Smart Car System Design

Brett Bloethner | CSCI E-97

Table of Contents

1. Smart Car System Overview

1.1 Introduction

This section provides a high level overview of the design of the Smart Car System. This section will provide some detail on specific components included in the Smart Car System. However, each component used in the Smart Car System includes its own design document section that should come included in the Smart Car System Design document you're reading right now.

1.2 Overview

The Smart Car System is designed to provide connected functionality to a fleet of vehicles in a way that would allow vehicles to autonomously provide rides to customers on demand. To do this, the Smart Car System implementation must include features to control individual vehicle hardware as well as controlling and monitoring the vehicles remotely as a fleet and in a single centralized tool. Additionally, end users (customers/riders) must be able to interface with the Smart Car System in order to request service without having to contact a central dispatch center like comparable taxi cab services may require.

1.3 High-level Requirements

The Smart Car System includes features designed to help maintain all aspects of a smart car system business model. Users (or Riders), administrator and vehicles all play a role in the proper functioning of the Smart Car System. Below is a list of all of the required functionality of this Smart Car System. Some of these features may be executed multiple actors in the Smart Car System. For example, a RiderProfile may be created and updated by both Riders and Administrators.

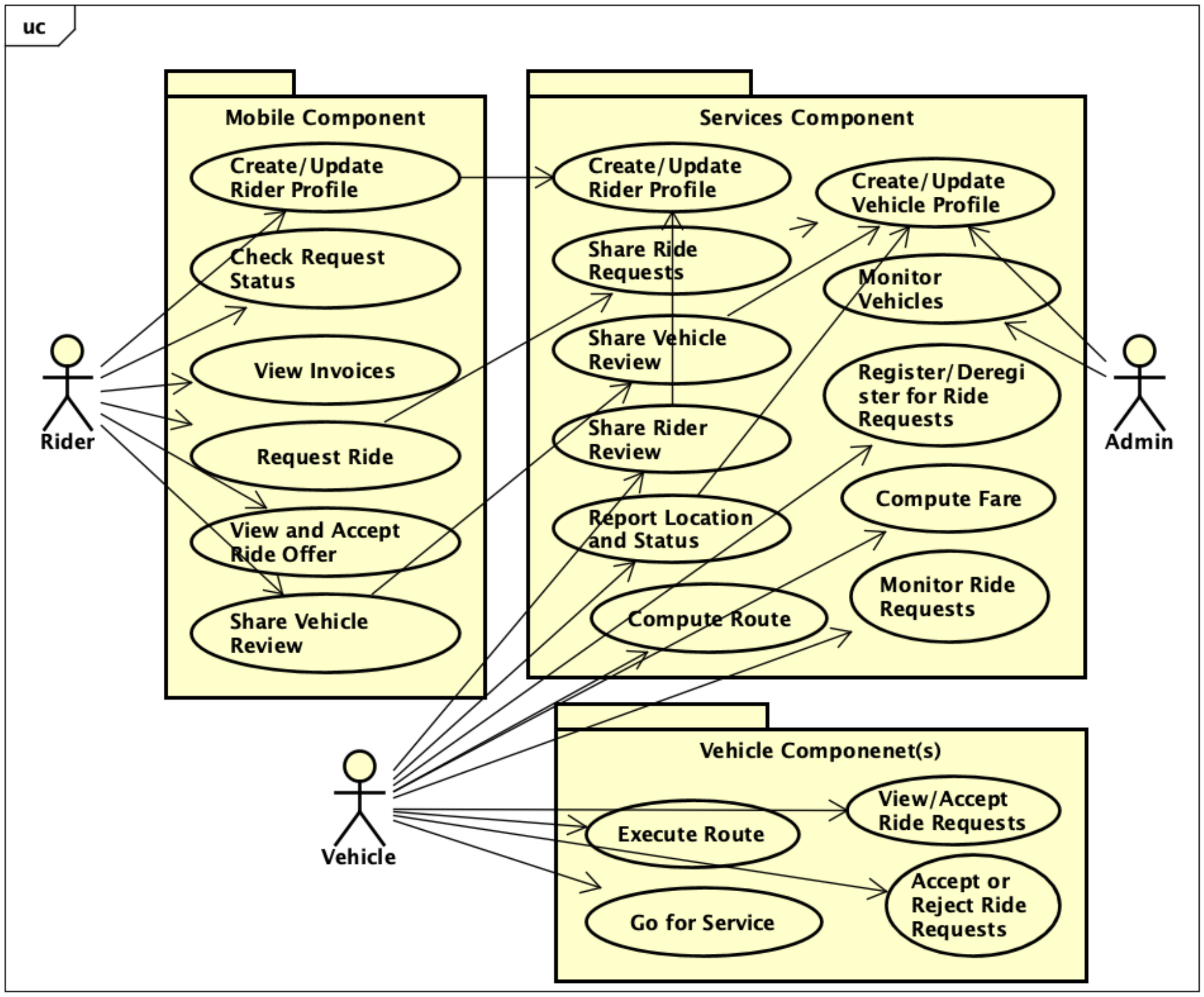
Creation of RiderProfiles	Updating of RiderProfiles	Creation of RideRequests
Monitoring of RideRequests	Accepting of RideRequests	Viewing of invoices
Reviewing of Vehicles	Reviewing of RiderProfiles	Sharing of RiderProfile reviews
Sharing of Vehicle reviews	Sharing of RideRequests	Monitoring of Vehicles
Viewing a Vehicles status	Execution of RideRequests	Computation of RideRequest routes
Computation of RideRequest Fares	Monitoring for new RideRequests	Vehicle self service

Smart Car System Design Requirements

1.4 High-Level Use Cases

Information about the use cases

Smart Car System Use Case Diagram

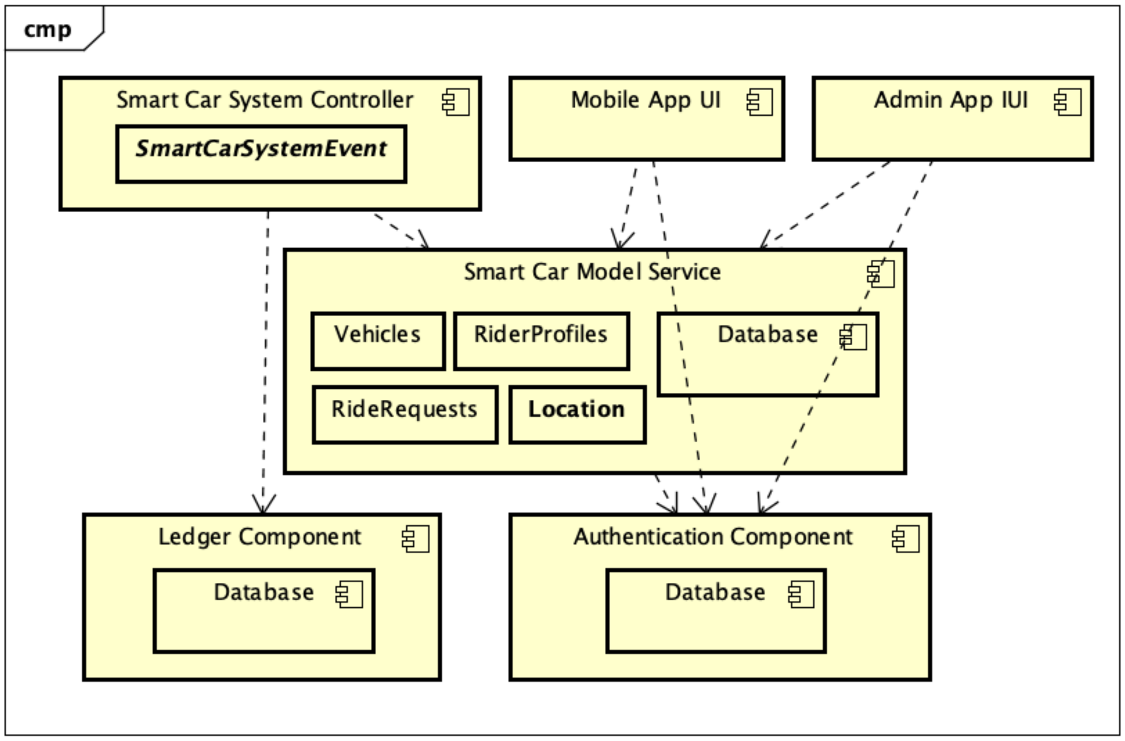


The above diagram outlines the use cases for each of the important pieces of the Smart Car System. The Administrator, the Rider and the Vehicle.

1.5 Components

The Smart Car System integrates a handful of components in order to provide full functionality to its users. Five components are interfaced together to create the Smart Car system, each of which is described in detail below.

Smart Car System Component Diagram



The above diagram outlines the components that make up the Smart Car System and important parts of each component as well as how they may depend on one another.

Smart Car System Controller

The Smart Car System Controller is responsible for all of the real time side effects of actions taken in the Smart Car Model Service. The observer pattern is used in order to let the Controller detect when important events happen in the Model Service so that the Controller and execute the appropriate next steps, like dispatching a car. Security is implemented in the Controller by working with the Authentication Service and the Command Pattern is also implemented in a way similar to what was in the Smart City System.

Mobile App UI

The mobile application component is responsible for providing a secure interface between the Smart Car System and the customer/rider. The Model Service is capable of performing actions well beyond the scope of what customers are provisioned to do, so the Mobile App UI acts as the gateway that customers can use in order to issue the commands they need to while still protecting the Model Service component by not implementing any features beyond only what a customer should be able to do. The Mobile App UI component also implements an easy to user graphical interface for the customer to take advantage of. Authentication for riders and customers in the Mobile App UI component is done by using the Authentication Component and an HTTP proxy is implemented using the Integration Broker pattern.

Admin App UI

The Admin App UI component is responsible for all of the interactions that an Administrator actor might have with the Smart Car System. In order to do this, the Admin App UI component must interface with the Authentication and Smart Car Model Service components. Many of the actions performed in the

Admin App UI Component will be executed by calling one or more features of the Model Service component while verification that that the user has Administrator level permissions will be conducted using features in the Authentication Component. The Administration Component also implements an easy to use graphical interface for the Administrator to take advantage of, similar to the Mobile App UI but in a desktop interface rather than a mobile interface and also with elevated privileges. An HTTP proxy is implemented using the Integration Broker pattern here as well.

Smart Car Model Service

The Smart Car Model Service is responsible for much of the core operations of the Smart Car System. The Model Service manages all object representations of things involved in the Smart Car System including Vehicles, Riders, Administrators, Reviews, Problem Reports and RideRequests. Standard CRUD operations are used in the Model Service to manage many of these objects and the Model Service implements security by interfacing with the Authentication Service. An implementation of the observable pattern is used to interact with the Smart Car System Controller while an implementation of the Integration Broker pattern is used to interface other components of the Smart Car System.

Ledger Component

The Ledger Component is the same component that was created in Assignment 2. This component is responsible for keeping track of the blockchain money balances of each of the RiderProfile and Vehicle blockchain accounts in the Smart Car system. Transactions are also created, executed and verified via the Ledger Component and the Authentication Component plays an instrumental role in making sure the identities on each side of the transaction are valid and able to properly conduct the transaction. All of the transactions in the Smart Car System originate as a result of actions conducted in the Smart Car System Controller that were a result of events detected in the Smart Car Model Service.

Authentication Component

The Authentication Component is the same component that was created in Assignment 4. It's responsible for all of the authentication and authorization of rider and administrators accounts. All features in the Smart Car System that require elevated privileges of any kind must verify the caller has the appropriate authorization to execute the feature. This is done by using an ephemeral AuthToken that is tied to an authenticated rider or administrator account and is issued once that valid account is logged in.

1.6 Component Implementation

All six components are implemented with the help of a few important design patters. The Integration Broker pattern is key in allowing the components to communicate with one another, especially when it comes to the Mobile GUI and Admin GUI components communicating with other components via HTTP. The Command pattern is used in the Controller Service and the Observable pattern is used across the Model and Command services which allows them to communicate effectively without being coupled which would result in circular dependency.

1.7 Data Persistance

The Smart Car System incorporate three databases. The Smart Car Model Service should utilize an SQL database and the Hibernate library in order to store the Vehicle, Rider, RideRequest, Review and ProblemReport entities its responsible for. The Authentication Service should also incorporate an SQL database and Hibernate in order to store user account authentication data as well as permissions and authentication tokens. The Ledger service should be backed by a secure immutable ledger persistence layer with by using SQL, Hibernate and clever hashing mechanisms (similar to the current design) or by using an all in one blockchain solution like HyperLedger and implementing the HyperLedger API into the Ledger Service in place of the current manual immutable ledger implementation.

1.8 High-level Risks

I think the biggest high-level of risk of this design could occur in regards to separation of concerns. Right now, the Smart Car System application is relatively simple in regards to features. However, as the application grows, more and more complex features could be implemented which could result in unmanageable growth in the two main components that make up much of the systems operations which are the Smart Car Model Service and the Smart Car System Controller. I see it as a real possibility that if this application we to become more feature rich then there might have to be more specialized components that would share the feature set of what is now the Smart City Model Service.

2. Graphical User Interface

2.1 Introduction

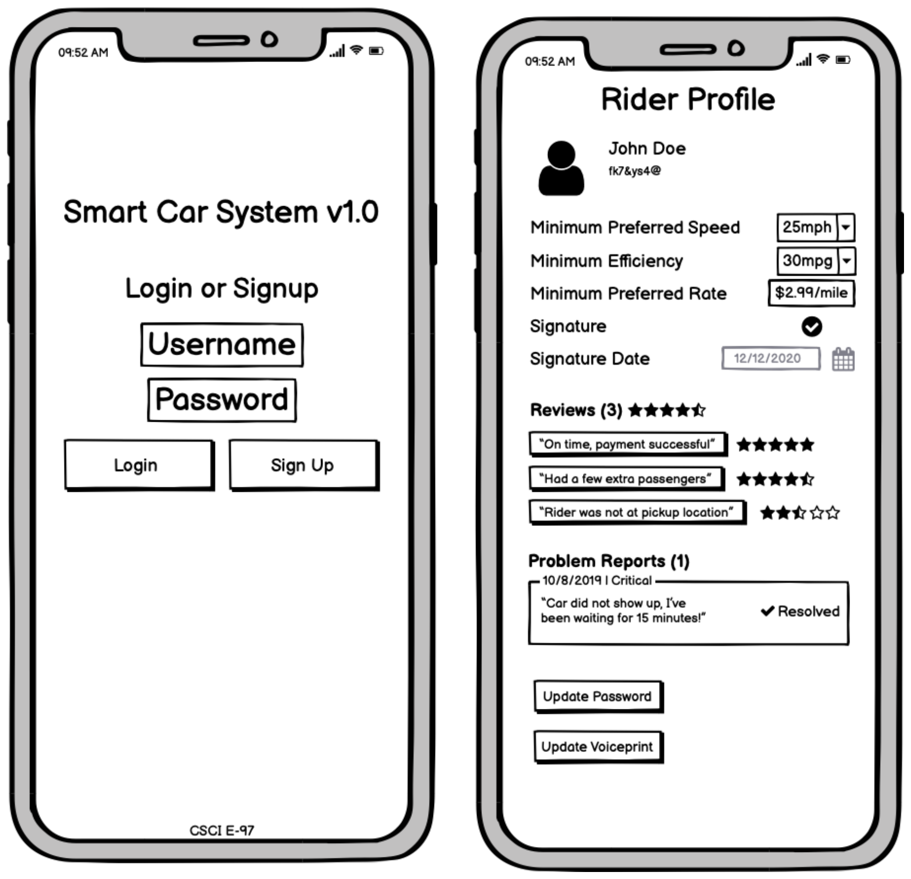
There are two graphical user interfaces in the Smart Car System. The first is the Mobile GUI. This user interface is designed to be used by end users or “Riders.” Rider need a few core features which are editing the RiderProfile and creating RideRequests. The other graphical user interface belongs to administrators and their requirements extend what Riders need. An admins should have complete control of the Smart Car System with the ability to monitor vehicles and riders from a high level as well as view and change Vehicles, RiderProfiles and RideRequests.

2.2 Customer Mobile Application

The rider facing Mobile GUI is responsible for letting the end user request rides from the Smart Car System. The Mobile GUI allows the Rider to create a RideRequest, view and change their Rider Profile, create Vehicle reviews and review previous ride invoices. This is all protected by an authentication entry point that requires the Rider to login or sign up before they can use any of the other features of the Mobile GUI.

2.2.1 Login, Signup and Rider Profile

Upon opening the Mobile Application for the first time, the rider is presented with a Login/Signup screen. If the rider is already signed up then they can login in, if they’re not then they can conveniently sign up on the same screen. Logged in riders can update their profiles anytime but riders who just signed up will be required to fill out their RiderProfile immediately after signing up. The RiderProfile screen on the left is used to view and update the riders RiderProfile and is presented to the user right after the initial signup.



John Doe
fk7&ys4@

Minimum Preferred Speed 25mph
Minimum Efficiency 30mpg
Minimum Preferred Rate $2.99/mile
Signature ✓
Signature Date 12/12/2020

Reviews (3) ★★★★☆
"On time, payment successful" ★★★★★
"Had a few extra passengers" ★★★★☆
"Rider was not at pickup location" ★★☆☆☆

Problem Reports (1)
10/8/2019 | Critical
"Car did not show up, I've been waiting for 15 minutes!" ✓ Resolved

The login/signup and RiderProfile pages provide an interface for managing rider preferences

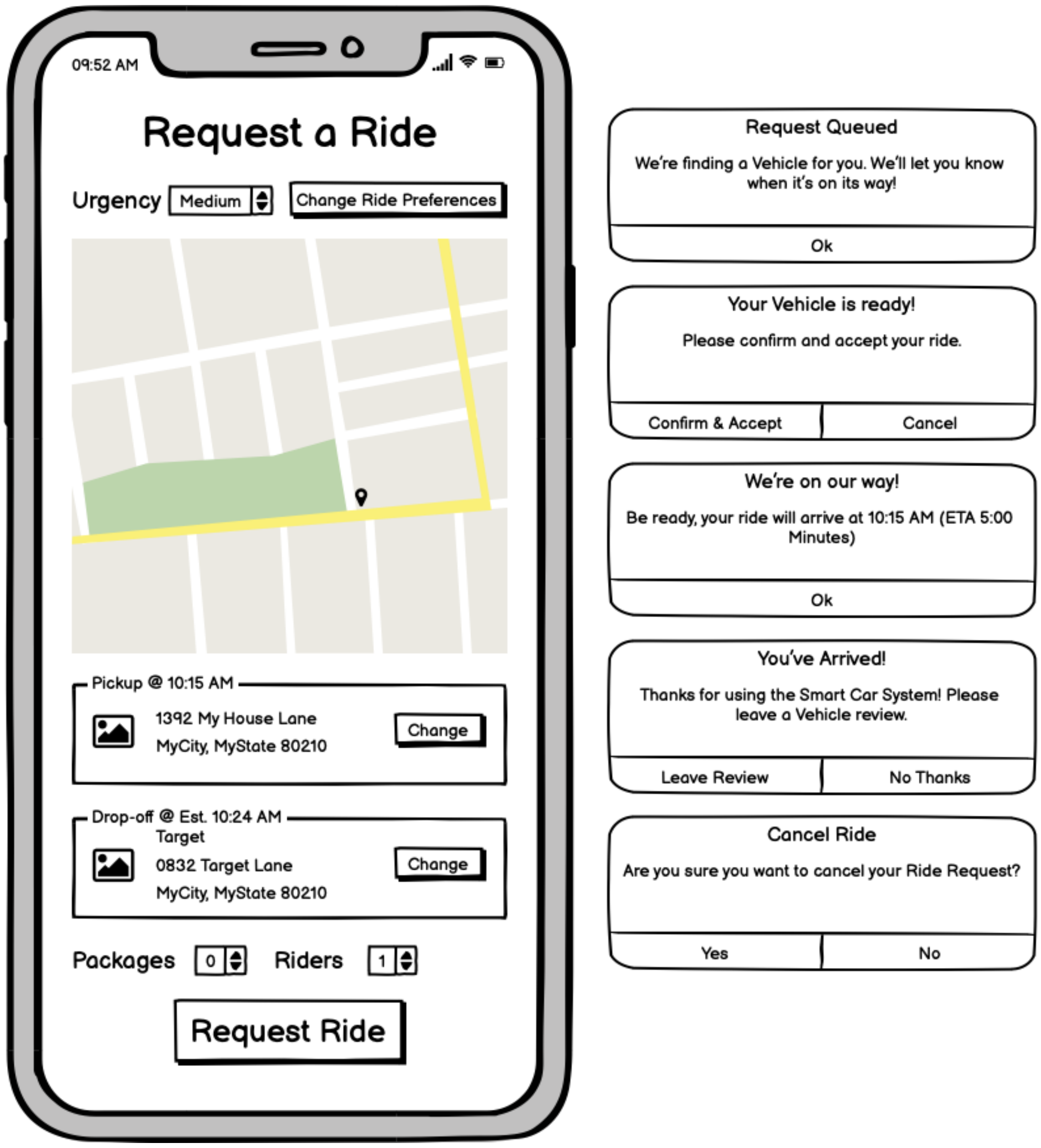
Login, Signup and RiderProfile Endpoints

Verb	Endpoint	Description
POST	/login	used to login a rider and get their issued auth token
POST	/signup	used to signup/create a rider and get their issued auth token
GET	/riderProfiles/{riderProfileId}	used to get single RiderProfile by ID
PUT	/riderProfiles/{riderProfileId}	used to update a RiderProfile by RiderProfile ID

2.2.2 Request a Ride

RideRequests are made on the below interface. The user can change their pickup and drop off destinations as well as their pickup time by clicking the "change" button in the location boxes at the bottom of the screen. Package and Rider count as well as Urgency can be changed by using simple dropdown menus. The Rider can change their ride preferences (like preferred trim and efficiency levels) by clicking the "Change Ride Preferences" button which will take them to their Rider Profile page.

Once a ride is requested, the Rider will get a notification that the request has been queued. Once a Vehicle is ready, the Rider receives an "offer" alert where they can "Confirm & Accept" the ride or cancel the ride. After they accept the ride, they are notified when the Vehicle is on its way. Once the ride is complete, the Rider is prompted with an alert that allows them to leave a review by redirecting to the review page.



Urgency: Medium

Pickup @ 10:15 AM
1392 My House Lane
MyCity, MyState 80210

Drop-off @ Est. 10:24 AM
Target
0832 Target Lane
MyCity, MyState 80210

Packages: 0 Riders: 1

Request Queued
We're finding a Vehicle for you. We'll let you know when it's on its way!

Your Vehicle is ready!
Please confirm and accept your ride.

We're on our way!
Be ready, your ride will arrive at 10:15 AM (ETA 5:00 Minutes)

You've Arrived!
Thanks for using the Smart Car System! Please leave a Vehicle review.

Cancel Ride
Are you sure you want to cancel your Ride Request?

The RideRequest screen provides a series of alerts to help the Rider understand what the status of their RideRequest is. After a RideRequest is created the "Request Ride" button turns into a "Cancel Request" button

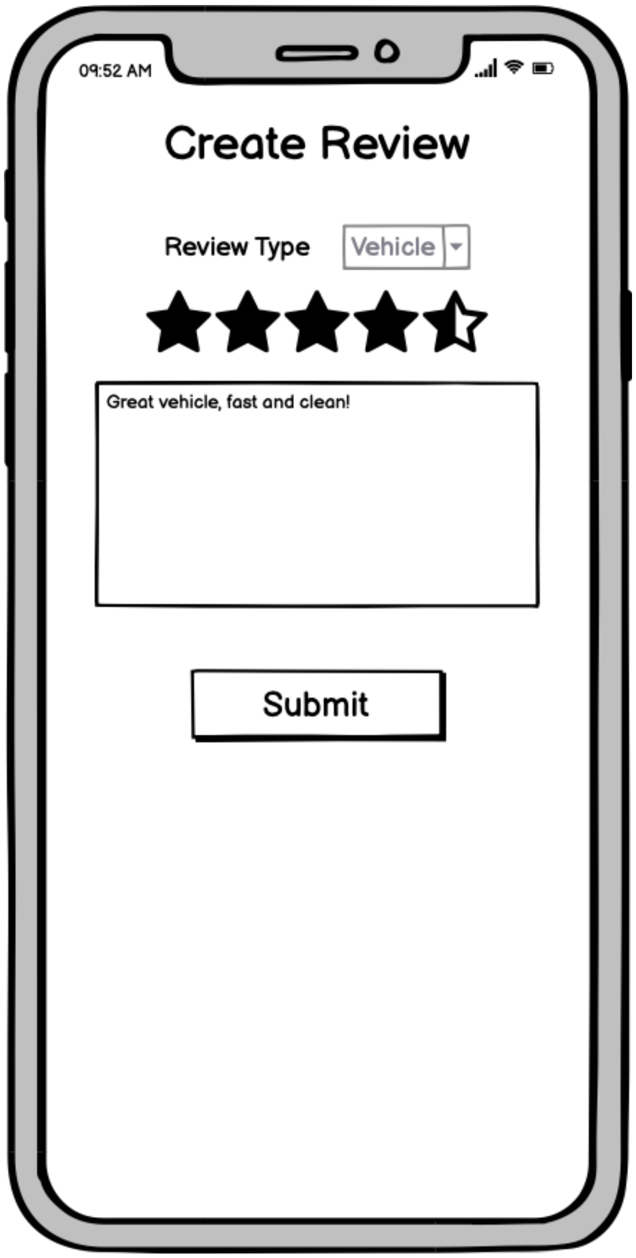
Request a Ride Endpoints

Verb	Endpoint	Description
POST	/rideRequests	used to create a new RideRequest

GET	/rideRequests/{requestId}	used to get a single RideRequest by ID
PUT	/rideRequests/{requestId}	used to update/cancel a RideRequest by RideRequest ID

2.2.3 Review

On the Review page, the Rider can give a star rating to the Vehicle as well as leave a text comment in the review. Review type is visible but is automatically filled in and not editable.



Great vehicle, fast and clean!

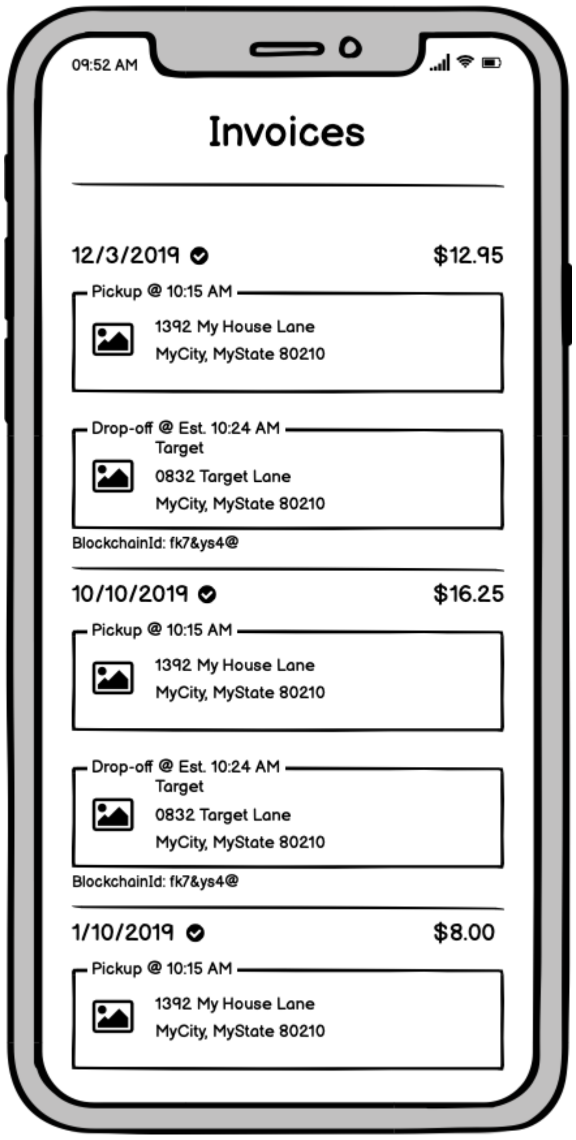
The create a review screen has a simple interface that makes leaving reviews quick and easy

Review Endpoints

Verb	Endpoint	Description
POST	/reviews	used to create a new Review

2.2.4 Review Invoices

The Review Invoices screen provides a convenient interface where riders can view previous RideRequests they have made. Each Ride Request entry shows the origin, destination, date and total cost of the Ride. An infinite scroll should be incorporate on this screen so all of the invoices can quickly been seen in one easy location.


09:52 AM

Invoices

12/3/2019 ✓ $12.95

Pickup @ 10:15 AM
1392 My House Lane
MyCity, MyState 80210

Drop-off @ Est. 10:24 AM
Target
0832 Target Lane
MyCity, MyState 80210

BlockchainId: fk7&ys4@

10/10/2019 ✓ $16.25

Pickup @ 10:15 AM
1392 My House Lane
MyCity, MyState 80210

Drop-off @ Est. 10:24 AM
Target
0832 Target Lane
MyCity, MyState 80210

BlockchainId: fk7&ys4@

1/10/2019 ✓ $8.00

Pickup @ 10:15 AM
1392 My House Lane
MyCity, MyState 80210

The Review Invoices screen provides an interface for Riders to look at their past RideRequests in detail

Review Invoices Endpoints

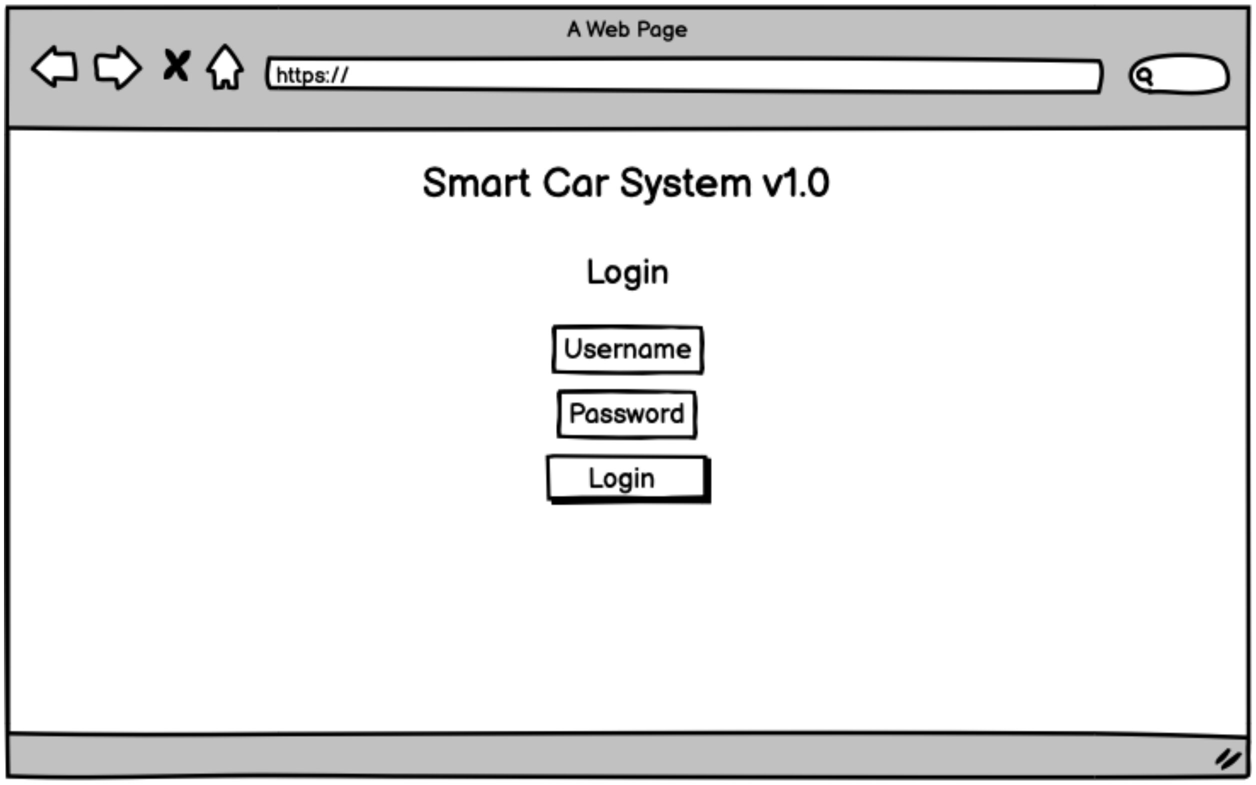
Verb	Endpoint	Description
GET	/riderProfiles/{riderProfileId}/rideRequests	used to find and view all of the RideRequests associated with a specific RiderProfile

2.3 Admin Web Application

The Admin Web Application is used by system administrators to monitor the system and make changes to Vehicles, Rider Profiles and RideRequests. The Admin Web Application is protected by a Login entry point and functionality from the Authentication Service.

2.3.1 Login

The Admin Web Application Login page is a simple login page that guards the Admin Web Application. Only users who have an Admin level account in the Authentication Service are allowed access into the Admin Web Application.



The Admin GUI is protected by a simple username password login

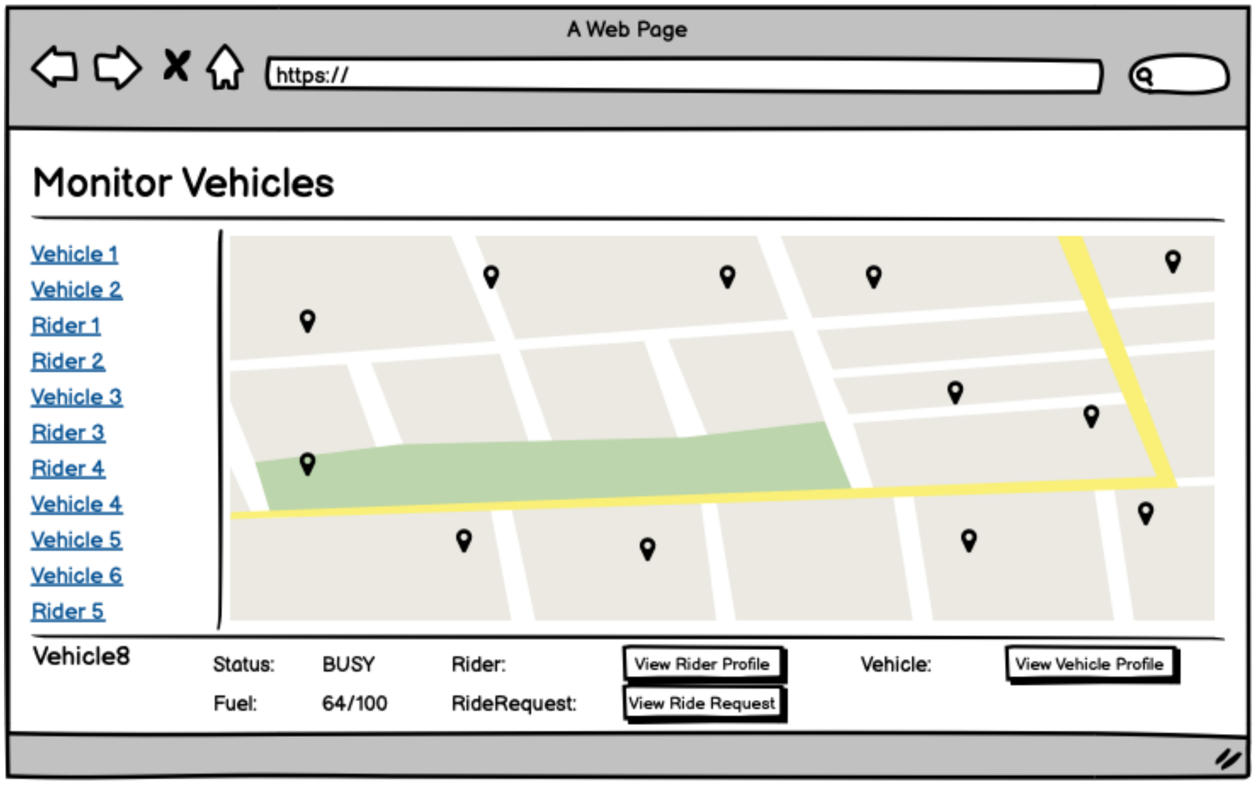
Login Endpoints

Verb	Endpoint	Description

POST	/login	used to login a admin and get their issued auth token

2.3.2 Monitor Vehicles

Administrators can monitor the system from a high level in the Monitor Vehicles page. On this page, admins can see the location of every Vehicle on a map and they can also click the map location markers or entries in a list to get more information about Vehicles and Riders currently using the Smart Car System. Details on the selected Vehicle or Rider are shown in the bottom portion of the screen as well as information about that selected items current status and relevant links. A google maps API could be used to provide the map graphics for this part of the GUI.



Vehicles can be monitored using the Admin GUI and an easy to use map shows where each Vehicle is

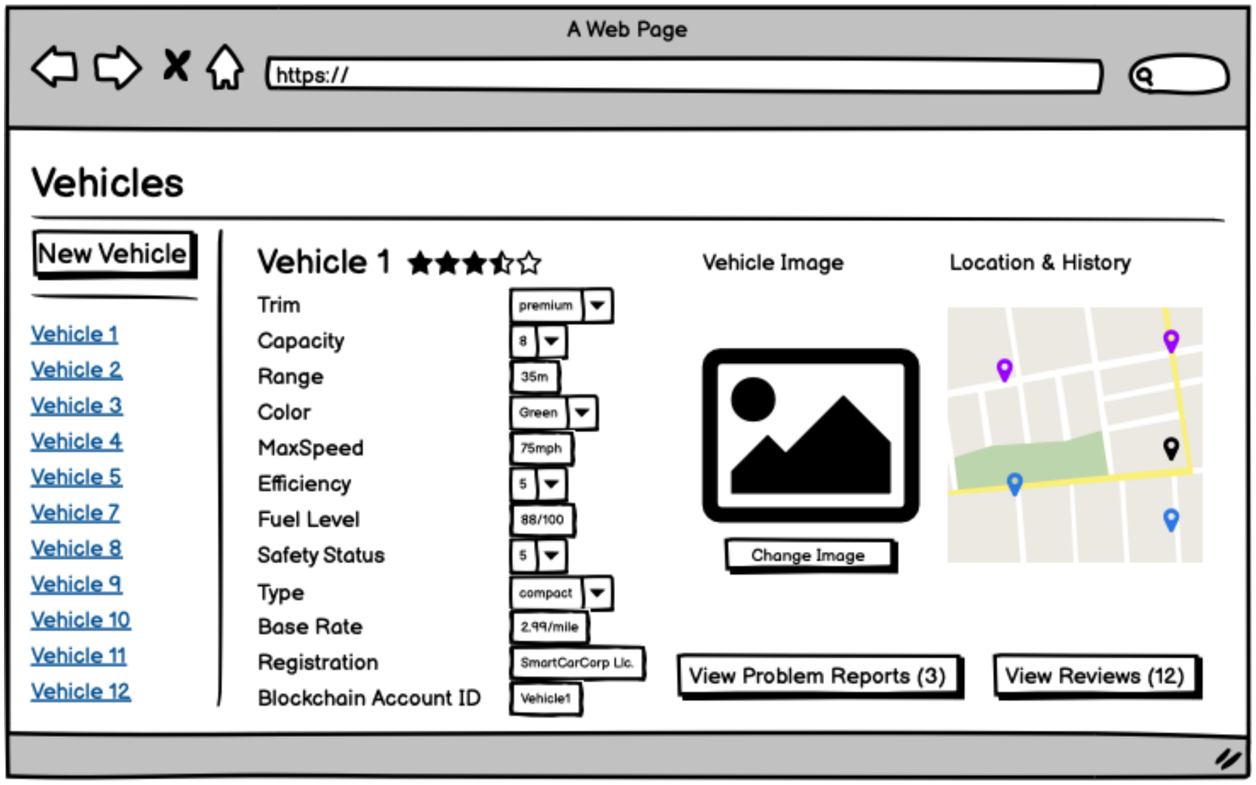
Monitor Vehicles Endpoints

Verb	Endpoint	Description
GET	/riderProfiles	used to get all RiderProfiles as once
GET	/riderProfiles/{riderProfilesId}	used to get a single RiderProfile by ID
GET	/vehicles	used to get all Vehicles at once
GET	/vehicles/{vehicleId}	used to get a single Vehicle by ID
GET	/rideRequests	used to get all RideRequests at once

GET	/rideRequests/{rideRequestId}	used to get a single RideRequest by ID

2.3.3 Create & Update Vehicles

Vehicles can be created and updated by an Admin on the Create & Update Vehicles page. Here the admin is shown a list of all of the available vehicles and a detailed view of properties they can mutate. Selecting "New Vehicle" provides the admin with an identical but empty Vehicle page which they can fill in with the new Vehicles details. Admins can also view ProblemReports and Reviews associated with the Vehicle and these would show in text in a pop up window.



Trim	premium
Capacity	8
Range	35m
Color	Green
MaxSpeed	75mph
Efficiency	5
Fuel Level	88/100
Safety Status	5
Type	compact
Base Rate	2.99/mile
Registration	SmartCarCorp Llc.
Blockchain Account ID	Vehicle1

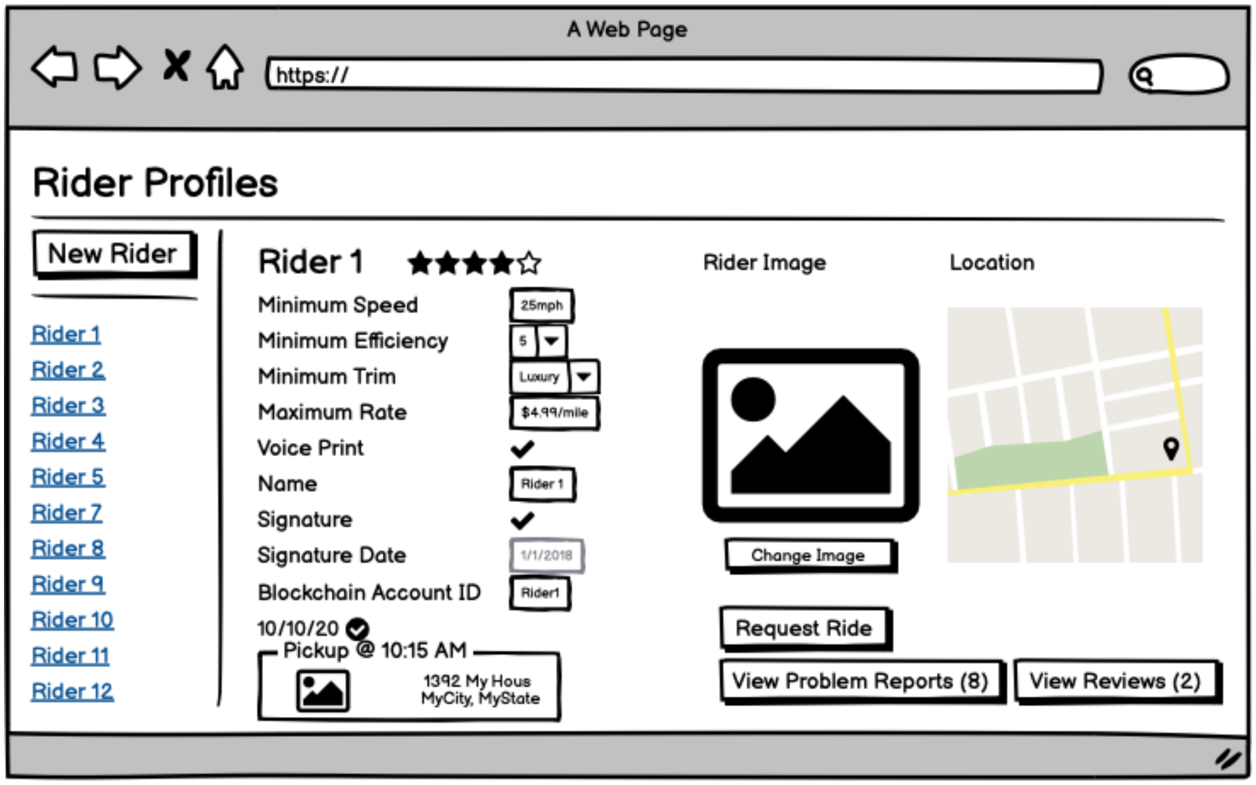
Vehicles can be updated by administrators on the Vehicle page. ProblemReports and Reviews can also be viewed.

Create & Update Vehicles Endpoints

Verb	Endpoint	Description
GET	/vehicles	used to get all Vehicles at once
GET	/vehicles/{vehicleId}	used to get a single Vehicle by the ID
POST	/vehicles	used to create a new Vehicle
PUT	/vehicles/{vehicleId}	used to update a Vehicle by the Vehicles ID

2.3.4 Create & Update RiderProfiles

Rider Profiles can be created and updated by an Admin on the Create & Update RiderProfiles page. Here the admin is shown a list of all of the available Rider Profiles and a detailed view of properties they can mutate. Selecting “New Rider” provides the admin with an identical but empty Rider Profile page which they can fill in with the new Rider Profile details. Admins can also view Problem Reports and Reviews associated with the RiderProfile and these would show in text in a pop up window. The riders RideRequest history is also visible on this screen.



Minimum Speed	25mph
Minimum Efficiency	5
Minimum Trim	Luxury
Maximum Rate	$4.99/mile
Voice Print	✓
Name	Rider 1
Signature	✓
Signature Date	1/1/2018
Blockchain Account ID	Rider1

10/10/20 ✓
Pickup @ 10:15 AM
1392 My Hous
MyCity, MyState

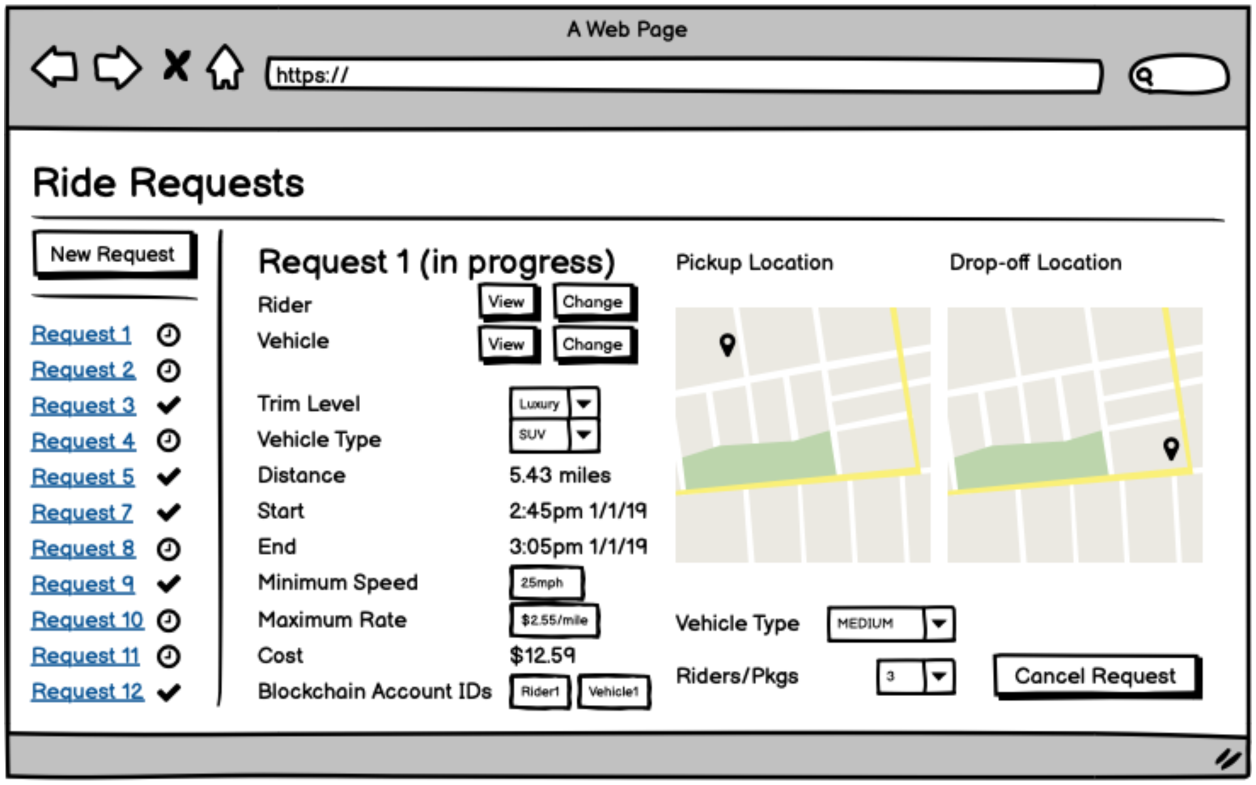
RiderProfiles can be updated by administrators on the Rider Profiles page. ProblemReports and Reviews can also be viewed.

Create & Update Rider Profiles Endpoints

Verb	Endpoint	Description
GET	/riderProfiles	used get all RiderProfiles at once
GET	/riderProfiles/{riderProfileId}	used to get a single RideProfile by ID
POST	/riderProfiles	used to create a new RiderProfile
PUT	/riderProfiles/{riderProfileId}	used to update the RiderProfile by the RiderProfiles ID
GET	/riderProfiles/{riderProfileId}/ rideRequests	used to get all of the RideRequests created by this RiderProfile

2.3.5 Create & Update RideRequests

Ride Requests can be created and updated by an Admin on the Create & Update RideRequests page. Here the admin is shown a list of all of the current and recent RideRequests. Each RideRequest can be selected and once selected, the RideRequests details will show in a panel on the right hand side. Selecting “New Request” provides the admin with an identical but empty RideRequest page which they can fill in with the new Ride Request details in order to manually create a RideRequest. Admins can also cancel ride request in this window. The left hand list used checkmarks to denote completed rides and clock to denote in progress rides. Some properties, like total cost (“cost”) cannot be changed even by admins since the are calculated based on other properties.



Request 1 (in progress)

Request 1	◷
Request 2	◷
Request 3	✓
Request 4	◷
Request 5	✓
Request 7	✓
Request 8	◷
Request 9	✓
Request 10	◷
Request 11	◷
Request 12	✓

Rider	View / Change
Vehicle	View / Change
Trim Level	Luxury
Vehicle Type	SUV
Distance	5.43 miles
Start	2:45pm 1/1/19
End	3:05pm 1/1/19
Minimum Speed	25mph
Maximum Rate	$2.55/mile
Cost	$12.59
Blockchain Account IDs	Rider1 Vehicle1
Vehicle Type	MEDIUM
Riders/Pkgs	3

RideRequests can be updated by administrators on the Rider Requests page. RideRequests can also be created and cancelled here.

Create & Update Ride Requests Endpoints

Verb	Endpoint	Description
GET	/rideRequests	used to get all RideRequests at once
GET	/rideRequests/{rideRequestId}	used to get a single RideRequest by ID
POST	/rideRequests	used to create a new RideRequest
PUT	/rideRequests/{rideRequestId}	used to update the RideRequest by the RideRequest ID

3. Smart Car Model Service Design Document

3.1 Introduction

The Smart Car Model Service is responsible for managing all of the objects in the Smart Car System. Standard CRUD operations are performed using the Model Service and can be executed by Admins and Riders. The Smart Car Model Service is also responsible for reporting events to observers after certain actions are performed on models. One example of this would be when an event is fired after a new RideRequest is created. The Model Service is secured by the Smart Car System Authentication Service and each method in the Model Service requires a valid AuthToken in order to execute successfully.

3.2 Requirements

Manage RiderProfiles
- Must be able to manage RiderProfiles by providing create, read and update functionality
- Must validate that no duplicates are persisted
- Must confirm that caller is authorized to manage RiderProfiles
- Must use Hibernate to create and update RiderProfile records

Manage RideRequests
- Must be able to manage RideRequests by providing create, read and update functionality
- Must validate that no duplicates are persisted
- Must confirm that caller is authorized to manage RideRequests
- Must use Hibernate to create and update RideRequest records
- Must also provide a way to get all of the RideRequests that belong to one particular RiderProfiles
- RideRequests must have a relationship with a RiderProfile and Vehicle that way one can tell which Vehicle executed the request and who ordered the ride

Manage Vehicles
- Must be able to manage Vehicles by providing create, read and update functionality
- Must validate that no duplicates are persisted
- Must confirm that caller is authorized to manage Vehicles
- Must use Hibernate to create and update Vehicle records
- Must also have methods to interface with the vehicles embedded system in order for the car to execute RideRequests in the physical world

Manage Reviews
- Must be able to manage Reviews by providing create, read and update functionality
- Must validate that no duplicates are persisted
- Must confirm that caller is authorized to manage Reviews
- Must use Hibernate to create and update Review records
- Must be able to get all of the Reviews for a Vehicle or RiderProfile

Manage Problem Reports
- Must be able to manage ProblemReports by providing create, read and update functionality
- Must validate that no duplicates are persisted
- Must confirm that caller is authorized to manage ProblemReports
- Must use Hibernate to create and update ProblemReport records
- Must be able to get all of the ProblemReports for a Vehicle or RiderProfile

Provide Utility Functions to Coordinate Smart Cars

- Must be able to mutate Vehicles and RideRequests in such a way that it's apparent that one Vehicle is currently executing a RideRequest and the progress on that RideRequest can be seen or inferred
- Must be able to send Vehicles to get serviced or to be put in storage in the event of a malfunction or low traffic day
- Must be able to explicitly send Vehicles to specific Location
- Must be able to command that a Vehicle begin/initiate executing a RideRequest
- Must be able to calculate trip routes in order to tell cars how to best execute RideRequests

Implementation as an Observable Subject
- Notification of observers must be done using the Observable pattern.
- Must be capable to registering observers
- Must be capable of notifying observers
- Must be free of circular dependencies

Implementation of an integration broker
- Must implement an integration broker pattern in order to allow HTTP communication between the Model Service and the Mobile and Admin GUIs

Security using the Authentication Service
- Must use the Auth Service and AuthTokens for authentication and authorization
- AuthTokens must be provided as args per execution and be verified at the time the method is called

---

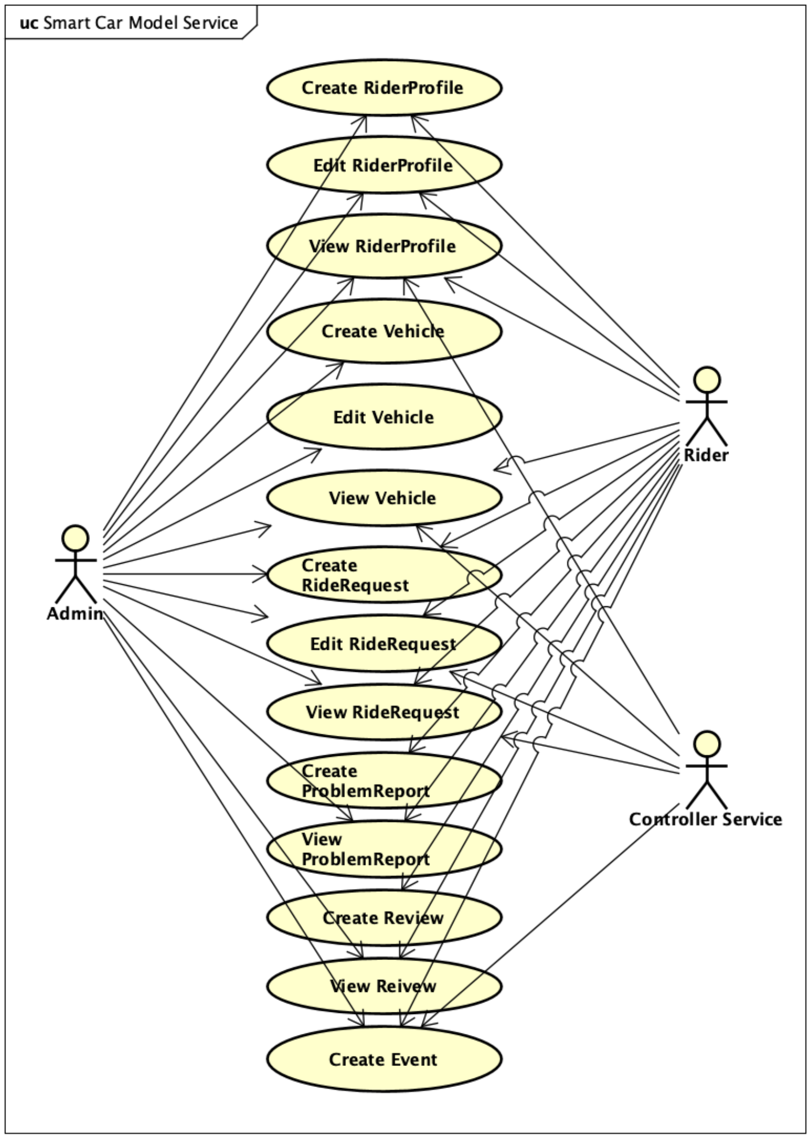
3.3 Use Cases

Actors

- **Admin:** Administrators have the greatest access to the Smart Car Model Service with the ability to execute any method on the class.

- **Rider:** Riders have restricted access to the Smart Car Model Service with most interaction being limited to mutating objects they have a relation with such as their RiderProfile, their RideRequests and their Reviews and ProblemReports.

- **Controller Service:** The Smart Car System Controller has pretty wide open access to most of the methods on the Smart Car Model Service although a vast majority of the operations conducted by the Controller Service involved simply mutating entities that are accessible via the Smart Car Model Service.



Above is the use case diagram which outlines what the expected actors are and what abilities they'll have in the Smart Car Model Service.

Create RiderProfile

- Verify that the method callers AuthToken is authorized access to this feature (else Error)
- Verify all the required arguments are present (else Error)
- Verify that the RiderProfiles ID is unique amongst all other RiderProfiles
- Create the a new RiderProfile based on the provided arguments
- Use Hibernate to persist the new RiderProfile

Get RiderProfile

- Verify that the method callers AuthToken is authorized access to this feature (else Error)
- Verify all the required arguments are present (else error)
- Use Hibernate to attempt to get the RiderProfile by the provided RiderProfile ID
- Return the RiderProfile (else NotFound Error)

Update RiderProfile

- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to attempt to get the RiderProfile by the provided RiderProfile ID
- Confirm the provided update info is valid
- Use Hibernate to update the RiderProfile with the provided new info
- Return the newly updated RiderProfile

Create Vehicle
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Verify that the Vehicle ID is unique amongst all other Vehicles (else error)
- Create the a new Vehicle based on the provided arguments
- Use Hibernate to persist the new Vehicle

Get Vehicle
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to attempt to get the Vehicle by the provided Vehicle ID
- Return the Vehicle (else NotFound Error)

Update Vehicle
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to attempt to get the Vehicle by the provided Vehicle ID
- Confirm the provided update info is valid
- If the Vehicles Location has been updated, then create a VehicleLocationEvent and notify observers
- Use Hibernate to update the Vehicle with the provided new info
- Return the newly updated Vehicle

Create RideRequest
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify that all the required arguments are present (else error)
- Use Hibernate to get the RiderProfile associated with the RideRequest
- Assign a capable and available Vehicle to the new RideRequest (using the "Finding/Assigning Vehicles to RideRequests" use case)
- Create an OfferAvailableEvent and notify observers
- Return the newly created RideRequest

Get RideRequest
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to attempt to get the RideRequest by the provided RideRequest ID
- Return the found RideRequest

Update RideRequest
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to attempt to get the RideRequest by the provided RideRequest ID
- Confirm the provided update info is valid
- Use Hibernate to update the RideRequest with the provided new info
- Return the newly updated RideRequest

Get RideRequests by RiderProfile ID
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to get the RiderProfile by ID to confirm that the RiderProfile exists

- Use Hibernate Search to get all of the RideRequests associate with the RiderProfile Id
- Return a collection of the RideRequests found

Create Review
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- User Hibernate find RiderProfile or Vehicle to determine if the provided ID argument is valid
- Create new Review review based on the provided arguments and associate it with the appropriate Vehicle or RiderProfile

Get Review
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to attempt to get the Review by the provided Review ID
- Return the Review (else NotFound Error)

Get Reviews by RiderProfileID
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to get the RiderProfile by ID to confirm that the RiderProfile exists
- Use Hibernate Search to get all of the Reviews associated with the RiderProfile Id
- Return a collection of the Reviews found

Get Reviews by VehicleID
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present (else error)
- Use Hibernate to get the Vehicle by ID to confirm that the Vehicle exists
- Use Hibernate Search to get all of the Vehicles associated with the Vehicle Id
- Return a collection of the Reviews found

Calculating Trip Routes
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present and that Location coordinates are valid (else error)
- Use the Google Maps api to request an efficient route for the trip from one Location to the other
- Return the Route

Finding/assigning Vehicles to RideRequests
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present
- Use Hibernate to get the RideRequest by ID
- Use Hibernate to search for a Vehicle matching the RideRequests specs
- Confirm the Vehicle has enough fuel
- Calculate/update the RideRequest.cost and .rate and confirm the Rider of the RideRequest has sufficient funds
- Assign the Vehicle to the RideRequest
- Use Hibernate to save the updated RideRequest
- Tigger a OfferAvailableEvent to let the Controller Service know a Vehicle has been assigned which implies that an offer has been made

Accepting RideRequest Vehicle Offer
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present
- Use Hibernate to get the Vehicle by ID
- Use Hibernate to get the RideRequest by ID
- Update the RideRequests status to “offerSelected” and set the .acceptanceDate as now, also calculate the ETA and update the .pickupDateTime to that.

- Use Hibernate to save the updated RideRequest
- Tigger an OfferAcceptedEvent to let the Controller Service know an offer has been accepted for a RideRequest

Releasing Vehicles from RideRequests
- Verify that the method callers AuthToken is authorized access to this feature (else error)
- Verify all the required arguments are present
- Use Hibernate to get the Vehicle by ID
- Use Hibernate to get the RideRequest by ID
- Update the RideRequests status
- Use Hibernate to save the updated RideRequest
- Tigger a SmartCarSystemEvent to let the Controller Service know a Vehicle has been released and can now be sent back to high traffic area, sent in for service, or sent into storage

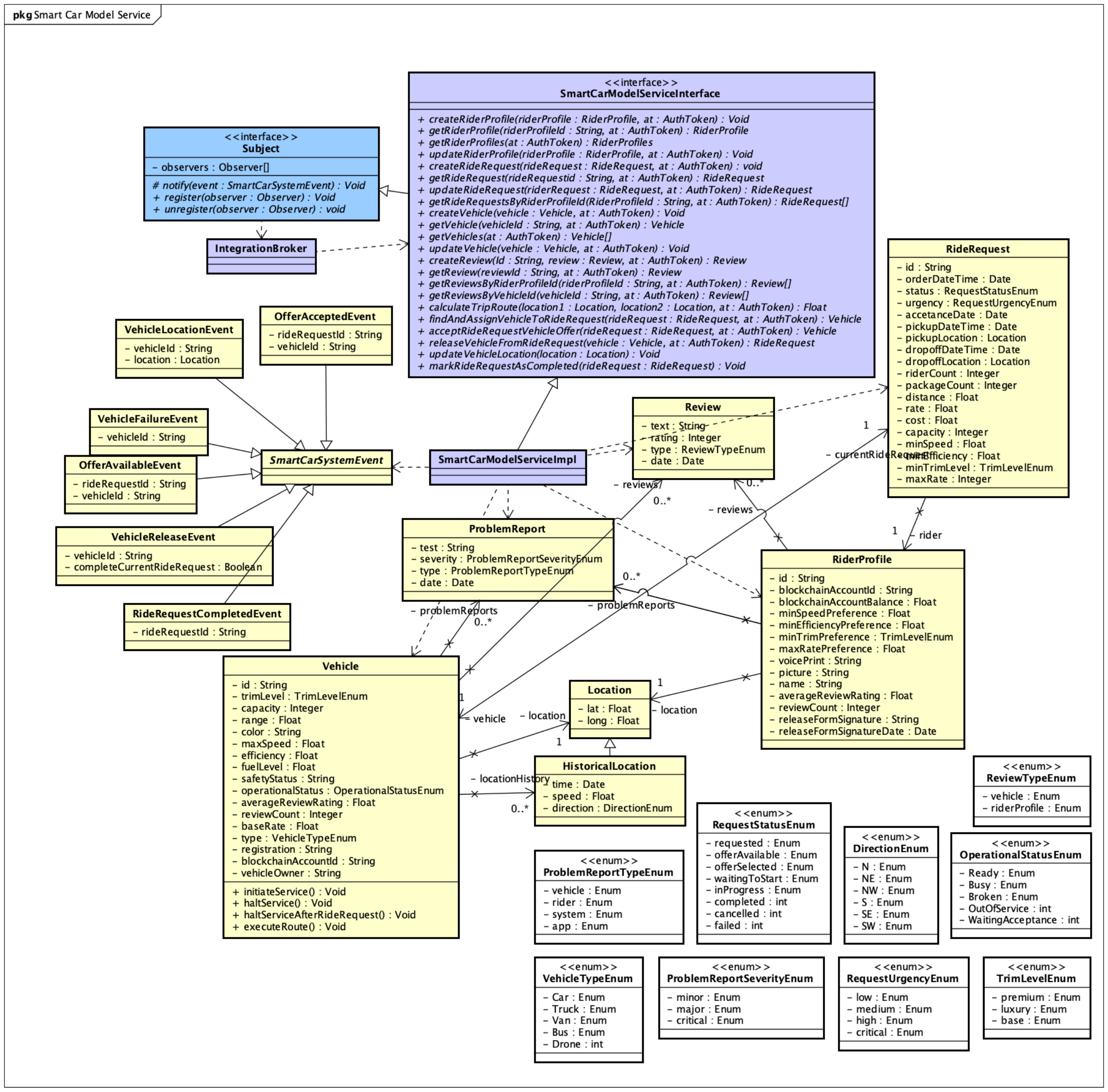
Observable Implementation
- Register as an observable on the Smart Car Model Service as long as the object type implements observable, store all registered observers in memory
- Allow in scope classes and callers to call method(s) that would notify observers when they're called and provide the option of including a payload

3.4 Implementation

The Smart Car System Model Service takes the responsibility for both managing all of the models in the Smart Car System as well as providing the methods to coordinate Vehicles in the Smart Car System. Timing of particular responses to SmartCarSystemEvents is left up to the Controller Service but many SmartCarSystemEvents originate in the Model Service as a side effect of a parameter change or a new association being made. Vehicles, RiderProfiles and RideRequests make up the bulk of the Model Service functionality and they're closely relate to one another. Vehicles are assigned RideRequests that they are or may soon be executing, that is considered an "offer". Riders are also assigned to RideRequests which they've ordered. Reviews can be either associated with a Vehicle or RiderProfile and each of those also store various Location data.

Often times, an observer notification will be used to let the Controller Service know that a relevant status has been changed or method called and that a certain side effect needs to occur. For example, once acceptRideRequestVehicleOffer is called, the Controller Service should receive an OfferAcceptedEvent after which it may direct the car to execute the RideRequest using Vehicle.initiateService. Another example can be seen in releaseVehicleFromRideRequest where the Vehicles properties are mutated and a VehicleReleaseEvent is emitted after which the Controller Service will be notified that its safe to direct the car to a new Location such as a service or storage location.

3.5 Class Diagram


pkg Smart Car Model Service
<<interface>> SmartCarModelServiceInterface
+ createRiderProfile(riderProfile : RiderProfile, at : AuthToken) : Void
+ getRiderProfile(riderProfileId : String, at : AuthToken) : RiderProfile
+ getRiderProfiles(at : AuthToken) : RiderProfiles
+ updateRiderProfile(riderProfile : RiderProfile, at : AuthToken) : Void
+ createRideRequest(rideRequest : RideRequest, at : AuthToken) : void
+ getRideRequest(rideRequestid : String, at : AuthToken) : RideRequest
+ updateRideRequest(riderRequest : RideRequest, at : AuthToken) : RideRequest
+ getRideRequestsByRiderProfileId(RiderProfileId : String, at : AuthToken) : RideRequest[]
+ createVehicle(vehicle : Vehicle, at : AuthToken) : Void
+ getVehicle(vehicleId : String, at : AuthToken) : Vehicle
+ getVehicles(at : AuthToken) : Vehicle[]
+ updateVehicle(vehicle : Vehicle, at : AuthToken) : Void
+ createReview(Id : String, review : Review, at : AuthToken) : Review
+ getReview(reviewId : String, at : AuthToken) : Review
+ getReviewsByRiderProfileId(riderProfileId : String, at : AuthToken) : Review[]
+ getReviewsByVehicleId(vehicleId : String, at : AuthToken) : Review[]
+ calculateTripRoute(location1 : Location, location2 : Location, at : AuthToken) : Float
+ findAndAssignVehicleToRideRequest(rideRequest : RideRequest, at : AuthToken) : Vehicle
+ acceptRideRequestVehicleOffer(rideRequest : RideRequest, at : AuthToken) : Vehicle
+ releaseVehicleFromRideRequest(vehicle : Vehicle, at : AuthToken) : RideRequest
+ updateVehicleLocation(location : Location) : Void
+ markRideRequestAsCompleted(rideRequest : RideRequest) : Void
<<interface>> Subject
– observers : Observer[]
# notify(event : SmartCarSystemEvent) : Void
+ register(observer : Observer) : Void
+ unregister(observer : Observer) : void
IntegrationBroker
RideRequest
– id : String
– orderDateTime : Date
– status : RequestStatusEnum
– urgency : RequestUrgencyEnum
– accetanceDate : Date
– pickupDateTime : Date
– pickupLocation : Location
– dropoffDateTime : Date
– dropoffLocation : Location
– riderCount : Integer
– packageCount : Integer
– distance : Float
– rate : Float
– cost : Float
– capacity : Integer
– minSpeed : Float
– minEfficiency : Float
– minTrimLevel : TrimLevelEnum
– maxRate : Integer
VehicleLocationEvent
– vehicleId : String
– location : Location
OfferAcceptedEvent
– rideRequestId : String
– vehicleId : String
VehicleFailureEvent
– vehicleId : String
OfferAvailableEvent
– rideRequestId : String
– vehicleId : String
SmartCarSystemEvent
SmartCarModelServiceImpl
Review
– text : String
– rating : Integer
– type : ReviewTypeEnum
– date : Date
VehicleReleaseEvent
– vehicleId : String
– completeCurrentRideRequest : Boolean
RideRequestCompletedEvent
– rideRequestId : String
ProblemReport
– test : String
– severity : ProblemReportSeverityEnum
– type : ProblemReportTypeEnum
– date : Date
RiderProfile
– id : String
– blockchainAccountId : String
– blockchainAccountBalance : Float
– minSpeedPreference : Float
– minEfficiencyPreference : Float
– minTrimPreference : TrimLevelEnum
– maxRatePreference : Float
– voicePrint : String
– picture : String
– name : String
– averageReviewRating : Float
– reviewCount : Integer
– releaseFormSignature : String
– releaseFormSignatureDate : Date
Vehicle
– id : String
– trimLevel : TrimLevelEnum
– capacity : Integer
– range : Float
– color : String
– maxSpeed : Float
– efficiency : Float
– fuelLevel : Float
– safetyStatus : String
– operationalStatus : OperationalStatusEnum
– averageReviewRating : Float
– reviewCount : Integer
– baseRate : Float
– type : VehicleTypeEnum
– registration : String
– blockchainAccountId : String
– vehicleOwner : String
+ initiateService() : Void
+ haltService() : Void
+ haltServiceAfterRideRequest() : Void
+ executeRoute() : Void
Location
– lat : Float
– long : Float
HistoricalLocation
– time : Date
– speed : Float
– direction : DirectionEnum
<<enum>> ReviewTypeEnum
– vehicle : Enum
– riderProfile : Enum
<<enum>> RequestStatusEnum
– requested : Enum
– offerAvailable : Enum
– offerSelected : Enum
– waitingToStart : Enum
– inProgress : Enum
– completed : int
– cancelled : int
– failed : int
<<enum>> DirectionEnum
– N : Enum
– NE : Enum
– NW : Enum
– S : Enum
– SE : Enum
– SW : Enum
<<enum>> OperationalStatusEnum
– Ready : Enum
– Busy : Enum
– Broken : Enum
– OutOfService : int
– WaitingAcceptance : int
<<enum>> ProblemReportTypeEnum
– vehicle : Enum
– rider : Enum
– system : Enum
– app : Enum
<<enum>> VehicleTypeEnum
– Car : Enum
– Truck : Enum
– Van : Enum
– Bus : Enum
– Drone : int
<<enum>> ProblemReportSeverityEnum
– minor : Enum
– major : Enum
– critical : Enum
<<enum>> RequestUrgencyEnum
– low : Enum
– medium : Enum
– high : Enum
– critical : Enum
<<enum>> TrimLevelEnum
– premium : Enum
– luxury : Enum
– base : Enum
– currentRideRequest
1
– reviews
0..*
– reviews
0..*
1
– rider
– problemReports
0..*
– problemReports
0..*
1
– vehicle
– location
1
– location
1
– locationHistory
0..*

Above is the class diagram outlining the design and relation of all the classes in the Smart System Model Service

3.6 Class Dictionary

Below are table representations of every class implemented in the Smart Car Model Service along with short descriptions of what each value represents.

SmartCarModelServiceImpl *(implements Subject & SmartCarModelServiceInterface)*

All methods are protected by the AuthService. Once the method is called, it will used the provided at:AuthToken to validate whether or not the caller has sufficient privileges to access the method.

SmartCarModelService Interface & Impl Description

Signature	Implementation Description
+ createRiderProfile(riderProfile : RiderProfile, at: AuthToken) : Void	Create a RiderProfile from the provided arg. Confirm a RiderProfile with that ID doesn't already exist (otherwise return error) and persist it using Hibernate.
+ getRiderProfile(riderProfileId : String, at: AuthToken) : RiderProfile	Get a RiderProfile using Hibernate via the provided ID
+ getRiderProfiles(at: AuthToken): RiderProfiles[]	Get all RiderProfiles in the system
+ updateRiderProfile(riderProfile : RiderProfile, at: AuthToken) : Void	Using Hibernate, update with the provided RiderProfile, the persisted RiderProfile whose ID matches the one in the provided RiderProfile. If there is no RiderProfile with that ID then return an error.
+ createRideRequest(rideRequest : RideRequest, at: AuthToken) : void	Create a RideRequest from the provided arg. Confirm a RideRequest with that ID doesn't already exist and confirm that the RideRequests associated RiderProfile exists otherwise return an error. Also set .orderDateTime to time of creation. Then proceed to .findAndAssignVehicleToRideRequest
+ getRideRequest(rideRequestid : String, at: AuthToken) : RideRequest	Get a RideRequest using Hibernate via the provided ID
+ updateRideRequest(riderRequest : RideRequest, at: AuthToken) : RideRequest	Using Hibernate, update with the provided RideRequest, the persisted RideRequest whose ID matches the one in the provided RideRequest. If there is no RideRequest with that ID then return an error. Update the associated Vehicle if necessary.
+ getRideRequestsByRiderProfileId(RiderProfileId : String, at: AuthToken) : RideRequest[]	Using Hibernate, get all of the RideRequests that are associated with a RiderProfile whose ID matches the one provided as an arg.
+ createVehicle(vehicle : Vehicle, at: AuthToken) : Void	Create a Vehicle from the provided arg. Confirm a Vehicle with that ID doesn't already exist (otherwise return error) and persist it using Hibernate.
+ getVehicle(vehicleId : String, at: AuthToken) : Vehicle	Get a Vehicle using Hibernate via the provided ID
+ getVehicles(at: AuthToken): Vehicle[]	Get all of the Vehicles in the system via Hibernate

Signature	Implementation Description
+ updateVehicle(vehicle : Vehicle, at: AuthToken) : Void	Using Hibernate, update with the provided Vehicle, the persisted Vehicle whose ID matches the one in the provided Vehicle. If there is no Vehicle with that ID then return an error. Update the associated currentRideRequest if necessary. If the Vehicles location has been updated (Vehicles make requests periodically in order to update their own location), then create a VehicleLocationEvent and notify observers.
+ createReview(Id : String, review : Review, at: AuthToken) : Review	Create a Review from the provided arg. Confirm a Review with that ID doesn't already exist (otherwise return error) and persist it using Hibernate. Also associate the Review to the correct Vehicle or RiderProfile and update the appropriate object.
+ getReview(reviewId : String, at: AuthToken) : Review	Get a Review using Hibernate via the provided ID
+ getReviewsByRiderProfileId(riderProfileId : String, at: AuthToken) : Review[]	Using Hibernate, get all of the Reviews that are associated with a RiderProfile whose ID matches the one provided as an arg.
+ getReviewsByVehicleId(vehicleId : String, at: AuthToken) : Review[]	Using Hibernate, get all of the Reviews that are associated with a Vehicle whose ID matches the one provided as an arg.
+ calculateTripRoute(location1 : Location, location2 : Location, at: AuthToken) : Route	Calculate the route from one Location to another Location and return that route.
+ findAndAssignVehicleToRideRequest (rideRequest : RideRequest, at: AuthToken) : Vehicle	Use Hibernate Search to find the best Vehicle capable to executing the RideRequest. Calculate the trip route using .calculateTripRoute in order to make sure the Vehicle has enough fuel otherwise search for another Vehicle. Call .updateRideRequest setting the RideRequest .status to "OfferAvailable," setting .vehicle to the best suited Vehicle found, setting .cost to (Vehicle.baseRate x rideRequest.distance), and setting .rate to the Vehicle.baseRate. Verify the Rider of the RideRequest has sufficient funds via their .blockchainAccountBalance. Also notify observers with a OfferAvailableEvent.
+ acceptRideRequestVehicleOffer (rideRequest : RideRequest, at: AuthToken) : Vehicle	Call .updateRideRequest to change RideRequest status to "OfferSelected" and set its acceptanceDate. Also notify observers with a OfferAcceptedEvent.
+ releaseVehicleFromRideRequest(vehicle : Vehicle, at: AuthToken) : RideRequest	Set the Vehicles currentRideRequest as null. Notify observers with a VehicleReleaseEvent.
+ markRideRequestAsCompleted(rideRequest: RideRequest): Void	Update the RideRequest.status to "completed" and set its .dropOffLocation and .dropOffDate. Also notify observer with a RideRequestCompletedEvent.

RideRequest

The RideRequest object is responsible for storing all of the data about a ride ordered by a Rider/Customer. The status property on RideRequest is responsible for much of the Vehicle coordination. The status property reflects wether or not there's an offer (which means a vehicle should be busy waiting for a response), the offer is accepted (which means the Vehicle should be in route) or the offer it inProgress (which means the Vehicle should be in the middle of completing the RideRequest).

RideRequest Associations

	Type	Description
vehicle	Vehicle	The Vehicle assigned to the RideRequest
rider	RiderProfile	The RiderProfile who ordered the RideRequest
pickupLocation	Location	Location where the Rider is picked up
dropoffLocation	Location	Location where the Rider is dropped off

RideRequest Properties

	Type	Description
id	String	Unique ID of the RideRequest in the system
orderDateTime	Date	Time/Date when the RideRequest was created
status	RequestStatusEnum	Current status of the request
urgency	RequestUrgencyEnum	Rider declared urgency of RideRequest
acceptanceDate	Date	Time/Date when the Rider accepted the ride offer
pickupDateTime	Date	Time/Date the Rider was picked up
dropoffDateTime	Date	Time/Date the Rider was dropped off
dropoffDateTime	Date	Time/Date the Rider was dropped off
dropoffLocation	Location	Location where the Rider is dropped off
riderCount	Integer	Number of Rider on the trip
packageCount	Integer	Number of packages on the trip
distance	Float	Total distance for pickup to drop-off Location
rate	Float	Per unit rate of ride
cost	Float	Total cost of the ride
minSpeed	Float	Min speed rating required of responding Vehicle
minEfficiency	Float	Min efficiency level required of responding Vehicle

	Type	Description
minTrimLevel	TrimLevelEnum	Minimum required Trim level of responding Vehicle
maxRate	Float	Maximum rate of responding Vehicle

RiderProfile

The RiderProfile class is responsible for all of the data related to the Rider/Customer. The properties in this class are use when filtering for a Vehicle that can fulfill a RideRequest from that Rider. Specifically, the “min” and “max” properties contribute to determining which Vehicles are capable of fulfilling a RideRequest for a particular Rider. ProblemReports authored by the Rider are associated with the RiderProfile as well as Reviews to which the Rider is a the subject.

RiderProfile Associations

	Type	Description
location	Location	Riders current Location
reviews	Reviews[]	All of the Reviews for this Rider
problemReports	ProblemReports[]	All of the ProblemReports for this Rider

RiderProfile Properties

	Type	Description
id	String	Unique ID used to id the Rider in the system
blockchainAccountId	String	blockchainAccount associated with Rider
blockchainAccountBalance	Float	Current balance for the RiderProfile
minSpeedPreference	Float	Riders min preferred Vehicle speed
minEfficiencyPreference	Float	Riders min preferred Vehicle efficiency
minTrimPreference	TrimLevelEnum	Riders min preferred Vehicle trim level
maxRatePreference	Float	Riders max Vehicle rate
voicePrint	String	Riders auth vocal print
picture	String	Url to Riders image
name	String	Riders name
averageReviewRating	Float	Riders average Rating from their Reviews
reviewCount	Integer	Count of all the Reviews for this Rider
releaseFormSignature	String	Riders secure and validated signature
releaseFormSignatureDate	Date	Date of releaseFormSignature creation

Vehicle

The Vehicle class represents an autonomous vehicle in the Smart Car System. Many of the properties in the Vehicle class are used simply to determine whether or not a particular Vehicle is suitable for a particular RideRequest. Vehicles can be manually sent to specific locations by using the executeRoute() method where the Vehicle will then call the SmartCarModelService updateVehicle with a new Location after it has reached its destination.

Vehicle Methods

Signature	Description
initiateService(): void	Tell the vehicle to execute the RideRequest. This method would interface with a hardware API that would coordinate the execution of the Route and pickup in something like an embedded system. Functions like throttle, stopping, opening doors and turning would be executed by the Vehicle as a result of initiatingService
haltService(goToLocation: Location): void	Tell the Vehicle to safely stop all activity and release the passenger if there is a currentRideRequest then idle. This can be done before returning a Vehicle to storage or service or if a Vehicle needs to end a RideRequest due to some failure.
haltServiceAfterRideRequest(goToLocation: Location): Void	Tell the Vehicle to safely stop all activity and idle after completing its currentRideRequest if there is a currentRideRequest. This can be done before returning a Vehicle to storage or service or if a Vehicle needs to end a RideRequest due to some failure.
executeRoute(route: Route)	Tell the Vehicle to follow a Route to a specific location immediately. (Route is a currently unknown shape provided by what would be a vendor API maybe somewhat like Google Maps)

Vehicle Associations

	Type	Description
location	Location	Vehicles current Location
reviews	Reviews[]	All of the Reviews for this Vehicles
problemReports	ProblemReports[]	All of the ProblemReports for this Vehicle
locationHistory	Location[]	Historical record of the Vehicles location
currentRideRequest	RideRequest	RideRequest current being fulfilled by the Vehicle

Vehicle Properties

	Type	Description
Id	String	Unique ID used to id the Vehicle in the system
trimLevel	TrimLevelEnum	Trim level of the Vehicle
capacity	Integer	Rider/package capacity of the Vehicle

	Type	Description
range	Float	Range of Vehicle with current fuel
color	String	Paint color of Vehicle
maxSpeed	Float	Current max speed setting of Vehicle
efficiency	Float	Estimated fuel efficiency of Vehicle
fuelLevel	Float	Current fuel level of Vehicle
safetyStatus	String	Safety rating of Vehicle
operationalStatus	OperationalStatusEnum	Current operational status of Vehicle
averageReviewRating	Float	Vehicles average Rating from its Reviews
reviewCount	Integer	Count of all the Reviews for the Vehicle
baseRate	Float	Base per unit rate for the Vehicle
type	VehicleTypeEnum	Type of Vehicle
registration	String	Government Vehicle registration info
blockchainAccountId	String	blockchainAccount ID associated with Vehicle
vehicleOwner	String	String to id legal Vehicle owner

Review

Reviews are simple classes that store customer opinions about a Vehicle (authored by a rider) or a rider (authored by a Vehicle). Reviews have no functionality beyond storing a few properties and the associations made between Vehicles and Reviews and RiderProfiles and Reviews.

Review Properties

	Type	Description
text	String	Review message content
rating	Integer	Review rating (out of 5)
type	ReviewTypeEnum	The type of object being reviewed
date	Date	Date the Review was created

ProblemReport

ProblemReports are simple classes that store information describing a problem that occurred in the Smart Car System. They are often created by riders to communicate problems that occur during a trip. ProblemReports have no functionality beyond storing a few properties and the associations made between Vehicles and ProblemReports and RiderProfiles and ProblemReports.

ProblemReport Properties

	Type	Description
text	String	Problem text description
severity	ProblemReportSeverityEnum	Severity of problem (minor, critical, etc…)
type	ProblemRerportTypeEnum	Type of problem (Rider, Vehicle, etc…)
date	Date	Date the ProblemReport was created

Location

Locations are simple classes that only have two properties that work to store the exact coordinates of a geographic location.

Location Properties

	Type	Description
lat	Float	Latitude of the location
long	Float	Longitude of the location

HistoricalLocation *(inherits from Location)*

HistoricalLocations extend the functionality of Location to include time of the reading and speed and direction at the time of the reading. In a collection, this data helps to communicate the route and location history of a Vehicle. Vehicles contain collections of HistoricalLocations in order to keep track of where they've been.

HistoricalLocation Properties

	Type	Description
time	Date	Date/time when the Location was recorded
speed	Float	Current speed of object when Location was recorded
direction	DirectionEnum	Current direction of travel of object when the Location was recorded

OfferAcceptedEvent *(implements SmartCarSystemEvent)*

The OfferAcceptedEvent is used to notify observers once a Vehicle RideRequest offer is accepted by a rider. This event is usually processed by the Controller where it will coordinate next steps such as initiating service from the Vehicle assigned to that RideRequest.

OfferAcceptedEvent Properties

	Type	Description
rideRequestId	String	The Id of the RideRequest relevant in the event
vehicleId	String	The Id of the Vehicle relevant in the event

VehicleLocationEvent *(implements SmartCarSystemEvent)*

The VehicleLocationEvent is used to notify observers once a Vehicle is subject to an update that includes a new Location. This event is usually processed by the Controller where it will coordinate next steps such as creating a new HistoricalLocation and saving in the Vehicles locationHistory collection.

VehicleLocationEvent Properties

	Type	Description
location	Location	The new Location of the Vehicle
vehicleId	String	The Id of the Vehicle relevant in the event

VehicleReleaseEvent *(implements SmartCarSystemEvent)*

The VehicleReleaseEvent is used to notify observers once a Vehicle directed to quit working on a RideRequest. This event is usually processed by the Controller where it will coordinate next steps such as updating the RideRequest as cancelled or failed and sending the Vehicle to a service center.

VehicleReleaseEvent Properties

	Type	Description
completeCurrentRideRequest	Boolean	Whether or not the vehicle should be allowed to completed its currentRideRequest before it halts service
vehicleId	String	The Id of the Vehicle relevant in the event

OfferAvailableEvent *(implements SmartCarSystemEvent)*

The OfferAvailableEvent is used to notify observers once a Vehicle capable of fulfilling a RideRequest has been found and is assigned to potentially fulfill the RideRequest. This event is usually processed by the Controller where it will coordinate next steps such as updating the RideRequest to notify the rider of an available offer.

OfferAvailableEvent Properties

	Type	Description
rideRequestId	String	The Id of the RideRequest relevant in the event

	Type	Description
vehicleId	String	The Id of the Vehicle relevant in the event

VehicleFailureEvent *(inherits from SmartCarSystemEvent)*

The VehicleFailureEvent is used to notify observers once a Vehicle has failed and no longer capable of providing services. This event is usually processed by the Controller where it will coordinate next steps such as halting the Vehicle and sending it to storage or a service station.

VehicleFailureEvent Properties

	Type	Description
vehicleId	String	The Id of the Vehicle relevant in the event

RideRequestCompletedEvent *(inherits from SmartCarSystemEvent)*

The RideRequestCompletedEvent is used to notify observers once a Vehicle has successful completed a RideRequest. This event is usually processed by the Controller where it will coordinate next steps such as sending the Vehicle back to a high traffic area or marking it as available to execute another RideRequest.

RideRequestCompletedEvent Properties

	Type	Description
rideRequestId	String	The Id of the RideRequest relevant in the event

Subject <<Interface>>

Implementation of the Subject of a standard Observable pattern.

Subject Methods

Signature	Description
# notify(event: SmartCarSystemEvent): Void	Method used to Notify observers of an event
register(observer: Observer): Void	Method used to register new observers to this subject
unregister(observer: Observer): Void	Method used to unregister observers from this event

Subject Associations

Name	Type	Description
observers	Observer[]	Collection of all the observer that will be notified of events

Enumerations

Name	**Values**
RequestStatusEnum	“requested”, “offerAvailable”, “offerSelected”, “waitingToStart”, “inProgress”, “completed”, “cancelled”, “failed”
VehicleTypeEnum	“compact”, “sedan”, “compact_suv”, “suv”
RequestUrgencyEnum	“low”. “medium”, “high”, “critical”
TrimLevelEnum	“premium”, “luxury”, “base”
OperationalStatusEnum	“ready”, “busy”, “broken”, “OutOfService”
ProblemReportTypeEnum	“vehicle”: Denotes a problem with a Vehicle “rider”: Denotes a problem with a Rider “system”: Denotes a problem with the Smart Car System “app”: Denotes a problem with the Mobile App UI
ProblemReportSeverityEnum	“minor”, “major”, “critical”
ReviewTypeEnum	“vehicle”, “riderProfile”
DirectionEnum	“N”,”NW”,”NE”,”S”,”SW”,”SE”

IntegrationBroker

The integration broker, while not extensively documented in the design, plays a critical role in the functioning of the Smart Car Model Service. The integration broker is responsible for making an HTTP proxy available for use by both the Mobile GUI and the Admin GUI that way those GUIs can communicate with the Smart Car System via standard HTTP RESTful patterns.

3.7 Implementation Details

The implementation of the Smart Car Model Service is relatively simple and relies on a few key design patterns to get the job done.

- The Observable pattern is used to allow the Smart Car Model Service to communicate with the Smart Car System Controller while avoiding circular dependencies since the Controller needs access to the Model Service
- The Integration Broker pattern is used to provide and RESTful HTTP interface to other components in the system, especially the Mobile GUI and the Admin GUI

3.8 Exception Handling

Exception handling is done using standard thrown errors and two more specific errors meant to help with debugging and error reporting.

SmartCarModelServiceException

This exception is thrown for errors that occur in the Command and concrete command classes.

SmartCarModelServiceException Properties

Name	Type	Description
className	String	Name of the class where the exception was thrown
reason	String	Reason for the exception

SmartCarModelServiceAuthException

This exception is thrown for errors that occur in the Smart Car Model Component but are related to authentication and authorization.

SmartCarModelServiceAuthException Properties

Name	Type	Description
className	String	Name of the class where the exception was thrown
reason	String	Reason for the exception

3.9 Testing

Teasing should be conducted using a mixture of automated integration and unit tests. This component takes advantage of using a database as well as an outside vendor for mapping and routing. Unit tests can mock the database and the vendor. However, integration tests should also be in place in order to verify that the component works when interfacing with the database and with the maps vendor API. Testing should be conducted before each code merge into the code pipeline and while new releases progress through each environment in the code pipeline. Integration tests can be conducted by creating a test script that uses the integration broker to send RESTful requests to the Model Service then querying the database to verify operations were executed successfully.

3.10 Risks

One of the biggest risks with the current design of the Smart Car Model Service is how it relies on the Controller service in order to complete some mutations on its models. While it makes sense in the spirit of the single responsibility principal, separating some of the functionality of the Smart Car System between the Model and Controller Services could create some issues if one of those components were to fail or if the connection between the somehow two got severed. One example would be the impact on Vehicle updates to Vehicle.location in the event that the Smart Car Controller Service was unreachable. If this were to happen, then real time Vehicle locations would still be properly recorded but Vehicle .locationHistory would no longer be accurate since it would be missing readings. I don't see a resolution to this problem since it seems like it's a natural occurrence due to the design of the observer pattern. Theres really no way for the Model Service to know if the Controller Service is listening and ready since the Model Service is supposed to be indifferent and decoupled from the Controller Service. I think the only solution to this is for developers and administrators to be cognizant of the design and risk and to develop features accordingly.

4. Smart Car System Controller Design Document

4.1 Introduction

The Smart Car System Controller is responsible for coordinating all of the real time activity that happens in the Smart Car System. One example of this is how the Controller responds to RideRequests changes. Once a RideRequest is updated with an “acceptanceDate,” the Controller receives an event which acts a notification to let the Controller know to send a particular Vehicle to the Riders Location and to set the RideRequest status accordingly. The Controller coordinates with the Smart Car System Authentication Service in order to get an AuthToken so it can access the Model Service securely.

4.2 Requirements

Handle the ingestion of notifications

- Implements the Observable pattern to enable the Controller Service to subscribe to events that occur on the Smart Car Model Service

Create and execute commands

- Implements the Command pattern to enable the Controller Service to queue the execution of responses to event notifications
- Create Commands based on the event received where the Command contains appropriate execution logic to respond to the event
- If appropriate as a event response, executes changes on models or sends out control messages to models via the Smart Car Model Service

Execute Transactions in the Ledger Service

- If appropriate as and event response, creates and executes Blockchain transactions by using the Ledger Service
- Appropriately handles successful transactions
- Appropriately handles failed transactions

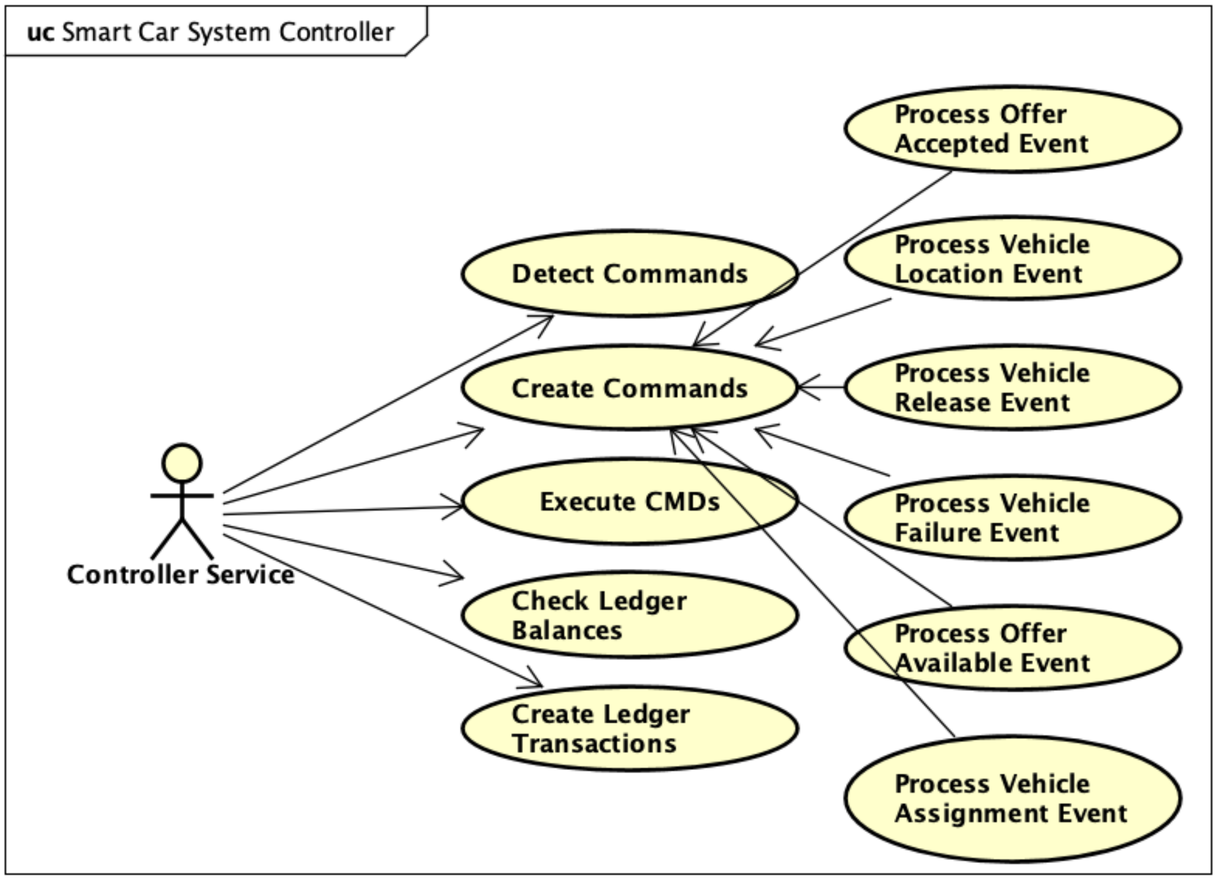
Security using the Authentication Service

- Must use the Auth Service and AuthTokens for authentication and authorization
- AuthTokens must be provided as args per execution and be verified at the time the method is called
- Must use Controller Auth Token for execution of commands
- May use pass through AuthToken for execution of methods on other modules

4.3 Use Cases

Actors

- **Controller Service:** The Controller Service is the main actor in the Smart Car Controller Service module. It is the main user of its own functionality and features since its main job is to listen for observable subjects notifications and to process commands based on what is hears.


uc Smart Car System Controller
Controller Service
Detect Commands
Create Commands
Execute CMDs
Check Ledger Balances
Create Ledger Transactions
Process Offer Accepted Event
Process Vehicle Location Event
Process Vehicle Release Event
Process Vehicle Failure Event
Process Offer Available Event
Process Vehicle Assignment Event

Above is the use case diagram which outlines what the expected actors are and what abilities they'll have in the Smart Car System Controller

Detect events

- The Smart Car Controller Services update() method is called by the Smart Car Model Service in order to notify the Controller that an event has just occurred

Create Commands

- A concrete command is created for each event type below. All concrete commands must include an execute method that performs the commands logic.
  - **Process Offer Accepted Event**
    - Get the RideRequest from the Smart Car Model Service using the provided rideRequestId
    - Get the Vehicle from the Smart Car Model Service using the provided vehicleId
    - Get the RiderProfile from the RideRequests association at .rider
    - Update the Vehicles operationalStatus to "Busy"
    - Call Vehicle.initiateService() for the Vehicles embedded system to process the .currentRideRequest and execute it.
    - Update the RideRequest status to "inProgress"
  - **Process Vehicle Location Event**
    - Get the Vehicle from the Smart Car Model Service using the provided vehicleId
    - Create a new HistoricalLocation
    - Get the Vehicles last previous HistoricalLocation

    - Calculate the Vehicle speed and direction based on the the current Vehicle.location v last previous HistoricalLocation
    - Update the Vehicle with the new HistoricalLocation added to its locationHistory collection

  - **Process Vehicle Release Event**
    - Get the Vehicle from the Smart Car Model Service using the provided vehicleId
    - If completeCurrentRideRequest is False, call haltService on the Vehicle to stop the Vehicle
    - If completeCurrentRideRequest is True, call haltServiceAfterRideRequest on the Vehicle to stop the Vehicle after is completed its current RideReqeust if it has one in progress
    - Update the Vehicle so that its operationalStatus is "OutOfService"

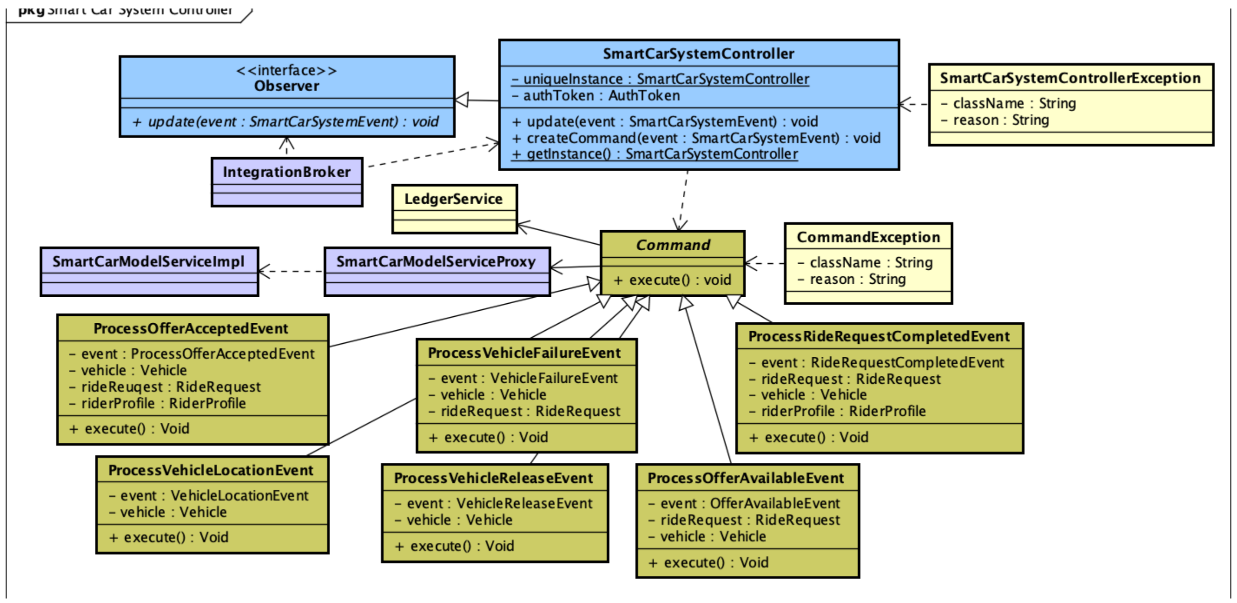
  - **Process Vehicle Failure Event**
    - Get the Vehicle from the Smart Car Model Service using the provided vehicleId
    - Call haltService on the Vehicle to stop the Vehicle
    - Update the Vehicle so that its operationalStatus is "OutOfService"
    - Get the RideRequest from its Vehicle association on .currentRideRequest
    - Update the RideRequest.status to "failed"
    - Calculate Route to service location
    - Call Vehicle.executeRoute() with the service location route in order to bring the car in for service

  - **Process Offer Available Event**
    - Get the RideRequest from the Smart Car Model Service using the provided rideRequestId
    - Get the Vehicle from the Smart Car Model Service using the provided vehicleId
    - Update Vehicle currentRideRequest to the to the RideRequest matching the rideRequestId
    - Update Vehicles operationalStatus to "WaitingAcceptance"

  - **Process RideRequest Completed Event**
    - Get the RideRequest from the Smart Car Model Service using the provided rideRequestId
    - Get the Vehicle from the Smart Car Model Service via RideRequest.vehicle
    - Get the RiderProfile from the Smart Car Model Service via RideRequest.rider
    - Update Vehicle .currentRideRequest null and its .operationalStatus to "Ready"
    - Create and execute the blockchain Transaction from the RiderProfile blockchain account to the Vehicle account for the amount specified in the RideRequest
    - Update the RiderProfile with their new Blockchain account balance

- **Execute Commands**
  - With the current implementation, Commands must be executed immediately after they're instantiated
  - Run the execute function on the concrete command to let the command execute the appropriate logic in repose to the event
  - Handle CommandExceptions and report them a ControllerServiceExceptions

4.4 Class Diagram


pkg Smart Car System Controller
<<interface>> Observer
+ update(event : SmartCarSystemEvent) : void
SmartCarSystemController
- uniqueInstance : SmartCarSystemController
- authToken : AuthToken
+ update(event : SmartCarSystemEvent) : void
+ createCommand(event : SmartCarSystemEvent) : void
+ getInstance() : SmartCarSystemController
SmartCarSystemControllerException
- className : String
- reason : String
IntegrationBroker
LedgerService
Command
+ execute() : void
CommandException
- className : String
- reason : String
SmartCarModelServiceImpl
SmartCarModelServiceProxy
ProcessOfferAcceptedEvent
- event : ProcessOfferAcceptedEvent
- vehicle : Vehicle
- rideReuqest : RideRequest
- riderProfile : RiderProfile
+ execute() : Void
ProcessVehicleFailureEvent
- event : VehicleFailureEvent
- vehicle : Vehicle
- rideRequest : RideRequest
+ execute() : Void
ProcessRideRequestCompletedEvent
- event : RideRequestCompletedEvent
- rideRequest : RideRequest
- vehicle : Vehicle
- riderProfile : RiderProfile
+ execute() : Void
ProcessVehicleLocationEvent
- event : VehicleLocationEvent
- vehicle : Vehicle
+ execute() : Void
ProcessVehicleReleaseEvent
- event : VehicleReleaseEvent
- vehicle : Vehicle
+ execute() : Void
ProcessOfferAvailableEvent
- event : OfferAvailableEvent
- rideRequest : RideRequest
- vehicle : Vehicle
+ execute() : Void

Above is the class diagram outlining the design and relation of all the classes in the Smart System Controller

4.5 Class Dictionary

SmartCarSystemController (implements Observer)

The SmarCarSystemController coordinates each response that happens as a result of an event in the Smart Car System. In oder for this to happen, the SmartCarSystemController implements part of the observer pattern by adhering the an observer interface and registering itself with the SmartCarModelService observable subject on bootstrap after which the SmartCarSystemController will received notifications of every event that occurs in the system.

SmartCarSystemController Methods

Signature	Description
update(event: SmartCarSystemEvent): Void	Interfaces with observable subjects and will receive an event that it will use to make a Command with .createCommand
createCommand(event: SmartCarSystemEvent): Void	Creates a concrete command based on the sub class of the SmartCarSystemEvent that's passed in.
getInstance(): SmartCarSystemController	Returns a singleton of the SmartCarSystemController or creates one if it doesn't exist

SmartCarSystemController Properties

Name	Type	Description
authToken	AuthToken	Stores the Controllers AuthToken that's fetched from AuthService on bootstrap
uniqueInstance	SmartCarSystemController	Stores SmartCarSystemController for the singleton pattern

Command

The Command is the parent class of the Concrete Commands in the SmartCityControllerService. There aren't many shared functions between the event responses/concrete commands so the functionality of the Command right now is limited to allowing access to outside services. The execute method is present on the Command and throws an error in order to encourage the implementation of execute method overrides in the concrete command subclasses. The LedgerService singleton is available in the Command in order to enable concrete commands to create blockchain ledger transactions. The SmartCarModelServiceProxy is also available in order to let concrete commands access operations provided by the SmartCarModelService via integration broker.

Command Methods

Signature	Description
execute(): Void	Throws error if execute() has not been overridden by child class

Command Associations

Name	Type	Description
SmartCarModelServiceProxy	SmartCarModelServiceInterface	For proxy (HTTP) access to operations in the SmartCarModelService
LedgerService	LedgerService	For access to operations in the LedgerService singleton

ProcessOfferAcceptedEvent (extends Command)

This Command handles the Smart Car Systems response to a RideRequestOffer being accepted.

ProcessOfferAcceptedEvent Methods

Type	Description
execute(): Void	Update the RideRequest to inProgress, the Vehicle status to busy and make sure the Vehicle is assigned to that RideRequest

ProcessOfferAcceptedEvent Properties

Name	Type	Description
event	OfferAcceptedEvent	Event that occurs after a RideRequest offer is accepted
vehicle	Vehicle	The Vehicle assigned to execute that RideRequest
rideRequest	RideRequest	The RideRequest to be executed
riderProfile	RiderProfile	The RiderProfile of the rider who ordered the RideRequest

ProcessVehicleLocationEvent (extends Command)

This Command handles the Smart Car Systems response to the location property being updated on a Vehicle.

ProcessVehicleLocationEvent Methods

Signature	Description
execute(): Void	Create a new HistoricalLocation and store it on the Vehicle for record keeping

ProcessVehicleLocationEvent Associations

Name	Type	Description
event	VehicleLocationEvent	Event that occurs after a Vehicle location is updated
vehicle	Vehicle	Vehicle whose location has been updated

ProcessVehicleReleaseEvent (extends Command)

This Command handles the Smart Car Systems response to a Vehicle being directed to halt its current service and do another task.

ProcessVehicleReleaseEvent Methods

Signature	Description
execute(): Void	Determine whether or not to complete the currentRideRequest before halting service then set status as “outOfService”

ProcessVehicleReleaseEvent Properties

Name	Type	Description
event	VehicleReleaseEvent	Event that occurs once a Vehicle has been released from a RideRequest
vehicle	Vehicle	Vehicle whose been released

ProcessVehicleFailureEvent (extends Command)

This Command handles the Smart Car Systems response to when a Vehicle experiences a failure.

ProcessVehicleFailureEvent Methods

Signature	Description
execute(): Void	Halt service on the Vehicle and mark the Vehicle operationalStatus as “outOfService”, the RideRequest status as “failed” and send the Vehicle to a service center

ProcessVehicleFailureEvent Properties

Name	Type	Description
event	VehicleFailureEvent	Event that occurs once a Vehicle detects a failure
vehicle	Vehicle	Vehicle that has failed
rideRequest	RideRequest	RideRequest that could not be fulfilled successfully

ProcessOfferAvailableEvent (extends Command)

This Command handles the Smart Car Systems response to when a Vehicle capable of fulfilling a RideRequest has been found and set aside for the RideRequest rider.

ProcessOfferAvailableEvent Methods

Signature	Description
execute(): Void	Set the Vehicles currentRideRequest and its operationalStatus as “waitingAcceptance”

ProcessOfferAvailableEvent Properties

Name	Type	Description
event	OfferAvailableEvent	Event that occurs once a Vehicle offer is available for a RideRequest
vehicle	Vehicle	Vehicle related to the offer

Name	Type	Description
rideRequest	RideRequest	RideRequest that the offer is for

ProcessRideRequestCompletedEvent (extends Command)

This Command handles the Smart Car Systems response a Vehicle successfully completed a RideRequest.

ProcessRideRequestCompletedEvent Methods

Signature	Description
execute(): Void	Free the Vehicle from the RideRequest by clearing its currentRideRequest property, set the RideRequest status as “completed” and process the appropriate blockchain Ledger transaction

ProcessRideRequestCompletedEvent Properties

Name	Type	Description
event	RideRequestCompletedEvent	Event that occurs once a Vehicle completed a RideRequest
vehicle	Vehicle	Vehicle that completed the RideRequest that will get paid
rideRequest	RideRequest	RideRequest that has been completed
riderProfile	RiderProfile	Rider of the Vehicle who will pay for the ride

4.6 Implementation Details

The implementation for the Smart Car Controller Service is very similar to the implementation of the Smart City Controller Service. The design relies heavily around taking advantage of the observer and the command patterns. Nearly every function in the Smart Car Controller Service is triggered after receiving an update notification from a subject that is being listened to by the Smart Car Controller Service. At a high level, the Controller Service is responsible for executing many of the side effects of operations that happen in observer subjects.

If you include an updated Location while updating a Vehicle via the Smart Car Model Service then the Vehicles .location will be mutated, this is the expected behavior out of any update method. A side effect also occurs but that’s left out of the Smart Car Model Service update method in order to help it adhere to the single responsibility principal which would require it to only update the Vehicle. Instead, the side effect is processed by the Smart Car Controller Service after it is notified via a VehicleLocationEvent that the Vehicles .location has change. Once notification is received, the Smart Car Controller Service will calculate a new HistoricalLocation and mutate the Vehicle .locationHistory to reflect a most up to date collection. This is the design flow I tried to stick with during the implementation of the Smart Car Controller Service.

4.7 Exception Handling

Exception handling is done using standard thrown errors and two more specific errors meant to help with debugging and error reporting.

CommandException

This exception is thrown for errors that occur in the Command and concrete command classes.

CommandException Properties

Name	Type	Description
className	String	Name of the class where the exception was thrown
reason	String	Reason for the exception

SmartCarSystemControllerException

This exception is thrown for errors that occur in the SmartCarSystemController class.

SmartCarSystemControllerException Properties

Name	Type	Description
className	String	Name of the class where the exception was thrown
reason	String	Reason for the exception

4.8 Testing

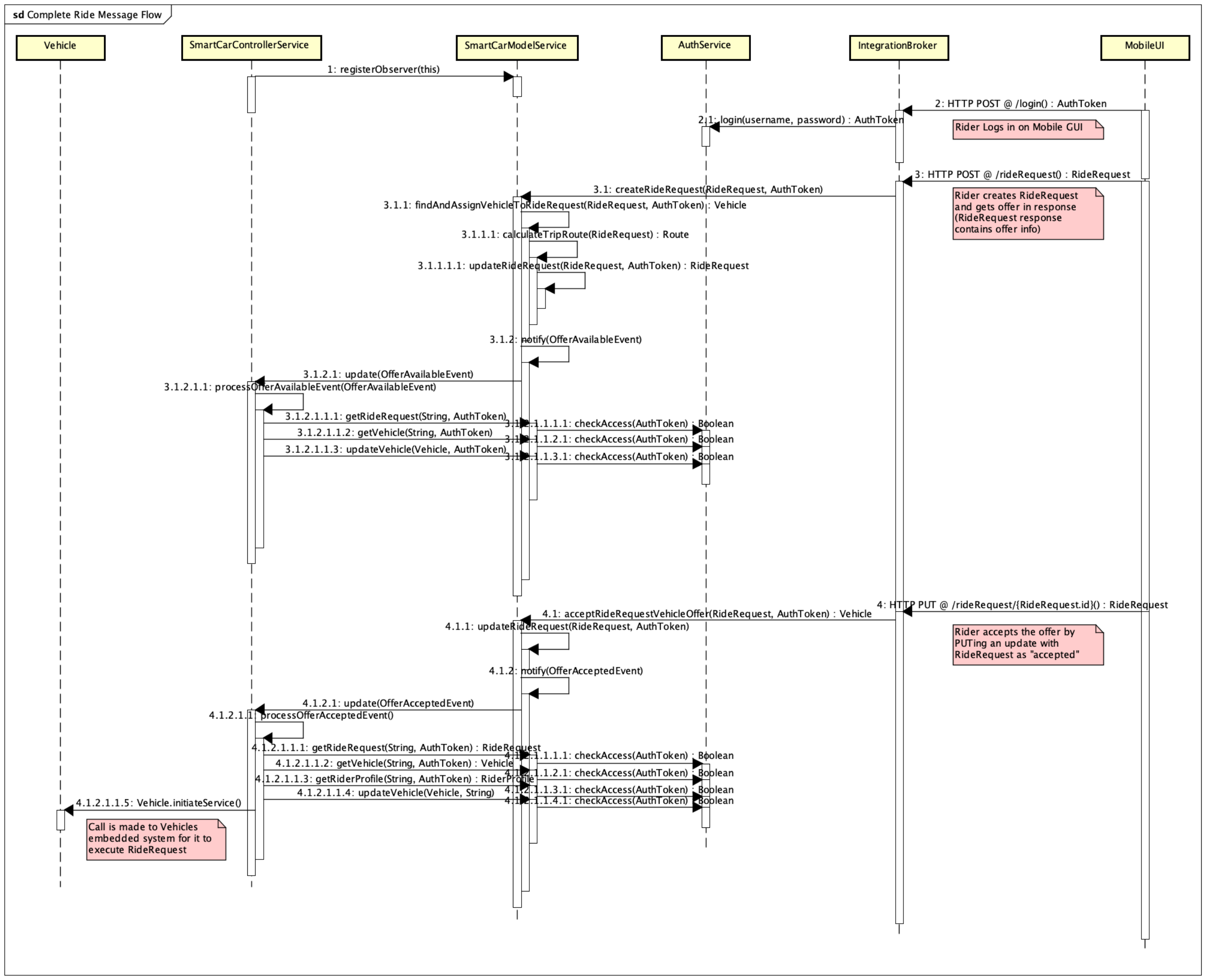
Testing for the Smart Car Controller Service is a bit more complicated than tests for the Smart Car System Model Service since many of the features on the Controller Service require access to the Smart Car System Model Service. Unit tests would not be able to test much, so integration tests are preferred for the Controller Service. Once a test environment is set up, a test script can be created that can uses the integration broker to RESTfully send requests to the Smart Car System Model Service. These should be requests that expect side effects from the Controller Service. Once the request is returned successfully, the test suite should verify that the side effects did take place by querying the database. Testing should be conducted before each code merge into the code pipeline and while new releases progress through each environment in the code pipeline.

4.9 Risks

One of the largest risks of Controller Service design is in the possible queueing of Commands. The Command pattern allows for queueing commands and for those commands to be executed at a later date. This is not an implemented feature right now but can be implemented at a later date. I think the risk with implementing this is in what might happen in the event of a critical failure while commands are queued. If Commands are stored in memory and a critical Controller component failure occurs then those Commands would be lost and data integrity issues could occur since Commands would never end up executing. One way to solve this could be to persists Commands temporarily in a database until they're executed that way there's a record of what Commands still need to be execute in the even that the component needs to be reinstantiated.

5. Other Diagrams

5.1 Ride Message Flow (Ride Message Flow.png)

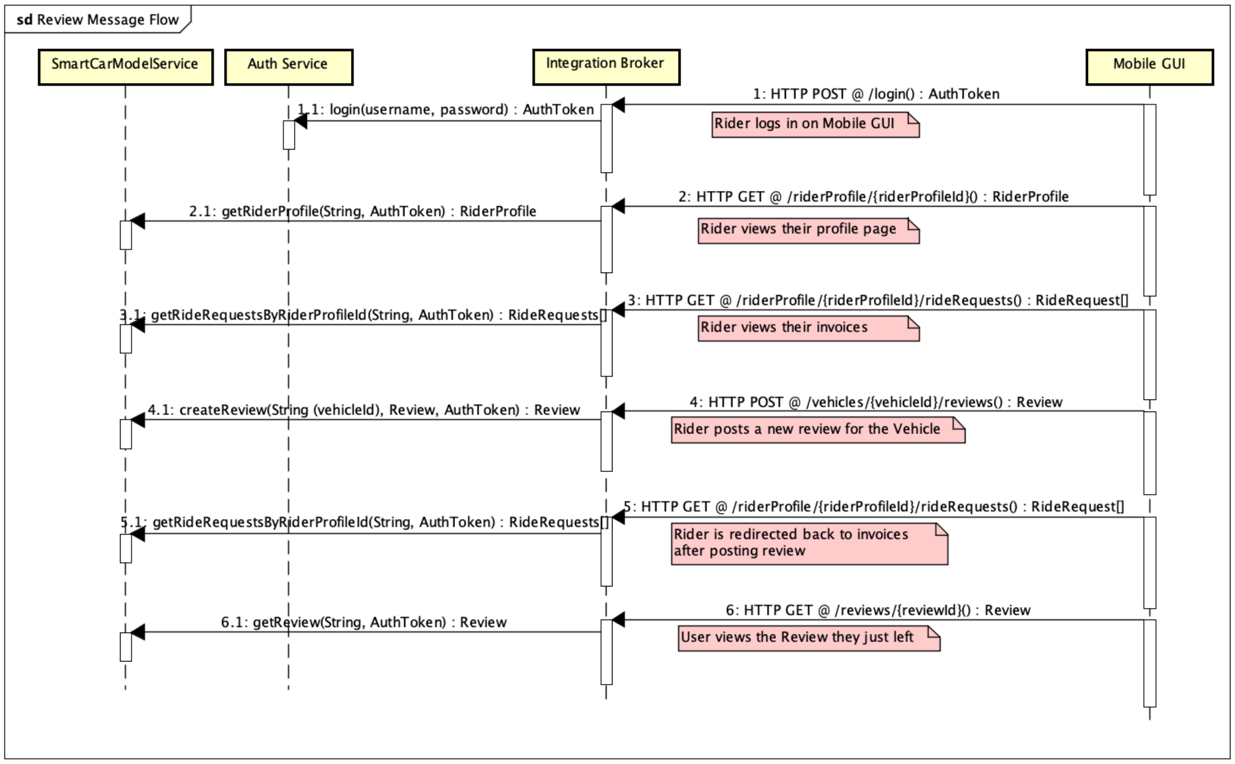


sd Complete Ride Message Flow
Vehicle
SmartCarControllerService
SmartCarModelService
AuthService
IntegrationBroker
MobileUI
1: registerObserver(this)
2: HTTP POST @ /login() : AuthToken
2.1: login(username, password) : AuthToken
Rider Logs in on Mobile GUI
3: HTTP POST @ /rideRequest() : RideRequest
3.1: createRideRequest(RideRequest, AuthToken)
Rider creates RideRequest and gets offer in response (RideRequest response contains offer info)
3.1.1: findAndAssignVehicleToRideRequest(RideRequest, AuthToken) : Vehicle
3.1.1.1: calculateTripRoute(RideRequest) : Route
3.1.1.1.1: updateRideRequest(RideRequest, AuthToken) : RideRequest
3.1.2: notify(OfferAvailableEvent)
3.1.2.1: update(OfferAvailableEvent)
3.1.2.1.1: processOfferAvailableEvent(OfferAvailableEvent)
3.1.2.1.1.1: getRideRequest(String, AuthToken)
3.1.2.1.1.1.1: checkAccess(AuthToken) : Boolean
3.1.2.1.1.2: getVehicle(String, AuthToken)
3.1.2.1.1.2.1: checkAccess(AuthToken) : Boolean
3.1.2.1.1.3: updateVehicle(Vehicle, AuthToken)
3.1.2.1.1.3.1: checkAccess(AuthToken) : Boolean
4: HTTP PUT @ /rideRequest/{RideRequest.id}() : RideRequest
4.1: acceptRideRequestVehicleOffer(RideRequest, AuthToken) : Vehicle
Rider accepts the offer by PUTing an update with RideRequest as "accepted"
4.1.1: updateRideRequest(RideRequest, AuthToken)
4.1.2: notify(OfferAcceptedEvent)
4.1.2.1: update(OfferAcceptedEvent)
4.1.2.1.1: processOfferAcceptedEvent()
4.1.2.1.1.1: getRideRequest(String, AuthToken) : RideRequest
4.1.2.1.1.1.1: checkAccess(AuthToken) : Boolean
4.1.2.1.1.2: getVehicle(String, AuthToken) : Vehicle
4.1.2.1.1.2.1: checkAccess(AuthToken) : Boolean
4.1.2.1.1.3: getRiderProfile(String, AuthToken) : RiderProfile
4.1.2.1.1.3.1: checkAccess(AuthToken) : Boolean
4.1.2.1.1.4: updateVehicle(Vehicle, String)
4.1.2.1.1.4.1: checkAccess(AuthToken) : Boolean
4.1.2.1.1.5: Vehicle.initiateService()
Call is made to Vehicles embedded system for it to execute RideRequest

The ride message flow is a relatively complex flow but its a great example to show the interaction between all of the components of the Smart Car System. The first step of the process involves the Rider using the Mobile GUI to login in in order to receive an AuthToken. The riders Mobile GUI then includes that AuthToken in all subsequent RESTful requests sent to the integration broker including the create RideRequest which is initiated next. As a result of that request, a capable Vehicle is found and a response is sent to the Mobile GUI, this response includes a mutated versions of the original RideRequest which now includes the Vehicle information of the newly assigned Vehicle. This is the Vehicles RideRequest offer. To accept the offer, the Rider/Mobile GUI updates the RideOffers status and acceptedOnDate through the integration broker executing acceptRideRequestVehicleOffer on the Model

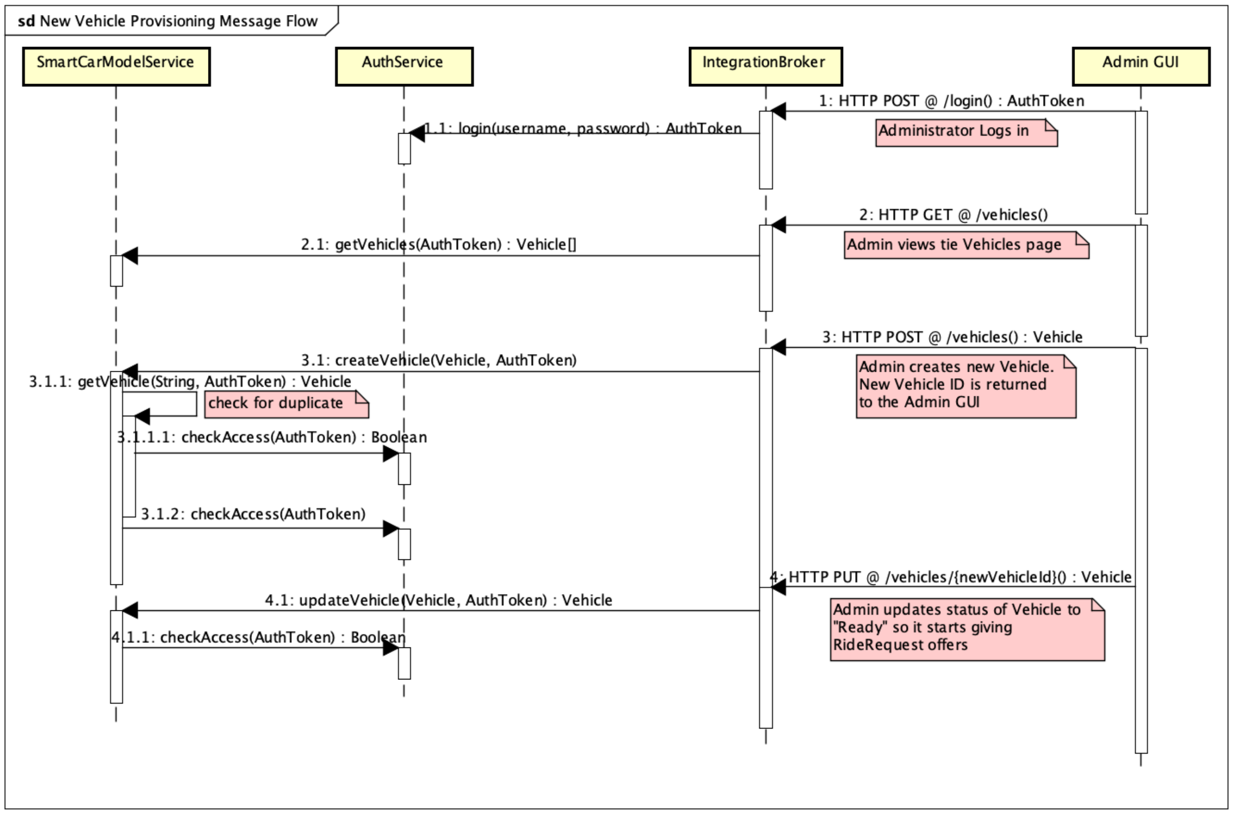
Service. Once this is done, an event is triggered and the Controller Service tells the Vehicle to being initiating service for that RideRequest. AuthTokens are verified through this flow by using the checkAccess method in the Authentication Service.

5.2 Review Message Flow (Review Message Flow.png)


sd Review Message Flow
SmartCarModelService
Auth Service
Integration Broker
Mobile GUI
1: HTTP POST @ /login() : AuthToken
1.1: login(username, password) : AuthToken
Rider logs in on Mobile GUI
2: HTTP GET @ /riderProfile/{riderProfileId}() : RiderProfile
2.1: getRiderProfile(String, AuthToken) : RiderProfile
Rider views their profile page
3: HTTP GET @ /riderProfile/{riderProfileId}/rideRequests() : RideRequest[]
3.1: getRideRequestsByRiderProfileId(String, AuthToken) : RideRequests[]
Rider views their invoices
4: HTTP POST @ /vehicles/{vehicleId}/reviews() : Review
4.1: createReview(String (vehicleId), Review, AuthToken) : Review
Rider posts a new review for the Vehicle
5: HTTP GET @ /riderProfile/{riderProfileId}/rideRequests() : RideRequest[]
5.1: getRideRequestsByRiderProfileId(String, AuthToken) : RideRequests[]
Rider is redirected back to invoices after posting review
6: HTTP GET @ /reviews/{reviewId}() : Review
6.1: getReview(String, AuthToken) : Review
User views the Review they just left

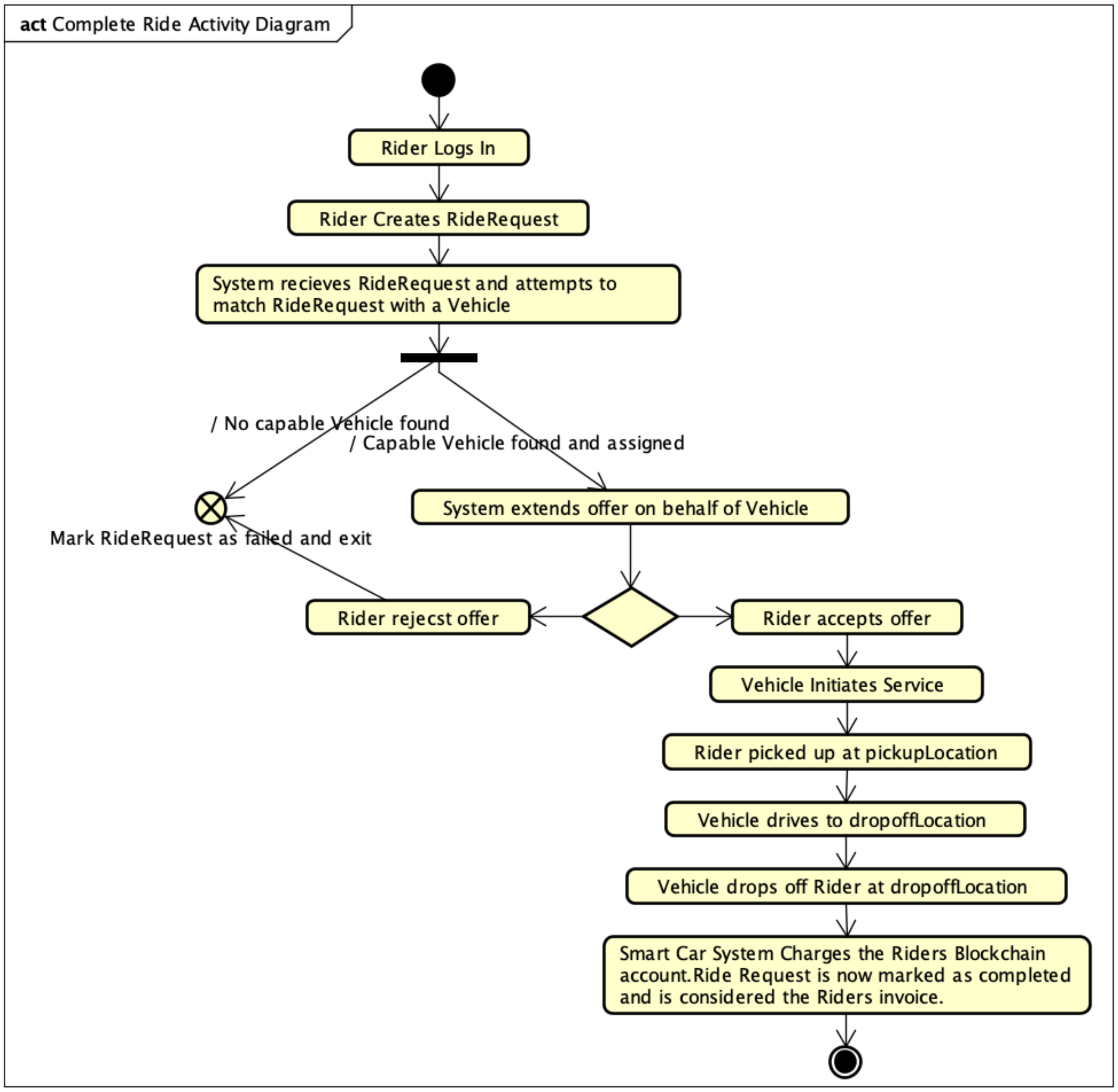
The review message flow is a pretty simple flow since it only uses standard CRUD RESTful operations with no sophisticated side effects from the Controller Service. To create review, the Rider must login, view their profile to get to their invoices/RideRequests and then via a RideRequest/invoice screen the rider is able to post a review. In this example flow, the rider is redirected back to the invoices/ RideRequests screen after they leave the review. Each of these actions, done through the Mobile GUI and with the help of the integration broker, are translatable to a single RESTful HTTP request. AuthTokens are verified through this flow by using the checkAccess method in the Authentication Service.

5.3 New Vehicle Provisioning Message Flow (New Vehicle Provisioning Message Flow.png)


sd New Vehicle Provisioning Message Flow
SmartCarModelService
AuthService
IntegrationBroker
Admin GUI
1: HTTP POST @ /login() : AuthToken
Administrator Logs in
1.1: login(username, password) : AuthToken
2: HTTP GET @ /vehicles()
Admin views tie Vehicles page
2.1: getVehicles(AuthToken) : Vehicle[]
3: HTTP POST @ /vehicles() : Vehicle
Admin creates new Vehicle. New Vehicle ID is returned to the Admin GUI
3.1: createVehicle(Vehicle, AuthToken)
3.1.1: getVehicle(String, AuthToken) : Vehicle
check for duplicate
3.1.1.1: checkAccess(AuthToken) : Boolean
3.1.2: checkAccess(AuthToken)
4: HTTP PUT @ /vehicles/{newVehicleId}() : Vehicle
Admin updates status of Vehicle to "Ready" so it starts giving RideRequest offers
4.1: updateVehicle(Vehicle, AuthToken) : Vehicle
4.1.1: checkAccess(AuthToken) : Boolean

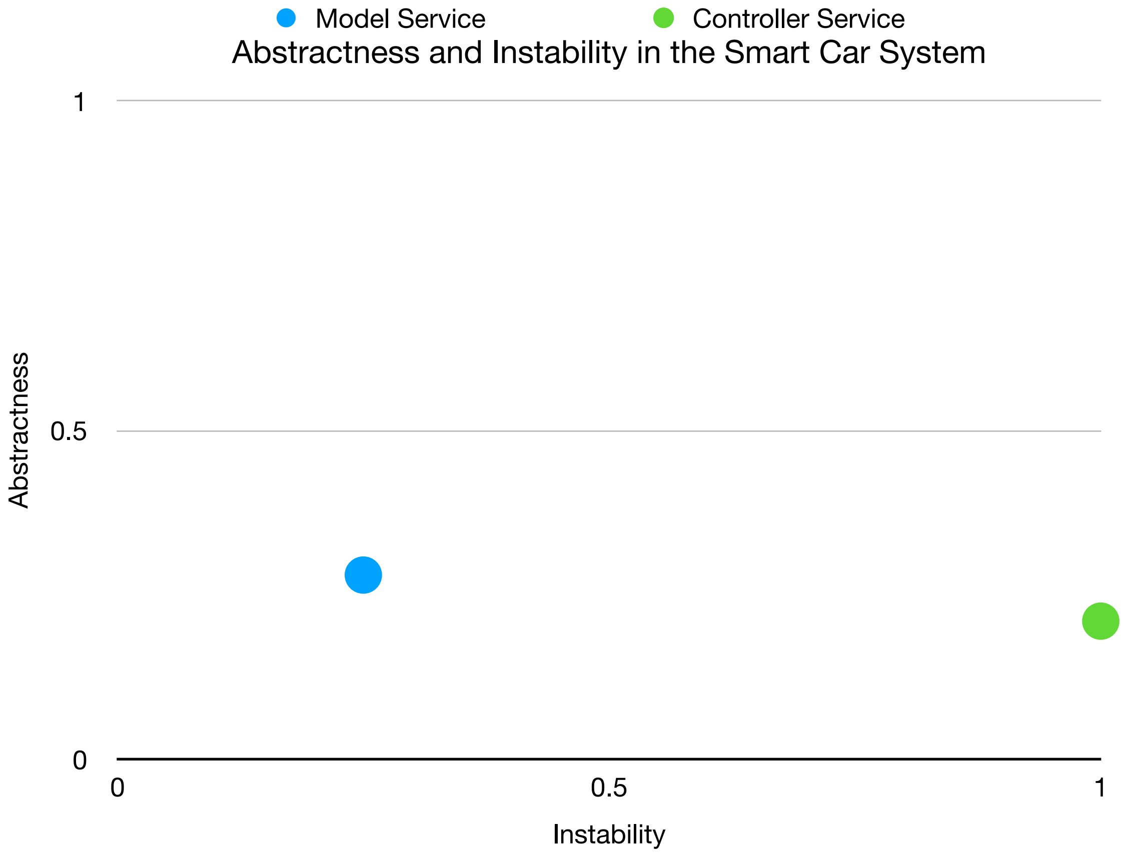
The new Vehicle provisioning message flow involved a new Vehicle being created by a logged in admin. After the new Vehicle is created the Vehicle is then update with a “ready” opreationalStatus which makes he Vehicle available to start executing RideRequests. AuthTokens are verified through this flow by using the checkAccess method in the Authentication Service.

5.4 Ride Activity Diagram (Ride Activity Diagram.png)


act Complete Ride Activity Diagram
Rider Logs In
Rider Creates RideRequest
System recieves RideRequest and attempts to match RideRequest with a Vehicle
/ No capable Vehicle found
/ Capable Vehicle found and assigned
System extends offer on behalf of Vehicle
Mark RideRequest as failed and exit
Rider rejecst offer
Rider accepts offer
Vehicle Initiates Service
Rider picked up at pickupLocation
Vehicle drives to dropoffLocation
Vehicle drops off Rider at dropoffLocation
Smart Car System Charges the Riders Blockchain account.Ride Request is now marked as completed and is considered the Riders invoice.

The Ride Activity Diagram represents the steps required to completed a ride. A Ride logs in and creates a RideRequest after which the Smart Car System either finds and capable Vehicle and proceeds or it fails to fulfill the RideRequest. If a capable Vehicle is found, then the rider can either accept the RideRequest offer and continue through the flow or reject the offer and start over later. If the rider accepts the offer the Vehicle executes the RideRequest and, upon completion of the RideRequest, a blockchain transaction is processed between the rider and the Vehicle.

5.5 Abstractness and Stability Metrics



After calculating the abstractness and instability of the modules in my design of the Smart Car System, I think some things could have been optimized in order to provide a more stable system. The Model Service is well within the “zone of pain,” which means that it's stable but its very concrete. While stability might be good for running the application, the components concreteness makes it less versatile and extensible. It might have been a good idea to implement more abstract classes in the Model Service in order to get the component out of the zone of pain. The Controller Service seems to have more issues. While it’s also low in abstractness it also scores a 1 on the instability metric. This may be due to the nature of the implementation of the observable pattern in the Controller Service and the required decoupling between it and the Model Service, so I’m unsure its instability can really be optimized much. However, its abstractness could be optimized through refactoring that could incorporate more interfaces and abstract classes into the Controller Service. The Controller Service is a closer approximation to the main sequence than the Model Service, so if I were to spend more time on the design I think I would start with increasing the stability and abstractness of the Model Service.