2.2.2 displayReceipt

```
displayReceipt(receipt)

   PUT receipt.orderID on screen

   PUT receipt.amount on screen

   PUT receipt.date on screen

   PUT receipt.details on screen
```

Name	1.1.1.2.2.3 displayReceipt
Purpose	Show the user what information is on a receipt.
Description	Accepts a receipt as a parameter and displays its information to the user.
Requirements	9
Elements	**receipt:** 1.2.4.2.A
Referenced By	1.1.1.2.2 Payment Screen
Viewpoint	Pseudocode

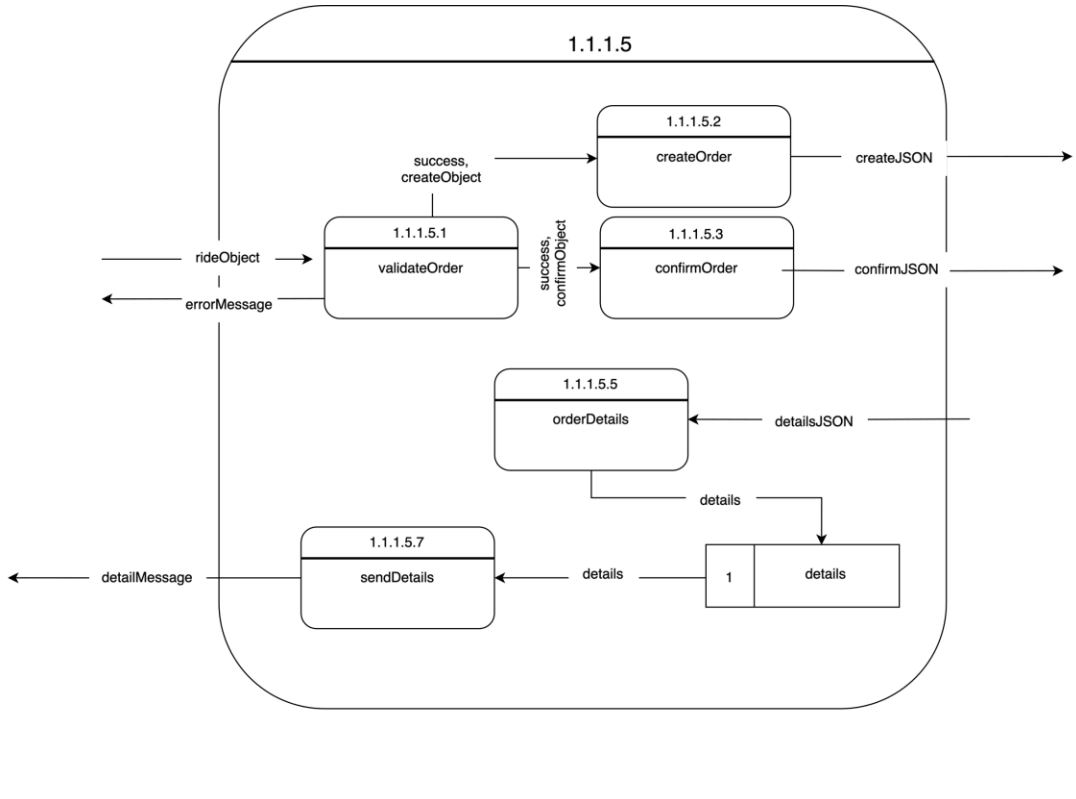
1.1.1.4 UserInterfaceController



Name	1.1.1.4 UserInterfaceController
Purpose	Allow the User to interact with the system
Description	Component that formats user input into objects compatible with backend processing and transforms backend data into readable displays for the User.
Requirements	1, 3-6, 9-12, 14-19, 21-22
Elements	**User**: The person using the program.
	input: Raw input data from the User.
	rideManagementService: 1.1.1.6

	paymentService: 1.1.1.2.1
	mapInterfaceService: 1.1.1.9
	orderService: 1.1.1.5
	chatService: 1.1.1.16
	rideInput, paymentInput, mapInput, orderInput, chatInput: This is the input coming from the presentationUI, the presentationUI will make a decision and it will send the input according to what the user wants. Ride information, payment information, map information, order information and chat information are the different inputs.
	rideMessage: Packed data regarding rides sent from rideManagementService.
	paymentMessage: Packed data regarding payments sent from paymentService.
	mapMessage: Packed data regarding maps/locations sent from mapInterfaceService.
	orderMessage: Packed data regarding orders sent from orderService.
	chatMessage: Packed data regarding chat messages sent from chatService.
	presentationUI: 1.1.1.13
	display: User-friendly display message built from the messages unpacked by presentationUI and sent to user.
Referenced By	1.1.1 UserInterface, 1.1.1.2.1 PaymentService, 1.1.1.5 OrderService, 1.1.1.16 ChatService, 1.1.1.6 Ride Service, 1.1.1.9 Map Service
Viewpoint	Data Flow Diagram

1.1.1.5 OrderService



Name	1.1.1.5 OrderService
Purpose	Send the order information to the backend to handle all the request of the PayPalAPI.
Description	Once a ride has been created, updated, or confirmed that information is sent to the PayPalAPI. When the users confirms an order they are then provided with OrderConfirmation details.
Requirements	2, 5
Elements	**RideObject:** 1.1.1.14
	ErrorMessage: 1.1.1.14
	ValidateOrder: 1.1.1.5.1
	Success: A boolean that indicates the information given is all there.
	ConfirmObject: 1.1.1.5.6
	CreateObject: 1.1.1.5.9
	ConfirmJSON: 1.1.1.5.6.A
	createJSON: 1.1.1.5.9.A
	DetailsJSON: 1.1.1.5.8
	detailsObject: 1.1.1.5.8.A
	confirmOrder: 1.1.1.5.3
	orderDetails: 1.1.1.5.5

	sendDetails: 1.1.1.5.7
Referenced By	1.1.1 UserInterface, 1.1.1.4 UI Controller
Viewpoint	Data Flow Diagram

1.1.1.5.1 validateOrder

```
validateOrder(trip)

   IF !trip OR typeof trip IS NOT 'object'
      RETURN error

   IF !trip.id
      RETURN error

   createObject ←{
      id: trip.id,
      paymentSource: paymentSource[]
      }

   confirmObject ←{
      id: trip.id,
      status: status,
      paymentSource: paymentSource[],
      payer: id
      }

   RETURN {
      success: true,
      createObject,
      confirmObject }
```

Name	1.1.1.5.1 validateOrder
Purpose	Verifies that information in the rideObject is there to generate the createObject and the confirmObject.
Description	Create the objects that will be used in the backend for the API requests.
Requirements	9
Elements	**trip:** The trip object that is given when the user selects a trip.
	trip.id: The id that is unique to the trip.
	Error: The message conveying that the object couldn't be retrieved, or an id could not be found.
	createObject: 1.1.1.5.9
	confirmObject: 1.1.1.5.6
	Payer: 1.1.1.5.6.A
	paymentSource: 1.1.1.5.6.A
	Success: A Boolean indicated that those objects were created.
Referenced By	1.1.1.5 Order Service
Viewpoint	Pseudocode

1.1.1.5.2 createOrder

```
createOrder(createOrderObject)

    createJson ← JSON.stringify(orderObject)

    response ← SEND createJson to createOrderCall

    IF response.status = "success"
        RETURN success
    ELSE
        RETURN error
```

Name	1.1.1.5.2 createOrder
Purpose	To compile the createObject into a JSON file to send to the backend.
Description	The JSON file is created to send to the backend for the API requests.
Requirements	9
Elements	**createJSON:** The object is compiled into a JSON file to send through the controller.
	JSON.stringify: 1.1.1.5.3
	Response: Tells the function where to send the JSON file.
	createOrderCall: 1.1.1.5.10.1
	response.status: The status of the file being delivered to the correct location or not.
	success: A boolean that indicates the file was delivered successfully
	error: A boolean that indicates the file couldn't be delivered for some reason.
Referenced By	1.1.1.5 Order Service
Viewpoint	Pseudocode

1.1.1.5.3 confirmOrder

```
confirmOrder(confirmObject)

   confirmJson ← JSON.stringify(confirmObject)

   response ← SEND confirmJson to confirmOrderCall

   IF response.status = "success"
      RETURN success
   ELSE
      RETURN error
```

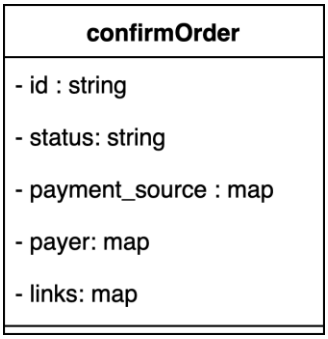
Name	1.1.1.5.3 confirmOrder
Purpose	To compile the confirmObject into a JSON file to send to the backend.
Description	The JSON file is created to send to the backend for the API requests.
Requirements	9
Elements	**confirmOrder:** 1.1.1.5
	confirmJSON: The object is compiled into a JSON file to send through the controller.
	response: 1.1.1.5.2
	confirmOrderCall: 1.1.1.5.10.2
	response.status: 1.1.1.5.2
	success: 1.1.1.5.2
	Error: 1.1.1.5.2
Referenced By	1.1.1.5 Order Service, 1.1.1.5.3 confirmOrder
Viewpoint	Pseudocode

1.1.1.5.5 orderDetails

```
orderDetails(detailsJSON)

   detail <- READ detailsJSON

   IF detail != NULL
      details <- new Details
      details.id <- details[“id”]
      details.status <- details[“status”]
      details.paymentSource <- details[“paymentSource”]
      details.payer <- details[“payer”],
      details.CreateTime <- details[“createTime”]

      RETURN details

   RETURN error
```

Name	1.1.1.5.5 orderDetails
Purpose	To compile detailsJSON into a JSON file to send to the UI.
Description	The detailsObject is created after the user has confirmed their order and the details of their order are displayed to them.
Requirements	9
	detailsJSON: 1.1.1.5 .8
	details: 1.1.1.5.8.A
	error: 1.1.1.5.2
Referenced By	1.1.1.5 Order Service
Viewpoint	Pseudocode

1.1.1.5.6 confirmOrder



Name	1.1.1.5.6 confirmOrder
Purpose	An object that takes data from the ride and the user and puts it into an instance.
Description	This object can be read and/or transformed to meet the needs of the of confirming an order before it gets sent to the backend of the system.
Requirements	8
Elements	id: 1.1.1.5.6.A
	status: 1.1.1.5.6.A
	payment_source: 1.1.1.5.6.A
	payer: 1.1.1.5.6.A
	links: 1.1.1.5.6.A
Referenced By	1.1.1.5 OrderService , 1.1.1.5.1.2 confirmOrderCall
Viewpoint	Class Diagram

1.1.1.5.6.A ConfirmOrder

```
{
  "id": "string",
  "status": "string",
  "paymentSource": {
    "string": {
      "name": {
        "givenName": "string",
        "surname": "string"
      },
      "emailAddress": "string"
    }
  },
  "payer": {
    "name": {
      "givenName": "string",
      "surname": "string"
    },
    "emailAddress": "string"
  },
  "links": [
    {
      "href": "string",
      "rel": "string",
      "method": "string"
    },
    {
      "href": "string",
      "rel": "string",
      "method": "string"
    }
  ]
}
```

Name	1.1.1.5.6.A ConfirmOrder
Purpose	A POST request for the PayPalAPI to confirm an order.
Description	A JSON schema that shows the payer (rider) has the intent to pay for the order.
Requirements	8
Elements	**id:** The ID of the order for which the payer confirms their intent to pay.
	status: The action needed from the API.
	paymentSource: The application the payment is being done in.
	payer: The PayPal users' information.
	name: Stores the users first and last name.

	givenName: Clients first name.
	surname: Clients last name.
	emailAddress: Clients registered email address.
	href: Hyperlink the client is directed too.
	rel: Defines the relationship between a linked resource and the current document.
	links: All the hyperlinks that are navigated too.
	method: Retrieves data in the JSON format from the API.
Referenced By	1.1.1.5 OrderService
Viewpoint	JSON Schema

1.1.1.5.7 sendDetails

```
sendDetails(details)

   details <- orderDetails()
   PUT detailsMessage
```

Name	1.1.1.5.7 sendDetails
Purpose	Send message to user about order confirmation details that occurred in PayPal.
Description	Gathers information for order details and gives it to the user interface.
Requirements	9
	sendDetails: 1.1.1.5
	details: 1.1.1.5
	detail: The value that corelates to the key.
Referenced By	1.1.1.5 Order Service
Viewpoint	Pseudocode

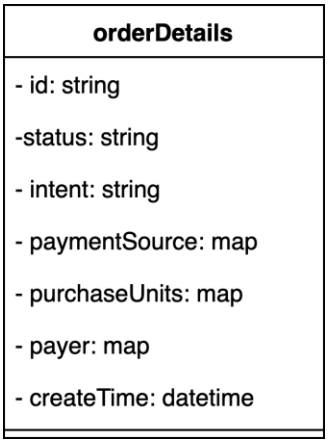
1.1.1.5.8 OrderDetails

```
{
  "id":"string",
  "status":"string",
  "intent":"string",
  "paymentSource":{
     "paypal":{
        "name":{
           "givenName":"string",
           "surname":"string"
        },
        "emailAddress":"string",
        "accountId":"string ID"
     }
  },
  ],
  "payer":{
     "name":{
        "givenName":"string",
        "surname":"string"
     },
     "emailAddress":"string",
     "payerId":"string"
  },
  "createTime":"dateTime",
  "links":[
     {
        "href":"string",
        "rel":"string",
        "method":"string"
     }
  }
```

Name	1.1.1.5.8 OrderDetails
Purpose	To show the details of an order.
Description	Once a user has confirmed an order, they are given the order details.
Requirements	9
Elements	**id:** 1.1.1.5.6.A
	status: 1.1.1.5.6.A
	intent: The intended action or Purposebehind a set of data.
	paymentSource: 1.1.1.5.6.A
	name: 1.1.1.5.6.A
	givenName: 1.1.1.5.6.A

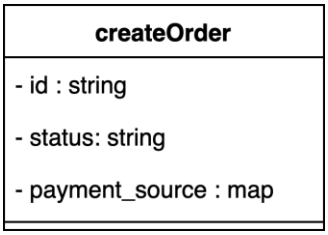
	surname: 1.1.1.5.6.A
	purchaseUnits: The quantity of trips the user is purchasing.
	referenceId: A specific code that has been assigned to the transaction.
	amount: The price of the product.
	currencyCode: The currency the amount is being set in.
	value: The numerical price of the product.
	payer: The client who is paying for the ride.
	emailAddress: 1.1.1.5.6.A
	paypal: The type of payment that is being used to order.
	accountId: PayPals unique id for the registered user.
	payerId: The user's unique identifier of their PayPal account.
	createTime: The time when the order has been processed.
	links: 1.1.1.5.6.A
	href: 1.1.1.5.6.A
	rel: 1.1.1.5.6.A
	method: 1.1.1.5.6.A
Referenced By	1.1.1.5 OrderService , 1.1.1.5.5 orderDetails, 1.1.1.5.10.3 validateDetails
View	JSON Schema

1.1.1.5.8.A OrderDetails



Name	1.1.1.5.8.A OrderDetails
Purpose	An object that takes data from the detailsJSON and turns it into an instance.
Description	This object can be read and/or transformed to the details of the order before it is displayed to the user.
Requirements	8
Elements	**id:** 1.1.1.5.6.A
	status: 1.1.1.5.6.A
	paymentSource: 1.1.1.5.6.A
	purchaseUnits: 1.1.1.5.8
	payer: 1.1.1.5.6.A
	createTime: 1.1.1.5.8
Referenced By	1.1.1.5 OrderService , 1.1.1.5.5 orderDetails
Viewpoint	Class Diagram

1.1.1.5.9 createOrder



Name	1.1.1.5.9 CreateOrder
Purpose	An object that takes data from the ride and the user and puts it into an instance.
Description	This object can be read and/or transformed to meet the needs of the of confirming an order before it gets sent to the backend of the system.
Requirements	8
Elements	**id:** 1.1.1.5.6.A
	status: 1.1.1.5.6.A
	paymentSource: 1.1.1.5.6.A
Referenced By	1.1.1.5 OrderService
Viewpoint	Class Diagram

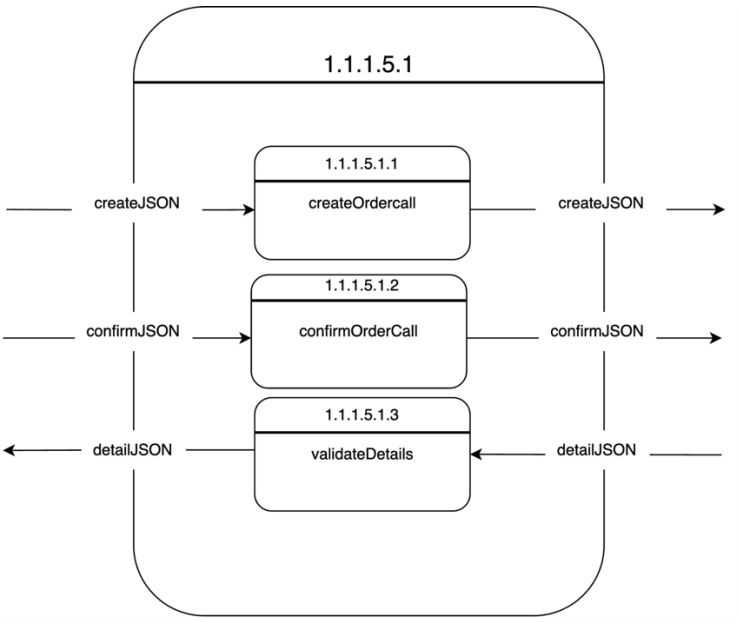
1.1.1.5.9.A CreateOrder

```
{
       "id": "string",
      "status": "string",
      "payment_source": {
            "paypal":{
                  "experience_context":{
                        "payment_method_preference":"string",
                        "landing_page":"string",
                        "user_action":"string"
                                    }
                  },
            },
      "links": [
      {
            "href": "string”,
            "rel": "string",
            "method": "string"
      },
      {
            "href": "string”,
            "rel": "string",
            "method": "string"
      }
      ]
}
```

Name	1.1.1.5.9.A CreateOrder
Purpose	A POST request for the PayPalAPI to create an order.
Description	A JSON schema that creates an order on the API side.
Requirements	8
Elements	**id:** 1.1.1.5.6.A
	status: 1.1.1.5.6.A
	payment_source: 1.1.1.5.6.A
	experience_context: Customizes the payer experience during the approval process for payment with PayPal.
	payment_method_preference: The merchant-preferred payment methods.

	login_page: The type of landing page to show on the PayPal site for customer checkout.
	user_action: Configures a Continue or Pay Now checkout flow on the PayPal Screen.
	href: 1.1.1.5.6.A
	links: 1.1.1.5.6.A
	rel: 1.1.1.5.6.A
	method: 1.1.1.5.6.A
Referenced By	1.1.1.5 OrderService
Viewpoint	JSON Schema

1.1.1.5.10 OrderConfirmation



Name	1.1.1.5.10 Order Service: Order Confirmation
Purpose	To notify the Ride Manager a seat has been reserved upon successful order completion.
Description	Once the user has officially paid for their trip and is sent to the chatroom with a confirmation message, the details of the payment and the seat they reserved are given to the Database Querier for Ride Manager to receive.
Requirements	9
Elements	**createJSON:** 1.1.1.5.9.A
	confirmJSON: 1.1.1.5
	detailJSON: 1.1.1.5
	createOrderCall: 1.1.1.5.10.1
	validateDetails: 1.1.1.5.10.3
	ConfirmOrdercall: 1.1.1.5.10.2
Referenced By	1.1.1.5 Order Service, 1.1.1.14 Order Screen , 1.3.4 PayPalAPI
Viewpoint	Data Flow Diagram

1.1.1.5.10.1 createOrderCall

```
createOrderCall(createOrder)

    FOR item IN createOrder:
        items.append({
            "name": createOrder[“name”],
            "description": createOrder[“description”],
            "unitAmount": {
                "currencyCode": createOrder[“currency”],
                "value": FORMAT(createOrder[“price”], "0.00")
            },
        })
        itemTotal ← item.price

    paypalPayload ← {
        "intent": "CAPTURE",
        "paymentSource": {
            "paypal": {
                "experienceContext": {
                    "paymentMethodPreference":
"IMMEDIATE_PAYMENT_REQUIRED",
                    "landingPage": "LOGIN",
                    "userAction": "PAY_NOW",
                    “returnUrl":
"https://example.com/returnUrl",
                    "cancelUrl": "https://example.com/cancelUrl"
                }
            }
        },
        "purchaseUnits": [
            {
                "invoiceId": createOrder[invoiceId],
                "amount": {
                    "currencyCode": createOrder[currency],
                    "value": FORMAT(itemTotal, "0.00"),
                "items": items
            }
        ]
    }

    authResponse ← POST(
     'https://api-m.sandbox.paypal.com/v1/oauth2/token', {
        'Accept': 'application/json',
        'Accept-Language': 'enUS'
    }, {
        'grantType': 'clientCredentials'
```

```
}, AUTH ←('YOUR_CLIENT_ID', 'YOUR_CLIENT_SECRET'))

IF authResponse.statusCode != 200:
    THROW ERROR
accessToken ← authResponse.json().accessToken
headers ← {
    'Content-Type': 'application/json',
    'PayPal-Request-Id': GENERATE_UUID(),
    'Authorization': 'Bearer ' + accessToken
}

response ← POST(
 'https://api-m.sandbox.paypal.com/v2/checkout/orders',
 headers,
 JSON.stringify(paypalPayload)
)

IF response.status_code == 200:
    RETURN response.json
ELSE:
    THROW ERROR
```

Name	1.1.1.5.10.1 createOrderCall
Purpose	Request sent to the PayPal API to create an order on their end.
Description	Post the order to PayPal using the createOrder object that was generated in the frontend.
Requirements	9
Elements	**Item:** A single element within createJSON file.
	Items: An array of all the elements.
	CreateOrder: 1.1.1.5.9.A
	Name: 1.1.1.5.6.A
	Description: 1.1.1.5.6.A
	unitAmount: The price and currency.
	currencyCode: The currency type.
	Value: The price of the item.
	itemTotal: Total price of the trip.
	paypalPayload: Data payload that is sent in the body of the HTTP request sent to the PayPal API.
	Intent: Purposeof the request.
	paymentSource: 1.1.1.5.6.A
	Paypal: Payment type.
	experienceContext: 1.1.1.5.5.A
	paymentMethodPreference: 1.1.1.5.5.A

	IMMEDIATE_PAYMENT_REQUIRED: Action required from PayPal.
	landingPage: The page where the client is directed to.
	LOGIN: Action required on login page.
	userAction: Action needed by the client.
	PAY_NOW: The action the user will complete.
	returnUrl: Where the user will be directed to.
	cancelUrl: Url page if the order gets cancelled.
	purchaseUnits: Order details.
	invoiceId: Order id that is specific to a single trip.
	Amount: Payment details.
	authResponse: Post the request to PayPal.
	Accept: File type acceptance.
	Accept-Language: Language the request is in.
	grantType: Type of access given to the client.
	AUTH: Both unique ids for the user provided by the PayPal API.
	accessTokens: Access for authentication in PayPal. Generated by PayPal.
	Headers: Data headers for the requests.
	Content-Type: File type acceptance.
	PayPal-Request-ID: The server stores keys for 6 hours. The API callers can request the times up to 72 hours by speaking to their Account Manager. It is mandatory for all single step create order calls.
	Authorization: An API-caller-provided JSON Web Token (JWT) assertion that identifies the merchant.
	Response: Returned JSON Schema.
	200: A successful request returns the HTTP 200 OK status code and a JSON response body that shows order details.
Referenced By	1.1.1.5.10 OrderConfirmation, 1.1.1.5.5.A createOrderJSON
Viewpoint	Pseudocode

1.1.1.5.10.2 confirmOrderCall

```
confirmOrderCall(confirmOrder)

    orderId ← confirmOrder[id]
    paymentSource ← confirmOrder[paymentSource]
    paypalPayload ← {
        "paymentSource": {
            "paypal": {
                "name": {
                    "givenName": paymentSource.name.givenName,
                    "surname": paymentSource.name.surname
                },
                "emailAddress": paymentSource.emailAddress,
                "experienceContext": {
                    "paymentMethodPreference":
"IMMEDIATE_PAYMENT_REQUIRED",
                    "brandName": "Rideshare",
                    "locale": "en-US",
                    "landingPage": "LOGIN",
                    "userAction": "PAY_NOW",
                    "returnUrl":
"https://example.com/returnUrl",
                    "cancelUrl":
"https://example.com/cancelUrl"
                }
            }
        }
    }
    headers ← {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer YOUR_ACCESS_TOKEN'
    }

    response ← POST(
        'https://example.com/v2/checkout/orders/' + orderId +
'/confirm-payment-source', headers,
JSON.stringify(paypalPayload))

    IF response.status_code = 200:
        RETURN response.json()
```

```
    ELSE:
        THROW ERROR
```

Name	1.1.1.5.10.2 confirmOrderCall
Purpose	Request sent to the PayPal API to confirm an order on their end.
Description	Payer confirms their intent to pay for the Order with the given payment source.
Requirements	9
Elements	**orderId:** The ID of the order for which the payer confirms their intent to pay.
	confirmOrder: 1.1.1.5.6
	paypal: Payment type.
	paypal: Payment type.
	brandName: The name of the application.
	locale: Location of the client.
	JSON.stringify : converts the object to a JSON string to send to the API.
Referenced By	1.1.1.5.10 Order Confirmation
Viewpoint	Pseudocode

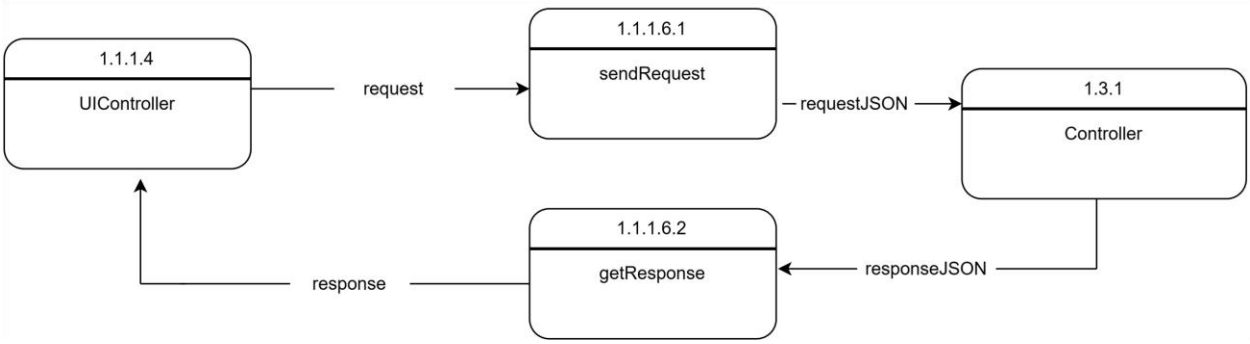
1.1.1.5.10.3 validateDetails

```
validateDetails(detailsJSON):

    requiredFields <- ["id", "status", "intent",
"paymentSource", "purchaseUnits", "payer", "createTime",
"links"]

    FOR EACH field IN requiredFields
        IF field NOT IN detailsJSON
            RETURN FALSE

    IF "paymentSource" IN detailsJSON
        IF "paypal" NOT IN detailsJSON["paymentSource"]
            RETURN FALSE
    ELSE:
        IF "name" NOT IN detailsJSON["paymentSource"["paypal"]
                RETURN FALSE
        IF "emailAddress" NOT IN
detailsJSON["paymentSource"]["paypal"]
                RETURN FALSE
        IF "accountId" NOT IN
detailsJSON["paymentSource"]["paypal"]
                RETURN FALSE

    IF "purchaseUnits" IN detailsJSON
        FOR EACH unit IN detailsJSON["purchaseUnits"]
            IF "referenceId" NOT IN unit
                RETURN FALSE
            IF "amount" NOT IN unit
                RETURN FALSE
            IF "currencyCode" NOT IN unit["amount"]
                RETURN FALSE
            IF "value" NOT IN unit["amount"]
                RETURN FALSE

    IF "payer" IN detailsJSON
        IF "name" NOT IN detailsJSON["payer"]
            RETURN FALSE
        IF "emailAddress" NOT IN detailsJSON["payer"]
            RETURN FALSE
```

```
        IF "payerId" NOT IN detailsJSON["payer"]
            RETURN FALSE

    IF "links" IN dataJSON
        FOR EACH link IN dataJSON["links"]
            IF "href" NOT IN link
                RETURN FALSE
            IF "rel" NOT IN link
                RETURN FALSE
            IF "method" NOT IN link
                RETURN FALSE

    RETURN true
```

Name	1.1.1.5.10.3 validateDetails
Purpose	Verify the requests to retrieve the details of order are all there before sending it to front end.
Description	The method checks all required information is present in the file to make sure it gives the correct results to the user.
Requirements	9
Elements	**RequiredFields:** The necessary data needed to show the user their order has been confirmed through PayPal.
	orderDetails: 1.1.1.5.8
Referenced By	1.1.1.5.10 orderConfirmation
Viewpoint	Pseudocode

1.1.1.6 RideManagementService



Name	1.1.1.6 RideManagementService
Purpose	This diagram defines the data flow through the RideManagementService.
Description	This diagram depicts how information from the user is used in the application layer of the front-end of the application to retrieve and update information about rides.
Requirements	4
Elements	**Controller:** 1.3.1
	UI Controller: 1.1.1.4
	Send Request: Send the requests as JSON to the Controller.
	Get Response: 1.1.1.6.2
	Request: 1.1.1.6.5
	Response JSON: A JSON list of trips
	Response: The response in object form
Referenced By	1.1.1 UserInterface
Viewpoint	Data Flow Diagram

1.1.1.6.1 sendRequest

```
sendRequest(request)
   requestJSON = convertToJSON(request)
   sendToController(requestJSON)
```

Name	1.1.1.6.1 sendRequest
Purpose	The sendRequest function takes a UI request, converts it to JSON, sends it to the controller, waits for the response, converts the JSON response into an object, and returns it for frontend use.
Description	This pseudocode describes an UI request, sends it to the backend and returns a usable response.
Requirements	4, 5
Elements	**request:** holds the request object from the UI, containing user info such as trip selection or any filtering criteria.
	requestJSON: the request object converted into JSON to be sent to the backend.
	responseJSON: the response received in the backend in the JSON format, containing the trip data.
	convertToJSON: Converts the request object to JSON.
	sendToController: Sends the JSON data to the controller over http.
Referenced By	1.1.1.6 Ride Management Service
Viewpoint	Pseudocode

1.1.1.6.2 getResponse

```
getResponse(responseJSON)

   FOREACH trip in responseJSON

      tripObjects.append(new Trip(trip))

   RETURN tripObjects
```

Name	1.1.1.6.2 getResponse
Purpose	Converts a response from the rideManagementGateway from JSON into objects usable by the Frontend.
Description	Receives request information and interprets it into a Trip Object. This object is then passed to the UI controller.
Requirements	4
Elements	**ResponseJSON:** a list that contains JSON objects of type trip.
	Trip: 1.3.2.1.1 Trip
	TripObject: 1.2.3.7 Trip
Referenced By	1.1.1.6 Ride Management Service
Viewpoint	Pseudocode

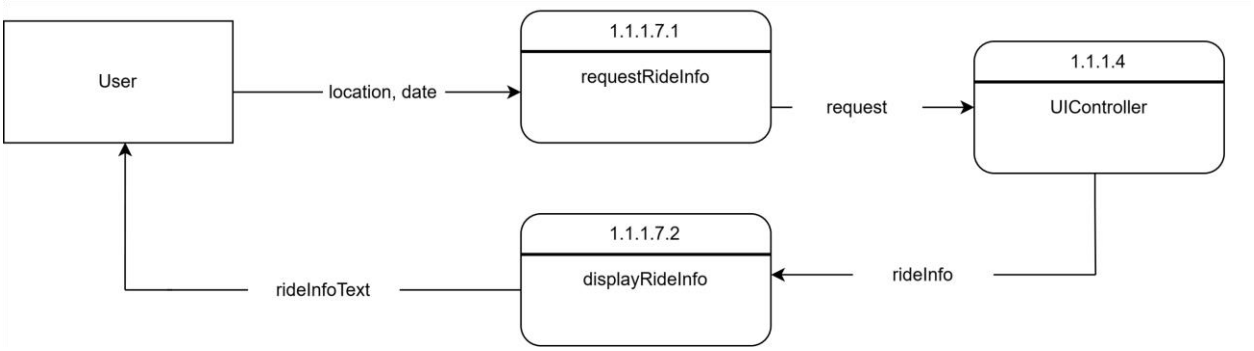
1.1.1.6.5 Request

```
{
   "startlocation":{
      "coordinates":"string",
      "address":{
         "line1":"string",
         "line2":"string",
         "city":"string",
         "state":"string",
         "zip":"string"
      }
   },
   "endLocation":{
      "coordinates":"string",
      "address":{
         "line1":"string",
         "line2":"string",
         "city":"string",
         "state":"string",
         "zip":"string"
      }
   },
   "customRadius": "float”,
   "tripTime":"string"
}
```

Name	1.1.1.6.5 Request
Purpose	This diagram defines the structure of the JSON requesting a list of trips from the backend.
Description	This diagram depicts how information will be formatted so that trips can be found which correspond to the start and end locations as well as the time given in the request.
Requirements	4
Elements	**Start Location:** The place the rider wants to start from.
	End Location: The place the rider wants to arrive at.
	Trip Time: The date and time which the rider would like to approximately leave at, encoded into a string.
	Coordinates: A string representing an exact spot on the earth. Can be null if the user entered an address instead.
	Address: A set of information which describes a certain location. Can be null if the user entered coordinates instead.

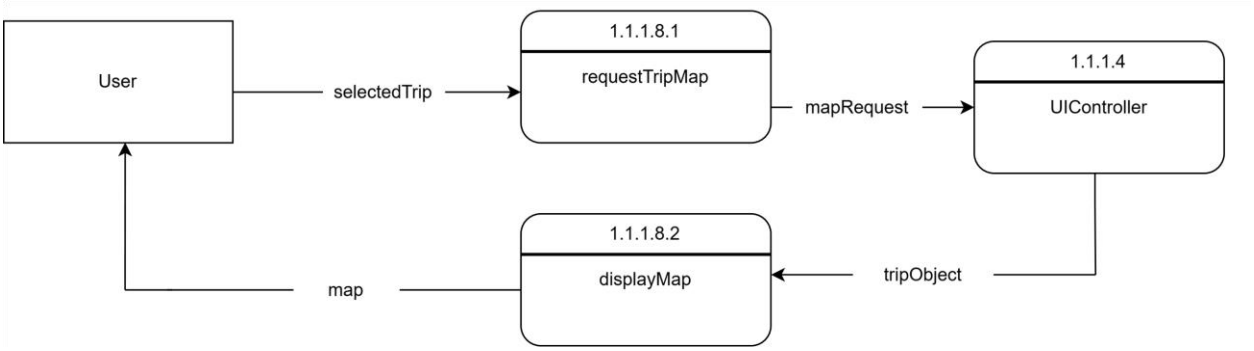
	Line 1: Information in the first line of an address, including the house number and street name.
	Line 2: Additional address information such as unit or apartment number.
	City: The Nameof the city on the address.
	State: The Nameof the state on the address.
	Zip: The zip code of the address as a string.
	Custom Radius: The radius the user has chosen which the trip must be within from their chosen start and end locations.
Referenced By	1.1.1.6 RideManagementService
Viewpoint	JSON Schema

1.1.1.7 RideManagementScreen



Name	1.1.1.7 RideManagementScreen
Purpose	This diagram defines the data flow required in the RideManagementScreen so that the user can use the rides.
Description	This diagram depicts how information from the user is used to display applicable rides to them.
Requirements	5
Elements	**User:** The person interacting with the app.
	UI Controller: 1.1.1.4
	Request Ride Info: Sends a request to the application layer for rides applicable to the location and date provided.
	Display Ride Info: Displays the ride options to the user
	Location, Date: the location will be given as an address in order to retrieve relevant rides.
	Request: A request to retrieve rides from the application layer.
	Ride Info: A collection of ride info objects which have been provided by the application layer.
	Ride Info Text: A list of rides presented to the user.
Referenced By	1.1.1 UserInterface
Viewpoint	Data Flow Diagram

1.1.1.8 MapScreen



Name	1.1.1.8 MapScreen
Purpose	This diagram defines the data flow required to maintain the MapScreen for the user.
Description	This diagram depicts how information from the user is used to retrieve and display a map of the trip.
Requirements	5
Elements	**User:** The person interacting with the app.
	UI Controller: 1.1.1.4
	Request Trip Map: Sends a request to the application layer for the information needed to display a map.
	Display Map: 1.1.1.8.2 displayMap
	Selected Trip: The user's selection of a trip to view.
	Map Request: A request for map information on a specific trip, represented by the trip's id.
	Map Data: The data needed for the UI to display a map.
	Map: The actual map displayed to the user to easily view the trip.
Referenced By	1.1.1 UserInterface
Viewpoint	Data Flow Diagram

1.1.1.8.1 requestTripMap

```
requestTripMap(selectedTrip)
   MapRequest = [“trip.startLocation”, “trip.endLocation”,
“trip.ID”]
   SendToUIController(mapRequest)
```

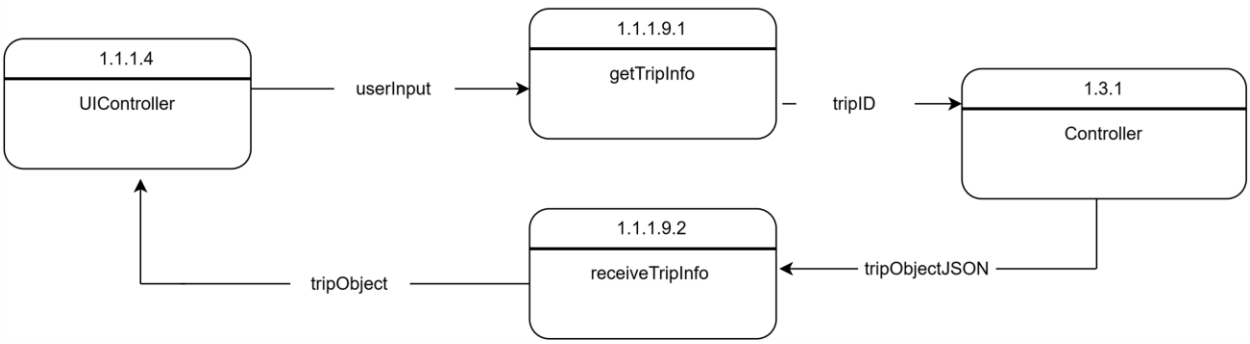
Name	1.1.1.8.1 requestTripMap
Purpose	Processes the user's selected trip information to generate a request for map data.
Description	This function takes the trip information from the selected trip and creates a ‘mapRequest’. The request is sent to the application layer to retrieve the necessary data for displaying a map.
Requirements	5
Elements	**requestTripMap:** Generates a request for trip-related map data based on the selected trip.
	mapRequest: 1.3.2.1.1 Trip
	selectedTrip: 1.1.1.8 MapScreen
	trip: 1.3.2.1.1 Trip, 1.3.3.1.1 TripLocations
Referenced By	1.1.1.8 MapScreen
Viewpoint	Pseudocode

1.1.1.8.2 displayMap

```
displayMap(mapData)
   map = generateMap(mapData)
   RETURN map
```

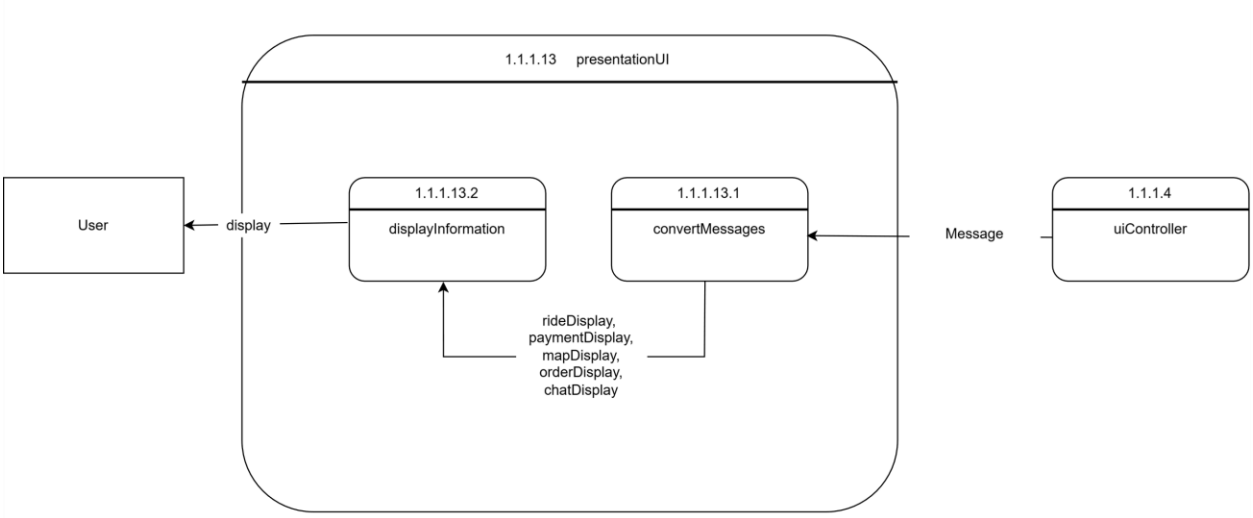
Name	1.1.1.8.2 displayMap
Purpose	Shows how Display Map will operate within the system.
Description	Map data from the UI Controller is used to create a map to display for the user.
Requirements	5
Elements	**mapData:** The data needed for the UI to display a map.
	generateMap: A function provided by Google Maps that processes 'mapData' and generates a map. Google Maps SDK
	Map: The visual representation of the trip, created from 'mapData' and displayed to the User. Google Maps SDK
Referenced By	1.1.1.8 MapScreen
Viewpoint	Pseudocode

1.1.1.9 MapInterfaceService



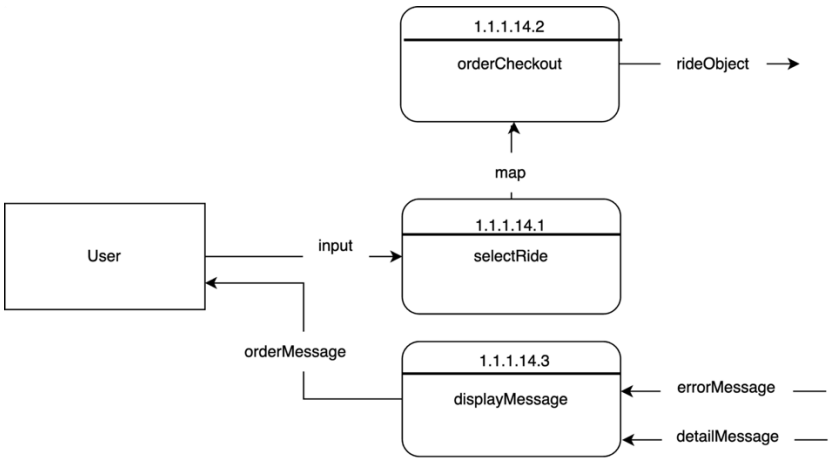
Name	1.1.1.9 MapInterfaceService
Purpose	This diagram defines the data flow through the MapInterfaceService.
Description	This diagram depicts how information from the user is used in the application layer of the Frontend of the application to retrieve map information for visually displaying a map to the user.
Requirements	5
Elements	**UIController:** 1.1.1.4
	Get Trip Info: Sends the trip Id to the backend for retrieval of any data needed for displaying the map.
	Controller: 1.3.1
	Request Map: Requests map information from the Maps API based on the trip info.
	Maps: 3rd party MapAPI Google Maps API Documentation
	Send Map Information: Sends the information acquired from the Maps API to the UI Controller.
	User Input: The User's selection which determines which trip they want to view.
	Trip ID: The identifier of the trip which the user has selected.
	Trip Object JSON: 1.3.2.1.1
Referenced By	1.1.1 UserInterface
Viewpoint	Data Flow Diagram

1.1.1.13 PresentationUI



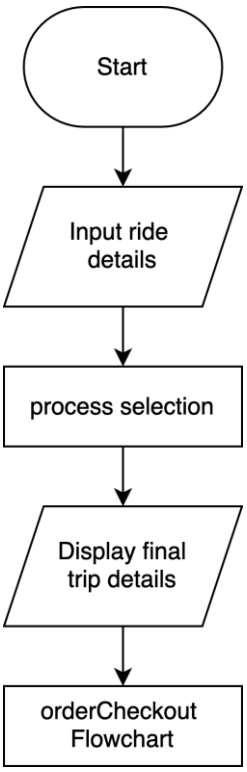
Name	1.1.1.13 PresentationUI
Purpose	Unpack messages regarding rides, payments, maps, orders, and chats, then show them to the user.
Description	Displays information in a user-friendly way.
Requirements	4-6, 10, 18-20, 22
Elements	**User:** Person using the app
	Message: Message is the type of message the controller is sending so it can be displayed. There are different types of messages such as display, rideMessage, paymentMessage, mapMessage, orderMessage, chatMessage, and uiController found in 1.1.1.4
	convertMessages: Unpacks the various messages into a display the user can read well.
	rideDisplay: Readable, user-friendly display regarding the information about a ride.
	paymentDisplay: Readable, user-friendly display regarding the information about a payment.
	mapDisplay: Readable, user-friendly display regarding the information about a map.
	orderDisplay: Readable, user-friendly display regarding the information about an order.
	chatDisplay: Readable, user-friendly display regarding the information about a chat.
	displayInformation: Combines the different displays and puts them on the screen for the user.
Referenced By	1.1.1.2 PaymentUserInterface , 1.1.1.18 ChatUserInterface
Viewpoint	Data Flow Diagram

1.1.1.14 OrderScreen



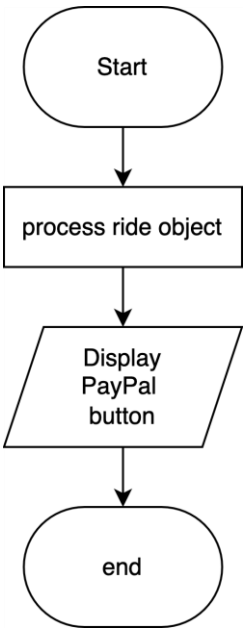
Name	1.1.1.14 Order Screen
Purpose	The user's interaction with the Order Screen.
Description	One the rider has selected a ride they are navigated to the Order Screen. The Order Screen is where the rider is prompted to input their desired pickup and drop off location which will be stored into an object. The final trip details are displayed back to the user before they are navigated to the payment screen.
Requirements	5
Elements	**User**: The user is the rider.
	input: Input request from the user for a specific ride.
	selectRide: 1.1.1.14.1
	map: Specific identification number that is assigned when a new trip is posted by a driver.
	orderCheckout: 1.1.1.14.2
	rideObject: data that pertains to the ride
	errorMessage: The message that is sent to the screen if the order cannot be validated.
	detailMessage: The message with the order confirmation details
	displayMessage: 1.1.1.14.3
	orderMessage: Displays the final order information or error to the user
Referenced By	1.1.1 User Interface
Viewpoint	Data Flow Diagram

1.1.1.14.1 selectRide



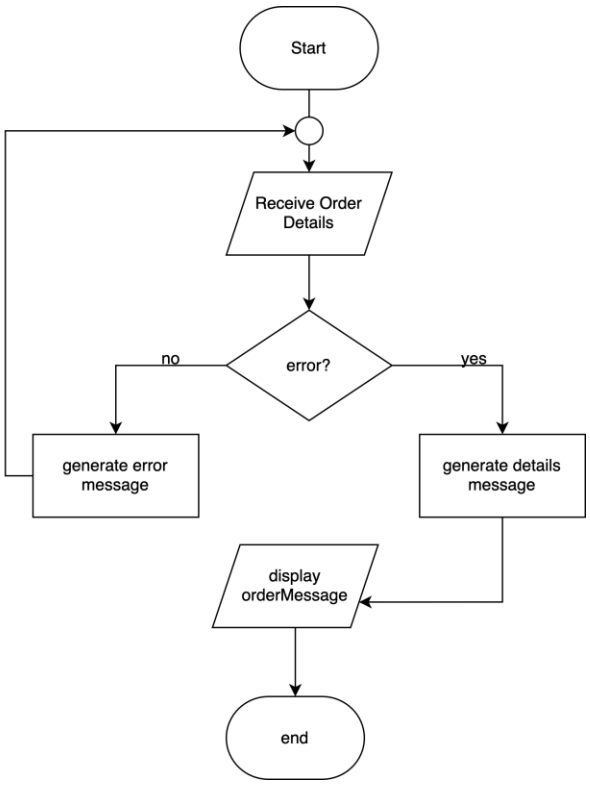
Name	1.1.1.14.1 selectRide
Purpose	The users interact with the trip they selected and views the order before proceeding to checkout.
Description	The rider inputs details about start location and end location for the ride. Once the selection has been processed, the details are displayed for verification before they are directed to orderCheckout.
Requirements	5
Elements	**Input ride details:** User inputs the details of the trip.
	Process selection: 1.2.3.0.1
	Display final trip details: The details of the trip are presented back to the user before they proceed to checkout.
	orderCheckout flowchart: 1.1.1.14.2
Referenced By	1.1.1.14 Order Screen
Viewpoint	Flowchart

1.1.1.14.2 orderCheckout



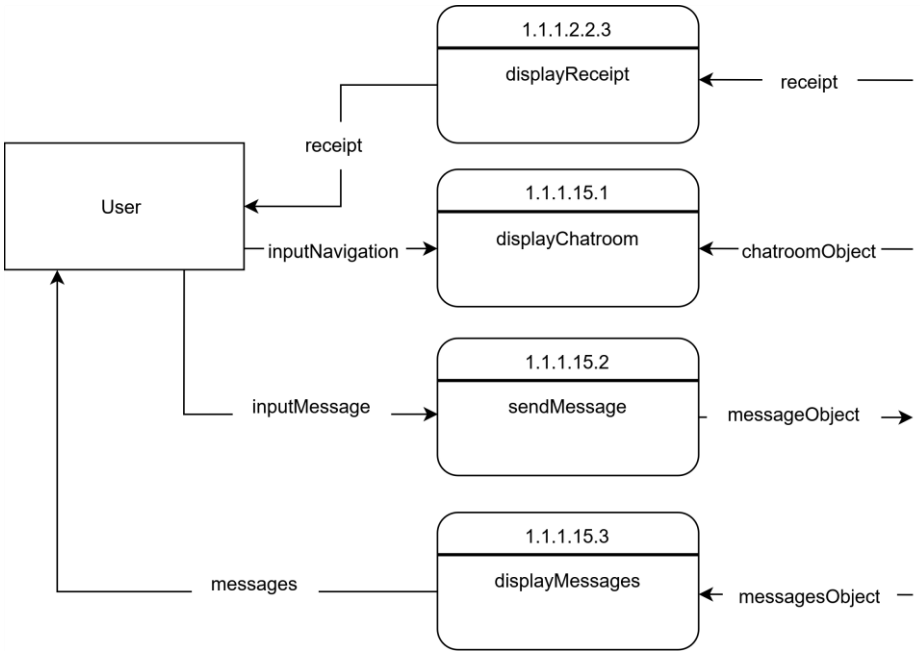
Name	1.1.1.14.2 orderCheckout
Purpose	The ride object is generated from the user's details, and then the paypal button to check out on the API is given.
Description	Once a trip is confirmed by the user and they want to purchase, the object of a trip is generated, and the user is directed to the PayPal screen to continue their order on their end.
Requirements	9
Elements	**Process ride object:** Compiles all the trip data into a trip object.
	Display PayPal button: The users directed to the PayPal screen.
Referenced By	1.1.1.14 Order Screen , 1.1.1.14.1 selectRide
Viewpoint	Flowchart

1.1.1.14.3 displayMessage



Name	1.1.1.14.3 displayMessage
Purpose	When the user confirms the user's intent to pay, the PayPal screen will close and they are presented with an orderMessage.
Description	The user is presented with an orderMessage that confirms they reserved a trip.
Requirements	9
Elements	**Receive order details:** The function is given the details object.
	error?: Checks to make sure all elements exist in the object before presenting it to the user.
	yes: If there is an error, the user is given an orderMesssage that says there was an error.
	no: If all information is there, the user is given an orderMessage that gives the order confirmation details.
	generate error message: An error message if all the elements are not found.
	Generate details message: A message that contains details about the order confirmation that occurred in PayPal.
	Display orderMessage: 1.1.1.14
Referenced By	1.1.1.14 Order Screen
Viewpoint	Flowchart

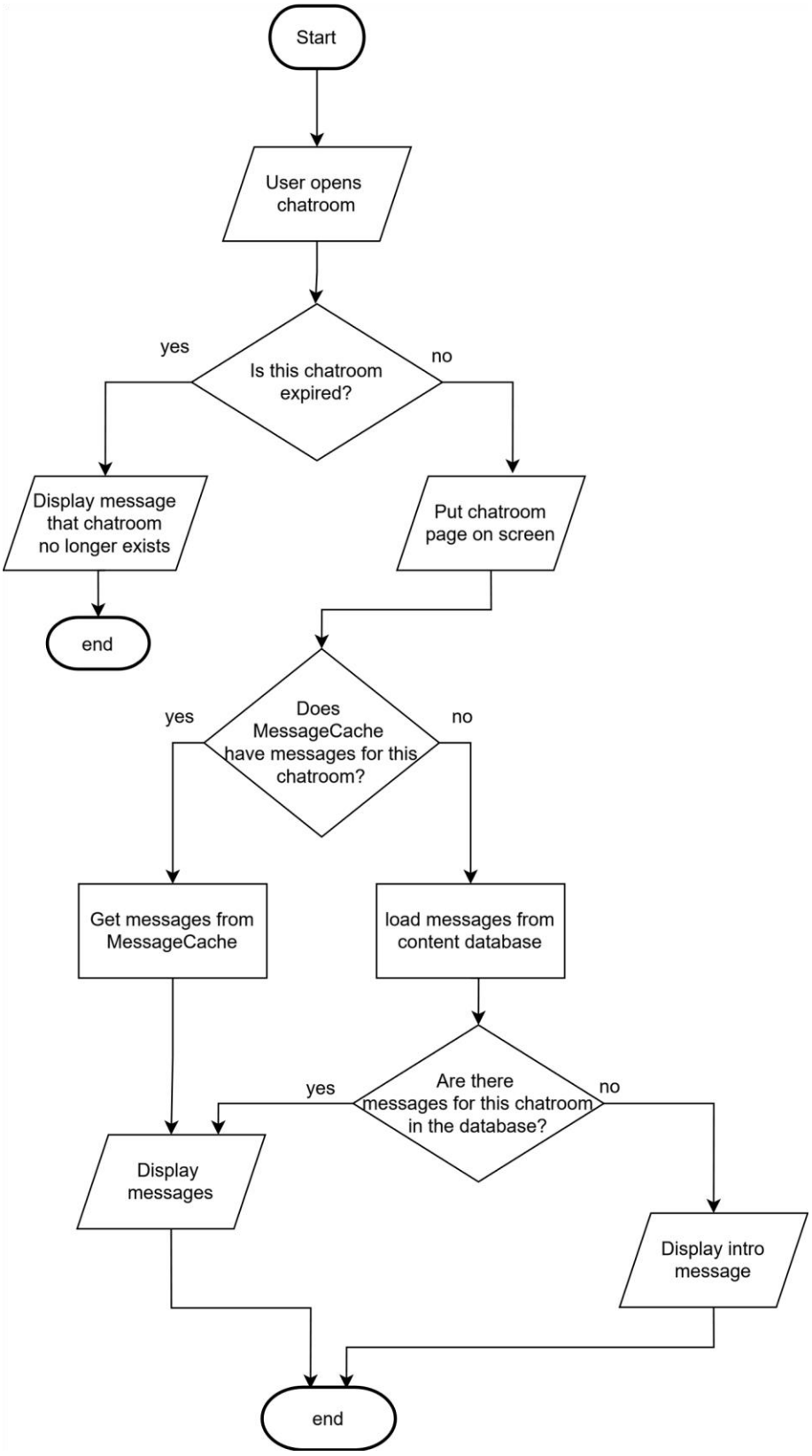
1.1.1.15 ChatScreen



Name	1.1.1.15 User Interface: Chat Screen
Purpose	The Purposeis to show how the rider or the driver will interact with the chatroom screen.
Description	When a ride has been paid for, they are directed to a chatroom with other users who have also secured that ride along with the driver. A confirmation message is sent and stored to go the Chatroom Service and further messages continue to be displayed.
Requirements	24 - 26
Elements	**User:** User of the app, either a Rider or a Driver.
	UI Controller: 1.1.1.4
	displayReceipt: Provides the user with a receipt detailing their order and payment.
	inputNavigation: Prompts the user for a type of request to move to the next screen.
	displayChatroom: 1.1.1.15.1
	receipt: 1.2.4.2
	chatroomRoomObject: Data that provides the necessary information of who can view the specific chat.
	sendMessage: 1.1.1.15.2
	inputMessage: Input text from the user.
	messageObject: 1.2.6.5
	messages: Displays messages between the users onto the chatroom screen

	displayMessages: Formats and shows the user the current messages between the users.
Referenced By	1.1.1 User Interface
Viewpoint	Data Flow Diagram

1.1.1.15.1 Chat Screen: displayChatroom



Name	1.1.1.15.1 displayChatroom
Purpose	To illustrate the steps taken when displaying a chatroom on the user screen.
Description	A flowchart that displays a chatroom and the messages in a chatroom.
Requirements	24-26
Elements	**chatroom:** 1.2.6.1.1
	MessageCache: 1.1.1.16.4
	loadMessages: 1.1.1.17.5
	contentDatabase: 1.3.2.1
	displayMessages: 1.1.1.15.4
	Display intro message: Display a message at the start of the chatroom that introduces the chatroom to the users.
Referenced By	1.1.1.15 ChatScreen
Viewpoint	Flowchart

1.1.1.15.2 Chat Screen: sendMessage

```
sendMessage()

   GET INPUT from user

   message ← new Message
   message.text ← user input
   message.timestamp ← DateTime.now
   message.senderID ← user.ID

   FOR member in chatroom.members
      IF member.ID != message.senderID
         message.targetID ← member.ID

   messages ← [message]
   displayMessages(messages)

   messageJSON ← chatService.transfromMessageToJSON(message)

   CMGateway.sendMessage(messageJSON)
```

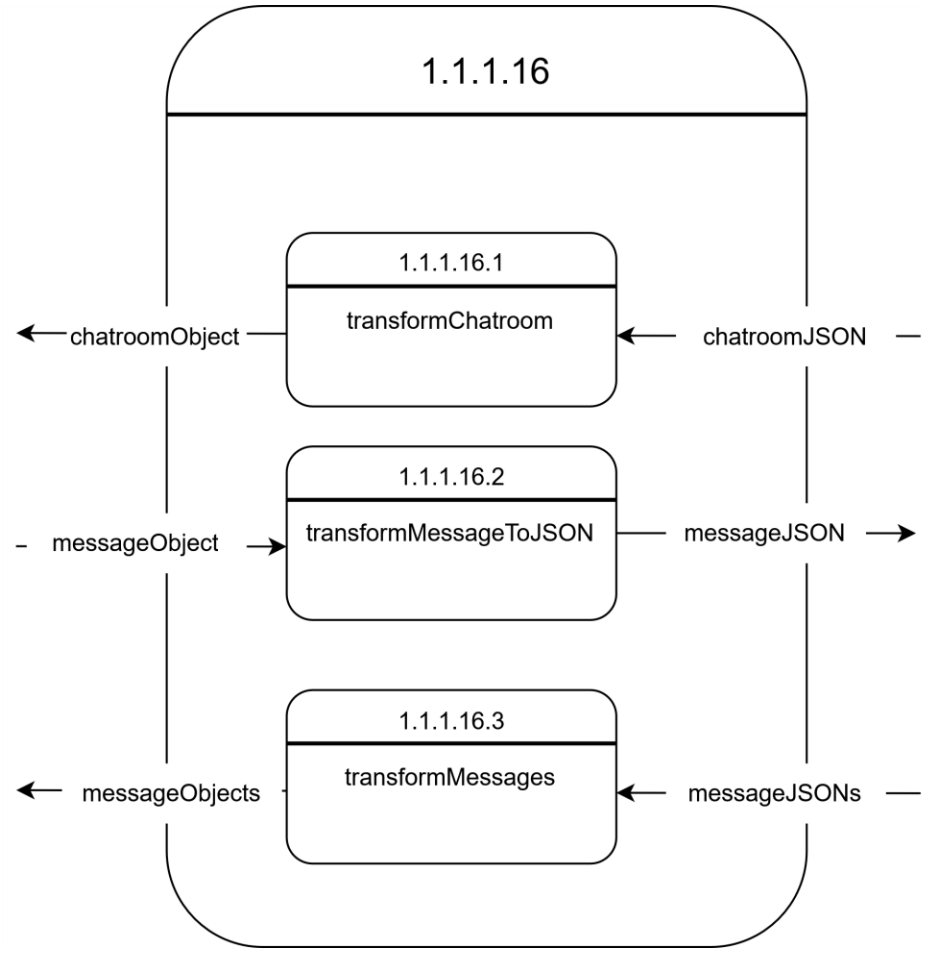
Name	1.1.1.15.2 Chat Screen: sendMessage
Purpose	To demonstrate how messages are sent from the chat screen.
Description	A function that takes input from the user, creates a message object from that input, displays the message to the screen, converts the object to json, and then sends the json to the CMGateway.
Requirements	24-26
Elements	**Message:** 1.2.6.5
	message: A variable to hold the message object.
	chatroom: 1.2.6.1.1
	displayMessages(): 1.1.1.15.4
	messageJSON: A variable to hold the message JSON object.
	CMGateway: 1.1.1.17
Referenced By	1.1.1.15 Chat Screen
Viewpoint	Pseudocode

1.1.1.15.4 Chat Screen: displayMessages

```
displayMessages(messages)

   FOR message IN messages
      IF screenUserID == message.senderID
         message.display(true)
      ELSE
         message.display(false)
```

Name	1.1.1.15.4 Chat Screen: displayMessages
Purpose	To display messages on the user screen.
Description	A function that displays a group of messages.
Requirements	24-26
Elements	**messages:** A list of message objects.
	message: 1.2.6.5
	screenUserID: The ID associated with the account of the user whose screen is receiving the display.
	message.display(): 1.2.6.5.1
Referenced By	1.1.1.15 Chat Screen, 1.1.1.15.1 Chat Screen: displayChatroom
Viewpoint	Pseudocode

1.1.1.16 Chat Service



Name	1.1.1.16 User Interface: Chat Service
Purpose	This diagram expresses the flow of receiving message history that existed in the backend and loading the current messages. The Chat messages are then displayed to the user.
Description	The messages that have expired and are stored in the backend are received in the Chat Service, once those have been received the recent messages are loaded and stored in the frontend. All the messages are then displayed to the screen.
Requirements	24-26
Elements	**chatroomObject:** 1.2.6.1.1
	transformChatroom: 1.1.1.16.1
	messageJSON: 1.3.2.1.6
	chatroomJSON: 1.3.2.1.5
	messageObject: 1.2.6.5
	transformMessagesToJSON: 1.1.1.16.2
	transformMessages: 1.1.1.16.3
Referenced By	1.1.1 User Interface, 1.1.1.17 CMGateway

Viewpoint	Data Flow Diagram

1.1.1.16.1 transformChatroom

```
transformChatroom(chatroomJSON)

     chatroom <- new Chatroom
     chatroom.ID <- chatroomJSON[“id”]

     creationDateParts <- SPLIT chatroomJSON[“creationDate”] BY
“-”

     creationDate <- new DateTime
     creationDate.day <- creationDateParts[0]
     creationDate.month <- creationDateParts[1]
     creationDate.year <- creationDateParts[2]
     chatroom.creationDate <- creationDate

expirationDateParts <- SPLIT chatroomJSON[“expirationDate”] BY
“-”

     expirationDate <- new DateTime
     expirationDate.day <- expirationDateParts[0]
     expirationDate.month <- expirationDateParts[1]
     expirationDate.year <- expirationDateParts[2]
     chatroom.expirationDate <-  expirationDate

     chatroom.memberIDs <- []

     FOR id in chatroomJSON[“members”]
        chatroom.addMember(id)

     messages <- transformMessages(chatroomJSON[“messages”])

FOR message in messages
        chatroom.addMessage(message)

     RETURN chatroom
```

Name	1.1.1.16.1 transformChatroom
Purpose	To demonstrate the function that converts a chatroom JSON schema into a chatroom object.

Description	Pseudocode that takes a chatroom JSON object as an argument, creates a new chatroom object, and assigns the coordinating variables from the JSON to the object member variables.
Requirements	25, 26
Elements	**chatroomJSON**: 1.3.2.1.5
	chatroom: 1.2.6.1.1
	creationDate: A DateTime object to represent the date when the chatroom was created.
	expirationDate: A DateTime object to represent a future date when the chatroom will be disabled/deleted.
	messages: A local variable to hold all the message objects.
Referenced By	1.1.1.16 Chat Service
Viewpoint	Pseudocode

1.1.1.16.2 transformMessageToJSON

```
transformMessageToJSON(message)

   messageJson <- {}
   messageJson [“senderID”] <- message.senderID
   messageJson [“targetID”] <- message.targetID
   messageJson [“text”] <- message.text
   messageJson [“timestamp"][“hour”] <- message.timestamp.hour
   messageJson [“timestamp"][“minute”] <-
message.timestamp.minute

   dateString <- “”
   dateString += message.timestamp.day formatted as String
   dateString += “-”
   dateString += message.timestamp.month formatted as String
   dateString += “-”
   dateString += message.timestamp.year formatted as String
   messageJson [“timestamp”][“date”] <- dateString

   RETURN messageJson
```

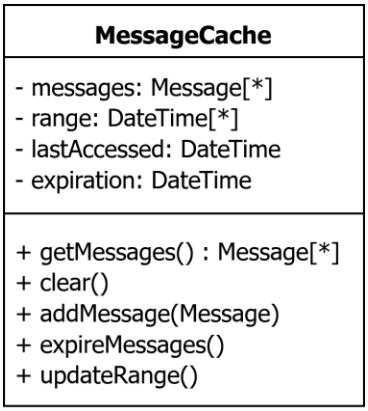
Name	1.1.1.16.2 transformMessageToJSON
Purpose	To demonstrate how one might code the process of converting a group of message objects into a group of message JSONs.
Description	A function that takes a list of message objects, creates message JSON objects based off of them, and then returns a list of message JSONs.
Requirements	25, 26
Elements	**json:** a variable to hold the message JSON object that is made.
	message: 1.2.6.5
Referenced By	1.1.1.16 Chat Service
Viewpoint	Pseudocode

1.1.1.16.3 transformMessages

```
transformMessages(messageJSONs)

   messages <- []

   FOR messageJSON in messageJSONs
      message <- new Message
      message.senderID <- messageJSON[“senderID”]
      message.targetID <- messageJSON[“targetID”]
      message.text <- messageJSON[“text”]

      timestamp <- new DateTime

      dateParts <- SPLIT messageJSON[“timestamp”][“date”] BY
“_”
      timestamp.day <- dateParts[0]
      timestamp.month <- dateParts[1]
      timestamp.year <- dateParts[2]

      timestamp.hour <- messageJSON[“timestamp”][“hour]
      timestamp.minute <-  messageJSON[“timestamp”][“minute”]
      message.timestamp <- timestamp

      messages.append(message)

   RETURN messages
```

Name	1.1.1.16.3 transformMessages
Purpose	Demonstrate how to convert a message JSON to a message object.
Description	A function that takes a message JSON as an argument and creates a message object using the JSON data.
Requirements	25, 26
Elements	**messageJSON:** 1.3.2.1.6
	message: A variable to hold the new message object.
	new Message: 1.2.6.5
	messages: An empty list to hold all the message objects.
Referenced By	1.1.1.16 Chat Service
Viewpoint	Pseudocode

1.1.1.16.4 MessageCache



Name	1.1.1.16.4 MessageCache
Purpose	To demonstrate an object that holds all the necessary information and functionality needed for MessageCache.
Description	A class diagram of a MessageCache object.
Requirements	23, 25
Elements	**messages:** a list of Messages.
	range: a list containing two DateTime objects that determine the two dates that all of the dates belonging to the Messages must be within, in order for the messages to be stored in the cache.
	lastAccessed: The date and time when the cache was last accessed.
	expiration: A calculated future date and time when the cache is no longer needed and expires.
	getMessages(): A method meant to retrieve the messages attribute.
	clear(): Method that replaces the messages list with an empty list.
	addMessage(): 1.1.1.16.4.3
	expireMessages(): 1.1.1.16.4.4
	updateRange(): A method that updates the dates in range.
Referenced By	1.1.1.17 CMGateway, 1.1.1.15.1 displayChatroom
Viewpoint	Class Diagram

1.1.1.16.4.3 addMessage

```
addMessage(message)

   messages.append(message)

   IF range[0].Date != message.timestamp.Date
      updateRange()
```

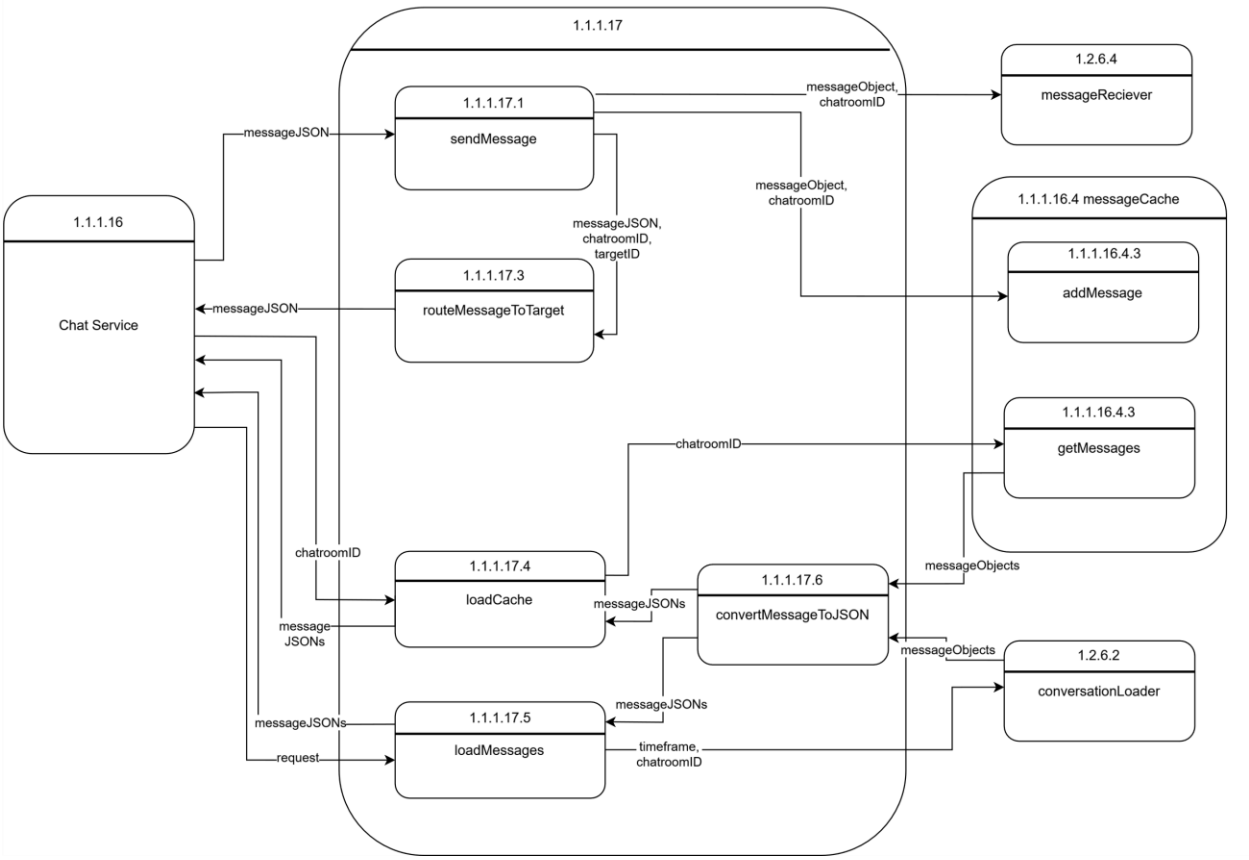
Name	1.1.1.16.4.3 addMessage
Purpose	To demonstrate how the addMessage method from MessageCache will function.
Description	A function that adds a message to the messages member variable, then calls updateRange() to update the MessageCache date range to include the date of the most recently added message.
Requirements	23, 25
Elements	**message:** 1.2.6.5
	messages: 1.1.1.16.4
	range[0].Date: The date portion of the first DateTime object in the range member variable of MessageCache.
	message.timestamp.Date: The date portion of the DateTime object of the timestamp of the message object.
	updateRange(): 1.1.1.16.4
Referenced By	1.1.1.16.4 MessageCache
Viewpoint	Pseudocode

1.1.1.16.4.4 expireMessages

```
expireMessages()

   IF messages[messages.length -1].timestamp.Date <
range[0].Date
      clear()
   ELSE
      FOR i IN range(0, messages.length -1)
         IF messages[i].timestamp.Date < range[0].Date
            messages.remove(messages[i])
```

Name	1.1.1.16.4.4 expireMessages
Purpose	Demonstrate the expireMessage function, which makes sure that the cache storage doesn't store unnecesary messages.
Description	Pseudocode for a function that checks to make sure all the messages within the cache are in the correct date range.
Requirements	23, 26
Elements	**messages[messages.length-1].timestamp.Date:** The date portion of the timestamp of the most recent message in the list member variable "messages".
	range[0].Date: The date portion of the first date in the list member variable range (not to be confused with the function range()). The older of the two dates, or the start of the date range.
	messages.length: The number of objects in messages.
	clear(): 1.1.1.16.4.2
Referenced By	1.1.1.16.4 MessagesCache
Viewpoint	Pseudocode

1.1.1.17 CMGateway



Name	1.1.1.17 CMGateway
Purpose	To illustrate the data flow facilitated by the Communications Gateway Controller.
Description	A data flow diagram of the CMGateway connecting the ChatScreen to the backend CommunicationManager.
Requirements	23, 25, 26
Elements	**ChatService:** 1.1.1.16
	sendMessage: Receives the incoming message from a user and sends it to the MessageReceiver and the cache.
	loadCache: Collects the stored messages from messageCache, formats them for the UI, and sends them back to the ChatScreen.
	loadMessages: 1.1.1.17.5
	convertMessageToJSON: Extracts the necessary information from the Message and turns it into a JSON object.
	routeMessageToTarget: Finds the chatroom using the chatroomID and senderID, and sends the messages to that chatroom.
	messageReciever: 1.2.6.4
	conversationLoader: 1.2.6.2
	messageCache: 1.1.1.16.4

	addMessage: 1.1.1.16.4
	getMessages: 1.1.1.16.4
	messageObject: 1.2.6.5
	messageJSON: 1.3.2.1.6
	chatroomID: A key unique to the chatroom, meant to identify it among other chatrooms. This key is generated when the chatroom is created in 1.2.6.1 Chatroom Manager. It is stored and passed in the messageJSON.
	targetID: The userID of the one who will receive the message. The ID is generated when the user's account is created and then identified as the targetID when the message is sent. It is stored and passed in the messageJSON.
	timeframe: A range consisting of a start time and end time, to be used to determine which messages to retrieve.
	request: 1.1.1.17.5.1
Referenced By	1.1.1.15.2 sendMessage, 1.2.6 CommunicationManager, 1.2.6.4 MessageReceiver, 1.2.6.2 Conversation Loader, 1.2.6.1 ChatroomManager
Viewpoint	Data Flow Diagram

1.1.1.17.5 loadMessages

```
loadMessages(request)

   chatroomID <- request["chatroomID"]

   timeframeStart <- new DateTime
   timeframeStart.day <- request["from"]["day"]
   timeframeStart.month <- request["from"]["month"]
   timeframeStart.year <- request["from"]["year"]

   timeframeEnd <- new DateTime
   timeframeEnd.day <- request["to"]["day"]
   timeframeEnd.month <- request["to"]["month"]
   timeframeEnd.year <- request["to"]["year"]

   timeframe <- [timeframeStart, timeframeEnd]

   messageJSONs <-
conversationLoader.retrieveMessages(chatroomID, timeframe)


   RETURN messageJSONs
```

Name	1.1.1.17.5 loadMessages
Purpose	Start the process of loading a group of messages by retrieving them from the Conversation Loader.
Description	A function that takes a request from the user, sends the request information to the Conversation Loader, recieves messages back from the Conversation Loader, converts the message objects into JSON objects, then returns the list of message JSON objects.
Requirements	26
Elements	**request:** 1.1.1.17.5.1
	chatroomID: The key, unique to the chatroom, used to identify it among other chatrooms.
	timeframeStart: A variable to hold the DateTime object that represents the beginning of the timeframe range.
	timeframeEnd: A variable to hold the DateTIme object that represents the end of the timeframe range.
	timeframe: A list that holds timeframeStart and timeframeEnd.
	conversationLoader.retrieveMessages: 1.2.6.2
	messageJSONs: A local list of message JSON objects.
	messageJSON: 1.3.2.1.6
Referenced By	1.1.1.17 CMGateway, 1.1.1.15.1 displayChatroom

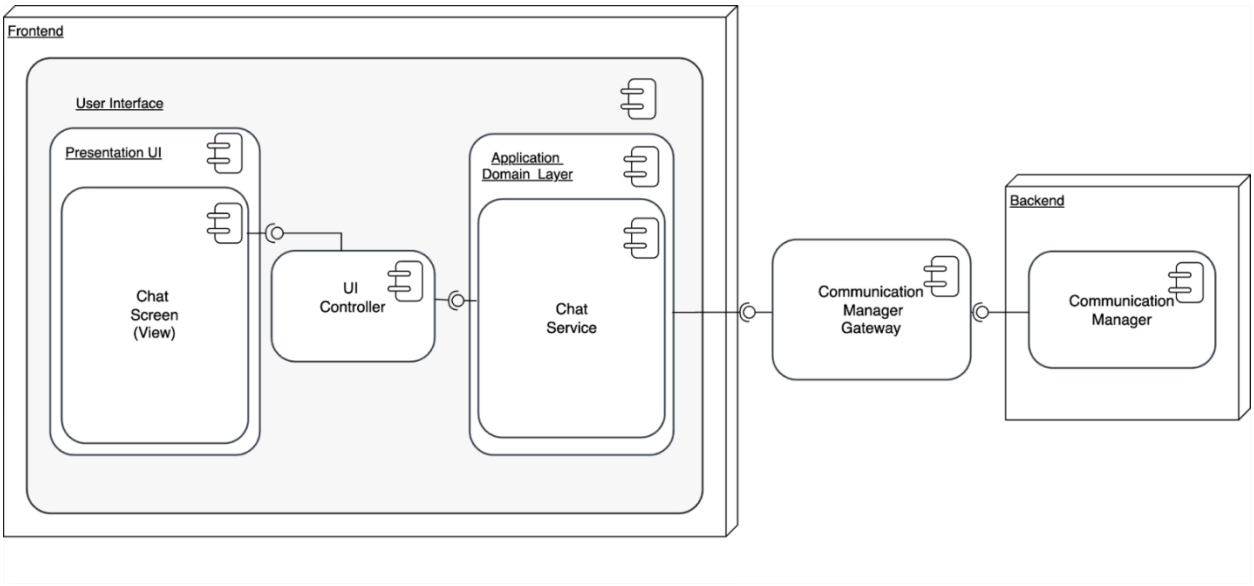
Viewpoint	Pseudocode

1.1.1.17.5.1 requestJSON

```
{
   "chatroomID": "int",
   "from": [
      "day": "int",
      "month": "int",
      "year": "int"
   ],
   "to": [
      "day": "int",
      "month" :"int",
      "year": "int"
   ]
}
```

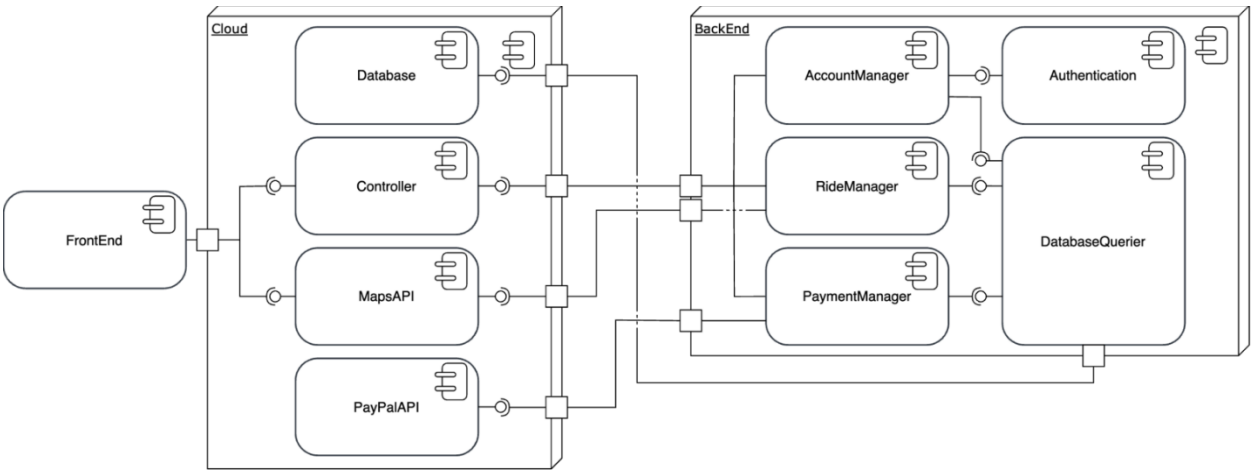
Name	1.1.1.17.5.1 requestJSON
Purpose	To demonstrate the structure of the request used to load messages.
Description	A JSON schema for a request to load messages within a certain time frame.
Requirements	26
Elements	**chatroomID**: The key, unique to the chatroom, used to identify the chatroom among other chatrooms.
	from: The date that represents the beginning of the timeframe range used to determine which messages need to be retrieved.
	to: The date that represents the end of the timeframe range.
	day: An integer representing the day part of the date.
	month: An integer representing the month part of the date.
	year: An integer representing the year part of the date.
Referenced By	1.1.1.17.5 loadMessages, 1.1.1.17 CMGateway
Viewpoint	JSON Schema

1.1.1.18 ChatUserInterface



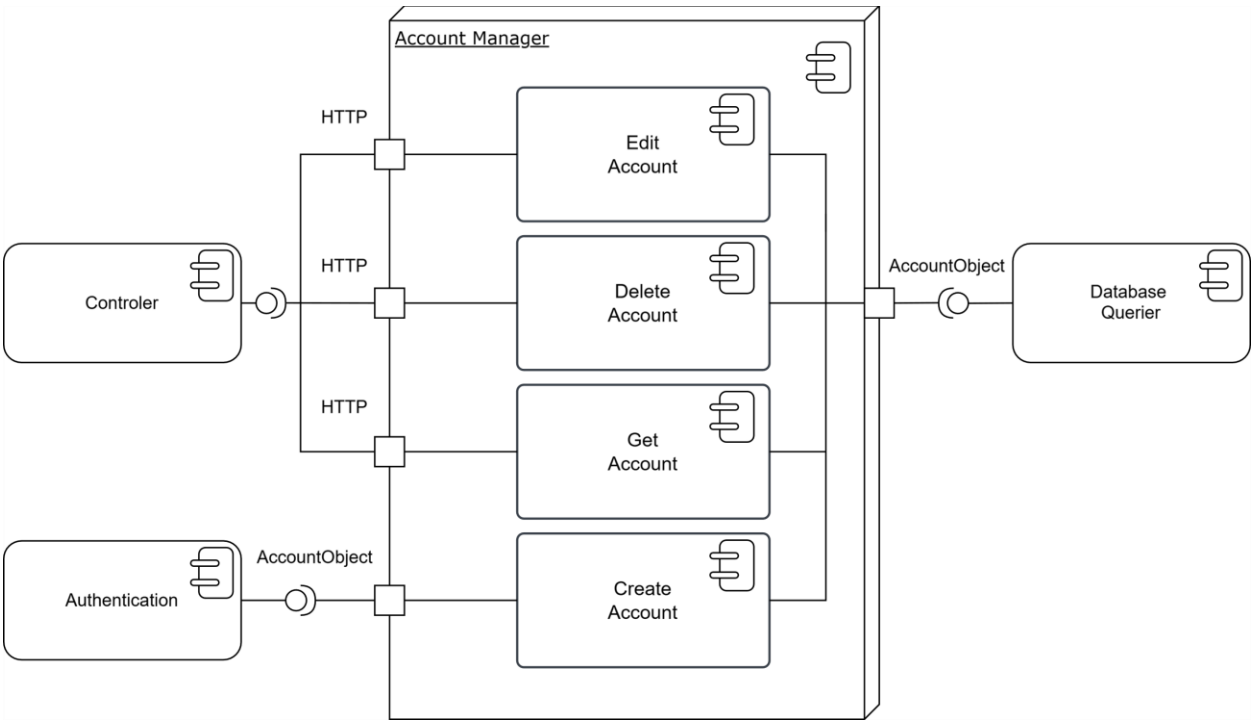
Name	1.1.1.18 ChatUserInterface
Purpose	The subsystems, Interaction Layer and View within UI communicate with the PaymentManager in the backend.
Description	Basic UI to Backend architecture where the two subsystems communicate to show the payment data of the trip to the user before and after the payment has been processed.
Requirements	19.0-20.0
Elements	UserInterface: 1.1.1
	Frontend: 1.1
	CommunicationManager: 1.2.6
	Application Domain Layer: 1.1.1.2
	ChatService: 1.1.1.16
	PresentationUI: 1.1.1.13
	ChatScreen View: 1.1.1.15
	UI Controller: 1.1.1.4
	Chat Gateway: 1.1.1.17
	Backend: 1.2
Referenced By	1 System
Viewpoint	Component Diagram

1.2 Backend



Name	1.2 BackEnd
Purpose	Component Diagram to illustrate the business logic of the app to encapsulate the functionality.
Description	Holds all the business logic and data maintenance functionality.
Requirements	1-27
Elements	**FrontEnd:** 1.1
	Cloud: 1.3
	Controller: 1.3.1
	Database: 1.3.2
	MapsAPI: 1.3.3
	PayPalAPI: 1.3.4
	AccountManager: 1.2.1
	Authentication: 1.2.2
	RideManager: 1.2.3
	PaymentManager: 1.2.4
	DatabaseQuerier: 1.2.5
Referenced By	1 System, 1.2.1.1.A EditAccount, 1.2.2.3.A ForgotPassword, 1.3 Cloud
Viewpoint	Component Diagram

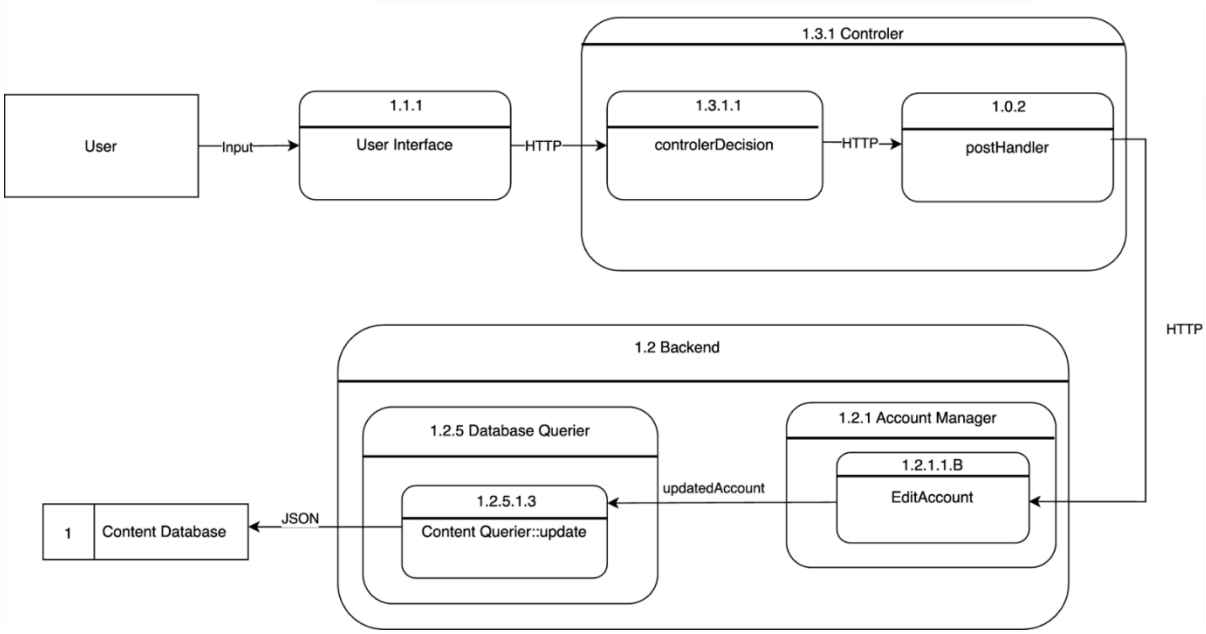
1.2.1 AccountManager



Name	1.2.1 AccountManager
Purpose	Describe the inner components of AccountManager and how they are connected
Description	Controls the actions that can be performed by an account.
Requirements	18
Elements	**HTTP:** Protocol that allows data transfer on the World Wide Web. Web standard since 1989.
	Get Profile: The process through which a user can get their account info.
	EditAccount: The process through which a user can edit their account.
	DeleteAccount: The process through which a user can delete their account.
	Create Account: The process through which a new account is created.
	DatabaseQuerier: 1.2.5
	Authentication: 1.2.2
	Controller: 1.3.1
Referenced By	1 System, 1.2.1.1.A EditAccount, 1.2 Backend, 1.2.2 Authentication, 1.2.5 DatabaseQuerier, 1.3.1.A Controller, 1.3.1.3 GETHandler, 1.3.1.4 MethodHandler, 1.2.1.5.B DeleteAccount Structure Chart, 1.2.1.3.A Display, 1.2.2.3.A ForgotPassword

Viewpoint	Component Diagram

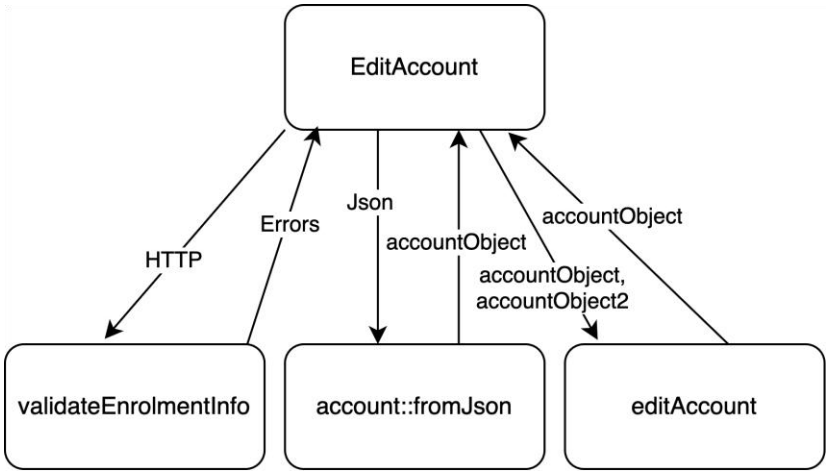
1.2.1.1.A EditAccount



Name	1.2.1.1.A EditAccount
Purpose	Shows the process of editing an account
Description	The flow of data for editing an account of a user
Requirements	18
Elements	**User**: A Person who is Registered in the System
	User Interface: 1.1.1 Validates form input, converts input into HTTP
	Controller: 1.3.1
	Controller Decision: Decides where in the backend the HTTP needs to go.
	Post Handler: Determines that HTTP is a post request
	HTTP: Protocol that allows data transfer on the World Wide Web
	Backend: 1.2
	Account Manager: 1.2.1
	Edit Account: 1.2.1.1.B
	JSON: The modified data in JSON format, being updated in the database.
	UpdatedAccount: Edited account Information
	Content Querier::update: 1.2.5.2.3 Method for updating JSON within the content database.
	Database Querier: 1.2.5
	Content Database: 1.3.2.1 Database that contains account information
Referenced By	1.2.1

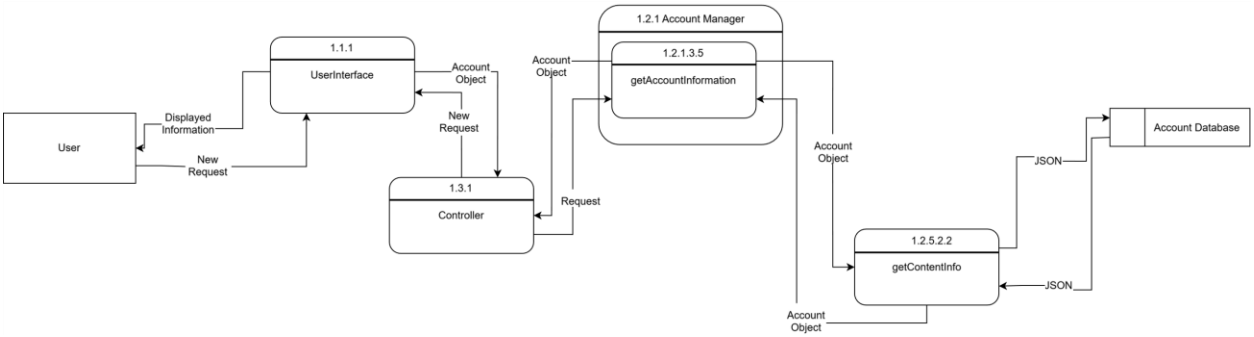
Viewpoint	Data Flow Diagram

1.2.1.1.B EditAccount



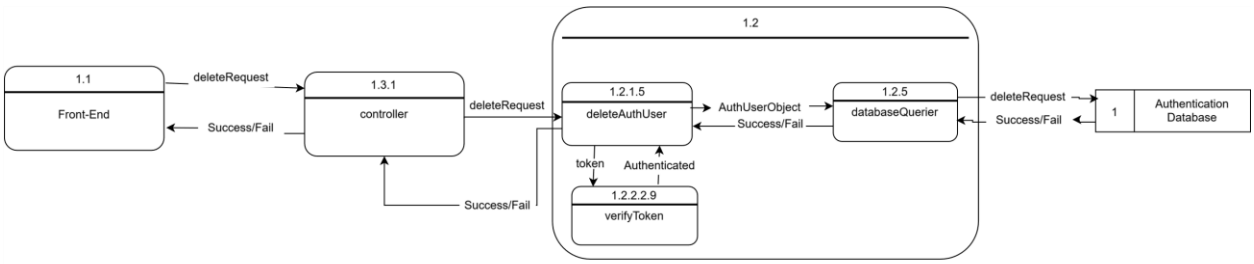
Name	1.2.1.1.B EditAccount
Purpose	Shows the functioned called to edit an account
Description	EditAccount receives HTTP and then compares it to the account object. The merged data is sent back to the database
Requirements	18
Elements	**HTTP**: Protocol that allows data transfer on the World Wide Web
	Account Object: The class on an account
	Account Object 2: Account object created from the HTTP
	ValidateEnrollmentInfo: 1.2.2.1.1
	Errors: Any errors from the validation
	Account::FromJson: 1.2.1.7.5
	JSON: account data in JSON format
	EditAccount: 1.2.5.1.3
Referenced By	1.2.1, 1.2.1.1.A EditAccount
Viewpoint	Structure Chart

1.2.1.3 DisplayAccount



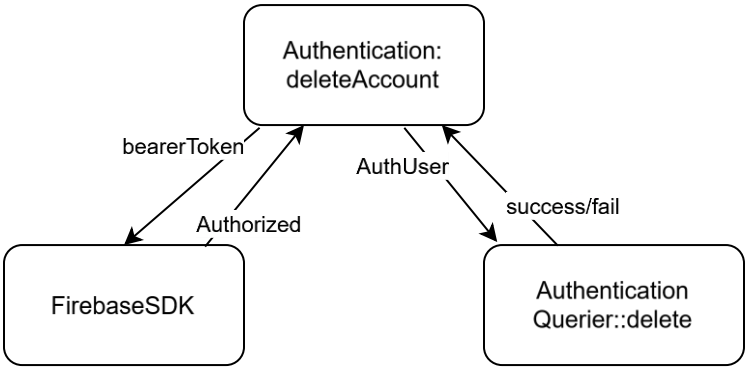
Name	1.2.1.3 DisplayAccount
Purpose	Shows the flow of data to display user information.
Description	This view shows the flow of data to display account information from the database that is not already cached in the front-end.
Requirements	18
	getAccountInformation: Process where a user request turns into a query object and an Account Object is returned to the User Interface.
	getContentInfo: 1.2.5.2.2
	Account Database: The database that stores account information.
	User: Representation of the user.
	User Interface: 1.1.1
	Account Manager: 1.2.1
	Controller: 1.3.1
	Account Object: 1.2.1.7
	Displayed Information: Displayed account information of requested account.
	JSON: 1.3.2.2.7
	Request: Request from user to get account information
Referenced By	1.1.1 User Interface
Viewpoint	Data Flow Diagram

1.2.1.5.A DeleteAccount



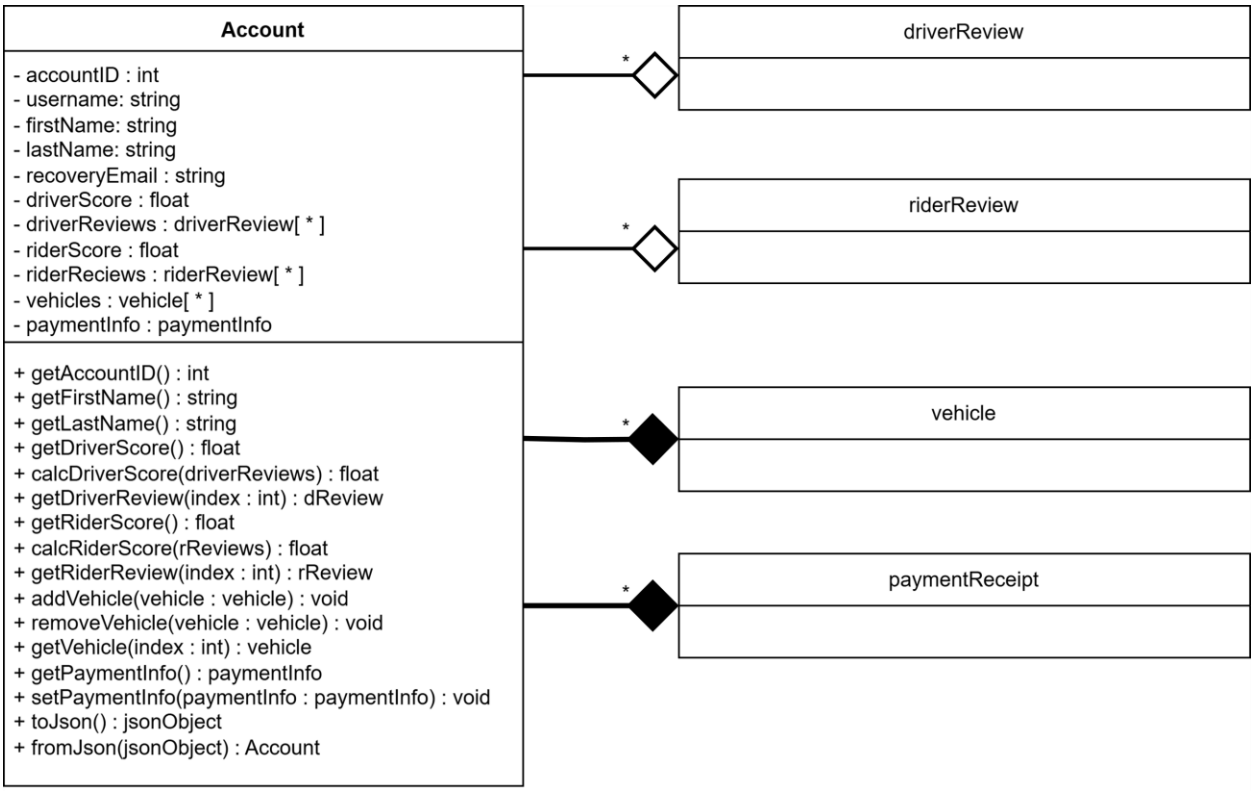
Name	1.2.1.5.A DeleteAccount
Purpose	Shows the data flow of deleting an account.
Description	The flow of data for editing a user account.
Requirements	18
Elements	**User**: A Person who is Registered in the System.
	controller: 1.3.1
	Backend: 1.2
	AuthUserObject: 1.2.2.4
	deleteAuthUser: Removes the login data of an account from the system.
	deleteRequest: The request to delete a specific user's account.
	databaseQuerier: 1.2.5
	authorizeRequest: The process that ensures the user has the authority to make the request.
	Authenticated: Whether or not the user is authorized to make the request.
	token: The token used to verify the user's session.
	verifyToken: The process to verify the user's session.
	Success/Fail: Whether or not the request succeeded.
	contentDatabase: 1.3.2.1
Referenced By	1.2.1
Viewpoint	Data Flow Diagram

1.2.1.5.B DeleteAccount



Name	1.2.1.5.B DeleteAccount
Purpose	Shows the methods that need to be called to delete a user account
Description	The methods used to delete a user's account.
Requirements	18
Elements	**Authentication:** 1.2.2
	AuthUser: a freshly created AuthUser object
	Authentication Querier::delete: Uses the username to delete an AuthUser from the authentication database
	FirebaseSDK: code that will handle verifying bearer tokens
	bearerToken: JWT Token the user can use to authenticate.
Referenced By	1.2.1 Account Manager
Viewpoint	Structure Chart

1.2.1.7 Account



Name	1.2.1.7 Account
Purpose	Shows everything related to an account.
Description	The variables, methods, and relations of account class.
Requirements	12, 16-18
Elements	**accountID:** Unique number associated with an account.
	firstName: First Nameof the user.
	lastName: Last Nameof the user.
	username: userNameof the user.
	email: Email used to send notifications to the user.
	driverScore: Number based average from driverReview to help others make choices.
	driverReview: List of driver review ID of a driver from the riders of their trips.
	riderScore: Number based on the average from riderReview to help others make choices.
	riderReview: List of rider review IDs of a rider from everyone on the trip.
	vehicles: List of vehicle IDs about the vehicles of the user.
	paymentReceipt: List of payment receipts IDs from PayPal.

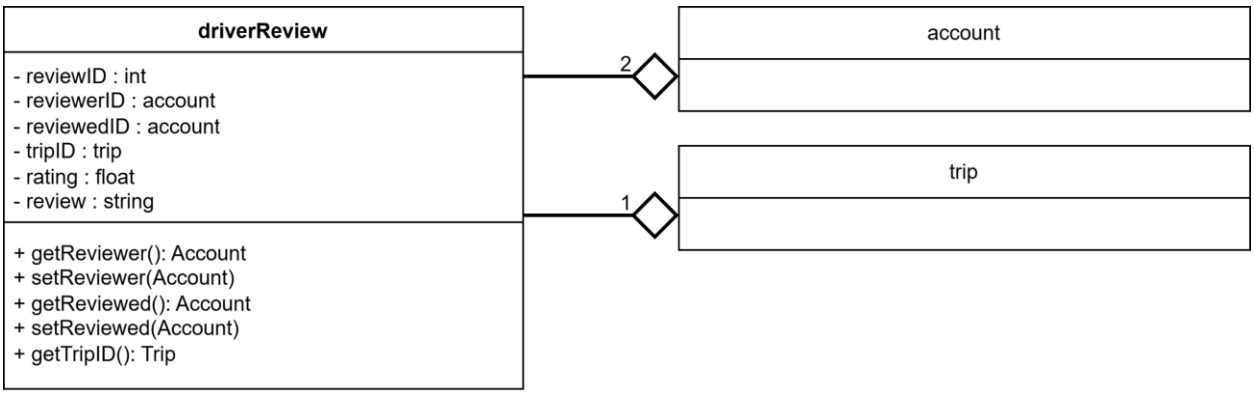
	setters: Sets individual attributes of the class.
	getters: Returns individual attributes of the class.
	calcScore: Calculates the average score of the reviews of an account.
	addReview: Adds a single review to a list of reviews.
	removeReview: Removes a single review from a list of reviews.
	fromJson: Turns the given JSON object into a new Account Object
	toJson: Converts the class to a JSON object.
Referenced By	1.2.1 AccountManager, 1.2.1.1.A EditAccount, 1.2.2.1.2 Create User, 1.2.1.3.A Display, 1.2.2 Authentication
Viewpoint	Class Diagram

1.2.1.7.1 AccountToJSON

```
toJSON()
   jsonObject = {}
   jsonObject["accountId"] <- this.accountId
   jsonObject["name"] <-
      "firstName": this.firstName
      "lastName": this.lastName
   jsonObject["username"] <- this.userName
   jsonObject["email"] <- this.email
   jsonObject["driverScore"] <- this.driverScore
   jsonObject["driverReviews"] <- this.driverReviews
   jsonObject["riderScore"] <- this.riderScore
   jsonObject["riderReviews"] <- this.riderReviews
   jsonObject["vehicles"] <- this.vehicles
   jsonObject["address"] <- this.address.addressToJson()
   jsonObject["paymentReceipt"] <- this.paymentReceipt

   RETURN jsonObject
```

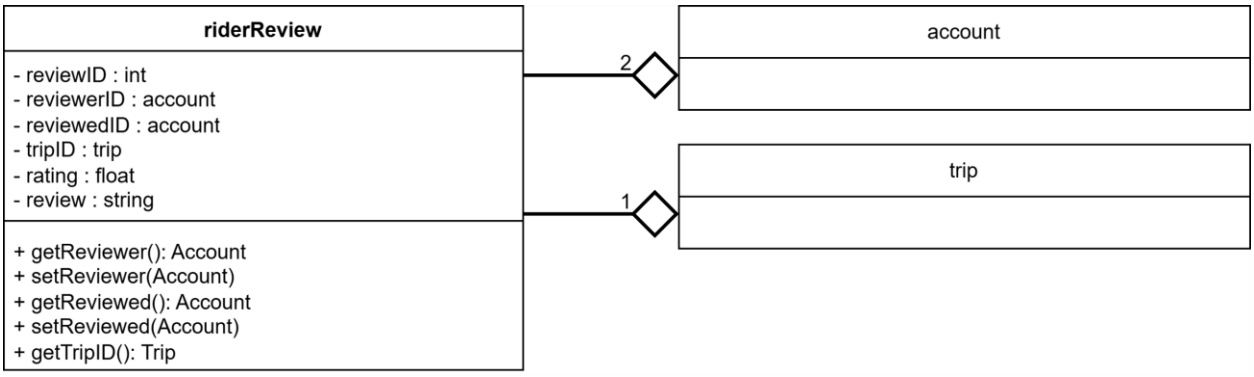
Name	**1.2.1.7.1 AccountToJSON**
Purpose	Shows a more focused look at the logic of the toJson function of account class.
Description	Takes all the variables from account class and makes them into key value pairs for the json object.
Requirements	12, 16-18
Elements	**accountId:** Unique number associated with an account.
	firstName: First Name of the user.
	LastName: Last Name of the user.
	username: username of the user.
	email: Email used to send notifications to the user.
	driverScore: Number based average from driverReview to help others make choices.
	driverReview: List of driver review ID of a driver from the riders of their trips.
	RiderScore: Number based on the average from riderReview to help others make choices.
	riderReview: List of rider review IDs of a rider from everyone on the trip.
	vehicles: List of vehicle IDs about the vehicles of the user.
	address: the address jsonObject
	PaymentReceipt: List of payment receipts IDs from PayPal.
	jsonObject: Return of pairs to create a json object.
Referenced By	1.2.1.7
Viewpoint	Pseudocode

1.2.1.7.3 driverReview



Name	1.2.1.7.3 driverReview
Purpose	Shows everything related to a review of a driver
Description	The variables, methods, and relations of driverReview class.
Requirements	18, 21, 22
Elements	**reviewID:** Unique number associated with a driver review.
	reviewerID: ID of the account making the review.
	reviewedID: ID of the account being reviewed.
	tripID: ID of the trip that the reviewer and reviewed shared.
	rating: Score out of five.
	review: Description or reason for the rating.
	account: 1.2.1.7
	trip: 1.2.3.7
Referenced By	1.2.1.7 Account, 1.2.3.7 Trip
Viewpoint	Class Diagram

1.2.1.7.4 riderReview



Name	1.2.1.7.4 riderReview
Purpose	Shows everything related to a review of a rider
Description	The variables, methods, and relations of riderReview class.
Requirements	18, 21, 22
Elements	**reviewID:** Unique number associated with a rider review.
	reviewerID: ID of the account making the review.
	reviewedID: ID of the account being reviewed.
	tripID: ID of the trip that the reviewer and reviewed shared.
	rating: Score out of five.
	review: Description or reason for the rating.
	account: 1.2.1.7
	trip: 1.2.3.7
Referenced By	1.2.1.7 Account, 1.2.3.7 Trip
Viewpoint	Class Diagram

1.2.1.7.6 CalculateDriverScore

```
calcDriverScore()
   totalScore <- 0
   FOR review IN driverReviews
      totalScore += review.getRating
   driverScore <- totalScore / driverReviews.count
```

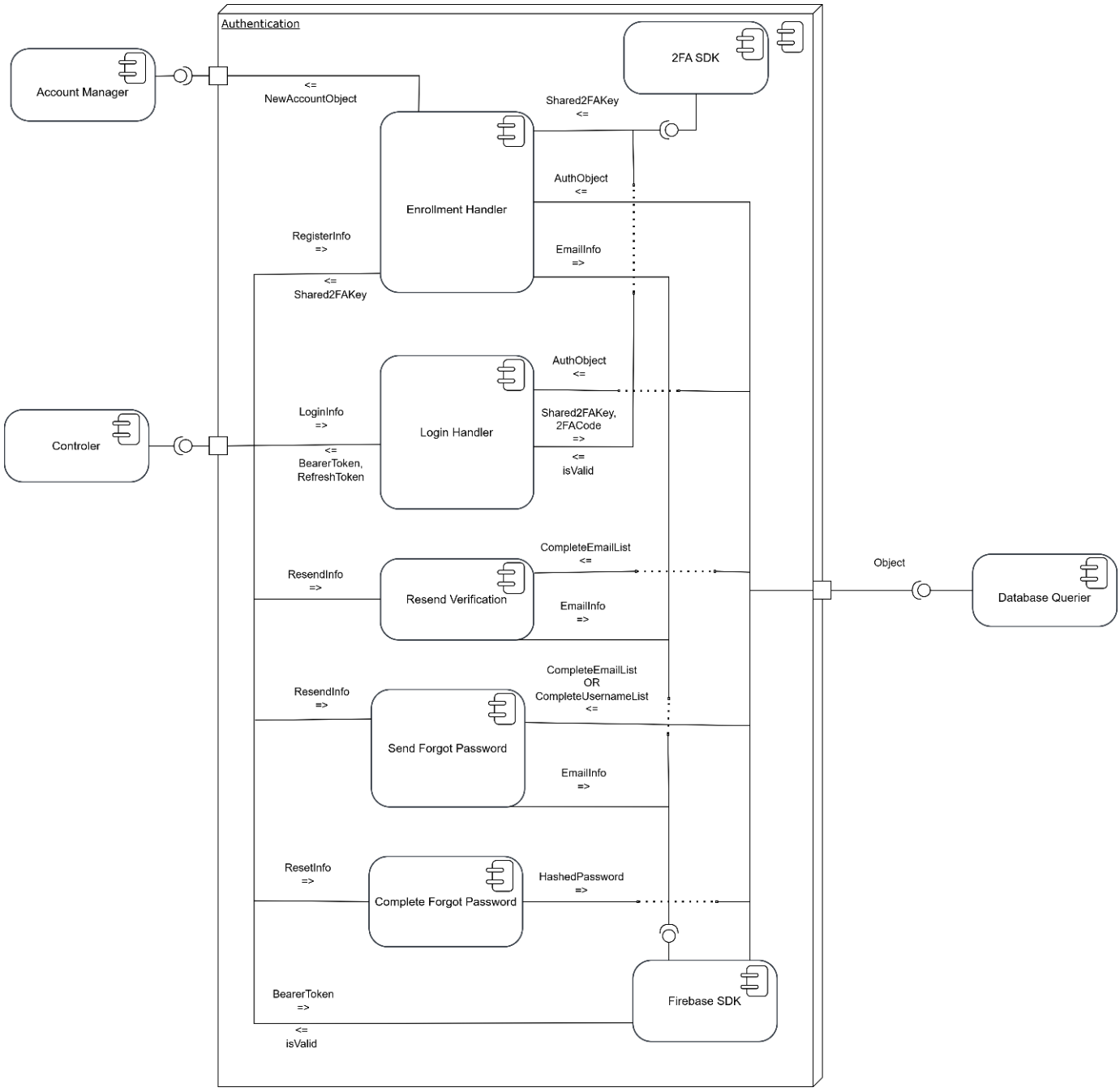
Name	1.2.1.7.6 CalculateDriverScore
Purpose	Shows how to calculate the full driver score
Description	The pseudocode for calculating the full driver score.
Requirements	18, 21, 22
Elements	**totalScore:** the total score of all the ratings given to the driver.
	Review: A single review that exists within the list.
	driverReviews: A member variable of the Account class containing list of all reviews associated with the account as a driver.
Referenced By	1.2.1.7 Account
Viewpoint	Pseudocode

1.2.1.7.7 CalculateRiderScore

```
calcRiderScore()
   totalScore <- 0
   FOR review IN riderReviews
      totalScore += review.getRating
   riderScore <- totalScore / riderReviews.count
```

Name	1.2.1.7.7 CalculateRiderScore
Purpose	Shows how to calculate the full rider score
Description	The pseudocode for calculating the full rider score.
Requirements	18, 21, 22
Elements	**totalScore:** the total score of all the ratings given to the rider.
	Review: A single review in the list of reviews.
	riderReviews: a member variable of the Account class that contains a list of all reviews associated with the account as a rider.
	riderScore: The average score for the Rider.
Referenced By	1.2.1.7 Account, 1.2.3.7 Trip
Viewpoint	Pseudocode

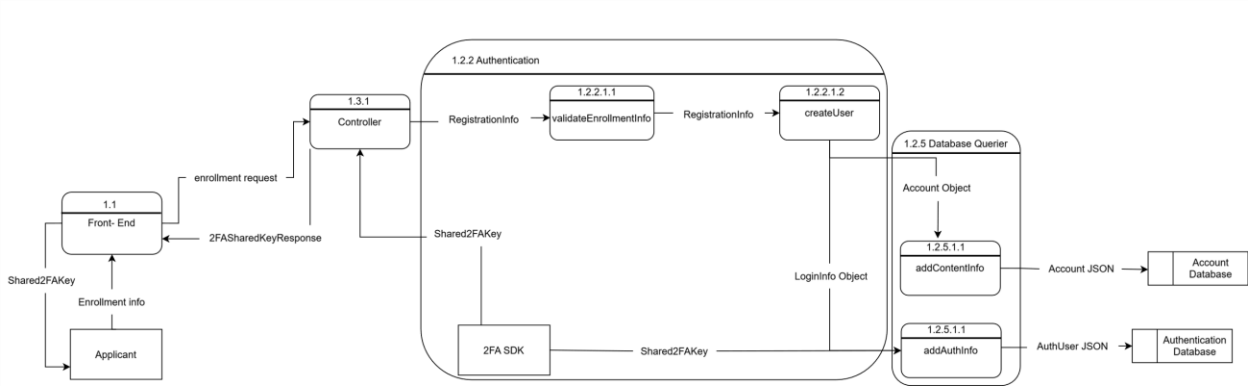
1.2.2 Authentication



Name	1.2.2 Authentication
Purpose	Describe the inner components of authentication.
Description	Authentication which can enroll/login users utilizing password and two-factor authentication, as well as verify tokens in requests for the controller.
Requirements	11-17
Elements	**Controller:** 1.3.1 Controller
	Enrollment Handler: Handles the enrollment new accounts with authentication and account
	Login Handler: Handles a request for login and sends bearer and refresh tokens on a successful login.
	2FA SDK: an external library that will generate shared keys and verify codes.
	Firebase SDK: an external library that will handle sending emails, issuing and verifying tokens.
	DatabaseQuerier: 1.2.5 DatabaseQuerier
	AccountManager: 1.2.1
	Resend Verification: Resends the verification email.
	Send Forgot Password: Sends a reset code via email to allow for password reset
	Complete Forgot Password: given a reset code resets, resets a user's password.
	NewAccountObject: a new account object created from the register request info. <accountObjectViewNumber>
	RegisterInfo: 1.2.1.6.
	LoginInfo: 1.2.1.3.A Display
	ResendInfo: 1.2.1.5
	ResetInfo: a 1.2.1.4
	BearerToken: a JWT bearer token used to authenticate users
	IsValid: Whether or not the token or object is valid
	AuthUser: Username, HashedPassword, Email, and Shared2FAKey. Everything is needed to authenticate a user. <AuthUserViewNumber>
	EmailInfo: Address, Subject, Body. The info required for FirebaseSDK to send an email.
	Shared2FAKey: A cryptographic key shared between the back-end server and the user for generating time-based one-time codes, 2FACodes
	2FACode: a 6 digit code generated using the Shared2FAKey and the current time.
	CompleteEmailList: a complete list of all emails in the system

	CompleteUsernameList: a complete list of all usernames in the system
	HashedPassword: a password that has been hashed and salted.
Referenced By	1 System, 1.2.1 AccountManager, 1.2.2.3.A ForgotPassword, 1.2.5 DatabaseQuerier, 1.3.1 Controller, 1.3.1.2 POSTHandler, 1.3.1.4 MethodHandler
Viewpoint	Component Diagram

1.2.2.1 NewUser



Name	1.2.2.1 NewUser
Purpose	Describe the flow of authentication
Description	The flow of data for authenticating and creating a new user.
Requirements	11-17
Elements	**Applicant**: A user wishing to create an account in the system.
	Enrollment Info: The information from the user needed to create an account.
	Controller: 1.3.1
	2FA SDK: Generates 2FA Shared Key on request.
	EnrollmentRequest: An HTTP Request containing RegisterInfo.
	RegisterInfo: 1.2.2.8
	validateEnrollmentInfo: Ensures the account information entered by the user is valid. (i.e. username is not an email, password conforms with requirements, phone number is formatted correctly, etc.)
	createUser: Creates an account in the system.
	LoginInfo Object: 1.2.2.5
	Account Object: 1.2.1.7
	Account Database: 1.3.2.1.7
	Authentication Database: 1.3.2.2
	Account JSON: 1.3.2.1.7
	AuthUser JSON: JSON according to the authentication database schema.
	addContentInfo: 1.2.5.2.1
	addAuthInfo: 1.2.5.1.1
	Shared2FAKey: A cryptographic key shared between the back-end server and the user for generating time-based one-time codes.

	2FASharedKeyResponse: HTTP response containing the new 2FASharedKey.
	Front-End: 1.1
	Authentication: 1.2.2
	Database Querier: 1.2.5
Referenced By	1.2.2
Viewpoint	**Data Flow Diagram**

1.2.2.1.1 Validate Enrollment Info

```
ValidateEnrollementInfo(RegisterInfo)

   passwordRegex =
      "^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[\W_]).{8,}$";

   usernameRegex = "^[a-zA-Z0-9]{5,}$";

   emailRegex =
      "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$";

   nameRegex = "^[a-zA-Z]+$";

   if(!MatchesRegex(PASSWORD_REGEX,RegisterInfo.getPassword()))

      RETURN false;

   if(!MatchesRegex(USERNAME_REGEX,RegisterInfo.getUsername()))

      RETURN false;

   if(!MatchesRegex(EMAIL_REGEX, RegisterInfo.getEmail())

      RETURN false

   if(!MatchesRegex(NAME_REGEX, RegisterInfo.getFirstName())

      RETURN false;

   if(!MatchesRegex(NAME_REGEX, RegisterInfo.getLastName())

      RETURN false;

   Usernames contentQuerier.getInfo(new Account)

   if(usernames.contains(RegisterInfo.getUsername())

      RETURN false;

   RETURN true;
```

Name	1.2.2.1.1 Validate Enrollment Info
Purpose	Validate user information for enrollment
Description	The process of validating enrollment information for creating a user

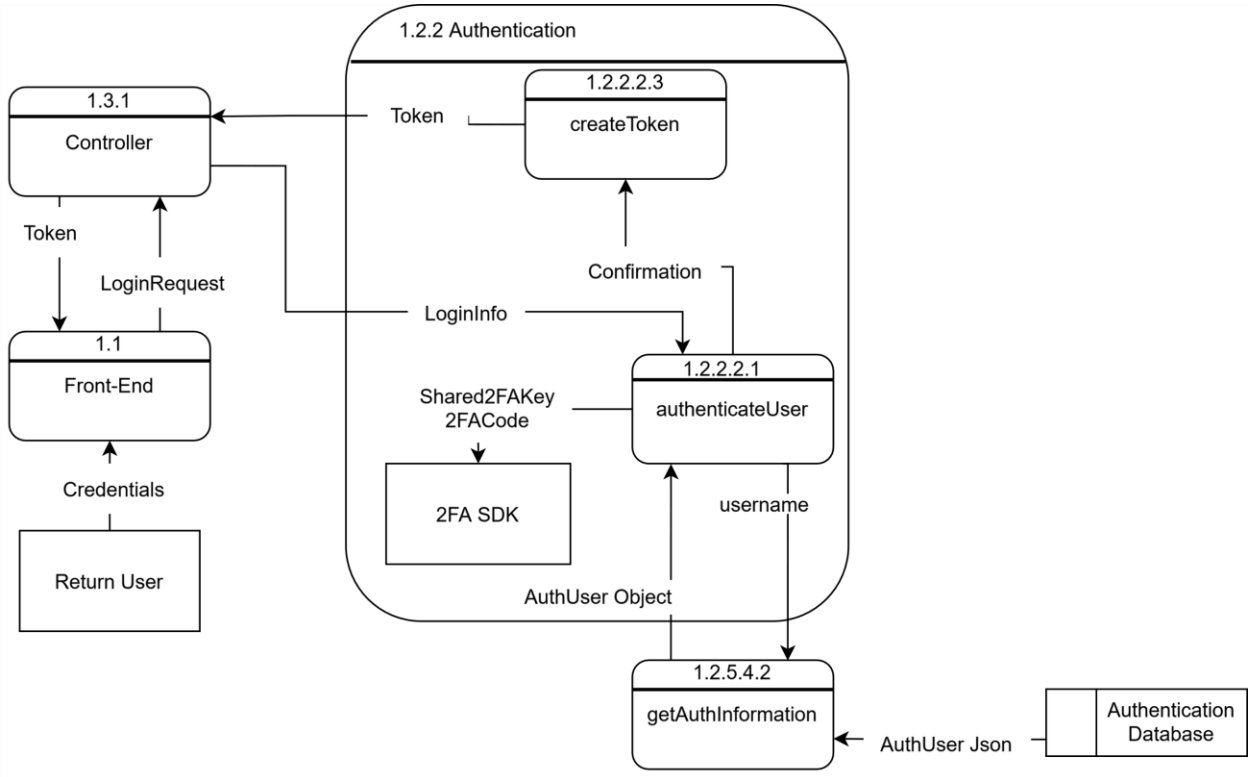
Requirements	13 - 16
Elements	**registerInfo:** 1.2.2.8
	matchesRegex: a method that checks if a string satisfies a regex
	passwordRegex: Require at least 8 chars, one lowercase, one upper case, one digit, and one special char
	usernameRegex: Require at least 5 chars of only upper lower or digits.
	nameRegex: at least one char, only one letter.
	emailRegex: require a valid email
	usernames: list of existing usernames from the account database
	ContentQuerier: 1.2.5.2
Referenced By	1.2.2.1.2 Create User
Viewpoint	Pseudocode

1.2.2.1.2 Create User

```
CreateUser(RegisterInfo){

   IF(!ValidateEnrolementInfo(RegisterInfo)){
      password = Hash(info.getPassword);

      twoFAKey = 2FASDK.Generate2FAKey();

      AuthenicationQuerier.addAuthInfo(new
AuthUser(RegisterInfo.getUsername(), password,
RegisterInfo.getEmail(), twoFAKey));

      ContentQuerier.addContentInfo(new Account
(RegisterInfo.getUsername, RegisterInfo.getEmail(),
RegisterInfo.getFirstName(), RegisterInfo.getLastName());

      return ("Shared2FAKey", twoFAKey);
   }

   else{
      return("error", "username already exists");
   }
}
```

Name	1.2.2.1.2 CreateUser
Purpose	Describes how a new user is created
Description	Process of creating a new user
Requirements	11, 12, 17
Elements	**RegisterInfo:** 1.2.2.8
	ValidateEnrolementInfo: 1.2.2.1.1
	ContentQuerier: 1.2.5.2
	Hash: Securely hashes the password
	2FASDK: third party SDK that will handle generating 2FA keys among other things.
	AuthenticationQuerier: 1.2.5.1
	AuthUser: 1.2.2.4
	Account: 1.2.1.7
Referenced By	1.2.2.1 New User, 1.2.2.1.1 Validate Enrollment Info
Viewpoint	Pseudocode

1.2.2.2 ReturnUser



Name	1.2.2.2 ReturnUser
Purpose	Describe the flow of authentication .
Description	The flow of data for authenticating a ReturnUser.
Requirements	13-15
Elements	**ReturnUser:** A user that already exists within the system.
	username: The username the user input.
	2FA SDK: Confirms 2FA Shared Key on request.
	getAuthInfo: 1.2.5.1.2
	authenticateUser: Authenticates the user through 2-factor authentication.
	Authentication Database: 1.3.2.1
	LoginInfo: 1.2.2.5
	AuthUser JSON: JSON according to the authentication database schema.
	Shared2FAKey: A cryptographic key shared between the back-end server and the user for generating time-based one-time codes.
	2FA code: A 6-digit code generated using the Shared2FAKey and the current time.
	Confirmation: The value that tells the system the user is authenticated.
	Token: Indicator of authentication from back-end.
	Controller: 1.3.1

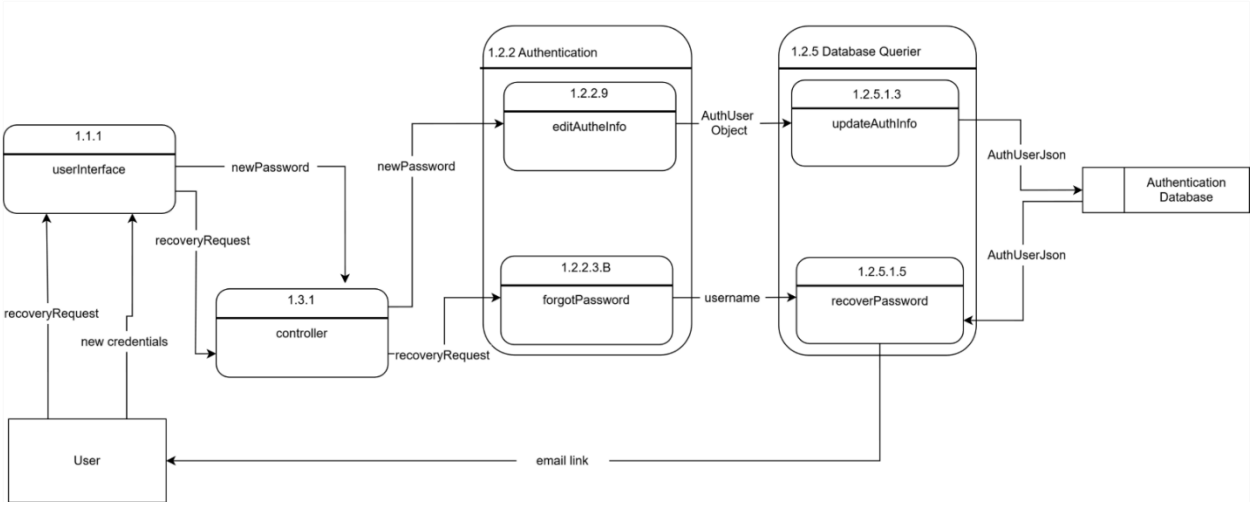
	createToken: Created token upon the confirmation of user credentials.
	Credentials: The users login information.
	Authentication: 1.2.2
	Request Object: A call to the database based on the unique account username.
	Front-End: 1.1
	Login Request: An HTTP request containing Login Info.
Referenced By	1.2.2 Authentication
Viewpoint	Data Flow Diagram

1.2.2.2.2 CreateToken

```
CreateToken()

   admin = new firebase admin
   uid = uuid()
   token = admin createCustomToken(uid)

   RETURN token
```

Name	1.2.2.2.2 CreateToken
Purpose	Showing the process of creating a token for an authenticated user.
Description	Pseudocode that demonstrates how the firebase SDK is used to create a session token.
Requirements	13
Elements	**admin**: instance of firebase admin SDK that will be used to create the token.
	uuid(): Language specific function to create uid.
	uid: Unique 16 digit number to create token.
	token: Unique token for the current authenticated session.
Referenced By	1.2.2.2 ReturnUser
Viewpoint	Pseudocode

1.2.2.3.A ForgotPassword



Name	1.2.2.3.A ForgotPassword
Purpose	Shows the process of recovering an account
Description	The flow of data for recovering an account of a user
Requirements	17
Elements	**User:** A Person who is Registered in the System
	Controller: 1.3.1
	userInterface: 1.1.1
	recoveryRequest: Request from user to recover account that contains their username.
	username: A unique identifier of a particular account.
	forgotPassword: 1.2.2.3.B The process of requesting account recovery from the querier.
	recoverPassword: 1.2.5.1.5 The process in handling account recovery.
	newPassword: New password for account retrieved from the user.
	Authentication: 1.2.2
	updateAuthInfo: 1.2.5.1.3
	editAuthInfo: 1.2.2.9
	EmailLink: Message sent to user to reset password.
	AuthUser Object: 1.2.2.4
	DatabaseQuerier: 1.2.5
	AuthUserJSON: 1.3.2.2
	Authentication Database: 1.3.2.1 Database that contains authentication information.
Referenced By	1.2.2 Authentication, 1.2.5.1.5 RecoverPassword, 1.2.2.3.B ForgotPassword

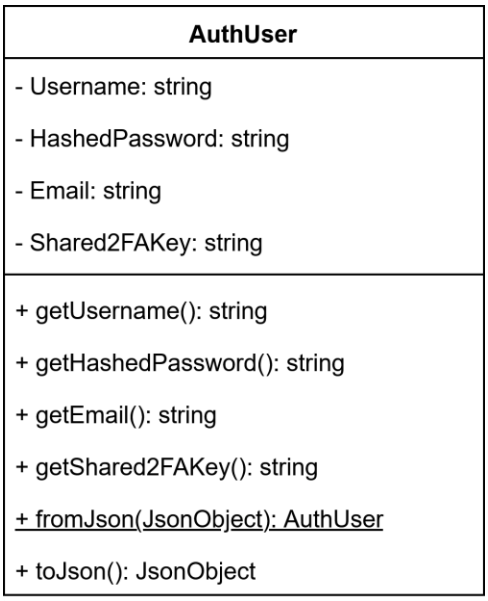
Viewpoint	Data Flow Diagram

1.2.2.3.B ForgotPassword

```
ForgotPassword(recoveryRequest)
   AuthentcationQuerying.RecoverPassword(rocoveryRequest
username)
```

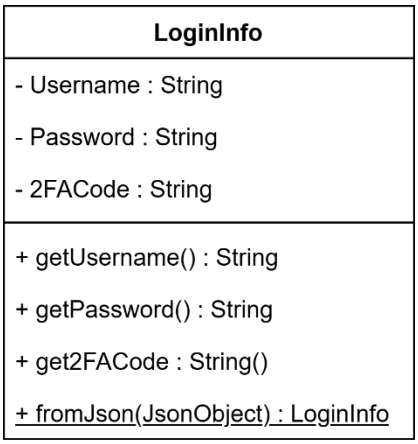
Name	1.2.2.3.B ForgotPassword
Purpose	Shows the process of recovering an account
Description	The flow of data for recovering an account of a user
Requirements	17
	username: A unique identifier of a particular account.
	RecoverPassword: 1.2.5.1.5 The process in handling account recovery.
	RecoveryRequest: Request for account recovery that includes the unique username for account to recover.
	AuthenticationQuerier: 1.2.5.1
Referenced By	1.2.2.3.A ForgotPassword
Viewpoint	Pseudocode

1.2.2.4 AuthUser



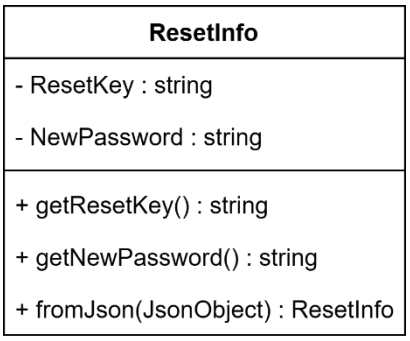
Name	1.2.2.4 AuthUser
Purpose	Describe the class used to store a user in the authentication system
Description	Authentication which can enroll/login users utilizing password and two-factor authentication, as well as verify tokens in requests for the controller.
Requirements	11-16
Elements	**Username:** a public unique identifier for the user
	HashedPassword: An encrypted string generated from the user's password
	Email: the email associated with the account.
	Shared2FAKey: A cryptographic key shared between the back-end server and the user for generating time-based one-time codes, 2FACodes
	getUsername: Returns the userNameof the user
	getHashedPassword: Returns the hashed password of the user
	getEmail: Returns the email of the user
	getShared2FAKey: A cryptographic key shared between the back-end server and the user for generating time-based one-time codes, 2FACodes.
	fromJson: a static method that returns a new AuthUser object generated based on json.
	toJson: converts the data to JSON data so it can be stored in the database.
Referenced By	1.2.2 Authentication, 1.2.2.1.2 Create User, 1.2.2.3.A ForgotPassword
Viewpoint	Class Diagram

1.2.2.5 LoginInfo



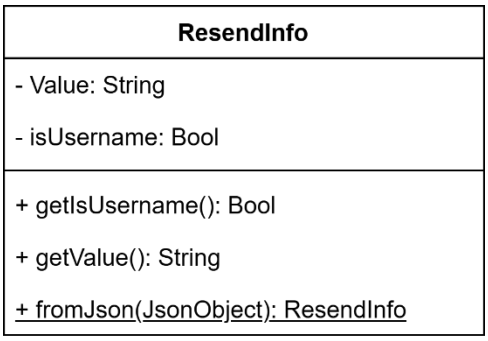
Name	1.2.2.5 LoginInfo
Purpose	Describe how the backend receives LoginInfo.
Description	A container for the information required to authenticate a user.
Requirements	13-15
Elements	**Username:** The unique identifier input by the user.
	Password: Unencrypted password input by a user.
	2FACode: A 6-digit code generated by the user using their copy of the Shared2FAKey.
	getUsername: Returns the UserNameproperty.
	getPassword: Returns the unencrypted Password property.
	Get2FACode: returns the 2FA code property.
	fromJson: Generates an instance of the class based on a JSON object.
Referenced By	1.2.2 Authentication, 1.2.2.1 Authentication: New User ,1.2.2.2 Authentication: Return User, 1.2.5 DatabaseQuerier
Viewpoint	Class Diagram

1.2.2.6 ResetInfo



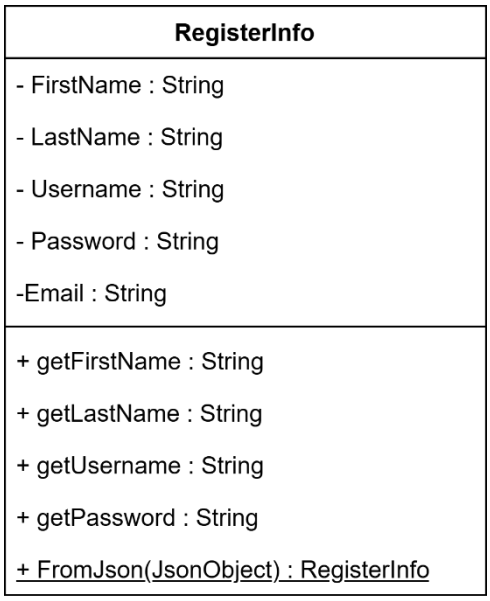
Name	1.2.2.6 ResetInfo
Purpose	Describe how the backend receives info to reset a password
Description	A container holding a reset key and a new password.
Requirements	13
Elements	**ResetKey:** An encrypted key that allows a user to reset their password.
	NewPassword: The new, unencrypted, password for the account.
	getResetKey: Returns the ResetKey property.
	getNewPassword: Returns the NewPassword property.
	fromJson: Generates an instance of the class based on a JSON object.
Referenced By	1.2.2 Authentication
Viewpoint	Class Diagram

1.2.2.7 ResendInfo



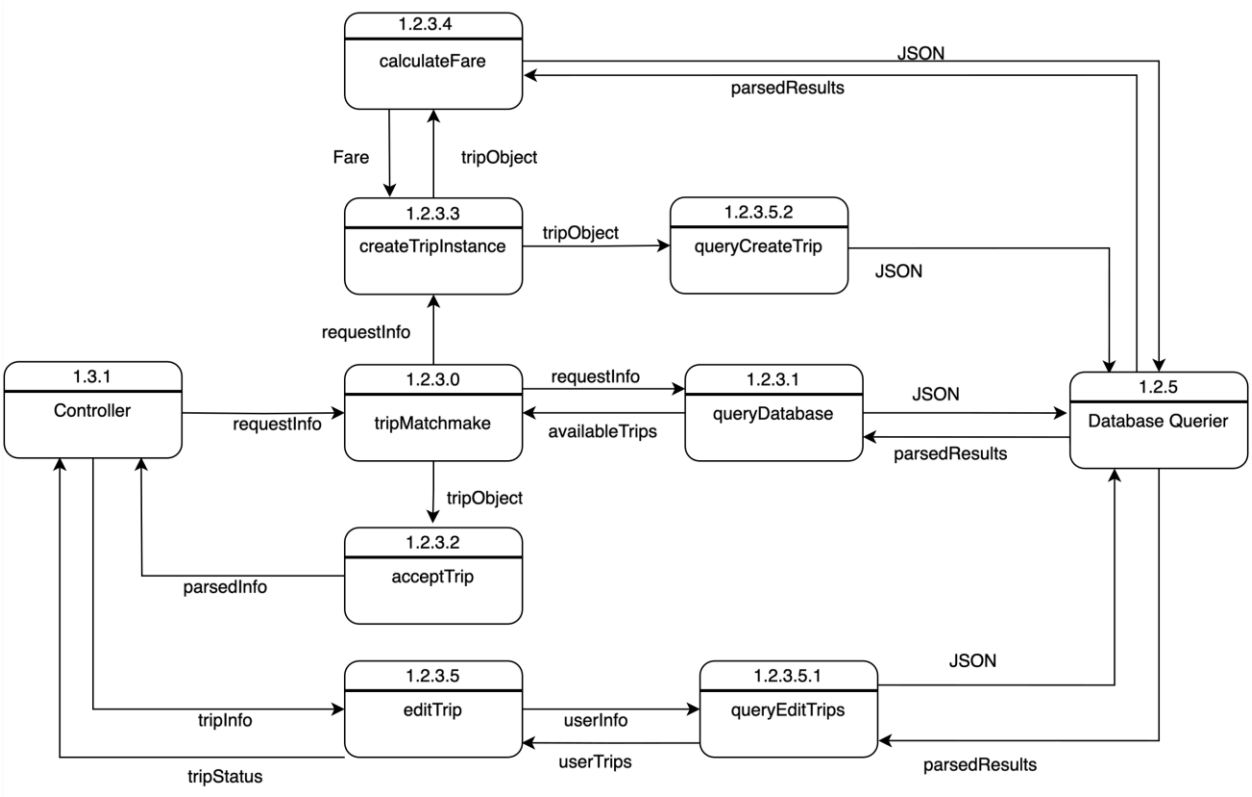
Name	1.2.2.7 ResendInfo
Purpose	Describe how the backend receives ResendInfo .
Description	A container holding either an email or a userName.
Requirements	13
Elements	**value:** Either an email or a username.
	isUsername: Remembers if it is storing an email or a username.
	getValue: returns the value property.
	getIsUsername: Returns whether this instance is storing a Username.
	fromJson: Generates an instance of the class based on a JSON object.
Referenced By	1.2.2 Authentication
Viewpoint	Class Diagram

1.2.2.8 RegisterInfo



Name	1.2.2.8 RegisterInfo
Purpose	Describe how the backend receives registration info.
Description	A container for the information required to register a user.
Requirements	11-17
Elements	**FirstName:** The first Nameof the user.
	LastName: The last Nameof the user .
	Username: A unique identifier for the account.
	Password: A Secret key used to log into the account. Should not be saved in raw from in any kind of long term way.
	Email: the email associated with the account.
	getFirstName: Returns the first name.
	getLastName: Returns the last name.
	getUsername: Returns the username.
	getPassword: Returns the password.
	fromJson: generates an instance of the class based on a JSON object.
Referenced By	1.2.2 Authentication, 1.2.2.1.1 Validate Enrollment Info, 1.2.2.1.2 Create User
Viewpoint	Class Diagram

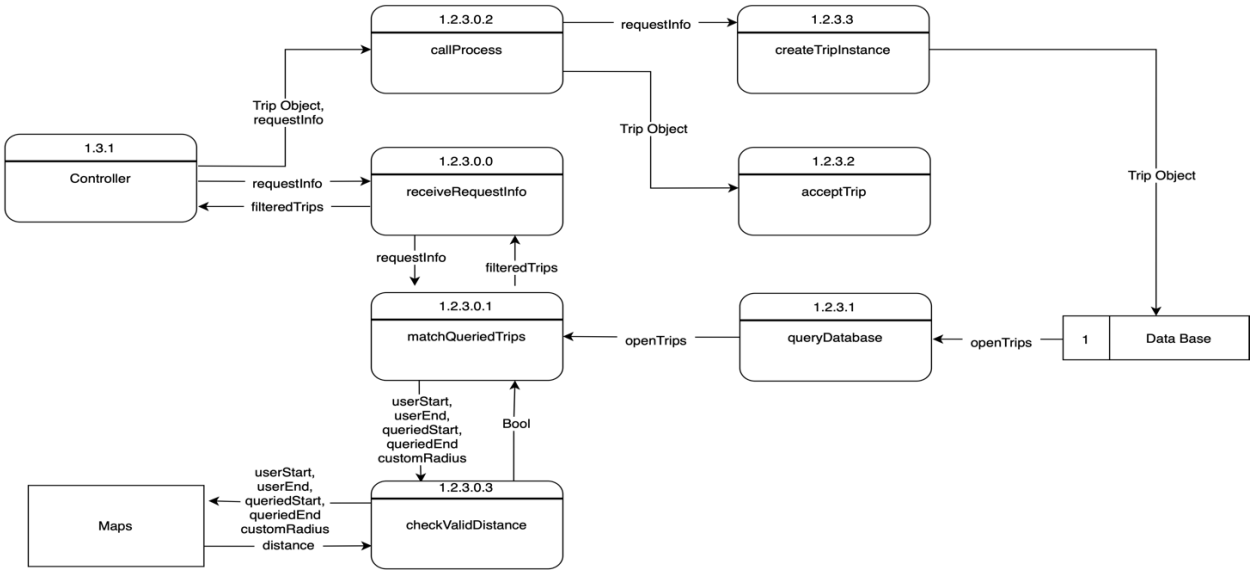
1.2.3 RideManager



Name	1.2.3 RideManager
Purpose	Manage the creation, state and information on all rides.
Description	How the user creates, accepts and changes the status of a ride.
Requirements	1-7
Elements	**Controller:** 1.3.1
	tripMatchmake: 1.2.3.0
	requestInfo: The user inputs information about the trip they want to list or enroll.
	availableTrips: Returns trips found from the database.
	DatabaseQuerier: 1.2.5
	JSON: Organization of data in a key-value pair.
	tripsDatabase:
	createTripInstance: 1.2.3.3
	tripInfo: information related to a trip.
	calculateFare: calculates the of a trip. 1.2.3.4
	tripDistance: it's the distance of a trip; it helps calculate the trip cost(fare).
	fare: price being charged for the trip.
	acceptTrip: user accepts the trip based on matchmaking. 1.2.3.2
	tripObject: 1.2.3.1

	queryEditTrips: queries Current Trips taken from the Trips Databases. 1.2.3.6
	userTrips: information about User Trips.
	userInfo: user information.
	editTrip: allows the user to change the user trips status, cancel, finish, etc. 1.2.3.5
	tripStatus: status of a trip.
	parsedResults: returns queried results from a database.
	Parsed Info: returns curated information requested by the user.
	tripDetails: details about trips being recorded to the database.
Referenced By	1 System, 1.1.1 UserInterface, 1.2 Backend, 1.2.5 DatabaseQuerier, 1.3.1.A Controller, 1.3.1.2 POSTHandler, 1.3.1.3 GETHandler, 1.3.1.4 MethodHandler
Viewpoint	Data Flow Diagram

1.2.3.0 TripMatchmake



Name	1.2.3.0 TripMatchmake
Purpose	Find and match existing trips to user request or create a new trip.
Description	Accepts user request information and then presents trips that match or are similar to the received specifications provided by the user. The user then chooses one of the trips to join or chooses to create a new trip.
Requirements	
Elements	**createTripInstance:**1.2.3.3_Create_Trip
	Controller: 1.3.1
	acceptTrip: See 1.2.3.2_Accept_Trip
	User: the source of all incoming data from the Frontend or the user.
	queryDatabase: See 1.2.5_Database_Querier
	Maps: See 1.3.3_Map_API
	callProcess: 1.2.3.0.2 callProcess
	receiveRequestInfo: 1.2.3.0.0 receiveRequestInfo
	matchQueriedTrips: 1.2.3.0.1 matchQueriedTrips
	checkValidDistance: 1.2.3.0.3 checkValidDistance
	requestInfo: a collection of variables representing account name, start location, end location, start date, end date, etc. See Trip Object.
	openTrips: a collection of all trips that are not locked or finalized.
	filteredTrips: a collection of trips that have been filtered through matchQueriedTrips.
	userStart: the start location contained in requestInfo.
	userEnd: the end location contained in requestInfo.
	queriedStart: the start location contained in an open trip.

	queriedEnd: the end location contained in an open trip.
	distance: a JSON file returned by Maps.
	TripObject: See 1.2.3.7_Trip_Object
	Data Base: the database in which all trips are stored.
Referenced by	1.2.3_Ride_Manager
Viewpoint	Data Flow Diagram

1.2.3.0.0 receiveRequestInfo

```
receiveRequestInfo(requestInfo)

   filteredTrips = matchQueriedTrips(requestInfo)

   RETURN filteredTrips
```

Name	1.2.3.0.0 receiveRequestInfo
Purpose	Gets the trip information from the user in order to matchmake and return relevant trips.
Description	This function receives the requested information from the user in the form of filtered trips.
Requirements	4, 7
Elements	**requestInfo:** gets the user input from the controller.
	filteredTrips: it accesses the matchQueriedTrips and inputs the information inide requestInfo.
	matchQueriedTrips: 1.2.3.0.1 matchQueriedTrips
Referenced By	1.2.3.0 tripMatchmake
Viewpoint	Pseudocode

1.2.3.0.1 Match Queried Trips

```
MatchQueriedTrips(requestInfo)

   openTrips <- queryDatabase()
   filteredTrips <- []

   userStart <- requestInfo.startLocation
   userEnd <- requestInfo.endLocation

   FOR trip IN openTrips
      queriedStart <- trip.startLocation
      queriedEnd <- trip.endLocation
      IF checkValidDistance(userStart, userEnd, queriedStart,
queriedEnd, requestInfo.customRadius)
         filteredTrips.add(trip)
```

Name	**1.2.3.0.1 MatchQueriedTrips**
Purpose	Collects a list of trips which are close enough to the user's desired start and end locations.
Description	Asks the querier for all the open trips and then adds each one which is within a valid distance of the user's desired start and end locations to an output list.
Requirements	
Elements	**openTrips:** a collection of all trips (1.2.3.7) that are not locked or finalized.
	userStart: The location the user wishes to start the trip from.
	userEnd: The location the user wishes to end the trip at.
	trip: 1.2.3.7
	queriedStart: The starting location of a specific trip from the database.
	queriedEnd: The ending location of a specific trip from the database.
	requestInfo: 1.1.1.6.5
	filteredTrips: a collection of trips (1.2.3.7) that are within a valid distance of the user's desired start and end locations.
	checkValidDistance: 1.2.3.0.3
Referenced By	1.2.3.0 Trip Matchmake
Viewpoint	Pseudocode

1.2.3.0.2 CallProcess

```
callProcess(tripObject, requestInfo)
   IF tripObject != NULL
      acceptTrip(tripObject)
   ELSE
      createTripInstance(requestInfo)
```

Name	1.2.3.0.2 callProcess
Purpose	The purpose of this pseudocode is to determine whether to accept an existing trip or create a new one based on user input. If a valid trip object is provided, the function proceeds with accepting that trip; otherwise, it creates a new trip instance using the details in the user's request.
Description	This pseudocode checks if an existing trip is provided; if so, it accepts that trip, otherwise it creates a new trip using the user's request information.
Requirements	3, 6, 7
Elements	**tripObject:** 1.2.3.7 Trip
	requestInfo: A parameter passed into callProcess that represents user-provided trip specifications (e.g., start/end locations, dates, etc.).
	acceptTrip: 1.2.3.2 acceptTrip
	createTripInstance: 1.2.3.3 createTripInstance
Referenced By	1.2.3.0 tripMatchMake
Viewpoint	Pseudocode

1.2.3.0.3 Check Valid Distance

```
checkValidDistance(userStart, userEnd, queriedStart, queriedEnd,
customRadius)


   IF userStart.coordinates == null
      userStart.coordinates <- GET from geocoding with
userStart.address
   IF userEnd.coordinates == null
      UserEnd.coordinates <- GET from geocoding with
userEnd.address
   IF queriedStart.coordinates == null
      queriedStart.coordinates <- GET from geocoding with
queriedEnd.address
   IF queriedEnd.coordinates == null
      queriedEnd.coordinates <- GET from geocoding with
queriedEnd.address



   GET distanceStart from Maps with userStart.coordinates,
queriedStart.coordinates


   GET distanceEnd from Maps with userEnd.coordinates,
queriedEnd.coordinates


   RETURN distanceStart < customRadius AND distanceEnd <
customRadius
```

Name	1.2.3.0.3 Check Valid Distance
Purpose	Checks if the user request for a trip is within their chosen radius.
Description	Utilizes Google Maps to calculate the distance between the start and end points provided by the user request and the start and end points of a given trip. If the distances are within the radius provided by the user then it returns true, otherwise false.
Requirements	1
Elements	**userStart:** user selected starting location, including coordinates and/or an address
	userEnd: user selected ending location, including coordinates and/or an address.
	queriedStart: trip starting location, including coordinates

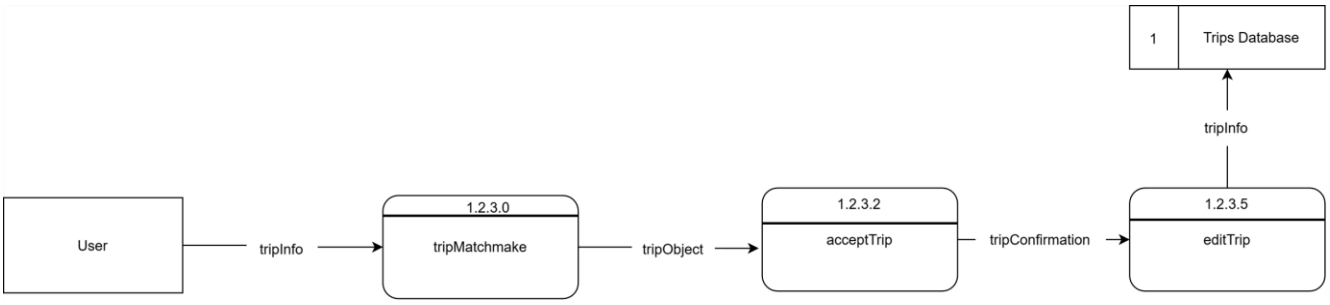
	queriedEnd: trip ending location, including coordinates
	customRadius: the distance trips should be within to be valid.
	geocoding: Google Maps Geocoding API
	Maps: Google Maps Distance API
Referenced By	1.2.3.0 Trip Matchmake, 1.2.3.0.1 MatchQueriedTrips
Viewpoint	Pseudocode

1.2.3.1 QueryDatabase

```
queryDatabase(tripObject)
 trip = {
 "owner": tripObject.owner,
 "driver": tripObject.driver,
 "startLocation": tripObject.StartLocation
 "endLocation": tripObject.EndLocation
 "tripTime": tripObject.tripTime
 }
 databaseQuerier(trip)
```

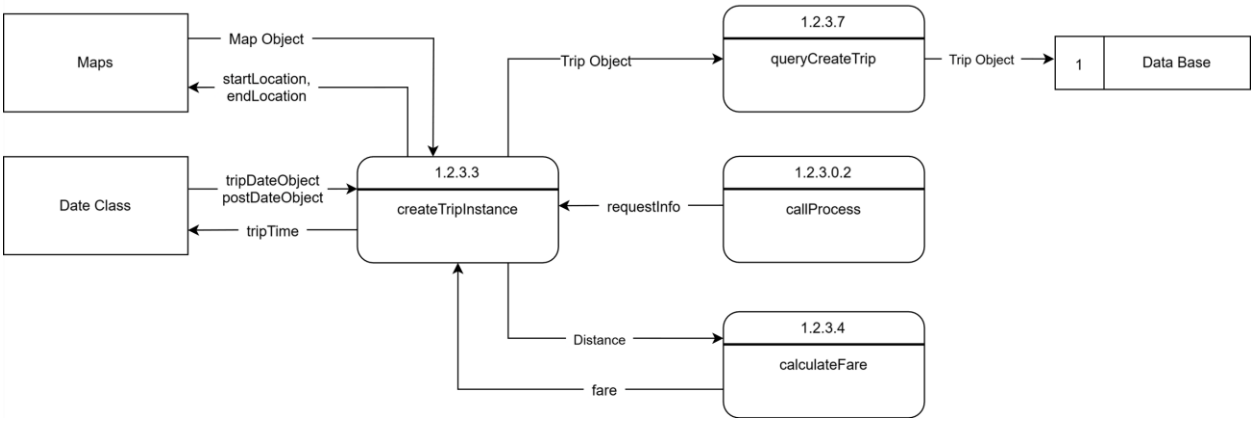
Name	1.2.3.1 QueryDatabase
Purpose	Retrieve multiple trips from the trip database
Description	Queries a trip from the trip database through its start and end point
Requirements	4, 5
Elements	**owner:** The requestor of the trip
	Driver: the person that posted the drive
	startLocation: starting geographical location point for the trip
	endLocation: starting geographical location point for the trip
	tripTime: total expected duration of the trip
	DatabaseQuerier: 1.2.5
	tripObject: 1.2.3.7_Trip_Object
Referenced By	1.2.3_Ride_Manager
Viewpoint	PseudoCode

1.2.3.2 AcceptTrip



Name	1.2.3.2 AcceptTrip
Purpose	Remove a rider from a given trip.
Description	Processes user requests to remove a rider from a trip.
Requirements	5
Elements	**User:** person interacting with the application.
	tripInfo: trip information used to get a trip from existing trips.
	tripMatchmake: see view 1.2.3.0
	tripObject: TripObject being sent in order to accept the trip. See 1.2.3.7_Trip_Object
	tripConfirmation: whether the trip was accepted or not.
	Trips Databases: stores all the information related to trips.
	editTrips: used to change the status of a trip. See 1.2.3.5
Referenced By	1.2.3_Ride_Manager
Viewpoint	Data Flow Diagram

1.2.3.3 CreateTripInstance



Name	1.2.3.3 CreateTripInstance
Purpose	Create a new trip object from received information.
Description	Receives request information and interprets it into a Trip Object. This object is then passed to another process that stores the object in a database.
Requirements	
Elements	**Maps:** 1.3.3
	Date Class: Library to transform tripTime into a tripDateObject and postDateObject.
	callProcess: receives a nullable trip object from the user interactor, then calls acceptTrip and passes it the trip object if the trip object is not null. If the trip object is null, then createTripInstance is called instead, passing it requestInfo.
	queryCreateTrip: 1.2.3.7
	calculateFare: 1.2.3.4
	convertToTrip: Receives requestInfo upon call and calls Maps and Date to receive date and map objects, then calls calculateFare with the map object to receive a fare for the trip. Then compiles all of this information into a trip object, which is then sent to queryCreateTrip.
	requestInfo: a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
	Distance: the expected distance a trip will take.
	Map Object: a JSON file returned by Maps.
	TripObject: 1.2.3.7
	startLocation: location for where the trip starts.
	endLocation: location for where the trip ends.
	tripTime: the time at which the trip takes place (date and time).
	tripDateObject: a time and date object formatted by a Date library for when the trip takes place.

	postDateObject: a time and date object formatted by a Date library for when a TripObject is created.
	fare: the monetary cost of a trip per rider.
	Data Base: the database in which all trips are stored.
Referenced By	1.2.3 RideManager, 1.2.3.4 CalculateFare, 1.2.3.0 TripMatchmake
Viewpoint	Data Flow Diagram

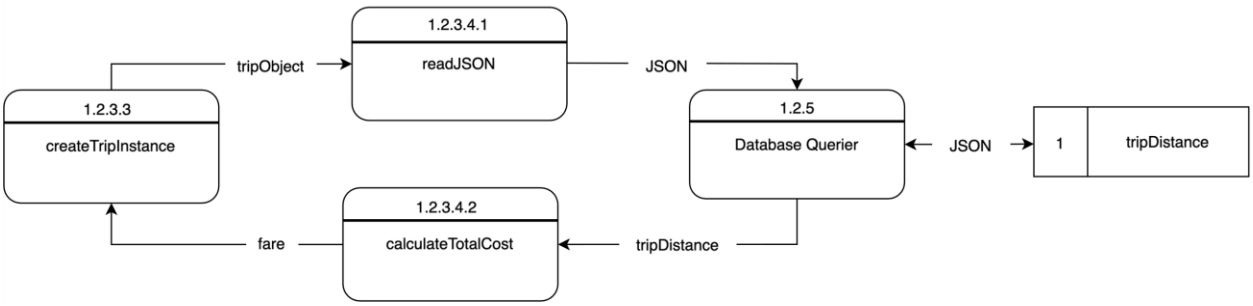
1.2.3.3.B createTripInstance

```
createTripInstance(tripDateObject, postDateObject, requestInfo)

   mapObject = Maps.getRoute(requestInfo.startLocation,
requestInfo.endLocation)

   tripDetails = {
      "startLocation": requestInfo.startLocation,
      "endLocation": requestInfo.endLocation,
      "tripDate": tripDateObject,
      "postDate": postDateObject,
      "tripTime": requestInfo.endDate - requestInfo.startDate
      "mapData": mapObject
   }

   trip = {
      "tripDetails": tripDetails,
      "fare": fare,
      "distance": distance
   }

   requestInfo = callProcess(tripDetails)
   distance = mapObject.getDistance()
   fare = calculateFare(distance)

   tripObject = new Trip(trip)

   RETURN tripObject
```

Name	1.2.3.3.B createTripInstance
Purpose	The primary purpose of this code is to create a new trip instance by retrieving routing information, calculating the trip fare, and storing all relevant trip data in a structured format.
Description	Creates a trip instance based on different parameters given by users.
Requirements	1-4
Elements	**tripDateObject:** object representing the date of the trip.
	postDateObject: object representing the post date of a trip.
	tripTime: an object or string representing the time of a trip.
	startLocation: start location of the trip.
	endLocation: end location of the trip.
	MapObject: an object that contains map information, usually addresses.
	getRoute: provides a route based on startlocation and endLocation.

	tripDetails: a dictionary containing all the details of a trip.
	tripObject: 1.2.3.7 Trip
	calculateFare: 1.2.3.4
	fare: price to be paid based on the ride distance.
	callProcess: 1.2.3.0.2
	requestInfo: a collection of variables representing account name, start location, end location, start date, end date, etc. See 1.2.3.7 TripObject.
	Maps: Google Maps API.
Referenced By	
Viewpoint	Pseudocode

1.2.3.4 CalculateFare



Name	1.2.3.4 CalculateFare
Purpose	Calculate the total cost of the ride.
Description	The Json data of the TripObject is pulled, and trip start location and the end location to find the number of miles. Based on the miles, there is a cent amount applied. The cent amount is added to a base rate and the total will be the fare. The fare is then be handed off to createTripInstance.
Requirements	5
Elements	**tripObject**: 1.2.3.1
	readJSON: Takes the object and reads into an accepted data type.
	DatabaseQuerier: 1.2.5
	tripTable: Extracted start location and end location from the Json data.
	tripDistance: storage for each location that determines the number of miles.
	JSON: Organization of data in a key-value pair.
	calculateTotalCost: determines the total cost of the ride
	fare: Stores the total cost of the ride.
	createTripInstance: 1.2.3.3
Referenced By	1.2.3 RideManager , 1.2.4.1 Get Total
Viewpoint	Data Flow Diagram

1.2.3.4.3 CalculateFare

```
calculateFare(distance: double)

 baseAmount ← 10.00

 centPerMile ← .10

 fare ← baseAmount + (centPerMile * distance)

 RETURN fare
```

Name	1.2.3.4.3 CalculateFare
Purpose	Calculate the total cost of the ride.
Description	The Json data of the Trip object is pulled, and trip start location and the end location to find the number of miles. Based on the miles, there is a cent amount applied. The cent amount is added to a base rate and the total will be the fare. The fare is then handed off to createTripInstance.
Requirements	5
Elements	**calculateFare:** The function that is computing the fare cost of the ride.
	distance: The variable that holds the total amount of miles from the start location to the end location.
	baseAmount: The initial price regardless of miles travelled.
	centPerMile: The cent amount that will be applied per mile.
	Fare: The total price of the trip that is calculated.
Referenced By	1.2.3 RideManager
Viewpoint	Psuedocode

1.2.3.5.1 QueryEditTrips

```
queryEditTrips (trip info)

 WRITE to tripDatabase trip info by tripID
```

Name	1.2.3.5.1 requestData
Purpose	Queries the MapAPI for a selected trip
Description	Makes an API call and returns the JSON data
Requirements	1, 2, 3
Elements	**Trip Info**: all relevant information required for the MapAPI, which is trip time, start location, end location, and stops
	TripID: a unique numeric value given to each trip contained in tripID
	Trip Database: contains data about each trip
Referenced By	1.2.3_Ride_Manager
Viewpoint	PseudoCode

1.2.3.5.2 QueryCreateTrip

```
queryCreateTrip (tripObject)
 json = {
 “owner”: tripObject.owner,
 “driver”: tripObject.driver,
 “startLocation”: tripObject.StartLocation
 “endLocation”: requestInfo.EndLocation
 “tripTime”: tripObject.tripTime
 }
 databaseQuerier(trip)
```

Name	1.2.3.5.2 QueryCreateTrip
Purpose	Create a new trip in the trip database
Description	Send the trip info and to the database
Requirements	1, 2, 3, 19
Elements	**owner:** The requestor of the trip
	Driver: the person that posted the drive
	startLocation: starting geographical location point for the trip
	endLocation: starting geographical location point for the trip
	tripTime: total expected duration of the trip
	DatabaseQuerier: 1.2.5
	tripObject: 1.2.3.7_Trip_Object
Referenced By	1.2.3_Ride_Manager
Viewpoint	Pseudocode

1.2.3.5 EditTrip

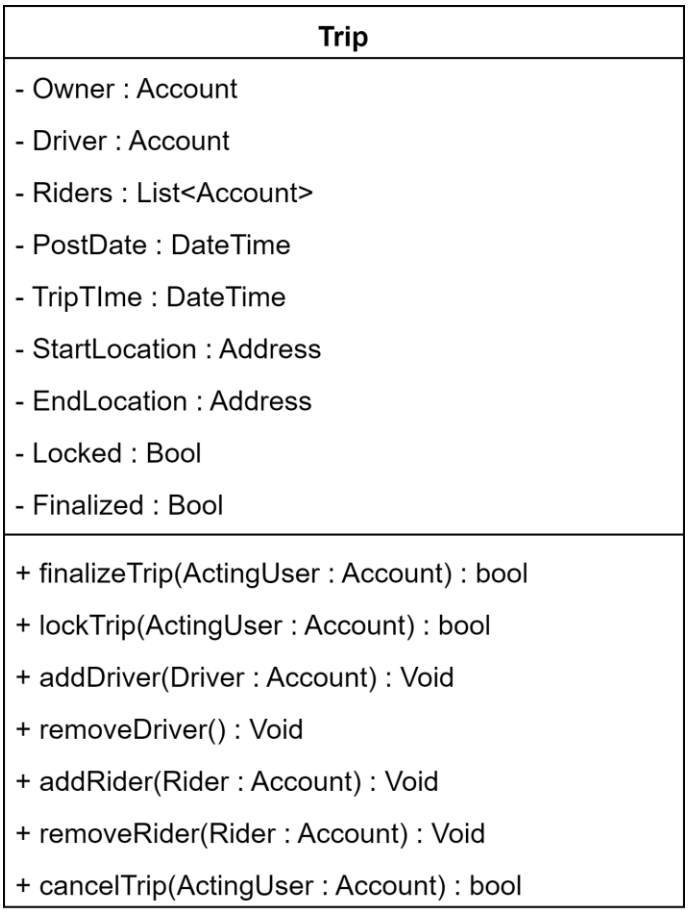
Name	1.2.3.5 EditTrip
Purpose	Allows a user or admin to modify trip details, including date, time, passenger count, and more.
Description	Receives user input on the trip to be edited and the updated details. Retrieves the trip information via query and applies the necessary updates.
Requirements	2
Elements	**QueryDatabase**: 1.2.5_Database_Querier
	EditTrip: Enables a user to select and update trip details.
	tripInfo: A return of a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
	requestInfo: A request for a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
	updatedInfo: The new, updated information for the trip formatted as TripObject json. See TripObject json.
Referenced By	1.2.3_Ride_Manager, 1.2.3.7.2 RemoveRider, 1.2.3.0 TripMatchmake
Viewpoint	Data Flow Diagram

1.2.3.5.A EditTrip

```
editTrip(tripInfo)
   trip = queryDatabase(tripInfo)
   updatedInfo = applyUpdates(trip)
   updateDatabase(updatedInfo)
   RETURN updatedInfo
```

Name	1.2.3.9.A editTrip
Purpose	Allows a user or admin to modify trip details, including date, time, passenger count, start location, end location, and more.
Description	Receives the original trip information and updated details from the user or admin. Retrieves the current trip information via a database query, applies the necessary updates, and returns the modified trip details.
Requirements	2
Elements	**queryDatabase**: 1.2.5_Database_Querier
	editTrip: Enables a user or admin to select and update trip details, modifying the trip based on provided updated information.
	tripInfo: 1.2.3.7 tripObject
	updatedInfo: 1.3.2.1.1 Trip JSON
	applyUpdates: A function that takes the current trip data (tripInfo) and applies the changes based on user or admin input, returning the modified trip details.
	UpdateDatabase: Sends the modified trip information (updatedInfo) back to the database to update the trip record.
Referenced By	1.2.3.9 Edit Trip
Viewpoint	Pseudocode

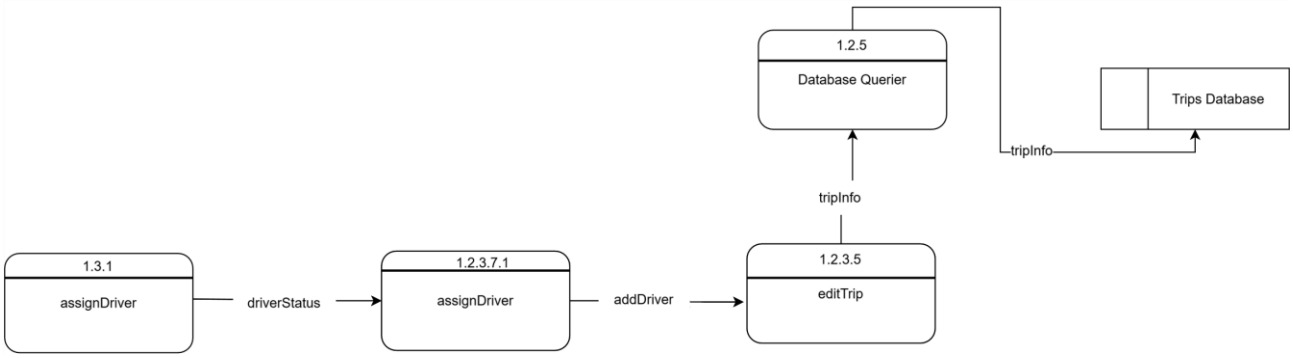
1.2.3.7 Trip



Name	1.2.3.7 Trip
Purpose	This class is the object that will store the information that make up trips after we get it from the database.
Description	The object we will store trip data in after we get it from the database
Requirements	2, 11
Elements	**Owner:** The account who owns, is responsible for, the trip
	Driver: The account who will drive on the trip
	Riders: A list of users who will be passengers
	PostDate: The datetime the trip was posted to the public
	StartLocation: Start location for the trip
	EndLocation: Destination for the trip
	Locked: When a trip is locked new riders may not be added and a new driver may not be added. However, riders and drivers may still leave. Once locked, trips may be unlocked by the owner.
	Finalized: When the trip is finalized, we send invoices for payment and start processing payments. Once finalized, a trip may only be un-finalized by an admin.

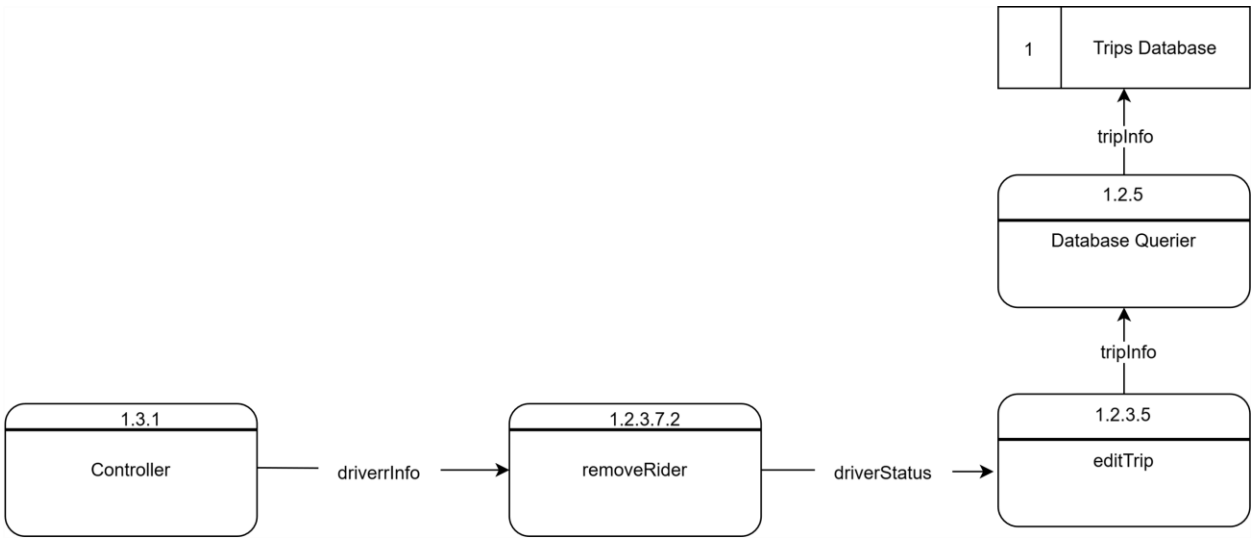
	finalizeTrip: Allows the owner to finalize the trip
	lockTrip: Allows the owner to lock the trip
	unlockTrip: Allows the owner to unlock the trip
	addDriver: Adds a driver to the trip
	removeDriver: Removes any driver from the trip
	addRider: Adds a rider to the trip
	removeRider: Removes a specific rider, if present, from the trip
	cancelTrip: Cancels the trip
Referenced By	1.2.3 RideManager, 1.2.3.3 CreateTripInstance, 1.2.3.7.2 RemoveDriver, 1.3.3.1 GenerateTripView, 1.2.3.0 TripMatchmake, 1.2.4.2.3 createReceipt, 1.2.6.1 ChatroomManager, 1.2.6.1.2 createChatroom
Viewpoint	Class Diagram

1.2.3.7.1 AddDriver



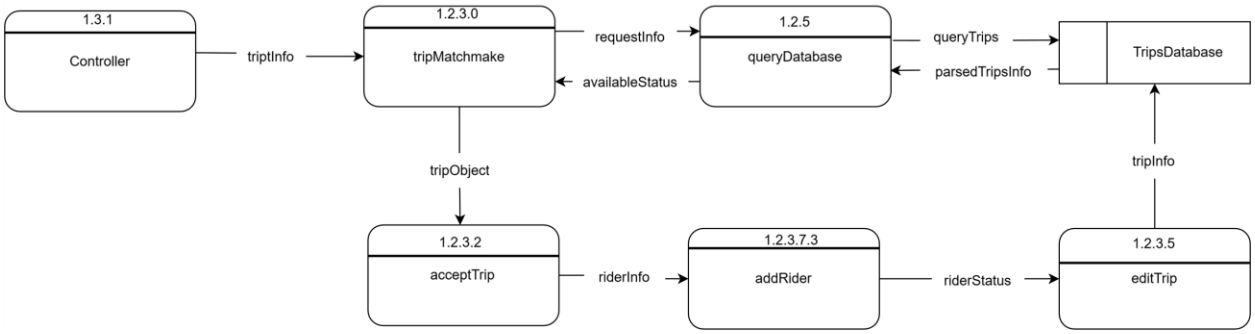
Name	1.2.3.7.1 AddDriver
Purpose	Add a driver to an existing trip.
Description	Processes user requests to add a driver to a trip.
Requirements	2
Elements	**Controller:** 1.3.1
	driverStatus: status of the driver, is the owner of the ride the driver or not.
	assignDriver: assigns a driver if the ride has not been assigned one, or if the owner of the ride is not the default driver.
	a**ddDriver:** send the driver information to the EditTrips processor.
	editTrips: See 1.2.3.5_Ride_Manger: EditTrip
	tripInfo: See 1.3.2.1.1
	DatabaseQuerier: 1.2.5
	Trips Databases: stores all the information related to trips.
Referenced By	1.2.3.7_Trip_Object
Viewpoint	Data Flow Diagram

1.2.3.7.2 RemoveDriver



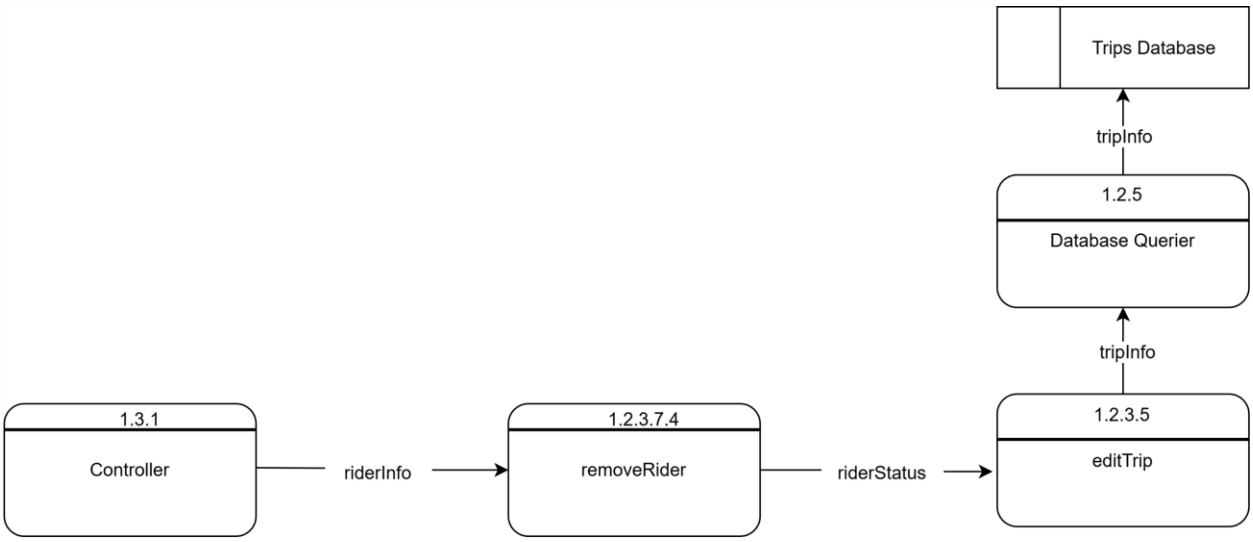
Name	1.2.3.7.2 RemoveDriver
Purpose	Remove a driver from an existing trip.
Description	Processes user requests to remove a driver from a trip.
Requirements	2
Elements	**Controller:** 1.3.1
	driverStatus: sends the status of the driver, whether he is a driver or not.
	removeDriver: removed a driver from an existing ride.
	driverInfo: send the driver information to the removed driver directly from the user.
	editTrips: See 1.2.3.5_Ride_Manger: EditTrip
	tripInfo: see 1.3.2.1.1
	DatabaseQuerier: 1.2.5
	Trips Databases: stores all the information related to trips.
Referenced By	1.2.3.7_Trip_Object
Viewpoint	Data Flow Diagram

1.2.3.7.3 AddRider



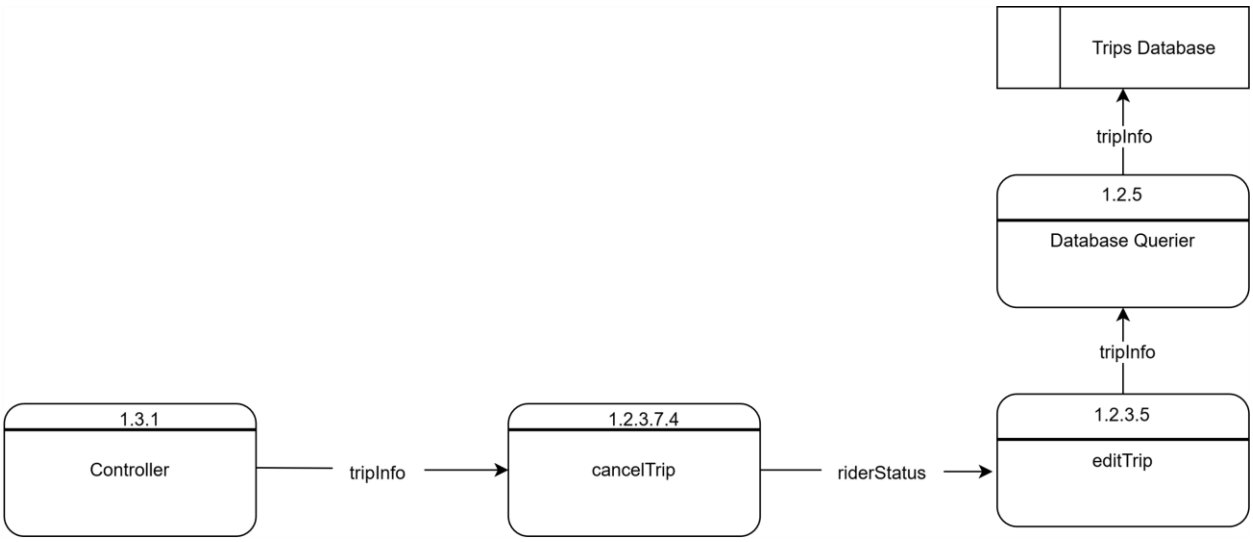
Name	1.2.3.7.3 AddRider
Purpose	Add a rider to a trip.
Description	Processes user requests to add a rider to a trip.
Requirements	2
Elements	**Controller:** 1.3.1
	tripInfo: trip information to input in the matchmake algorithm.
	tripMatchMake: See 1.2.3.0 TripMatchmake
	queryDatabase: See 1.2.5_Database_Querier
	requestInfo: information being requested from the database.
	Trips Databases: stores all the information related to trips.
	parsedTripInfo: information coming out from the database to be read by the user.
	availableSeats: number of available seats on a given ride.
	editTrips: See 1.2.3.5_Ride_Manger: EditTrip
	tripObject: trip being passed so user can accept a trip it can be by the rider or driver.
	riderInfo: rider information being passed to the addRider processor.
	addRider: adds a rider to a given ride.
	riderStatus: passes the riderStatus to the editTrip processor.
	tripInfo: trip information being updated to the database.
Referenced By	1.2.3.7 TripObject
Viewpoint	Data Flow Diagram

1.2.3.7.4 RemoveRider



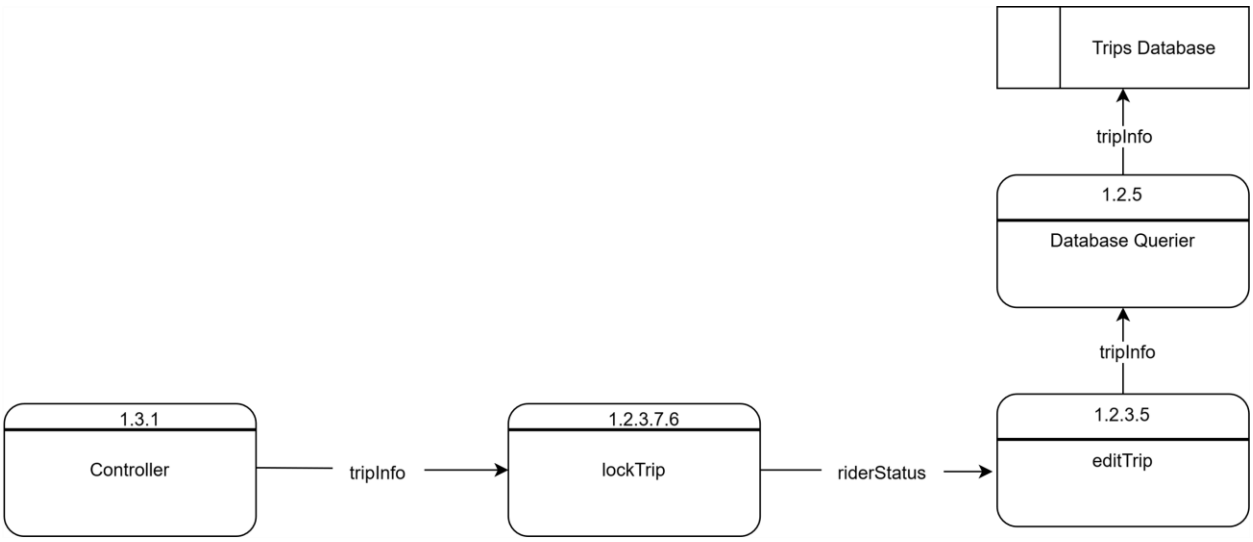
Name	1.2.3.7.4 RemoveRider
Purpose	Remove a rider from a given trip.
Description	Processes user requests to remove a rider from a trip.
Requirements	2
Elements	**Controller:** 1.3.1
	riderInfo: rider information being moved to the removeRider processor.
	removeRider: remove a rider from a given trip.
	riderstatus: status of a rider, whether absent or present.
	editTrips: edits a given trip. 1.2.3.5
	Trips Databases: stores all the information related to trips.
	tripInfo: see 1.3.2.1.1
Referenced By	1.2.3.7_Trip_Object
Viewpoint	Data Flow Diagram

1.2.3.7.5 CancelTrip



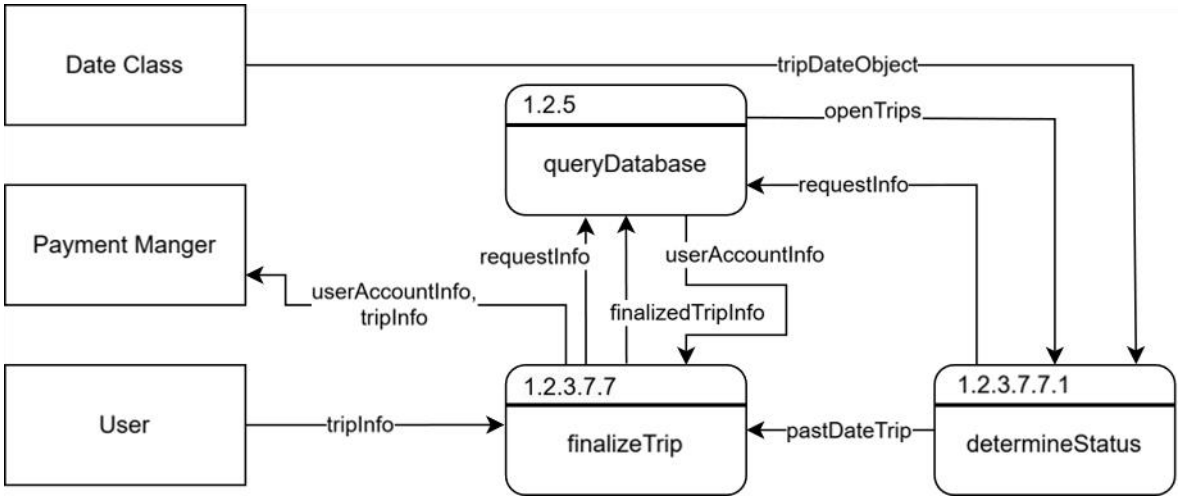
Name	1.2.3.7.5 CancelTrip
Purpose	Allows a user—primarily a Driver—to cancel a trip. In certain cases, a Rider or an Admin may also cancel, such as when a Rider requests a trip that remains unfulfilled.
Description	Receives information pertaining to what trip to cancel, along with information with the database, it will then update the database with the new “cancelled” status.
Requirements	2
Elements	**Controller:** 1.3.1
	queryDatabase:1.2.5_Database_Querier
	cancelTrip: Enables a user or admin to cancel a trip, rendering it unavailable.
	tripInfo: see 1.3.2.1.1
	cancelledTripInfo: Updates the trip status to "Cancelled," preventing access or participation.
	requestInfo: A request for a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
Referenced By	1.2.3.7_Trip_Object
Viewpoint	Data Flow Diagram

1.2.3.7.6 LockTrip



Name	1.2.3.7.6 LockTrip
Purpose	Locks the trip, this can only be done by the driver or an admin. Once locked a trip's status and riders cannot change without penalty.
Description	Receives user input on which trip to lock, retrieves the trip details via query, and updates its status to "Locked."
Requirements	2
Elements	**Controller:** 1.3.1
	queryDatabase: See 1.2.5 DatabaseQuerier
	lockTrip: This will change the state of the trip to "Locked". A user or admin can do this, once done, status and riders cannot change without penalty.
	tripInfo: see 1.3.2.1.1
	requestInfo: A request for a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
	lockedTripInfo: Information of the trip updated to "locked" so it cannot be easily changed.
Referenced By	1.2.3.7 TripObject
Viewpoint	Data Flow Diagram

1.2.3.7.7.A FinalizeTrip



Name	1.2.3.7.7 FinalizeTrip
Purpose	Allow a user, specifically a Driver, to finalize a trip to process payment and complete the trip finalization process. FinalizeTrip can also be done automatically from the trip passing the trip date.
Description	Processes user requests to finalize a trip, updating its state to “finalized” and triggering billing for associated users. Additionally, it retrieves trip data to check if the arrival date and time have passed without manual finalization, ensuring automatic finalization when necessary.
Requirements	2,5,7,8
Elements	**Date Class**: Date class library to transform tripTime and tripDate into a tripDateObject.
	queryDatabase: See 1.2.5_Database_Querier
	PaymentManager: See 1.2.4_Payment_Manager
	User: The source of all incoming data from the Frontend or the user.
	finalizeTrip: Allows the owner, or system to finalize the trip.
	determineStatus: This will receive date and time and return to FinalizeTrip if the trip has passed its scheduled time of arrival.
	tripDateObject: A time and date object formatted by a Date library for when the trip takes place.
	openTrips: A collection of trips that remain available for interactions, excluding those that are Finalized, Closed, or Locked.
	userAcountInfo: Provides user account information for FinalizeTrip, and PaymentManager to complete the process.

	requestInfo: A request for a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
	tripInfo: A return of a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
	finalizedTripInfo: Information of the trip updated to “finalized” so it cannot be easily changed.
	pastDateTrip: A trip that has exceeded their latest arrival date and is not in a Closed, Locked, or Finalized state.
Referenced By	1.2.3_Ride_Manager
Viewpoint	Data Flow Diagram

1.2.3.7.7.B FinalizeTrip

```
finalizeTrip(userAccountInfo, tripInfo, pastDateTrip)
  FOREACH tripInfo.rider IN tripInfo
     AccountInfo = queryDatabase(userAccountInfo)
      paymentManager(userAccountInfo, tripInfo)
      FinalizedTripInfo
      IF pastDateTrip = true
         updateTripStatus(tripInfo, "finalized": true)
      QueryDatabase(finalizedTripInfo)
```

Name	1.2.3.7.7.2 FinalizeTrip
Purpose	The finalizeTrip function allows a driver (or the system) to finalize a trip by marking it as completed, triggering payment processing, and ensuring the trip status is updated. This process can also be automated when the trip's scheduled date and time have passed.
Description	The finalizeTrip function takes in a trip that has passed its scheduled date, retrieves the associated user account information, processes the payment, and updates the trip status to "finalized." The system ensures that the trip cannot be modified after finalization, and the user is billed accordingly.
Requirements	2,5,7,8
Elements	**tripInfo**: 1.3.2.1.1 Trip
	userAccountInfo: The user accountInfo, with a type being rider. 1.3.2.1.7 Account
	queryDatabase: A function that retrieves tripInfo and userAccountInfo from the database based on the provided tripInfo. **1.2.5_Database_Querier**
	paymentManager: 1.2.4 PaymentManager
	finalizedTripInfo: The trip data after it has been updated with the status "finalized" to indicate the trip's completion.
	pastDateTrip: we set this to true if the date of the trip has passed and the trip is not in a Closed, Locked, or finalized state, hence finishing the trip.
	updateTripStatus: A function that updates the status of the trip (e.g., from "false" to "true").
	tripInfo.rider: All riders within a trip and their info.
Referenced By	1.2.3.7.7.A FinalizeTrip
Viewpoint	Pseudocode

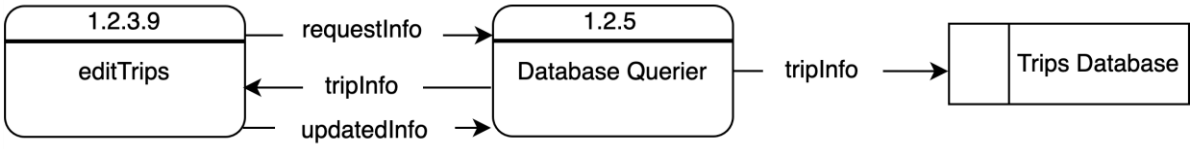
1.2.3.7.7.1 determineStatus

```
determineStatus(requestInfo)
   openTrips = queryDatabase(requestInfo)
   FOR EACH trip IN openTrips
      TripDateObject = DateClass(trip.tripTime)
      IF currentDateTime > tripDateObject
         PastDateTrip = trip
         FinalizeTrip(pastDateTrip)
```

Name	1.2.3.7.7.1 determineStatus
Purpose	Determines which trips have passed their scheduled date and time, and if so, triggers the finalization process by passing those trips to the finalizeTrip function.
Description	The determineStatus function retrieves all open trips from the database, checks if each trip's scheduled time has passed, and if a trip is overdue, it sends that trip to the finalizeTrip function for finalization. If a trip has not yet passed its scheduled time, no action is taken.
Requirements	2,5,7,8
Elements	**queryDatabase**: 1.2.5_Database_Querier
	requestInfo: A request containing the necessary information to query the database for open trips.
	openTrips: A collection of trips retrieved from the database that are still available for interactions.
	trip: A single trip from the collection of openTrips that is being processed. 1.3.2.1.1 Trip
	tripTime: The time at which the trip is scheduled to take place, used for comparison.
	tripDateObject: A Date object representing the scheduled trip time converted from trip.tripTime.
	currentDateTime: The current date and time, used to compare against the trip's scheduled time.
	pastDateTrip: A trip that has passed its scheduled time and is ready for finalization.
	finalizeTrip: Function that finalizes a trip, marking it as complete and processing payment if necessary.
	DateClass: A class or library used to convert trip time into a Date object for comparison.
Referenced By	1.2.3.7.7.A FinalizeTrip, 1.2.3.7.7.B FinalizeTrip
Viewpoint	Pseudocode

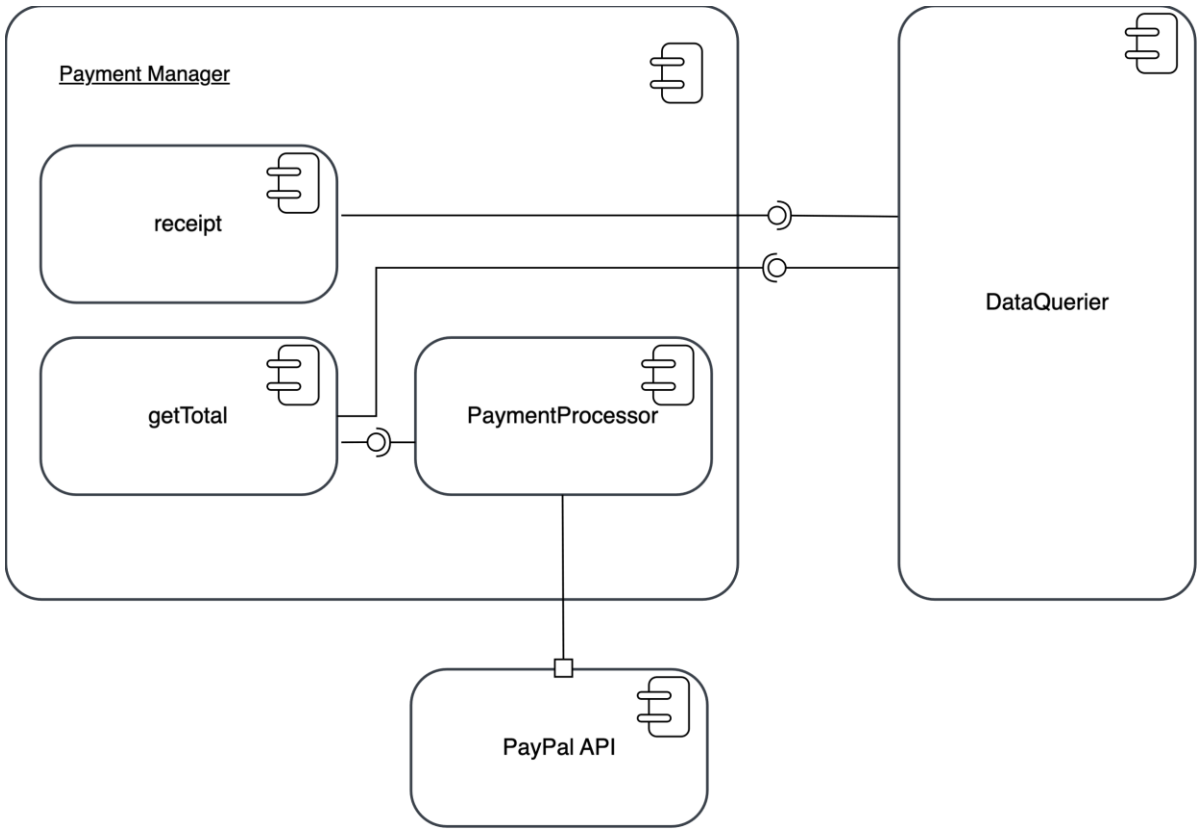
Name	1.2.3.8 AcceptTrip
Purpose	Remove a rider from a given trip.
Description	Processes user requests to remove a rider from a trip.
Requirements	5
Elements	**User:** person interacting with the application.
	tripInfo: trip information used to get a trip from existing trips.
	tripMatchmake: see view 1.2.3.0
	tripObject: TripObject being sent in order to accept the trip. See 1.2.3.7_Trip_Object
	tripConfirmation: whether the trip was accepted or not.
	Trips Databases: stores all the information related to trips.
	editTrips: used to change the status of a trip. See 1.2.3.9_Ride_Manger: EditTrip
Referenced By	1.2.3_Ride_Manager
Viewpoint	Data Flow Diagram

1.2.3.9.A EditTrip



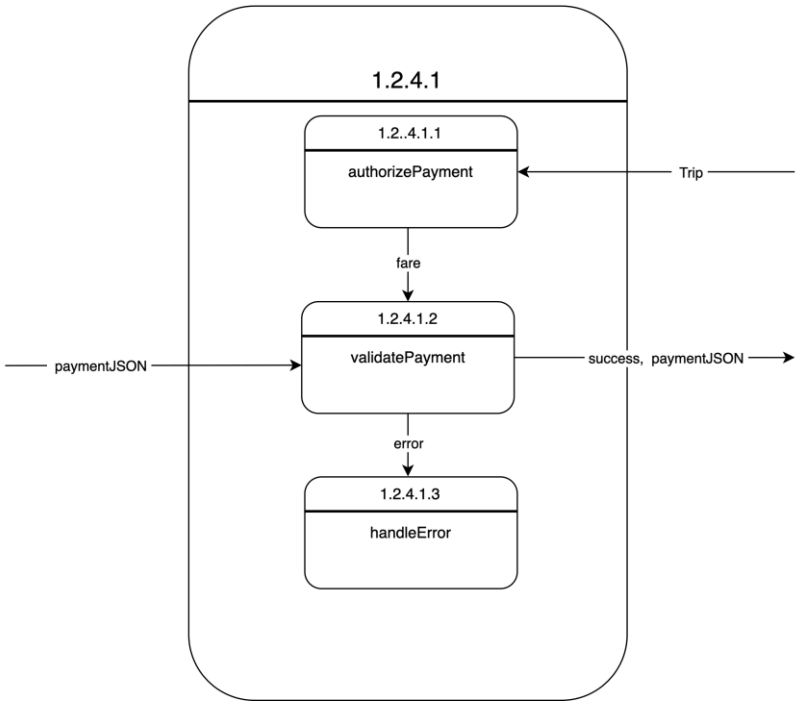
Name	EditTrip
Purpose	Allows a user or admin to modify trip details, including date, time, passenger count, and more.
Description	Receives user input on the trip to be edited and the updated details. Retrieves the trip information via query and applies the necessary updates.
Requirements	2
Elements	**QueryDatabase**: 1.2.5_Database_Querier
	EditTrip: Enables a user to select and update trip details.
	tripInfo: A return of a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
	requestInfo: A request for a collection of variables representing account name, start location, end location, start date, end date, etc. See TripObject.
	updatedInfo: The new, updated information for the trip formatted as TripObject json. See TripObject json.
Referenced By	1.2.3_Ride_Manager, 1.2.3.7.2 RemoveRider, 1.2.3.0 TripMatchmake
Viewpoint	Data Flow Diagram

1.2.4 PaymentManager



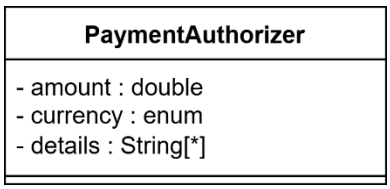
Name	1.2.4 PaymentManager
Purpose	The subsystems within PaymentManager have the ability to communicate with other systems on the server side.
Description	Business Logic of Subsystems communicates with the PayPalAPI to process payments and generate receipts for specific trips for the user.
Requirements	9.0
Elements	**PaymentProcessor:** 1.2.4.4
	Receipt: Class allowing generation of payment receipts, creating a transaction history, and telling rides that a seat has been purchased.
	Get Total: Function for retrieving the total amount of the transaction
	Data Querier: 1.2.5
	PayPalAPI: 1.3.4
Referenced By	1 System, 1.1.1 UserInterface, 1.1.1.2 PaymentUserInterface, 1.2 Backend, 1.2.5 DatabaseQuerier, 1.3.1.A Controller, 1.2.6 CommunicationManager, 1.2.6.1 ChatroomManager, 1.2.3.7.7 FinalizeTrip, 1.3.1.2 POSTHandler, 1.3.1.3 GETHandler
Viewpoint	Component Diagram

1.2.4.1 GetTotal



Name	1.2.4.1 GetTotal
Purpose	The function that retrieves the fare cost and sends it through the Payment Gateway Controller to the PaymentService
Description	The fare cost that was calculated in RideManager is given to the DatabaseQuerier. getTotal retrieves the fare and stores it and sends it to the Frontend through the Payment Gateway Controller.
Requirements	9
Elements	**Trip:** The object of the Trip.
	fare: The cost of the ride.
	paymentJSON: 1.1.1.2.1
	error: A variable indicating that the paymentJSON couldn't be validated.
	success: 1.1.1.2.1
	handleError: Displays a message stating that the total was not found.
	validatePayment: 1.1.1.2.1.1
Referenced By	1.2.4.2 Receipt, 1.2.4 PaymentManager
Viewpoint	Data Flow Diagram

1.2.4.1.1 AuthorizePayment



Name	1.2.4.1.1 AuthorizePayment
Purpose	An object that requests payment authorization.
Description	The object will gather total, currency, and details regarding the order, which will then be sent to get authorized by PayPal service.
Requirements	9
Elements	**amount:** The total cost of the payment.
	currency: The currency type of the payment (e.g. USD, CAD).
	details: Additional details regarding the order (e.g. rider, driver, destination).
Referenced By	1.3.3.1.1.A
Viewpoint	Class Diagram

1.2.4.1.1.A AuthorizePayment

```
{
    "id":"int",
    "status":"enum",
    "amount":{
        "total":"double",
        "currency_code":"enum"
    },
    "details":{
        "name":"String"
    }
}
```

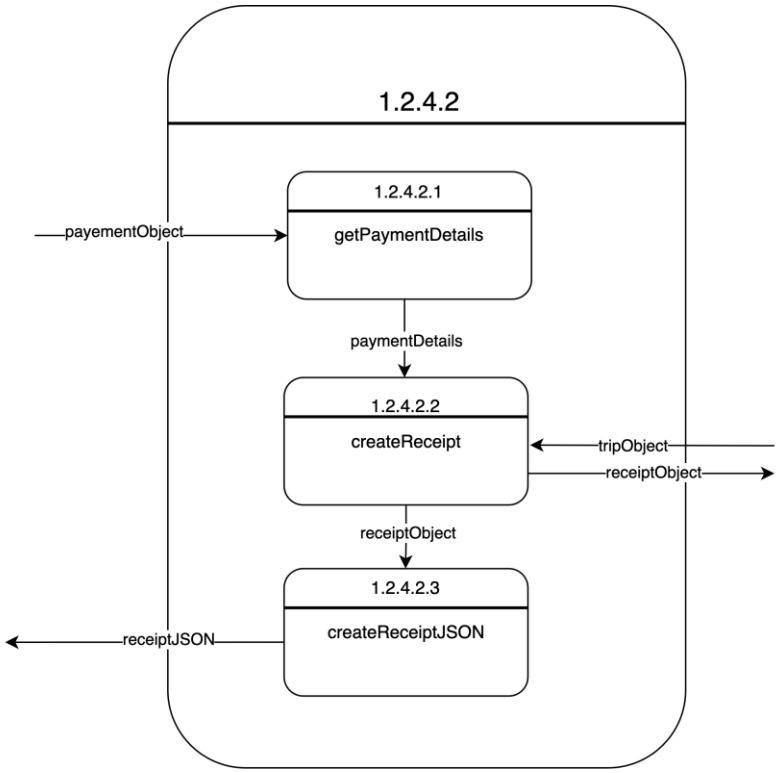
Name	1.2.4.1.1.A AuthorizePayment
Purpose	Authorizes payment.
Description	A JSON Schema that authorizes an order.
Requirements	9
Elements	**id:** The ID of the order for which the payer confirms their intent to pay.
	status: The action needed from the API.
	amount: 1.2.4.1.1
	total: The sum of payment.
	currency: 1.2.4.1.1
	details: 1.2.4.1.1
	name: The Nameof the payee
Referenced By	1.2.4.1.1
Viewpoint	JSON Schema

1.2.4.1.1.B authorizePayment

```
authorizePayment(trip)

   IF trip.fare IS NOT NULL
      validatePayment(fare)
   ELSE
      RETURN error
```

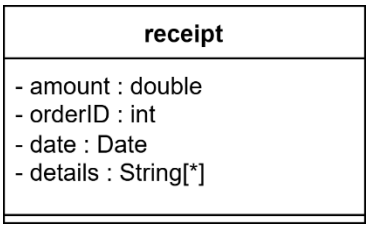
Name	1.2.4.1.1.B authorizePayment
Purpose	Verify the cost of the trip exist in the map Object.
Description	Accesses the map object from the database querier to make sure a fare has been assigned to a trip before generating payment requestion for the Paypal API.
Requirements	9
Elements	**Trip.fare:** Accesses the fare that was calculated in Ride Manager.
	fare: Local variable that is set to the data that was in trip.
	ValidatePayment: 1.2.4.1.2
	error: A boolean that says a fare could not be identified in the trip object.
Referenced By	1.2.4.1 Payment Manager: Get Total
Viewpoint	Pseudocode

1.2.4.2 Receipt



Name	1.2.4.2 Receipt
Purpose	Generate a receipt upon a payment completion
Description	PaymentProcessor sends a request to generateReceipt, it grabs the total and then sends that to the payment controller.
Requirements	5
Elements	**paymentObject:** 1.1.1.2.1
	getPaymentDetails: extract the neccessary information from the object.
	paymentDetails: Data from payment that is incorporated into the receipt.
	rideObject: Readable object to transform the information
	createReceipt: 1.2.4.2.3
	receiptObject: 1.1.1.2.1
	receiptJSON: 1.1.1.2.1
	createReceiptJSON: 1.2.4.2.3
Referenced By	1.2.4.4 PaymentProcessor, 1.2.4 PaymentManager, 1.1.1.2.1 PaymentService
Viewpoint	Data Flow Diagram

1.2.4.2.A Receipt



Name	**1.2.4.2.A** Receipt
Purpose	An object that creates a receipt.
Description	The object will gather total, date, and details regarding the order.
Requirements	9
Elements	**amount:** The total of the payment.
	date: The date and time the order was placed.
	details: Additional details regarding the order (e.g. rider, driver, destination).
	orderID: The ID for the order.
Referenced By	1.1.1.2.1 PaymentService, 1.2.4.2.3 createReceipt, 1.2.5 DatabaseQuerier, 1.1.1.2.1.3 transformReceipt, 1.1.1.2.2.3 displayReceipt, 1.1.1.2.1.4 sendReceipt
Viewpoint	Class Diagram

1.2.4.2.2 createReceipt

```
createReceipt(paymentObject, rideObject)

   receiptObject <- new receiptObject
   receiptObject.orderID <- generateRandomID()
   receiptObject.amount <- paymentObject.total
   receiptObject.date <- paymentObject.fulfillmentDate
   receiptObject.details <- [rideObject.Driver,
                             rideObject.Riders,
                             rideObject.endLocation]

   RETURN receiptObject
```

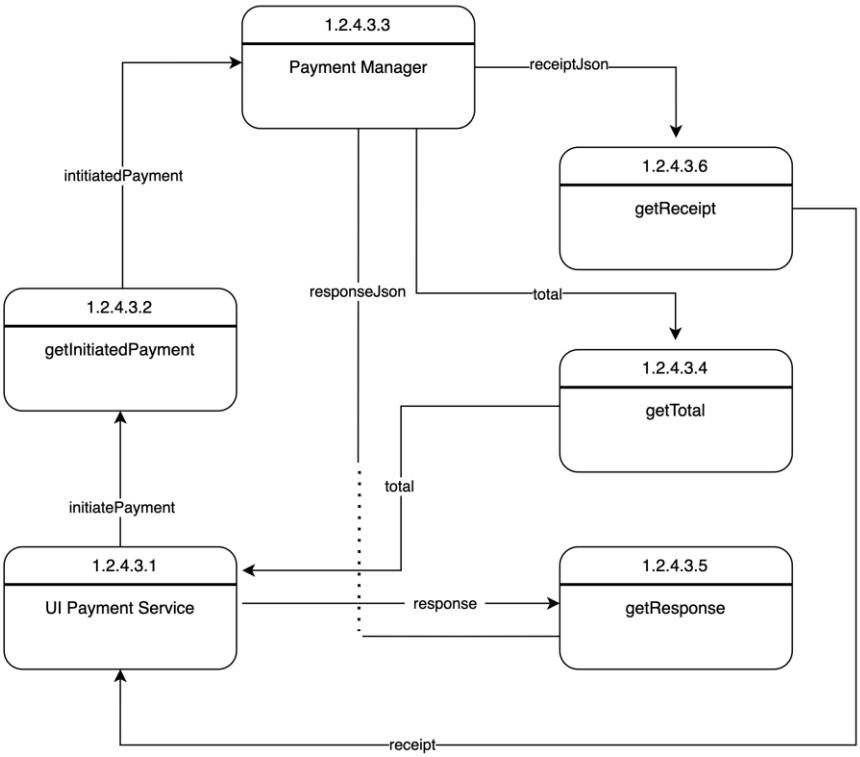
Name	1.2.4.2.2 createReceipt
Purpose	To compile the confirmObject into a JSON file to send to the backend.
Description	The JSON file is created to send to the backend for the API requests.
Requirements	9
Elements	**paymentObject:** 1.2.4.5
	rideObject: 1.2.3.7
	receiptObject: 1.2.4.2.A
	generateRandomID: Generates a random ID that hasn't been used yet for a receiptObject.
Referenced By	1.2.4.2 Payment Manager: Receipt
Viewpoint	Pseudocode

1.2.4.2.3 createReceiptJSON

```
createReceiptJSON(receiptObject)

   newJSON <- new receiptJSON
   NewJSON[receiptID] <- receiptObject.ReceiptID
   NewJSON[amount] <- receiptObject.amount
   NewJSON[date] <- receiptObject.date
   NewJSON[details] <- receiptObject.details

   RETURN newJSON
```

Name	1.2.4.2.3 createReceiptJSON
Purpose	To convert a receipt object into a receiptJSON for movement through the system
Description	Accepts a receiptObject as a parameter and returns a receiptJSON containing the information stored within the passed receiptObject.
Requirements	9
Elements	**receiptObject:** Receipt passed into the function as a parameter.
	receiptJSON: 1.3.2.1.4
Referenced By	1.2.4.2 Payment Manager: Receipt
Viewpoint	Pseudocode

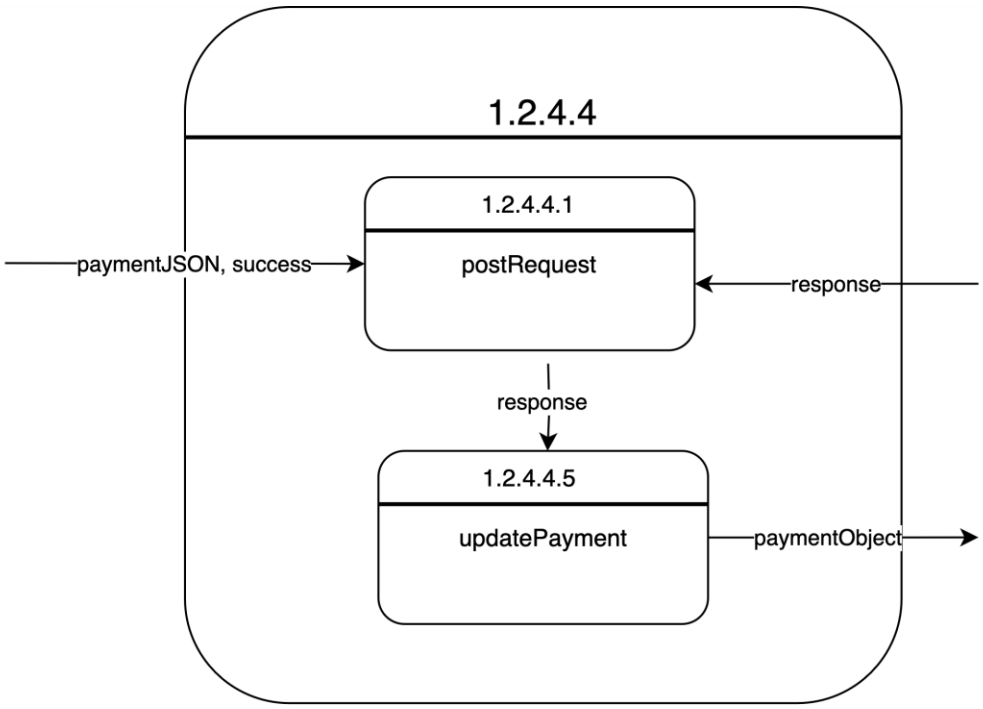
1.2.4.3 PaymentGatewayController



Name	1.2.4.3 PaymentManager: Payment Gateway Controller
Purpose	The component wherein data from the backend gets transferred to the Frontend and vice versa.
Description	A total gets sent to the controller, it receives a response from the PaymentService, the controller sends response to PaymentManager, PaymentManager then sends a receipt to the controller, which then sends the receipt to the user.
Requirements	9
Elements	**PaymentManager:** 1.2.4
	PaymentService: 1.1.1.2.1
	initiatePayment: The PayPal payment has been initiated on their end.
	responseJason: Json file containing response data.
	response: object with the response from PaymentService stating if a payment was successful or not.
	receipt: Object containing data for the order.
	receiptJason: Json file containing receipt data.
	getTotal: Function call to receive total from PaymentManager.
	getReceipt: Function call to receive receipt from PaymentManager.
	getResponse: Function to receive a response from Payment UI PaymentServices.

	initiatedPayment: Variable to tell PaymentManager that a payment has been initiated.
Referenced By	1.2.4.2 Receipt, 1.2.4.1 getTotal, 1.1.1.5.1 OrderConfirmation, 1.2.4 PaymentManager
Viewpoint	Data Flow Diagram

1.2.4.4 PaymentProcessor



Name	1.2.4.4 PaymentProcessor
Purpose	The process in which it is decided if a payment was successful.
Description	A rider attempts to initiate a payment. Once that happens, the PaymentProcessor will get called. If the payment was successful, a receipt is generated. If not, the user is taken back to the PaymentScreen.
Requirements	9
Elements	**paymentJSON:** 1.2.4.5.A
	success: 1.1.1.5
	postRequest: 1.2.4.4.1
	Response: 1.2.4.4.1
	updatePayment: 1.2.4.4.5
	PaymentObject: 1.2.4.5
Referenced By	1.2.4.2 Receipt, 1.2.4 PaymentManager
Viewpoint	Data Flow Diagram

1.2.4.4.1 PostRequest

```
postRequest(success, paymentData):
   IMPORT requests

   IF success == FALSE
      RETURN

   paymentHeaders <- {
      "contentType": "JSON",
      "authorization": "String",
      "payPalRequestId": "String",
      "prefer": "int",
       }

   response <- requests.post("https://exampleURL",
headers=paymentHeaders, data=paymentData)

   updatePayment(response)
```

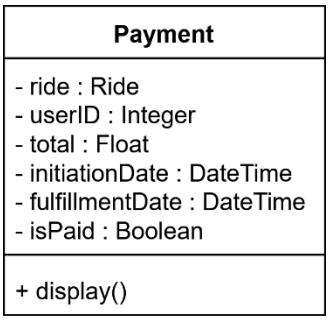
Name	1.2.4.4.1 postRequest
Purpose	Send a post request to PayPal API
Description	A post request is made that gets returned as a response.
Requirements	9
Elements	**requests:** PayPal specific object that allows requests to be made.
	paymentHeaders: A map containing PayPal specific data.
	headers: PayPal parameter set to the value of paymentHeaders.
	paymentData: A map containing information on the order (e.g. amount, currency, details).
	data: PayPal parameter set to the value of paymentData.
	response: Variable containing the response from PayPal.
	contentType: The format the file containing data is in.
	payPalRequestId: PayPal API key.
	prefer: Variable that lets PayPal know what variable type to return.
	authorization: PayPal-generated ID for the authorized payment.
	UpdatePayment: 1.2.4.4.5
	success: Bool that says whether a payment was successful or not.
Referenced By	1.2.4.4 Payment Processor, 1.1.1.2.2.1 redirectToPayPal
Viewpoint	Pseudocode

1.2.4.4.5 updatePayment

```
updatePayment(response):

   responseObject <- new Payment(response)

   RETURN responseObject
```

Name	1.2.4.4.5 updatePayment
Purpose	Turns a response into an object.
Description	Takes a response and transforms it into an object.
Requirements	9
Elements	**response:** 1.2.4.4.1
	Payment: 1.2.4.5
	responseObject: Object containing response data.
Referenced By	1.2.4.4 paymentProcessor
Viewpoint	Pseudocode

1.2.4.5 Payment



Name	1.2.4.5 Payment
Purpose	An object to hold all the necessary information and functionality of a payment object.
Description	A class diagram for a payment object.
Requirements	10
Elements	**ride**: Ride object that the payment connected to.
	userID: The key unique to the user, meant to identify them among other users.
	total: The total amount of money that the payment consists of.
	initiationDate: Date and time for when the payment process was initiated.
	fulfillmentDate: Projected date and time for when the payment hold is released and the payment goes through.
	isPaid: Boolean value representing if the payment has been fulfilled or not.
	display(): Function method to display the payment object.
Referenced By	1.1.1.2.1 PaymentService, 1.1.1.2.1.1 validatePayment, 1.1.1.2.1.2 transformPayement, 1.1.1.2.2.1 redirectToPayPal, 1.2.4.4 PaymentProcessor , 1.2.4.2.3 createReceipt
Viewpoint	Class Diagram

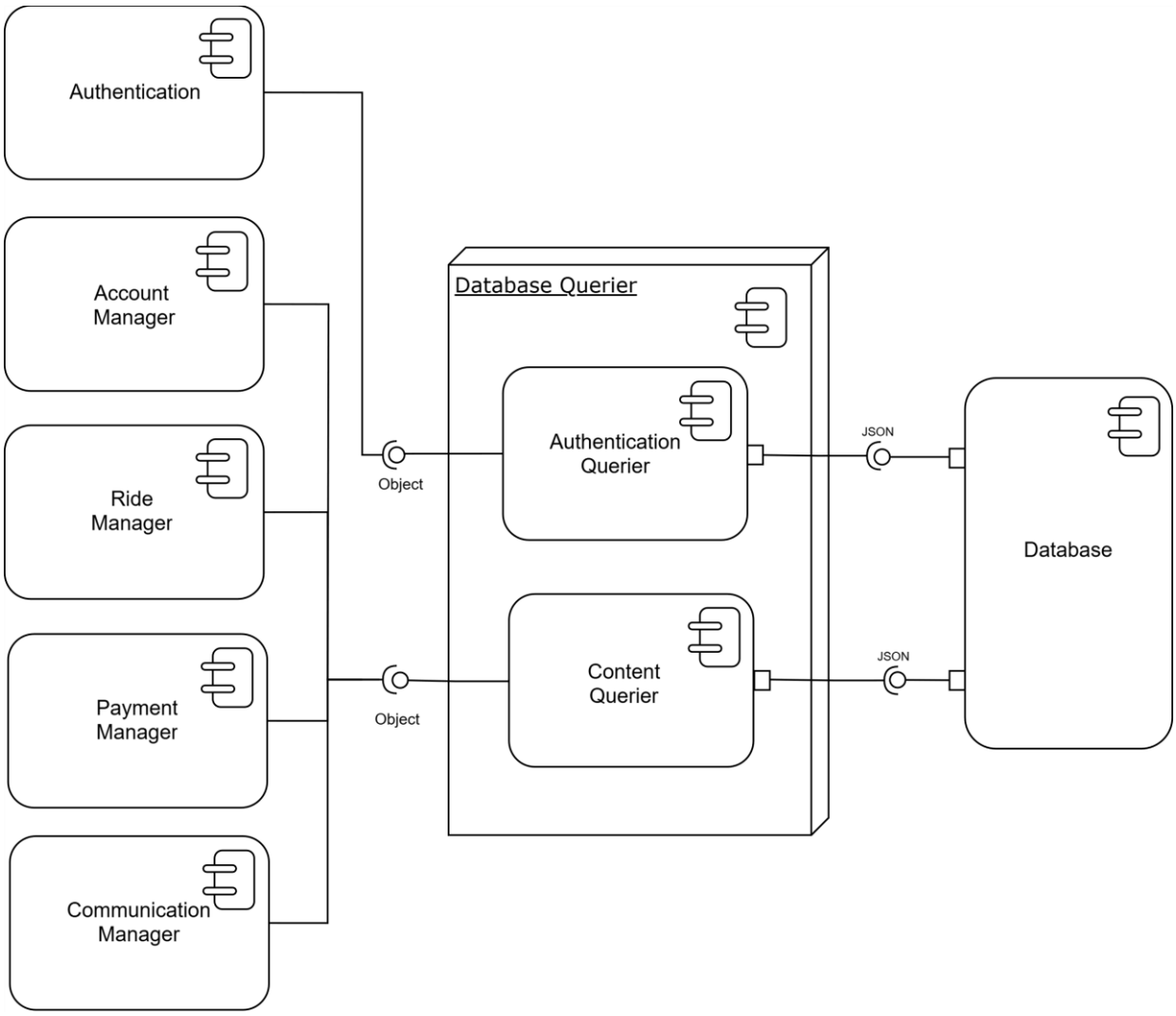
1.2.4.5.A Payment

```
{
    "rideID" : Integer,
    "userID" : Integer,
    "total" : Float,
    "initiationDate" :
    {
        "day" : Integer,
        "month" : Integer,
        "year" : Integer,
        "hour" : Integer,
        "minute" : Integer
    },
    "fulfillmentDate :
    {
        "day" : Integer,
        "month" : Integer,
        "year" : Integer,
        "hour" : Integer,
        "minute" : Integer
    },
    "isPaid" : Integer
}
```

Name	1.2.4.5.A Payment
Purpose	To illustrate the JSON Schema used when creating a payment entry in the database.
Description	The JSON schema for a payment object.
Requirements	10
Elements	**rideID:** The key unique to the ride object, meant to identify it among other rides.
	userID: The key unique to the user, meant to identify them among other users.
	total: The total amount of money that the payment consists of.
	initiationDate: Date and time for when the payment process was initiated.
	fulfillmentDate: Projected date and time for when the payment hold is released and the payment goes through.
	day: An integer value representing the day part of the date.
	month: An integer value representing the month part of the date.
	year: An integer value representing the year part of the date.
	hour: An integer value representing the hour part of the time.

	minute: An integer value representing the minute part of the time.
	isPaid: Boolean value representing if the payment has been fullfilled or not. In the schema it will be represented by a 0 for False and a 1 for True.
Referenced By	1.1.1.2.1 PaymentService,1.1.1.2.1.2 transformPayement, 1.2.4.4 PaymentProcessor, 1.2.5 DatabaseQuerier
Viewpoint	JSON Schema

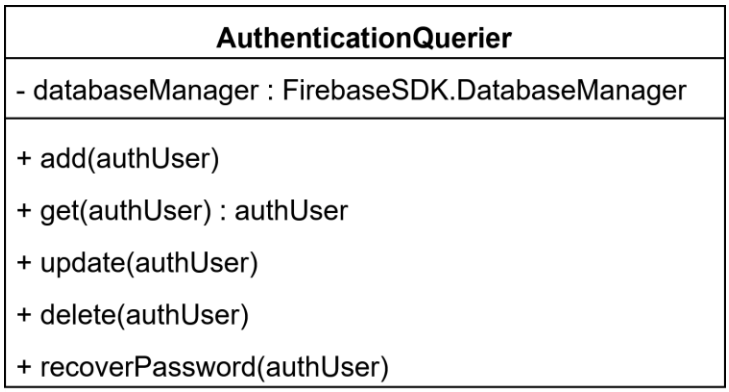
1.2.5 DatabaseQuerier



Name	1.2.5 DatabaseQuerier
Purpose	A view of the DatabaseQuerier and the connections to it.
Description	The components involved in retrieving and sending information from the database.
Requirements	11-12, 14, 16-18
Elements	**Database**: 1.3.2
	Authentication Querier: Handles queries and data manipulation for anything in the authentication database as well as authenticate and retrieve authentication token.
	Content Querier: Handles queries and data manipulation for anything going into the content database.
	AccountManager: 1.2.1

	RideManager: 1.2.3
	PaymentManager: 1.2.4
	CommunicationManager: 1.2.6
	Authentication: 1.2.2
	Object: Class objects that hold information for or retrieved from the database. 1.2.2.4, 1.2.2.5, 1.2.4.2.A, 1.2.6.5, 1.2.6.1.1
	JSON: The data that the database consumes or returns. 1.3.2.1.2, 1.3.2.1.1, 1.3.2.1.5, 1.3.2.1.6, 1.2.4.5.A, 1.2.3.2.C
Referenced By	1 System, 1.1.1.5.1 OrderConfirmation ,1.2.1 AccountManager, 1.2.1.1.A EditAccount, 1.2.2 Authentication, 1.2.2.1 Authentication: New User, 1.2.2.3.A ForgotPassword, 1.2.3.7.6 LockTrip, 1.2.4 PaymentManager, 1.2.4.1 Get Total, 1.3.1.A Controller, 1.2.6 CommunicationManager, 1.2.6.1 ChatroomManager, 1.2.6.2 Conversation Loader, 1.2.6.4 MessageReceiver, 1.2.3.0 TripMatchmake , 1.2.2.3.A ForgotPassword
Viewpoint	Component Diagram

1.2.5.1 AuthenticationQuerier



Name	1.2.5.1 AuthenticationQuerier
Purpose	Shows the ContentQuerying Class
Description	The methods, of the ContentQuerying class.
Requirements	1-10, 18-27
Elements	**RecoverPassword:** Takes in an authUser object, and requests the Firebase SDK send a recovery email to the user.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	Get: Takes in a system authUser object and returns the database version of the same object.
	Delete: Removes the user's authentictaion information from the database.
	update: Modifies the non-null fields of the object passed in.
	Add: Adds the object's information to the database.
Referenced By	1.2.2 Authentication, 1.2.2.1.2 Create User
Viewpoint	Class Diagram

1.2.5.1.1 AddAuthInfo

```
Add(AuthUserObject)

   json = AuthUserObject.ToJson()

    databaseManager(database

      collection(Authentication).Add(json))
```

Name	1.2.5.1.1 AddAuthInfo
Purpose	Showing the process of adding to the authentication database.
Description	The pseudocode that shows how adding to the authentication database will be handled.
Requirements	11 - 17
Elements	**AuthUserObject:** The class object that maps to the Authentication database.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	Collection: The Firebase admin SDK syntax for describing a database.
Referenced By	1.2.5.1 AuthenticationQuerier, 1.2.2.1.2 NewUser
Viewpoint	Pseudocode

1.2.5.1.2 GetAuthInfo

```
Get(AuthUserObject)

   json <- AuthUserObject.ToJson()

   database <- databaseManager(database
collection(Authentication))

   query <- database where(AuthUserId == json[AuthUserId])

   result <- run query

   AuthUserObject.fromJson(result)

   RETURN AuthUserObject
```

Name	1.2.5.1.2 GetAuthInfo
Purpose	Showing the process of retrieving from the authentication database.
Description	Pseudocode that demonstrates how retrieval from the authentication database is handled.
Requirements	11- 17
Elements	**AuthUserObject:** The class object that maps to the Authentication database.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	database: variable that holds a pointer directly to the database.
	Collection: The Firebase admin SDK syntax for describing a database.
	query: The way we are asking the database to find the correct instance for retrieving.
Referenced By	1.2.5.1 AuthenticationQuerying,
Viewpoint	Pseudocode

1.2.5.1.4 DeleteAuthInfo

```
delete(AuthUserObject)

   json <- AuthUserObject.ToJson()

   database <- databaseMangager(database
collection(Authentication))

   query <- database where(AuthUserId == json[AuthUserId])

   result <- run query

   FOR instance IN results

      DELETE instance
```

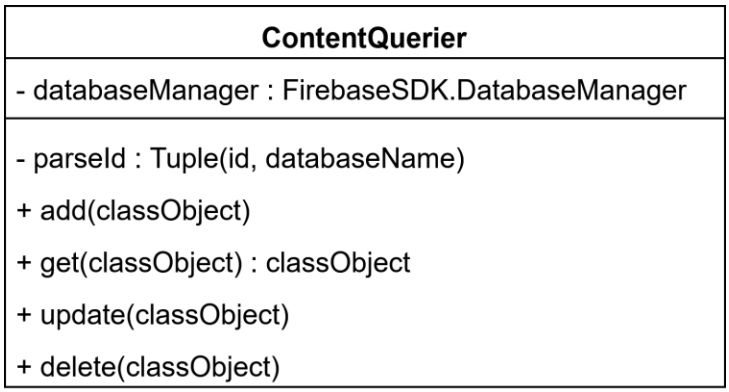
Name	1.2.5.1.4 DeleteAuthInfo
Purpose	Showing the process of deleting from the authentication database.
Description	The pseudocode that shows how deletion from the authentication database will be handled.
Requirements	11-17
Elements	**AuthUser:** The class object that maps to the Authentication database.
	Collection: The Firebase admin SDK syntax for describing a database.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	database: variable that holds a pointer directly to the database.
	instance: An item in results that comes from the JSON file.
	query: The way we are asking the database to find the correct instance for deleting.
Referenced By	1.2.5.1 AuthenticationQuerier
Viewpoint	Pseudocode

1.2.5.1.5 RecoverPassword

```
RecoverPassword(AuthUser)
   authJSON <- AuthUser.ToJSON()
   TRY
      query <- DatabaseManager (
                  database collection("Authentication")
                  where ("username", ==, authJSON["username"])
               )
      result <- query.run()
      link <- DatabaseManager
                  .generatePasswordResetLink(result["email"])
      msg <- {
         to: email,
         from: email for admin in company,
         subject: 'Password Reset',
         text: `Click the link to reset your password:
                   ${link}`,
         html: 'Click the link to reset your password: link`
            };
      send(msg);

   CATCH
      error("error sending reset email")
```

Name	1.2.5.1.5 RecoverPassword
Purpose	Showing the process of account recovery on the database querier end.
Description	The pseudocode involved with recovering an account when a password has been forgotten.
Requirements	17
Elements	**AuthUser:** 1.2.2.4
	AuthJSON: 1.3.2.2.1
	collection["Authentication"]: finding the authentication database.
	generatePasswordRestLink: Method that exists in the firebase admin SDK.
	msg: The necessary components for sending the email containing the reset link.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
Referenced By	1.2.5.1 AuthenticationQuerying, 1.2.2.3.A ForgotPassword
Viewpoint	Pseudocode

1.2.5.2 Content Querier



Name	1.2.5.2 Content Querier
Purpose	Shows the ContentQuerying Class
Description	The methods, of the ContentQuerying class.
Requirements	1-10, 18-27
Elements	**parseId:** Extracts the id and the database name from the id in the JSON key.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	get: Takes in a system class object and returns the database version of the same object.
	deelete: Removes the objects information from the database.
	update: Modifies the non-null fields of the object passed in.
	add: Adds the object's information to the database.
Referenced By	1.2.1.5A Delete Account, 1.2.2.1.1 Validate Enrollment Info, 1.2.2.1.2 Create User, 1.2.3 RideManager, 1.2.6.4 Message Reciever, 1.3.3 MapApi1,
Viewpoint	Class Diagram

1.2.5.2.1 AddContentInfo

```
add(ClassObject)
   json <- ClassObject.ToJson()
   idTuple <- parseId(json)
   DatabaseManager(database collection(idTuple[0].add(json))
```

Name	1.2.5.2.1 AddContentInfo
Purpose	Showing the process of adding to a content database.
Description	The pseudocode that shows how adding to a content database is handled.
Requirements	1 – 10, 18 - 27
Elements	**Class Object:** Class objects that hold information on database entry to be deleted from the database.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	Json: Store the object into a JSON schema to send to the database.
	Collection: The Firebase admin SDK syntax for describing a database.
	idTuple: returned tuple from parseId containing database name and the key for the id.
Referenced By	1.2.5.2 ContentQuerier
Viewpoint	Pseudocode

1.2.5.2.2 GetContentInfo

```
get(ClassObject)
   json <- ClassObject.ToJson()
   idTuple <- parseId(json)
   database <- DatabaseManager(database collection(idTuple[0]))
   query <- database where(idTuple[1] == json[idTuple[1]])
   result <- run query
   NewClassobject <- ClassObject.FromJson(result)
   RETURN NewClassobject
```

Name	1.2.5.2.2 GetContentInfo
Purpose	Showing the process of retrieving from a content database.
Description	The pseudocode shows how retrieval from the content database is handled.
Requirements	1-10, 18-27
Elements	**Class Object:** Class objects that hold information on database entry to be deleted from the database.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	idTuple: Returned tuple from parseId containing database name and the key for the id.
	Collection: The Firebase admin SDK syntax for describing a database.
	database: the database we are finding the instance to retrieve.
	query: The way we are asking the database to find the correct instance for retrieval.
Referenced By	1.2.5.2 ContentQuerier
Viewpoint	Pseudocode

1.2.5.2.3 Update

```
update(ClassObject)
   json <- ClassObject.ToJson()
   idTuple <- parseId(json)
   database <- DatabaseManager(database collection(idTuple[0]))
   query <- database where(idTuple[1] == json[idTuple[1]])
   result <- run query
   FOR instance IN result
      instance.update(json)
```

Name	1.2.5.2.3 Update
Purpose	Showing the process of updating an instance in a content database.
Description	The pseudocode shows how updating in a content database is handled.
Requirements	1-10, 18-27
Elements	**Class Object:** Class objects that hold information on database entry to be deleted from the database.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	idTuple: Returned tuple from parseId containing database name and the key for the id.
	Collection: The Firebase admin SDK syntax for describing a database.
	database: The database we are finding the instance to delete in.
	query: The way we are asking the database to find the correct instance for deleting.
	Instance: A single element from the result.
	Result: A list of data given from the query.
Referenced By	1.2.5.2 ContentQuerier ,
Viewpoint	Pseudocode

1.2.5.2.4 DeleteContentInfo

```
delete(ClassObject)
   json <- ClassObject.ToJson()
   idTuple <- parseId(json)
   database <- DatabaseManager(database collection(idTuple[0]))
   query <- database where(idTuple[1] == json[idTuple[1]])
   result <- run query
   FOR instance IN result
      DELETE instance
```

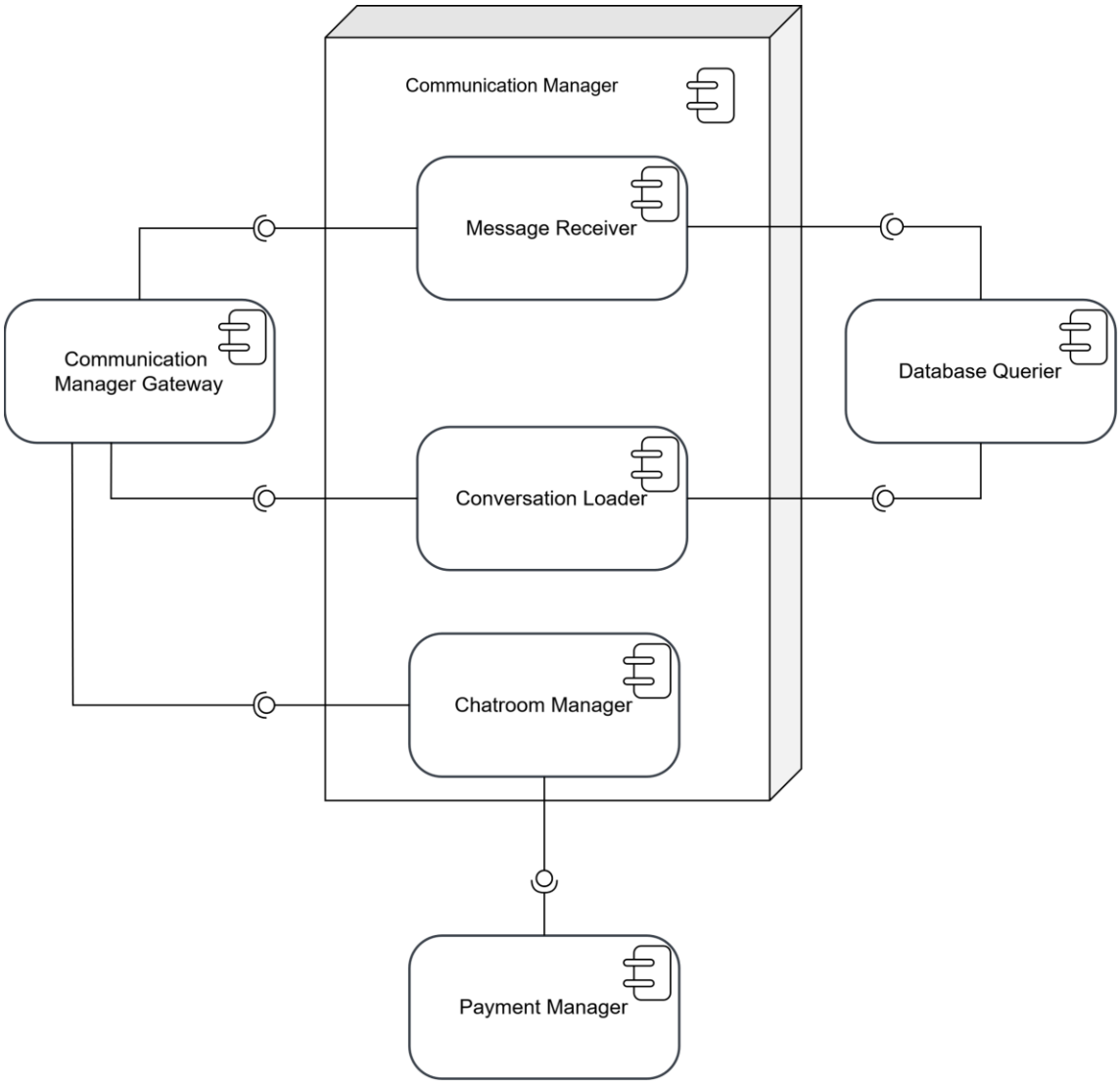
Name	1.2.5.2.4 DeleteContentInfo
Purpose	Showing the process of deleting from a content database.
Description	The pseudocode that shows how deletion from the content database is handled.
Requirements	1-10, 18-27
Elements	**Class Object:** Class objects that hold information on database entry to be deleted from the database.
	databaseManager: Pointer to an object from the FirebaseSDK for managing the connection to the database.
	Json: Store the converted object.
	Collection: The Firebase admin SDK syntax for describing a database.
	idTuple: returned tuple from parseId containing database name and the key for the id.
	database: the database we are finding the instance to delete in.
	Result: A list of data from the query.
	Instance: A single element from result.
	query: The way we are asking the database to find the correct instance for deleting.
Referenced By	1.2.5.2 ContentQuerier
Viewpoint	Pseudocode

1.2.5.2.5 ParseId

```
parseId(JSONObject)
   key <- JSONObject[0]
   splitKey <- key split at "I"
   RETURN tuple(splitKey[0],key)
```

Name	1.2.5.2.5 ParseId
Purpose	Showing the process of getting the database name and object Id.
Description	The pseudocode that parses from JSON the database name and the object Id.
Requirements	1-10, 18-27
Elements	**JSON Object:** The JSON that is being retrieved from one of the content databases**.**
	splitKey: Split the key up to get the name of the needed database.
	I: The key ends at the capital letter "I" so we split at that character.
	key: The key for the Id.
Referenced By	1.2.5.2.2 ContentQuerier
Viewpoint	Pseudocode

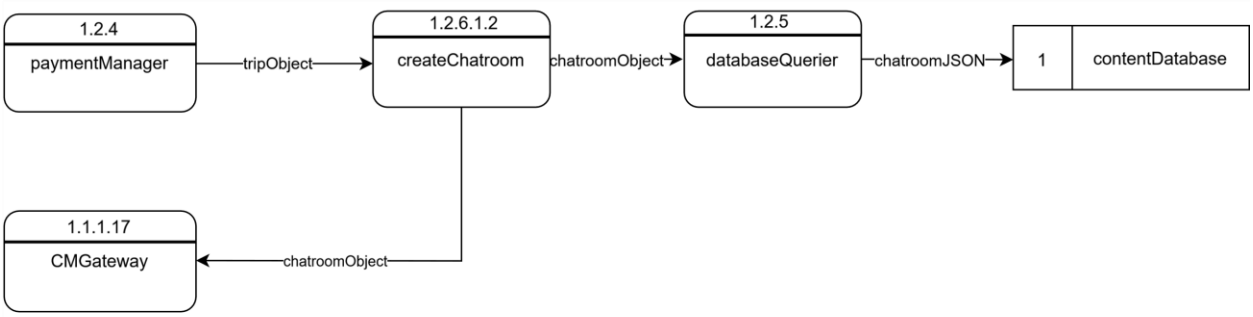
1.2.6 CommunicationManager



Name	1.2.6 CommunicationManager
Purpose	Describe the inner components of CommunicationManager
Description	A representation of the different components within the CommunicationManager, how they connect to each other, and how they connect with the database and the UI controller to facilitate communication between users.
Requirements	23-28
Elements	**CommunicationManager Gateway**: 1.1.1.17
	DatabaseQuerier: 1.2.5
	MessageReceiver: 1.2.6.4
	Conversation Loader: 1.2.6.2

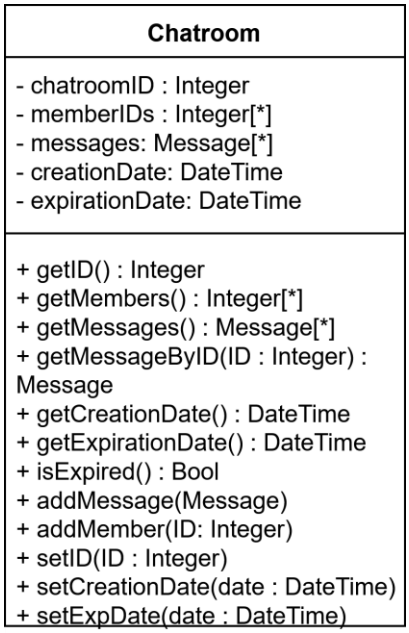
	ChatroomManager: 1.2.6.1
	PaymentManager: 1.2.4
Referenced By	1 System View , 1.1.1.18 ChatUserInterface, 1.2.5 DatabaseQuerier
Viewpoint	Component Diagram

1.2.6.1 ChatroomManager



Name	1.2.6.1 ChatroomManager
Purpose	Demonstrate the Data Flow regarding the creation of a chatroom.
Description	A Data Flow diagram that illustrates the Data Flow involved in creating a chatroom, starting at the PaymentManager and ending at UI.
Requirements	23, 24
Elements	**CommunicationManager (CM) Gateway:** 1.1.1.17
	paymentManager: 1.2.4
	DatabaseQuerier: 1.2.5
	contentDatabase: 1.3.2.1
	createChatroom: 1.2.6.1.2
	tripObject: 1.2.3.7
	chatroomObject: 1.2.6.1.1
	chatroomJSON: 1.3.2.1.5
Referenced By	1.2.6 CommunicationManager
Viewpoint	Data Flow Diagram

1.2.6.1.1 Chatroom



Name	1.2.6.1.1 Chatroom
Purpose	An object to hold all of the necessary information and functionality for a chatroom.
Description	A class diagram for a Chatroom.
Requirements	25, 26
Elements	**chatroomID:** A key unique to the chatroom, meant to identify it among other chatrooms.
	memberIDs: A list of all of the userIDs of the users who are members of the chatroom.
	messages: A list of all the messages sent in the chatroom, as Messages. 1.2.6.5
	creationDate: The date and time for when the chatroom was created.
	expirationDate: The calculated future date and time for when the chatroom expires and is deleted.
	getMessageByID(Id): A method that uses a given messageID number to find and retrieve a message object from the messages list.
	isExpired(): A method that returns true if the expiration date has passed and false if the expiration date is still yet to come.
	addMessage(Message): A method that adds a message object to the list of messages.
	addMember(ID): A method that adds a memberID to the memberIDs list.

Referenced By	1.2.6.1 ChatroomManager, 1.2.6.1.2 createChatroom, 1.2.5 DatabaseQuerier, 1.1.1.15.1 displayChatroom, 1.1.1.15.2 sendMessage, 1.1.1.16.1 transformChatroom
Viewpoint	Class Diagram

1.2.6.1.1.B Chatroom

```
getChatroomByChatroomID(chatroomID)

 READ from chatroomDatabse in contentDatabase by chatroomID as
chatroom

 RETURN chatroom
```

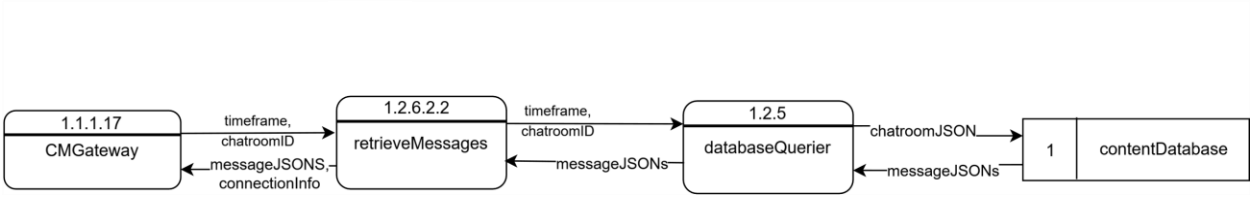
Name	1.2.6.1.1.B Chatroom
Purpose	To demonstrate the query that would be used to retrieve a Chatroom from the Communication Database.
Description	Pseudocode that demonstrates the retrieval of a specific chatroom from the communication database using the chatroom ID.
Requirements	25, 26
Elements	**chatroomID:** The key unique to a chatroom entry that identifies it among other chatrooms.
	chatroomDatabase: The section of the contentDatabase where chatroom information is stored.
	contentDatabase: 1.3.2.1
	chatroom: The variable that holds the Chatroom object retrieved from the database.
Referenced By	1.2.6.5.B Message
Viewpoint	Pseudocode

1.2.6.1.2 createChatroom

```
createChatroom(trip)

   chatroom ← new Chatroom

   chatroom.setID(generateChatroomId())

   FOR rider in trip.riders
      Chatroom.addMember(rider.accountID)

   chatroom.setCreationDate(DateTime.now)

   expirationDate ←  trip.TripTime
   expirationDate.day += 1
   Chatroom.setExpDate(expirationDate)

   RETURN chatroom
```

Name	1.2.6.1.2 createChatroom
Purpose	To create a new chatroom object.
Description	A function that creates a new chatroom object and populates its member variables.
Requirements	25, 26
Elements	**trip:** 1.2.3.7
	Chatroom: 1.2.6.1.1
	chatroom: A variable to hold the new chatroom object.
	generateChatroomId(): A function that returns a unique ID number for the chatroom.
	DateTime.now: A DateTime object of the current time.
	expirationDate: A variable to hold the expiration date DateTime object so that it can be edited before the chatroom expirationDate variable is set to it.
Referenced By	1.2.6.1 Chatroom Manager
Viewpoint	Pseudocode

1.2.6.2 ConversationLoader

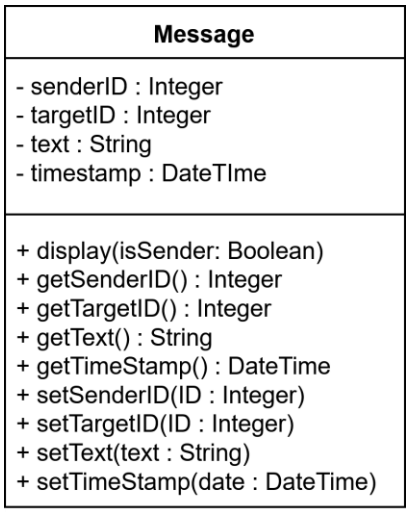


Name	1.2.6.2 ConversationLoader
Purpose	Describe the flow of data through Load Conversations in the CommunicationManager.
Description	Data Flow starts from the user requesting for a conversation to load.
Requirements	26
Elements	**CM Gateway:** 1.1.1.17
	retrieveMessages: Queries the database for all of the messages in the specified chatroom within the given timeframe being loaded.
	DatabaseQuerier: 1.2.5
	timeframe: A range consisting of a start time and end time, to be used to determine which messages to retrieve.
	chatroomID: A key unique to the chatroom, meant to identify it among other chatrooms.
	messageJSON: 1.3.2.1.6
	chatroomJSON: 1.3.2.1.5
	contentDatabase: 1.3.2.1
Referenced By	1.2.6 CommunicationManager, 1.2.6.1 ChatroomManager , 1.1.1.17 CMGateway
Viewpoint	Data Flow Diagram

1.2.6.4 MessageReceiver

Name	1.2.6.4 MessageReceiver
Purpose	Describe flow of data through the MessageReceiver
Description	A representation of the main way that message data travels through the MessageReceiver.
Requirements	23, 25
Elements	**Communication Gateway (CM) Gateway:** 1.1.1.17
	receiveMessage: Function that the controller sends the message information to. It facilitates the necessary database entry and then passes the information to Message Sender.
	createDBEntry: Formats the message information and then sends it to the database. Entry is created for sender and target.
	DatabaseQuerier: 1.2.5
	messageObject: 1.2.6.5
	messageJSON: 1.3.2.1.6
	contentDatabase: 1.3.2.1
Referenced By	1.2.6 CommunicationManager , 1.1.1.17 Communcation Manager CMGateway
Viewpoint	Data Flow Diagram

1.2.6.5 Message



Name	1.2.6.5 Message
Purpose	An object to hold all the necessary information for one message.
Description	A class diagram for a Message.
Requirements	25, 26
Elements	**senderID:** The userID of the one who sends the message.
	targetID: The userID of the one who receives the message.
	text: The text that makes up the message.
	timestamp: The date and time for when the message is sent.
	display: 1.2.6.5.1
Referenced By	1.2.1.3.A Display, 1.2.6.4 CommunicationManager: MessageReceiver, 1.2.6.2 ConversationLoader, 1.2.6.1 ChatroomManager, 1.2.5 DatabaseQuerier, 1.1.1.16.4.3 addMessage,1.1.1.15.2 sendMessage, 1.1.1.15.4 displayMessages, 1.1.1.16.2 transformMessagesToJSON, 1.1.1.16.3 transformMessages
Viewpoint	Class Diagram

1.2.6.5.B Message

```
getMessageByMessageID(messageID, chatroomID)

   chatroom <- getChatroomByID(chatroomID)

   message <- chatroom.getMessageByID(messageID)

   RETURN message
```

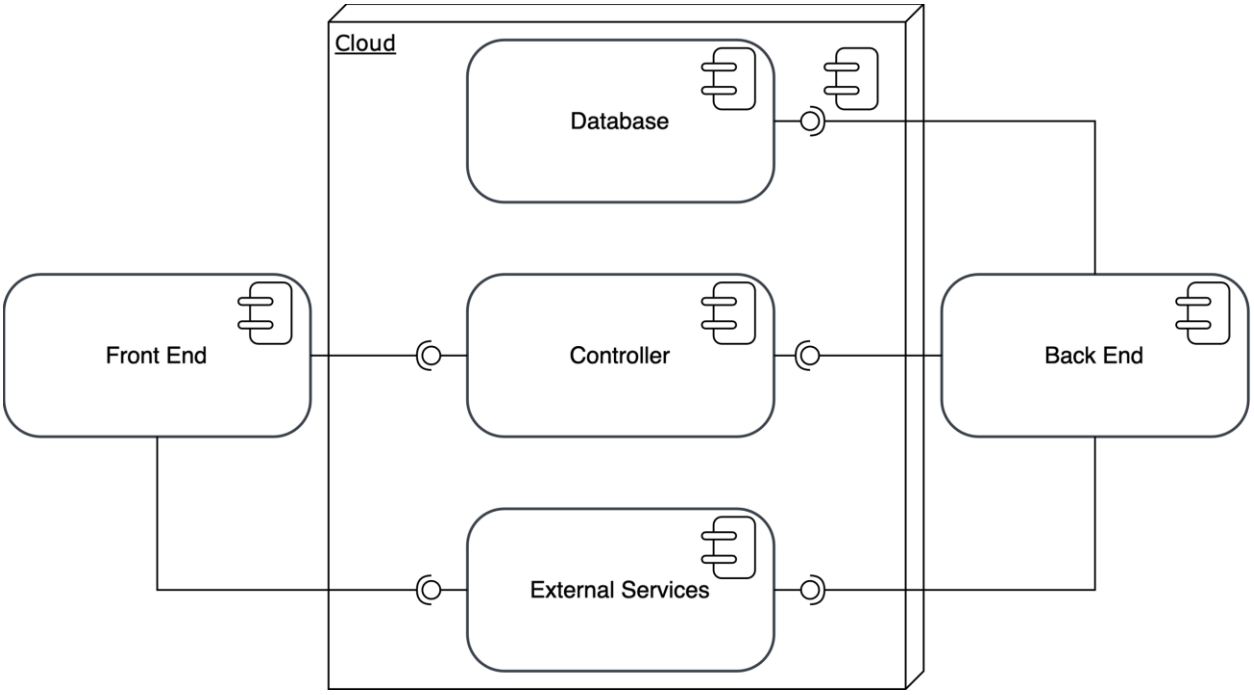
Name	1.2.6.5.B Message
Purpose	To demonstrate the query that would retrieve a Message from the Communication Database.
Description	Pseudocode that demonstrates the retrieval of a specific chatroom from the communication database using the chatroom ID, and then retrieves a specific message from the chatroom database entry using the message ID.
Requirements	25, 26
Elements	**messageID:** The key unique to a message entry that identifies it among other messages.
	chatroomID: The key unique to a chatroom entry that identifies it among other chatrooms.
	getChatroomByID(): 1.2.6.1.1.B Chatroom
	chatroom: The variable that holds the Chatroom object retrieved from the database.
	getMessageByID(): 1.2.6.1.1
	message: The variable that holds the Message object retrieved from the Chatroom.
Referenced By	1.2.5 DatabaseQuerier
Viewpoint	Pseudocode

1.2.6.5.1 Message: display

```
Message::display(isSender)

   IF isSender
      PUT senderBubble on screen
   ELSE
      PUT recieverBubble on screen

   PUT text on chatbubble on screen
   PUT timestamp on chatbubble on screen
```

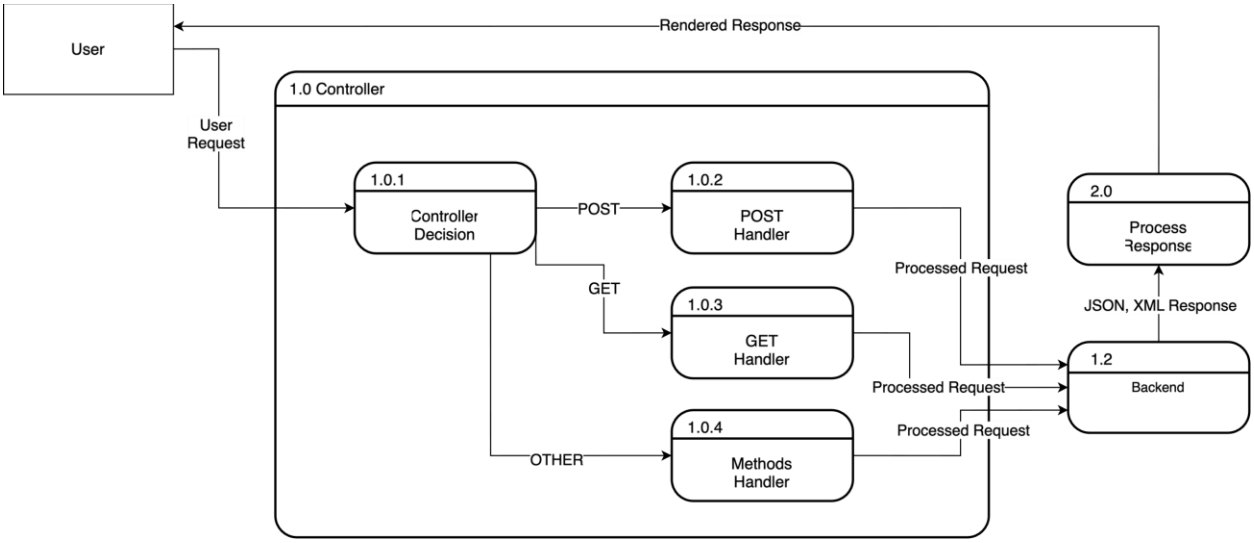
Name	1.2.6.5.1 Message: display
Purpose	To display a message on the user screen.
Description	A function that displays a message object.
Requirements	24-26
Elements	**isSender:** A boolean value that is true if the message is being displayed to the person who sent it and is false if the message is being displayed to someone receiving the message.
	chatbubble: The small square that holds the message in the chatroom.
	senderBubble: A chatbubble formatted to show it was sent by the user whose screen is receiving the display.
	recieverBubble: A chatbubble formatted to show it was received by the user whose screen is receiving the display.
Referenced By	1.2.6.5 Message Object, 1.1.1.15.4 Chat Screen: displayMessages
Viewpoint	Pseudocode

1.3 Cloud



Name	1.3 Cloud
Purpose	This Component Diagram is intended to illustrate the relationships in the Cloud.
Description	The Cloud Component Diagram is designed to demonstrate what goes on between the Frontend and the Back End. It includes all processes which occur within the Cloud.
Requirements	1-27
Elements	**Frontend:** 1.1
	Back End: 1.2
	Database: 1.3.2
	Controller: 1.3.1
	External Services: All leveraged services from outside the system, i.e. APIs.
Referenced By	1 System, 1.1 Frontend
Viewpoint	Component Diagram

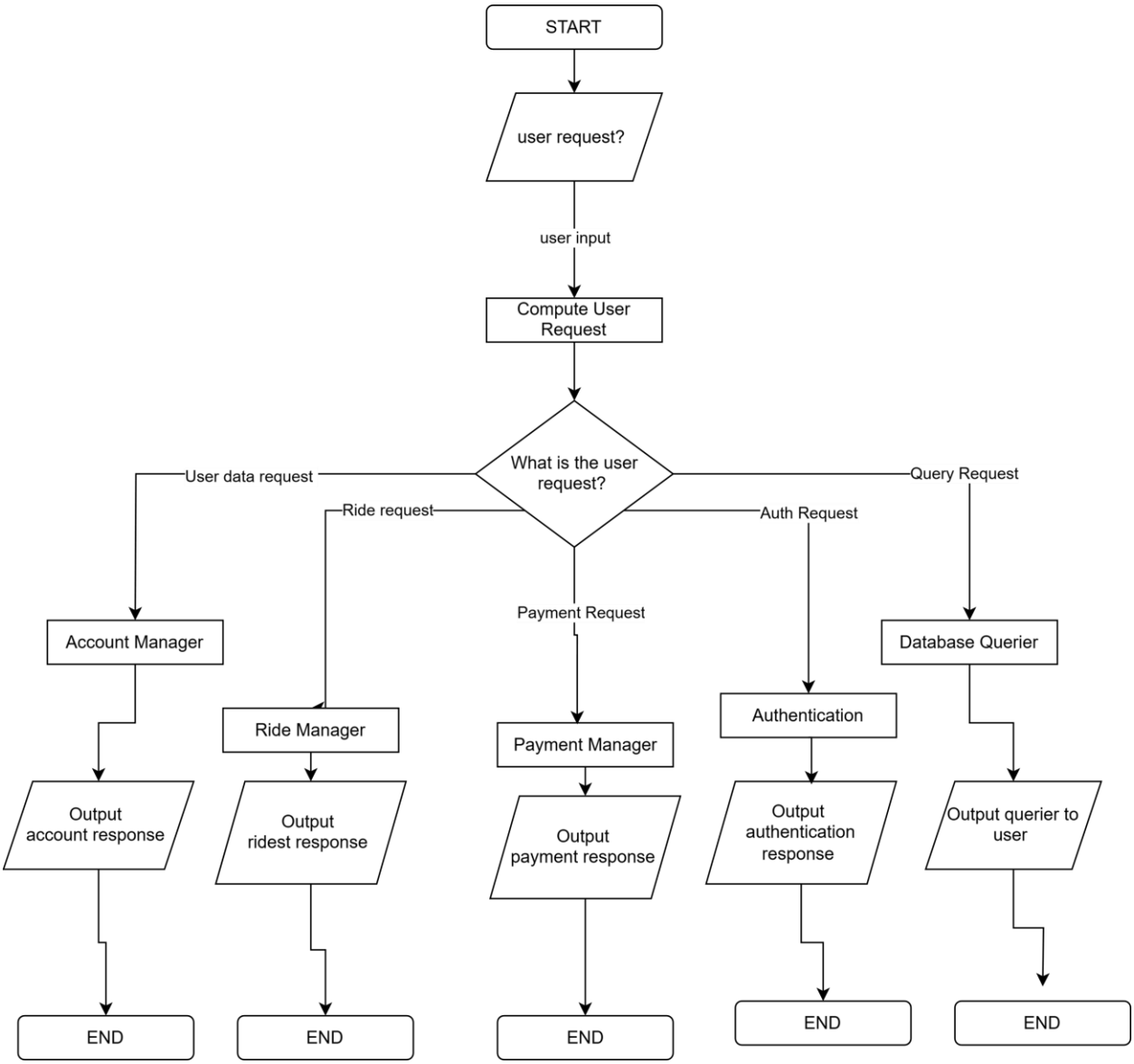
1.3.1 Controller



Name	1.3.1 Controller
Purpose	This DFD represents how data flows between the user, controller, response, and backend components in the controller
Description	The diagram illustrates a typical web application request-response cycle. It shows how user requests are processed through a controller, handled by a process response component, and communicated with the backend.
Requirements	User Interface, Backend Interface and Data formatting.
Elements	**User:** The user interface component that starts the request.
	Controller Subsystem: The controller subsystem.
	Controller Decision: Handles and routes the user request.
	POSTHandler: Handles the post requests when necessary.
	GETHandler: 1.3.1.3
	Methods Handler: Handles other types of requests when necessary.
	Backend: 1.2
	User Request: This is the input request from the user.
	POST: This is the flow HTTP post request.
	GET: This is the flow HTTP get request
	Other: This is the flow just in case there is another type of request rather that post or get request.
	Processed Data: The type of request already processed to the backend.
	JSON, XML Response: Response given by the backend according to the request given.
	Rendered Response: The response the user is going to received

Referenced By	1 System, 1.2.1 AccountManager, 1.2.1.1.A EditAccount, 1.2.2 Authentication 1.2.2.1 NewUser, 1.2.2.2Returning User, 1.2.2.3.A ForgotPassword, 1.3 Cloud
Viewpoint	DFD

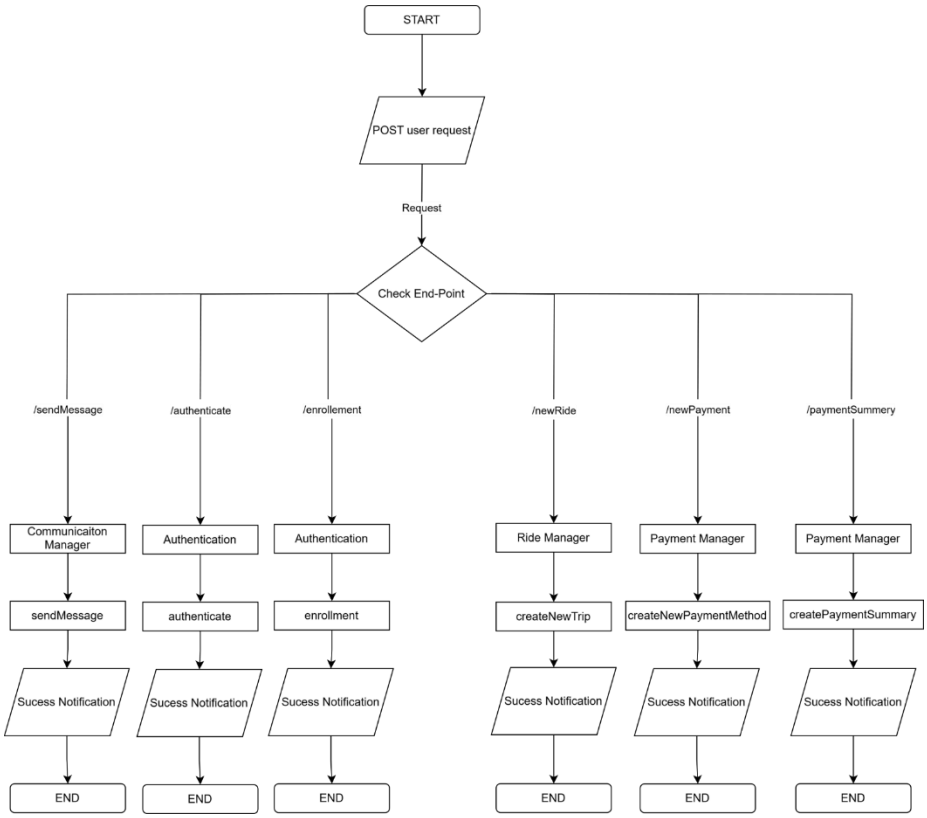
1.3.1.A Controller



Name	1.3.1.A Controller
Purpose	This flowchart represents how the controller will make the decisions based in the user's request
Description	The flowchart shows a request handling system that processes different types of user request. Starts from the user input, then the controller evaluates the request type and directs it to the different components in the backend. Each path has its own dedicated manager that handles the specific request type before terminating.
Requirements	User Interface 1.1.1 and Backend Interface 1.2
Elements	**Start:** Entry Point of the system
	Input: Asks the person for the type of request

	Compute User Request: Process the user request
	Decision Diamond (Controller): Evaluates request type and routes to appropriate handler
	AccountManager: 1.2.1
	RideManager: 1.2.3
	PaymentManager: 1.2.4
	Authentication: 1.2.2
	DatabaseQuerier: 1.2.5
	Output: Outputs the request to the user
	End: Termination points for different processes paths
Referenced By	1 System, 1.1.1.2 PaymentUserInterface
Viewpoint	Flowchart

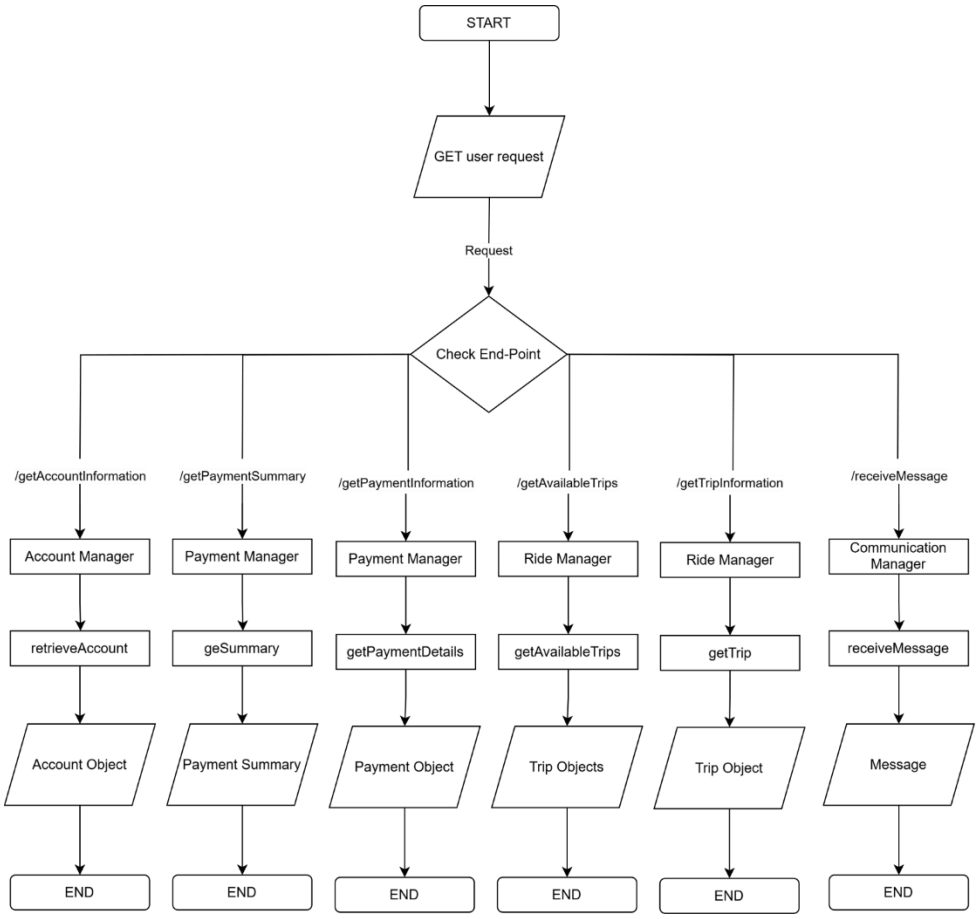
1.3.1.2 POSTHandler



Name	1.3.1.2 POSTHandler
Purpose	Shows the decision process of the POSTHandler.
Description	This view shows part of the decision process of the controller on where requests are sent based on the endpoint.
Requirements	1-22
Elements	**POST User Request:** A request from the user that requires a POST call.
	Check End-Point: The decision point to help direct the request to the correct place.
	Authentication: 1.2.2 Manages all authentication and begins the enrollment process.
	PaymentManager: 1.2.4 Manages all backend interaction to do with payments.
	RideManager: 1.2.3 Manages all backend interaction to do with rides.
	CommunicationManager: Manages the messaging process.
	sendMessage: The function that starts the process to create a payment summary.
	Authenticate: The function that handles the process of authenticating a user.

	Enrollment: The function that starts the process of enrollment as a NewUser.
	CreateNewTrip: The function that handles the process of creating a new ride.
	CreateNewPaymentMethod: The function that starts the process of creating a new payment method attached to the logged in user.
	CreatePaymentSummary: The function that handles the creation of a payment summary.
Referenced By	Controller 1.3.1
Viewpoint	Flow Chart

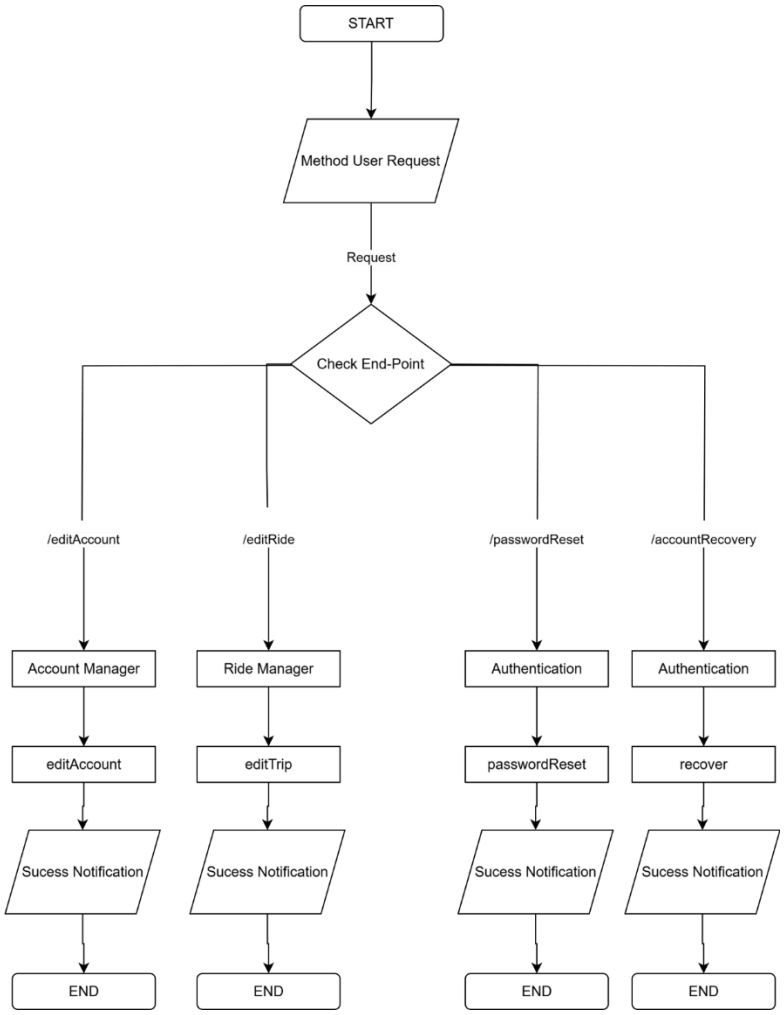
1.3.1.3 GETHandler



Name	1.3.1.3 GETHandler
Purpose	Shows the decision process of the GETHandler.
Description	This view shows part of the decision process of the controller on where requests are sent based on the endpoint.
Requirements	1-22
Elements	**GET User Request:** A request from the user that requires a GET call.
	Check End-Point: The decision point to help direct the request to the correct place.
	AccountManager: Manages all backend interaction to do with accounts.
	PaymentManager: 1.2.4 Manages all backend interaction to do with payments.
	RideManager: 1.2.3 Manages all backend interaction to do with rides.
	CommunicationManager: Manages the messaging process.
	ReceiveMessage: The function that handles the receiving of a message from another user.
	RetrieveAccount: The function that starts the process of retrieving the account from the database.

	GetPaymentDetails: The function that starts the process of retrieving the payment information from the database.
	GetAvailableTrips: The function that starts the process of retrieving all currently available rides from the database.
	GetTrip: The function that starts the process of retrieving the details of a particular ride from the database.
	GetSummary: The function that handles the process of getting the payment summary from the database.
	Payment Summary: An object that contains the payment summary details.
	Account Object: An object that holds account details.
	TripObject: An object that holds trip details.
	Payment Object: An object that holds payment details.
Referenced By	1.3.1 Controller, 1.2.2.3.A ForgotPassword
Viewpoint	Flow Chart

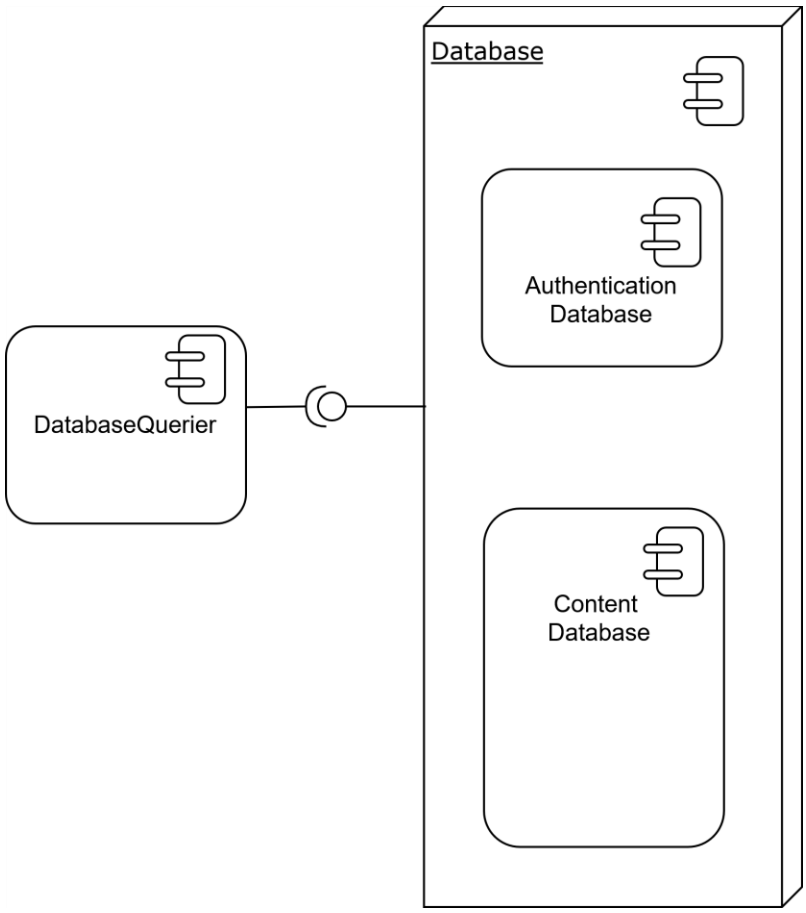
1.3.1.4 MethodHandler



Name	1.3.1.4 MethodHandler
Purpose	Shows the decision process of the MethodHandler.
Description	This view shows part of the decision process of the controller on where requests are sent based on the endpoint.
Requirements	1-22
Elements	**Method User Request:** A request from the user that requires a call of neither GET or POST.
	Check End-Point: The decision point to help direct the request to the correct place.
	AccountManager: 1.2.1
	RideManager: 1.2.3
	Authentication: 1.2.2
	EditAccount: Starts the process of sending an edit to an account in the database.

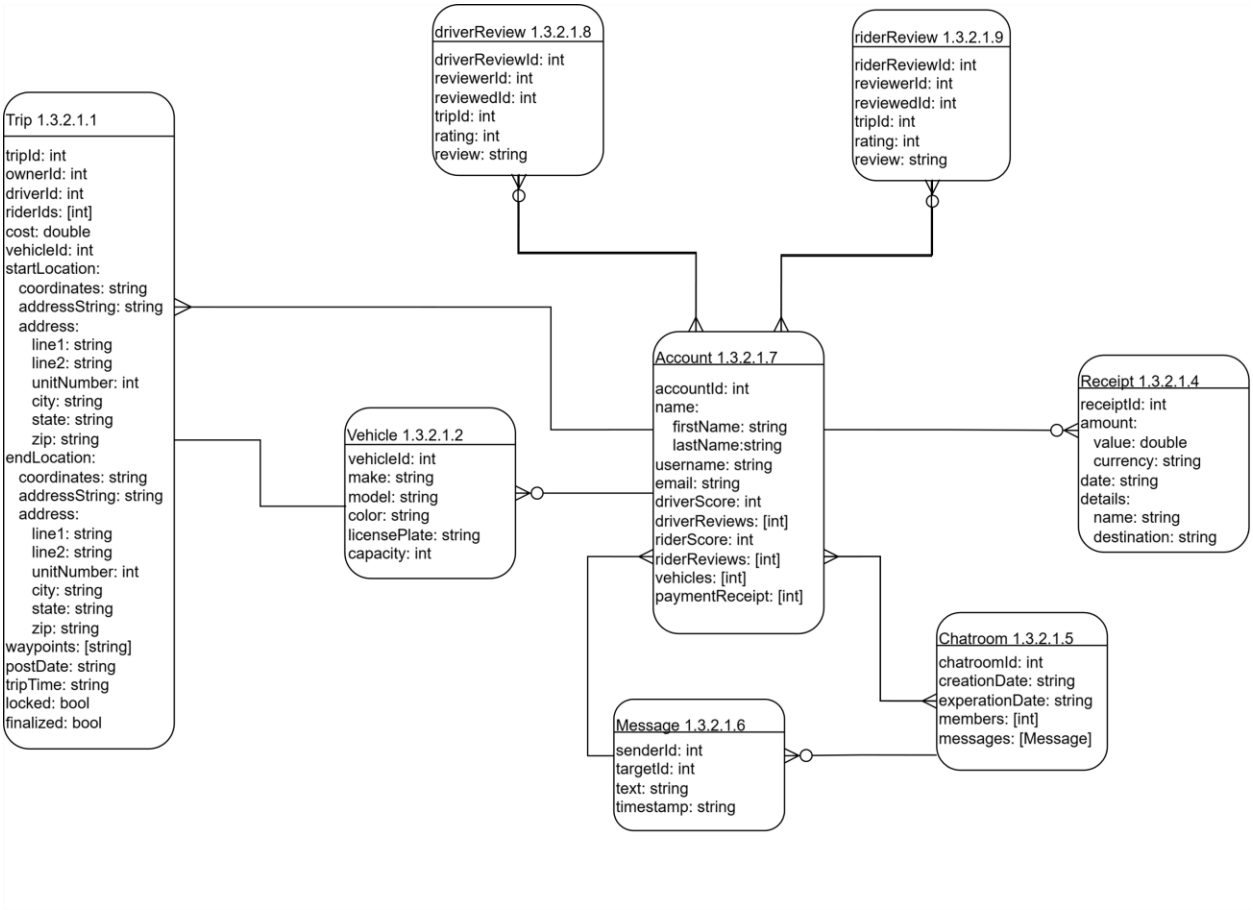
	EditTrip: Starts the process of sending an edit to a trip in the database.
	PasswordReset: Starts the process of resetting an existing user's password.
	Recover: Handles the process of taking the user through account recovery.
Referenced By	1.3.1 Controller
Viewpoint	Flow Chart

1.3.2 Database



Name	1.3.2 Database
Purpose	A view of the database.
Description	The place where data comes to rest.
Requirements	1-27
Elements	**Authentication Database**: 1.3.2.2
	ContentDatabase: 1.3.2.1
	Database Querier: 1.2.5
Referenced By	1 System, 1.2.5 DatabaseQuerier, 1.3 Cloud
Viewpoint	Component Diagram

1.3.2.1 Content Database



Name	1.3.2.1 Content Database
Purpose	An overview of the relationships between the document databases in the content database.
Description	The information that is being held in each of the sub-databases and how they connect to each other.
Requirements	1-10,18-27
Elements	**Trip Database:** 1.3.2.1.1
	Vehicle Database: 1.3.2.1.2
	Chatroom Database: 1.3.2.1.5
	Message Database: 1.3.2.1.6
	Account Database: 1.2.3.1.7
	Receipt Database: 1.3.2.1.4
	driverReview Database: 1.3.2.1.8
	riderReview Database: 1.3.2.1.9
	TripId: a number that uniquely identifies a trip.
	OwnerId: a number that uniquely identifies an account pointing to the owner.

	DriverId: a number that uniquely identifies an account pointing to the person who is driving.
	rideIds: a list number that uniquely identifies accounts that point to the riders.
	VehicleId: a number that uniquely identifies a vehicle.
	senderId: a number that uniquely identifies an account pointing to the person who is sending the message.
	targetId: a number that uniquely identifies an account pointing to the person who is receiving the message.
	driverReviewId: a number that uniquely identifies a driver review.
	riderReviewId: a number that uniquely identifies a rider review.
	reviewerId: a number that uniquely identifies an account pointing to the person who sent the review.
	reviewedId: a number that uniquely identifies an account pointing to the person who was reviewed.
	chatRoomId: a number that uniquely identifies a chat room.
	ReceiptId: a number that uniquely identifies a receipt.
	Waypoints: a list of stopping locations during the trip.
	DriverScore: the overall score as a driver.
	RiderScore: the overall score as a rider.
	DriverReviews: list of ids pointing to the reviews of a driver.
	RiderReviews: list of ids pointing to the reviews of a rider.
	Vehicles: a list number that uniquely identifies vehicles that point to the account's vehicles.
	paymentReceipts: a list number that uniquely identifies receipts that point to the account's receipts.
	members: a list number that uniquely identifies accounts that point to the members of a chat room.
	messages: a list of the JSON structure message.
Referenced By	1.3.2 Database, 1.1.1.15.1 displayChatroom, 1.2.1.1.A EditAccount, 1.2.6.1.1.B Chatroom
Viewpoint	ERD

1.3.2.1.1 Trip

```
{
 "tripId": int,
 "ownerId": int,
 "driverId": int,
 "riderIds": [int],
 "cost": float,
 "vehicleId": int,
 "startLocation": {
 "coordinates": "string",
"addressString": "string",
"address": {
 "line1": "string",
 "line2": "string",
 "city": "string",
 "state": "string",
 "zip": "string"
}
 }
 "endLocation": {
 "coordinates": "string",
 "addressString": "string",
"address": {
 "line1": "string",
 "line2": "string",
 "city": "string",
 "state": "string",
 "zip": "string"
}
 }
 "waypoints": ["string"],
 "postDate": "string",
 "tripTime": "string",
 "locked": bool,
 "finalized": bool
}
```

Name	1.3.2.1.1 Trip
Purpose	This diagram defines the structure of the JSON representing the trip information needed to construct a map.
Description	This diagram depicts how information will be formatted so that a map can be embedded into the user interface which references the coordinates for the start and end of a trip.
Requirements	4
Elements	**tripId:** A unique identifier for the trip.
	ownerId: The user Id for the user which owns the trip.
	driverId: The user Id for the user which is driving the trip.
	riderIds: The collection of user Ids for the users which are riding on the trip.
	cost: The dollar amount a rider would need to pay for the trip.
	vehicleId: the vehicle Id for the vehicle being driven.
	capacity: The number of riders the vehicle can carry on a trip.
	startLocation: The place the driver will be starting from.
	endLocation: The place the driver will arrive at.
	coordinates: A string representing an exact spot on the earth.
	addressString: A string representing a location, formatted as “Number - Street Name, City, State ZIP-Code". Example: “111 Home Dr, Rexburg, Idaho 83440”.
	address: A set of information which describes a certain location.
	line1: Information in the first line of an address, including the house number and street name.
	line2: Additional address information such as unit or apartment number.
	city: The Nameof the city on the address.
	state: The Nameof the state on the address.
	zip: The zip code of the address as a string.
	waypoints: A stop or multiple stops added to a trip, positioned between the start and end locations, where riders can exit or be dropped off along the route. 1.3.3.1.1 TripLocations
	postDate: The date and time when the trip was posted, encoded into a string.
	tripTime: The date and time when the trip is scheduled to start, encoded into a string.
	locked: The state of the trip being locked or not. When locked, trip is read only.
	finalized: The state of the trip being finalized or not.

Referenced By	1.1.1.6 RideManagementService, 1.1.1.9 MapInterfaceService, 1.2.3.0 TripMatchmake, 1.2.3.3 CreateTripInstance, 1.2.3.5.1 QueryEditTrips, 1.2.3.7.1 AddDriver, 1.2.3.7.2 RemoveDriver, 1.2.3.7.3 AddRider, 1.2.3.7.4 RemoveRider, 1.2.3.7.5 Trip Object: CancelTrip, 1.2.3.7.6 Trip Object: LockTrip, 1.2.3.2 AcceptTrip, 1.2.3.5 Ride Manger: EditTrip, 1.3.2.1 ContentDatabase, 1.3.3 MapAPI
Viewpoint	JSON Schema

1.3.2.1.2 Vehicle

```
{
 “vehicleId”: int,
 “make”: “string”,
 “model”: “string”,
 “color”: “string”,
 “licensePlate”: “string”,
 “capacity”: int,
}
```

Name	1.3.2.1.2 Vehicle
Purpose	This diagram defines the structure of the JSON representing a vehicle used for rides.
Description	This diagram depicts how information will be formatted to contain all necessary information about a vehicle in JSON.
Requirements	4
Elements	**make:** The car manufacturer or brand.
	vehicleId: Unique number used to identify vehicle and be references by other JSONs
	model: The specific vehicle Namewithin the brand.
	color: The overall color of the vehicle.
	licensePlate: The number which can identify the specific vehicle.
	capacity: The number of riders the vehicle can carry on a trip.
Referenced By	1.3.2 Database , 1.3.2.1 ContentDatabase
Viewpoint	JSON Schema

1.3.2.1.4 Receipt

```
{
   "ReceiptID":"int"
   "amount"{
      "value": "double"
      "currency": "string"
   },
   "date": "YEAR-MONTH-DAY"
   "details": {
      "name": "string"
      "destination": "string"
   }
}
```

Name	**1.3.2.1.4** Receipt
Purpose	Authorizes and shows capture payment details.
Description	The receipt is generated after the trip has been fulfilled. It contains Elementsfrom the ride object and the payment object that provides information about their trip and payment.
Requirements	9
Elements	**ReceiptID:** The ID of the order.
	date: The date and time when the order was placed.
	details: Specifics regarding order (e.g. Nameof payee, destination.)
	destination: The end location of the trip.
	name: Full Nameof the payee
	amount: Object with the total and currency of the order.
Referenced By	1.2.4.2 Receipt , 1.2.4.2.3 createReceiptJSON, 1.3.2.1 ContentDatabase, 1.1.1.2.1.3 transformReceipt, 1.1.1.2.1.4 sendReceipt
Viewpoint	JSON Schema

1.3.2.1.5 Chatroom

```
{
  "chatroomID": "int",
  "creationDate": "YEAR-MONTH-DAY",
  "expirationDate": "YEAR-MONTH-DAY",
  "members": {
    "type": "array",
    "items": {
      "type": "integer"
    }
  },
  "messages": {
    "type": "array",
    "items": {
      "type": "object"
    }
  }
}
```

Name	1.3.2.1.5 Chatroom
Purpose	JSON schema used when creating a chatroom database entry.
Description	The JSON schema for a Chatroom.
Requirements	25, 26
Elements	**chatroomID:** A key unique to the chatroom meant to identify it among chatrooms.
	creationDate: The date and time for when the chatroom was created.
	expirationDate: The calculated future date and time for when the chatroom expires and is deleted.
	members: A list of userIDs for the members of the chatroom.
	messages: 1.3.2.1.6
Referenced By	1.2.6.1 ChatroomManager, 1.2.6.2 ConversationLoader, 1.2.5. DatabaseQuerier, 1.3.2.1 ContentDatabase, 1.1.1.16.1 transformChatroom, 1.1.1.16.3 transformMessages
Viewpoint	JSON Schema

1.3.2.1.6 Message

```
{
  "senderID": "integer",
  "targetID": "integer",
  "text": "string",
  "timestamp": {
    "date": "YEAR-MONTH-DAY"
    "hour": "integer",
    "minute": "integer"
  }
}
```

Name	1.3.2.1.6 Message
Purpose	JSON schema to be used when creating a message database entry.
Description	The JSON schema for a Message.
Requirements	25, 26
Elements	**senderID:** The userID of the one who sends the message.
	targetID: The userID of the one who receives the message.
	text: The text that makes up the message.
	timestamp: The date and time for when the message was sent.
	date: A sting representing the date part of the timestamp date.
	hour: An integer value representing the hour part of the timestamp time.
	minute: An integer value representing the minute part of the timestamp time.
Referenced By	1.1.1.16.3 transformMessages, 1.1.1.17.5 loadMessages, 1.2.6.4 CommunicationManager: MessageReceiver, 1.2.6.2 ConversationLoader, 1.3.2.1.5 Chatroom, 1.2.5 DatabaseQuerier, 1.3.2.1 ContentDatabase
Viewpoint	JSON Schema

1.3.2.1.7 Account

```
{
 “accountID”: int,
 “name”: {
     “firstName”: string,
     “lastName”: string
 },
 “username”: string,
 “email”: string,
 “driverScore”: float,
 “driverReviews”: [int],
 “riderScore”: float,
 “riderReviews”: [int],
 “vehicles": [int],
 “address”: int,
 “paymentReceipt”: [int]
}
```

Name	1.3.2.1.7 Account
Purpose	Shows format of how data for account is stored in the database.
Description	JSON schema of account and everything in it.
Requirements	12, 16-18
Elements	**accountID:** Unique number associated with an account.
	firstName: First Name of the user.
	lastName: Last Name of the user.
	username: userName of the user.
	email: Email used to send notifications to the user.
	driverScore: Number based average from driverReview to help others make choices.
	driverReview: List of driver review ID of a driver from the riders of their trips.
	RiderScore: Number based on the average from riderReview to help others make choices.
	riderReview: List of rider review IDs of a rider from everyone on the trip.
	vehicles: List of vehicle IDs about the vehicles of the user.
	address: Address ID of the user.
	PaymentReceipt: List of payment receipts IDs from paypal
Referenced By	1.2.1 AccountManager, 1.2.1.1.A EditAccount, 1.2.1.3 A Display, 1.2.2 Authentication , 1.2.2.1 Authentication: New User, 1.3.2.1 ContentDatabase
Viewpoint	JSON Schema

1.3.2.1.8 driverReview

```
{
    "reviewId":"int",
    "reviewerId":"int",
    "reviewedId":"int",
    "tripId":"int",
    "rating":"float",
    "review":"string"
}
```

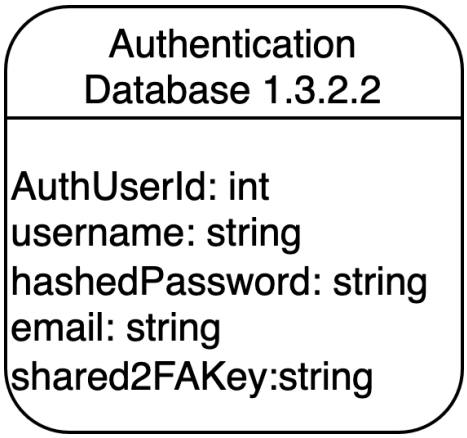
Name	1.3.2.1.8 driverReview
Purpose	Shows everything related to a review of a driver
Description	The variables, and relations of diverReview class.
Requirements	18, 21, 22
Elements	**reviewID:** Unique number associated with a driver review.
	reviewerID: ID of the account making the review.
	reviewedID: ID of the account being reviewed.
	tripID: ID of the trip that the reviewer and reviewed shared.
	rating: Score out of five.
	review: Description or reason for the rating.
Referenced By	1.3.2.1.7 Account, 1.3.2.1.1 Trip
Viewpoint	JSON Schema

1.3.2.1.9 riderReview

```
{
    "reviewId":"int",
    "reviewerId":"int",
    "reviewedId":"int",
    "tripId":"int",
    "rating":"float",
    "review":"string"
}
```

Name	1.3.2.1.9 riderReview
Purpose	Shows everything related to a review of a rider
Description	The values stored in the rider review database.
Requirements	18, 21, 22
Elements	**reviewID:** Unique number associated with a rider review.
	reviewerID: ID of the account making the review.
	reviewedID: ID of the account being reviewed.
	tripID: ID of the trip that the reviewer and reviewed shared.
	rating: Score out of five.
	review: Description or reason for the rating.
Referenced By	1.3.2.1.7 Account, 1.3.2.1.1 Trip
Viewpoint	JSON Schema

1.3.2.2 Authentication Database



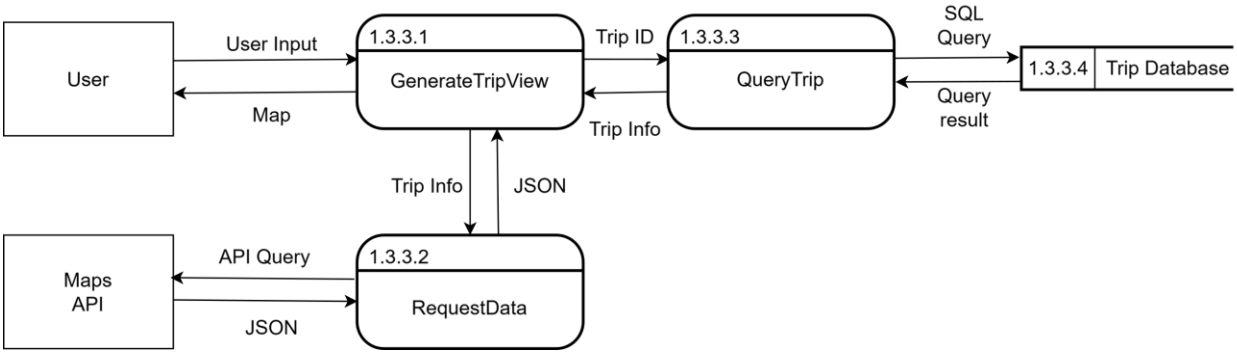
Name	1.3.2.2 Authentication Database
Purpose	Defines the tables and relationships of Authentication Database.
Description	The data stored in the authentication database.
Requirements	12, 13, 14, 15
Elements	**AuthUserId:** A primary key for this authUser.
	username: the username of the authUser.
	hashedPassword: the user's password that has been hashed.
	email: the user's recovery email.
	shared2FAKey: the key shared between the user and the system. Used to define 2FACode.
Referenced By	1.3.2 Database, 1.2.5.1 AuthenticationQuerieir
Viewpoint	ERD

1.3.2.2.1 Auth User Json

```
{
   "AuthUserId" : "Int",
   "username" : "String",
   "hashedPassword" : "String",
   "email" : "String",
   "shared2FAKey" : "String"
}
```

Name	1.3.2.2.1 Auth User Json
Purpose	Defines the structure of the AuthUser table in the Authentication Database.
Description	The flow of data for authenticating a return user.
Requirements	12
Elements	**AuthUserId:** A unique id for this authUser.
	username: the username of the auth user
	hashedPassword: the user's password that has been hashed.
	email: the user's recovery email
	shared2FAKey: the key shared between the user and the system. Used to define 2FACode.
Referenced By	1.3.2.2 Authentication Database
Viewpoint	Json Schema

1.3.3 MapAPI



Name	1.3.3 MapAPI
Purpose	Describe the use of the MapAPI and map generation.
Description	Generate a trip view that show a potential route the driver will take with their starting location, planned stops, and final destination.
Requirements	2, 4, 20
Elements	**GenerateTripView**: 1.3.3.1
	RequestData: 1.3.3.2
	QueryTrip: Makes a call to the trip Database to get the requested trips information .
	Trip Database: Contains data about each trip.
	MapAPI: 3rd party MapAPI Google Maps API Documentation
	User input: The trip the user wants to view.
	Trip ID: A numerical value used to query the trip.
	SQL Query: SQL used to access info from the database.
	Query Result: All information about the trip, which is: driver, riders, trip time, start location, end location, stops, locked, and finalized.
	Trip Info: see 1.3.2.1.1
	API Query: Request for data from the MapAPI using trip info.
	JSON: The return from the MapAPI Contains all the information in the style that we can use to generate a map.
	Map: The map view.
Referenced By	1 System., 1.1.1 UserInterface, 1.2.3.0 TripMatchmake, 1.2.3.3 CreateTripInstance
Viewpoint	Data Flow Diagram

1.3.3.1 GenerateTripView

```
generateTripView (tripID
   tripLocationsJSON = queryTrip(tripID)
   map = requestData(trip)
   PUT map
```

Name	1.3.3.1 GenerateTripView
Purpose	Describe the use of the MapAPI and map generation
Description	Generate a trip view that shows a potential route the driver will take with their starting location, planned stops, and final destination.
Requirements	5
Elements	**TripID:** A unique numeric value given to each trip.
	requestData: 1.3.3.2 RequestData
	queryTrip: 1.3.3.3_queryDatabase
	tripLocationsJSON: 1.3.3.1.1
	Map: The map view.
Referenced By	1.3.3 MapAPI
Viewpoint	Pseudocode

1.3.3.1.1 TripLocations

```
{
 “startLocation”: {
 “coordinates”: “string”,
 “addressString”: “string”,
 “address”: {
 “line1": “string”,
 “line2”: “string”,
 “city”: “string”,
 “state”: “string”,
 Courier New ”zip”: “string”
 }
 “waypoint”: {
 “coordinates”: “string”,
 “addressString”: “string”,
 “address”: {
 “line1": “string”,
 “line2”: “string”,
 “city”: “string”,
 “state”: “string”,
 “zip”: “string”
 }
 “endLocation”: {
 “coordinates”: “string”,
 “addressString”: “string”,
 “address”: {
 “line1": “string”,
 “line2”: “string”,
 “city”: “string”,
 “state”: “string”,
 “zip”: “string”
 }
}
```

Name	1.3.3.1.1 TripLocations
Purpose	This diagram defines the structure of the JSON representing information needed to construct a map.
Description	This diagram depicts how information will be formatted so that a trip can be saved into the database.

Requirements	4
Elements	**startLocation:** The place the driver will be starting from.
	coordinates: A string representing an exact spot on the earth.
	addressString: A string representation of a place in the format “111 Home Dr, Rexburg, Idaho 83440”.
	address: A set of information which describes a certain location.
	line1: Information in the first line of an address, including the house number and street name.
	line2: Additional address information such as unit or apartment number.
	city: The Nameof the city on the address.
	state: The Nameof the state on the address.
	zip: The zip code of the address as a string.
	waypoint: A stop or multiple stops added to a trip, positioned between the start and end locations, where riders can exit or be dropped off along the route.
	endLocation: The place the driver will arrive at.
Referenced By	1.3.3 MapAPI, 1.3.3.1 GenerateTripView, 1.3.3.2 RequestData
Viewpoint	JSON Schema

1.3.3.2 RequestData

```
requestData (trip info)
   READ Google Maps using trip info as map.JSON
   RETURN map.JSON
```

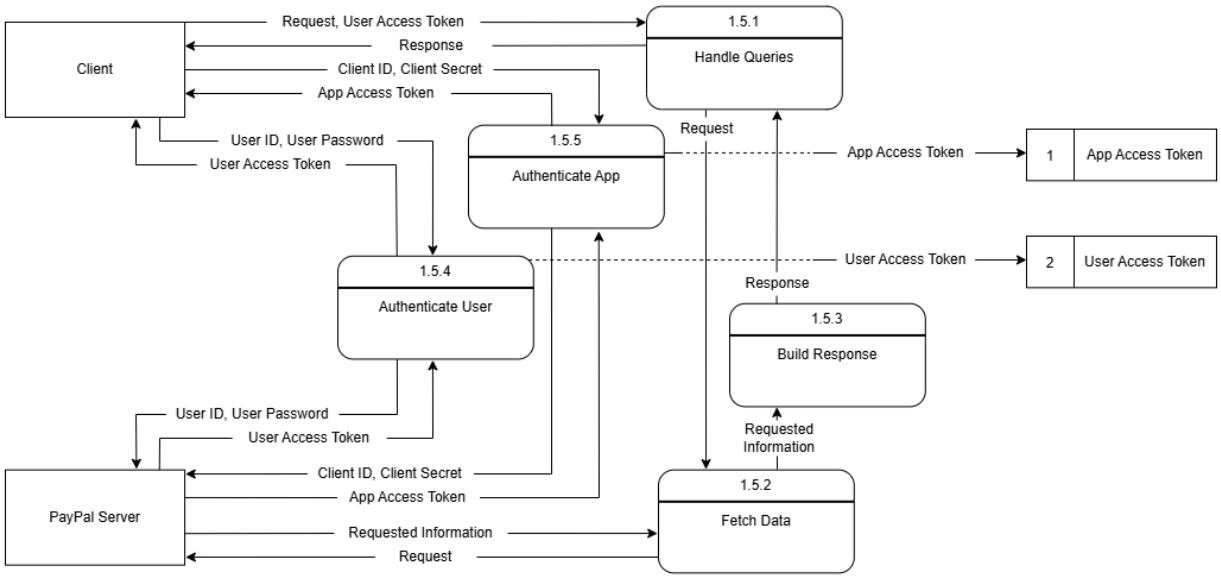
Name	1.3.3.2 RequestData
Purpose	Queries the MapAPI for a selected trip
Description	Makes an API call and returns the JSON data
Requirements	20
Elements	**Trip Info**: all relevant information required for the MapAPI, which is trip time, start location, end location, and stops. See Trip Object: 1.2.3.7_Trip_Object
	Map.JSON: The return from the MapAPI Contains all the information in the style that we can use to generate a map
	Google Maps: 3rd party MapAPI Google Maps Documentation
Referenced By	1.3.3 MapAPI , 1.3.3.1 GenerateTripView
Viewpoint	Psuedocode

1.3.3.3 QueryTrip

```
queryDatabase(tripID)
   READ from tripDatabase by tripID as trip
   RETURN trip
```

Name	1.3.3.3 QueryTrip
Purpose	Retrieve a trip from the trip database.
Description	Queries a trip from the trip database through its tripID.
Requirements	4, 5
Elements	**TripID:** A unique numeric value given to each trip.
	Trip Database: The database in which all trips are stored.
	Trip: JSON data containing all the trips information
Referenced By	1.3.3_Map_API, 1.3.3.1 GenerateTripView
Viewpoint	Pseudocode

1.3.4 PayPalAPI



Name	1.3.4 PayPalAPI
Purpose	The System can send Payments through PayPal
Description	How the Client converses with the PayPalAPI to access services and information hosted by PayPal .
Requirements	9-12, 14
Elements	**Client**: The Rideshare Program.
	Client ID: Element used by PayPal to identify an app.
	Client Secret: Element used by PayPal to authenticate a Client ID.
	User ID: ID provided by the Client for a single user.
	User Password: Password submitted by the Client provided by a user.
	PayPal Server: Server and services hosted by PayPal, including the PayPal Database. PayPalAPI Documentation.
	Request: A query from the Client that seeks the exchange of information with the PayPal Server.
	Response: An answer to the Request from the Client.
	Requested Information: Whatever information the Client wants from the PayPal Server, including payment information.
	App Access Token: Token used to prove the Client can access the PayPal Server.
	User Access Token: Token used to prove a certain User from the Client can access the PayPal Server.
	Handle Queries: Receives the Request and User Access Token from the Client, builds a Response from the Requested Information sent by Fetch Data, and sends the Response back to the Client.

	Fetch Data: Processes the Request if the User Access Token is valid and sends the Requested Information from the PayPal Server to Build Response.
	Build Response: Creates a Response from the Requested Information for Handle Queries to send back to the Client
	Authenticate App: Trades a valid Client ID and Client Secret with the PayPal Server for an App Access Token.
	Authenticate User: Trades a valid User ID and User Password from the Client with the PayPal Server for a User Access Token .
Referenced By	1 System, 1.2.4 PaymentManager
Viewpoint	Data Flow Diagram