IS213 Enterprise Solution Development

G2T4

Report Submission

Cedric Lim Shao Ming

Choo Zheng Yang

Edwin Lee Kian Yong

Lee Xin Ying

Teo Qian Qi Fiona

Charmaine Tan Ruo Xuan

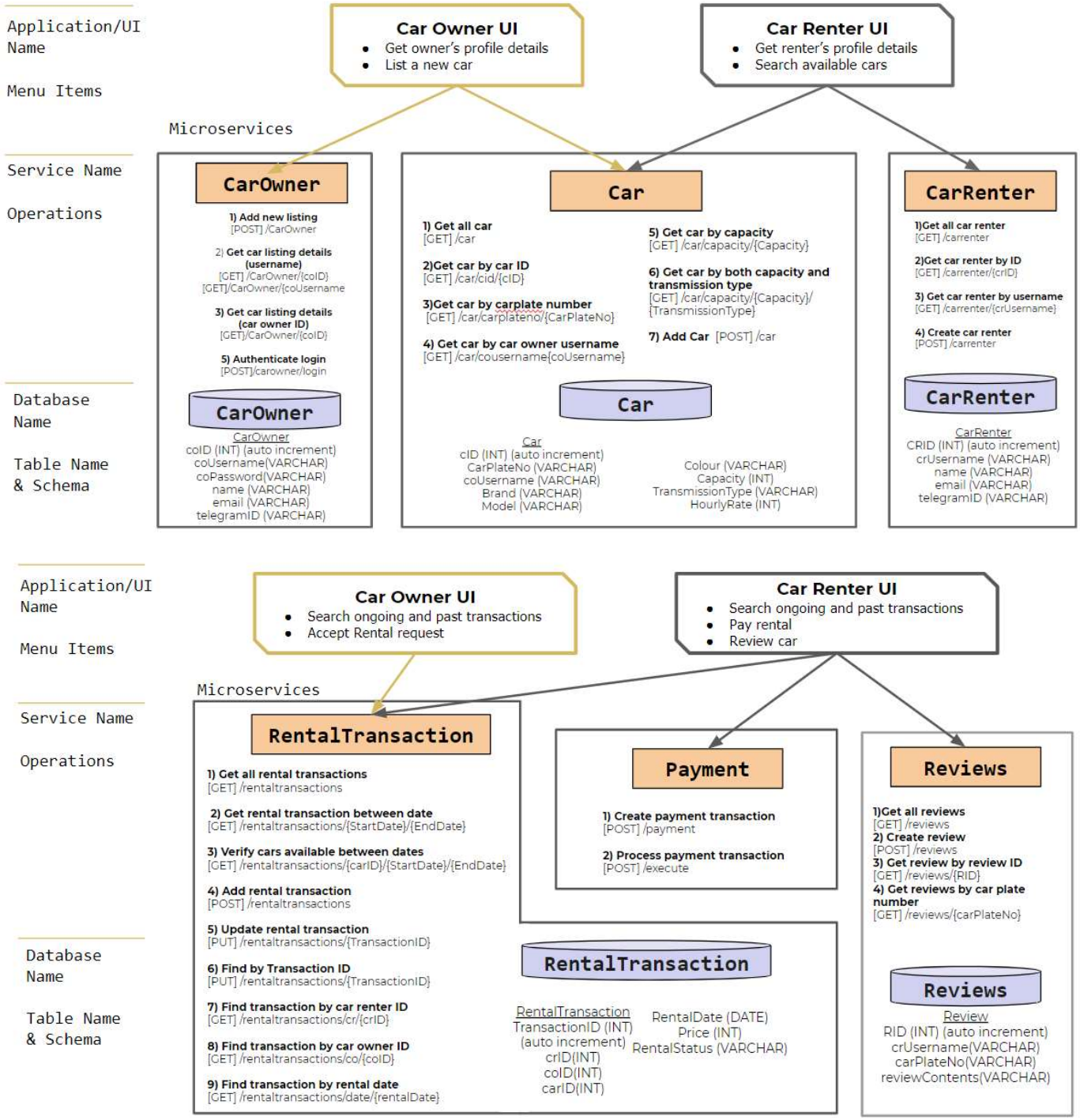
Introduction

Carma is a peer-to-peer car rental platform targeted to connect private vehicle owners with customers, known as car renters, looking to rent cars. As most car rental services provided in Singapore are from commercial companies, there are little to no services focused on private car owners looking to rent their cars out. Therefore, Carma bridges that gap in the market. In this report, we will be illustrating four different user scenarios:

1. Car Owner Lists Car for Rental
2. Car Renter Browses for Cars
3. Car Renter Confirms Car for Rental
4. Car Renter Makes Payment

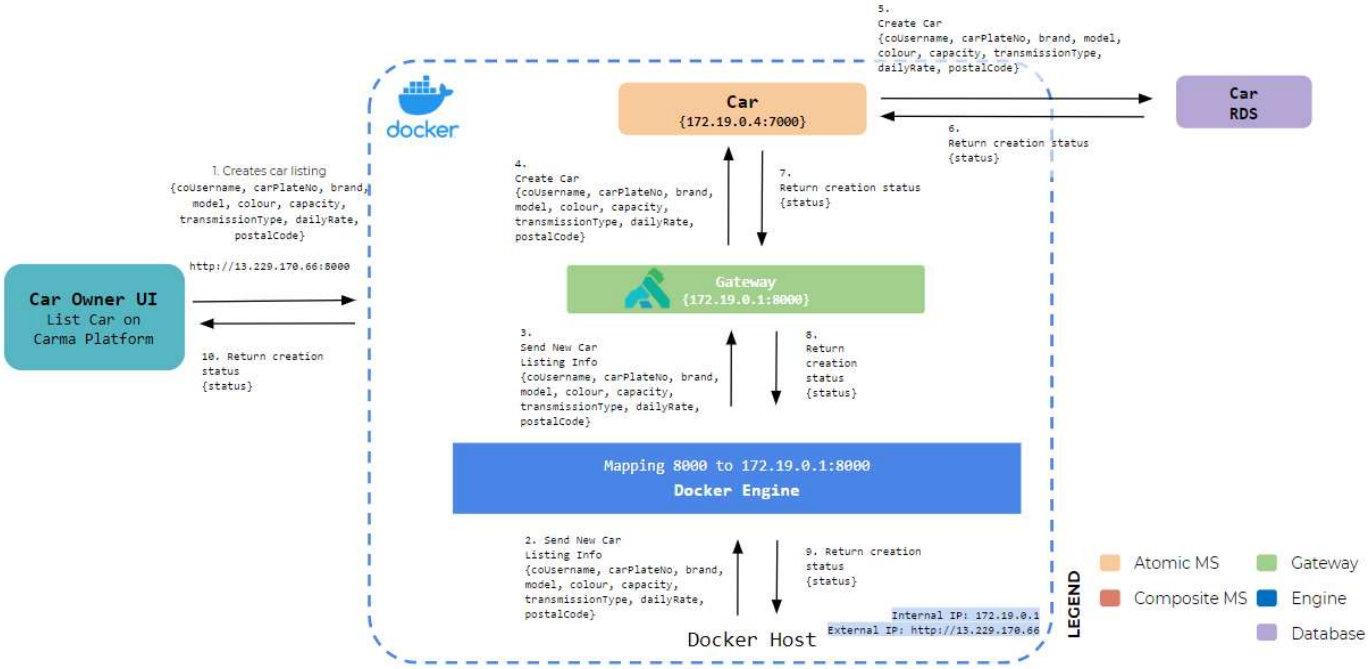
Using these scenarios, we aim to give a better understanding of how Carma works through the use of various microservices.

Technical Overview Diagram



User Scenarios

User Scenario 1: Car Owner Lists Car for Rental



1. Scenario assumes Car Owner has already logged in as user 'betaced'
2. Car Owner lists his car on his profile for rental on the UI, sent via HTTP POST.
3. The UI will invoke Car, an atomic microservice through Docker by sending the request through Port 8000.
4. Docker Host then routes the POST request to the API gateway's container on Port 8000.
5. The API Gateway, Kong, will then route the request to the Car microservice's container on Port 7000.
6. Car microservice will insert the new car information into the Car RDS database. If it's a successful insertion, a success status will be returned to the Car Owner UI, else, an error message will be shown on the UI.

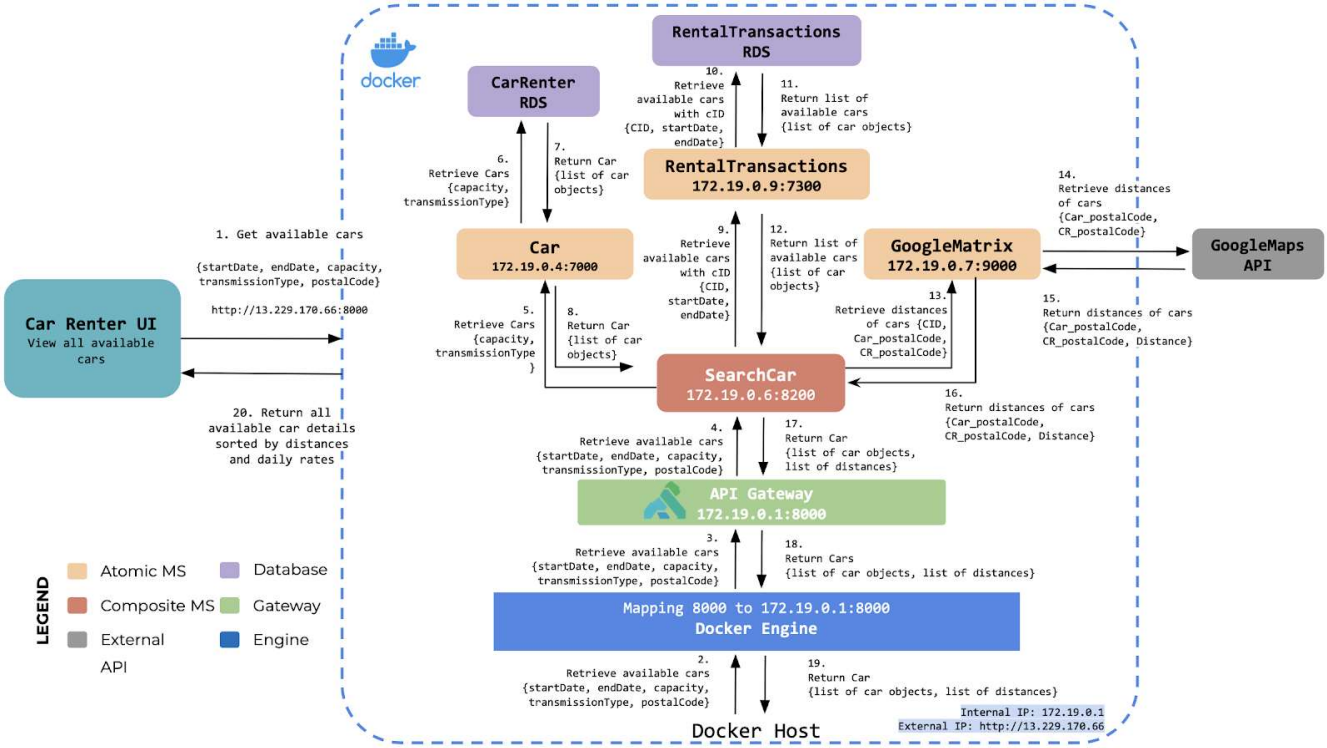
(Micro)Services

Service Name	Operational information	Description of the functionality	Input(if any)	Output(if any)
Car	[POST] /car	List a car on the Carma platform	{carPlateNo, brand, model, colour, capacity, transmissionType, dailyRate, postalCode}	If creation success: {car.json, status: 201} Else: {"A car with carPlateNo already exists", status: 400}

Beyond the Labs

1. Utilisation of API Gateway to access the Car microservice, instead of direct invocation of the service from UI
2. Storage of Car microservice's data on cloud with RDS MySQL database. Putting data on the cloud allows the server to retrieve and update the data easily.

User Scenario 2: Car Renter Browses for Cars



1. Assuming Car Renter has already logged in as user 'Cutie111'.
2. Car Renter uses the UI to retrieve available cars by invoking SearchCar microservice through Docker by sending a GET request via Port 8000.
3. Through Docker, it would route the request to the API gateway container mapped to Port 8000.
4. The API gateway, Kong, will then route the request to SearchCar, a composite microservice mapped to Port 8200.
5. Through the SearchCar microservice, it will first invoke Car, an atomic microservice, on Port 7000 to check for the cars that fit the capacity and transmission criteria.
6. The list of cars that fit the criteria will be sent back to SearchCar.
7. SearchCar will then send a request to RentalTransactions, an atomic microservice, on Port 7300 with the list of available cars based on the user's search criteria to check for its availability for booking.
8. After shortlisting the number of available cars, SearchCar will call GoogleMatrix on Port 9000, another atomic microservice, which would invoke an external API, GoogleMaps, to get the distance of the cars from the renter's specified postal code.
9. The results will be then sent back to the UI through Kong and Docker to the car renter. If there are no cars available, it will display a message for the renter as well.

(Micro)Services

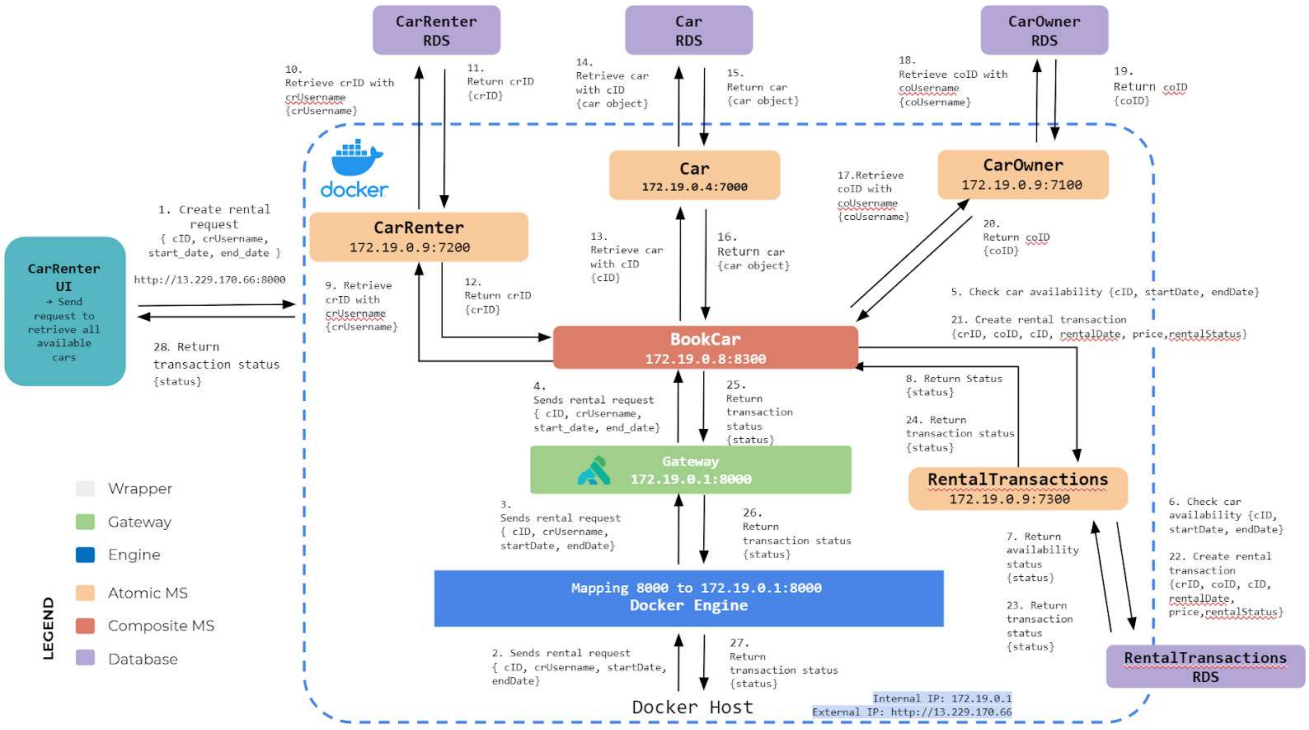
Service Name	Operational information	Description of the functionality	Input (if any)	Output (if any)
Search Car	[GET] /searchcar /<string:start_date> /<string:end_date> /<int:seating_capacity> /<string:transmissiontype> /<string:postalcode>	• Search for cars that matched the capacity, transmission type and date of availability • Interacts with Car microservice to retrieve cars that matched the capacity and transmission type • Interacts with rentalTransaction microservice to retrieve cars that's not available • Interacts with GoogleMatrix Wrapper to check for distance between locations	{start_date, end_date, capacity, transmissionType, postalCode}	{"car": availableCarList, "distance": List of distances}}

Car	[GET]/car/capacity /<int:seating_capacity> /<string:transmissiontype>	Get the cars that fits the search criterias of seating capacity and transmission type	{Capacity,TransmissionType}	{"car": availableCarList}
Rental Transactions	[GET] /rentaltransactions /<int:carID> /<date:StartDate> /<date:EndDate>	Get the available cars that fit the search criteria	{carID, StartDate, EndDate}	{"car": availableCarList}
GoogleMatrix Wrapper	[GET]getDistanceByPostalCodes<string:renterpostalcode>/<string:carpostalcode>	Get the distance of each car from the user	{renterpostalcode, carpostalcode}	{“distance”: List of distance}

Beyond the Labs

1. Utilisation of API Gateway to access the SearchCar microservice, instead of direct invocation from UI
2. Google Maps to retrieve the location of the cars from the user’s input location. We used the Google Maps wrapper that invokes an external API, Google Maps, to get the distance of the cars that matches the search query and the user’s location. The API takes in the 2 postal codes, one from the car’s location and another from the user’s location to generate the distance matrix.
3. Storage of CarRenter and RentalTransactions microservice’s data on cloud with RDS MySQL database. Putting data on the cloud allows the server to retrieve and update the data easily.
4. Creation of composite microservice, SearchCar, to interact with the atomic microservices such as Car and Rentaltransactions
5. Creation of wrapper microservice, GoogleMatrix, to invoke external API, Google Maps

User Scenario 3: Car Renter Confirms Car for Rental



1. Assuming Car Renter has already logged in as user ‘Cutie111’.
2. Car Renter choses a car he/she wants to rent and clicks on the “I want to rent this!” button.
3. Car Renter is taken to the booking page. When the page loads, BookCar composite microservice is activated.
4. Through Docker, it would route the request to the API gateway container mapped to Port 8000.
5. The API gateway, Kong, will then route to BookCar, a composite microservice, mapped to Port 8300.
6. In the BookCar microservice, it will first invoke the RentalTransactions microservice on Port 7300 to check if the car that the car renter wants to book is available according to the start and end dates.
7. After which, BookCar invokes CarRenter, an atomic microservice on Port 7200 to retrieve the car renter object and extract the crID, using the car renter’s username.

8. Next, BookCar will invoke Car atomic microservice on Port 7000 using the car ID to retrieve the car object.
9. BookCar will then invoke CarOwner microservice on Port 7100 using the car owner's username retrieved from the car object microservice. It will then retrieve the car owner's ID.
10. Lastly, BookCar will send all retrieved data to RentalTransactions to create a new rental transaction with the list of available cars based on the user's search criteria.
11. Upon successful booking, Car Renter will be able to view their current and past transactions on their profile page. The rental status of the car will thus be changed to reserved.

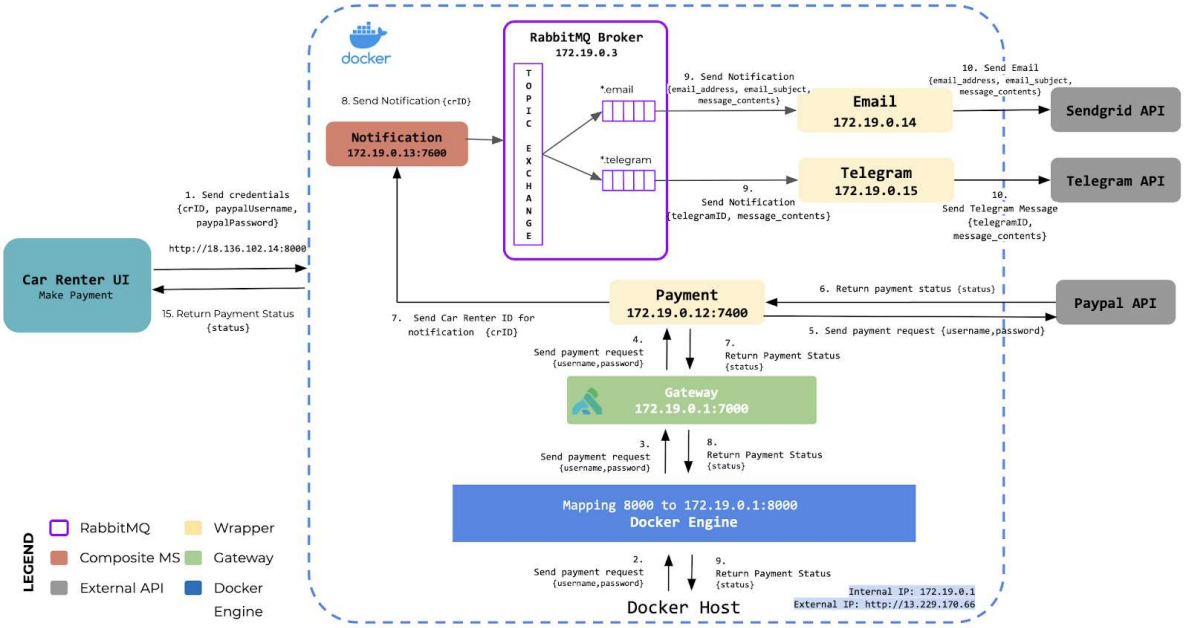
(Micro)Services

Service Name	Operational information	Description of the functionality	Input (if any)	Output (if any)
BookCar	[POST] /bookcar	Book a car	{cID, crUsername, startDate, endDate}	If creation success: {status: 201} Else: {"An error occurred creating the car.", status:500}
CarRenter	[GET] /carrenter	Used in the bookcar composite microservice to retrieve a car renter's ID number based on his/her username	{crUsername}	If successful: {carrenter.json, status: 201} Else: {"car renter ID not found.", status: 404}
Car	[GET] /car	Used in the bookcar composite microservice to retrieve a car object based on its carID. The daily rate and car owner's username will be extracted for further use.	{cID}	If successful: {car.json, status: 201} Else: {"Car not found.", status: 404}
CarOwner	[GET] /carowner	Used in the bookcar composite microservice to retrieve a car owner's ID number based on his/he username.	{coUsername}	If successful: {carowner.json, status: 201} Else: {"Car Owner not found.", status: 404}
RentalTransactions	[GET] /rentaltransactions & [POST] /rentaltransactions	2 use case in the bookcar composite microservie: A. Verifies if a car is available for rental transactions within a specified start & end date B. Adds a new rentaltransaction	For [GET]: {cID, startDate, endDate} For [POST]: {crID, coID, cID, rentalDate, price, rentalStatus}	[GET] If successful: {"Car is available", status: 200} Else: {"Car is not available.", status: 400} [POST] If successful: {rentaltransaction.json, status: 201} Else: {"An error occurred when creating Rental Transaction in DB.", status: 500}

Beyond the Labs

1. Utilisation of API Gateway to access the BookCar microservice, instead of direct invocation of the service from UI
2. Storage of each microservices' data on cloud with RDS MySQL database. This would allow the server to access, retrieve and update the data easily.
3. Creation of composite microservice, BookCar, to interact with the atomic microservices such as Car,CarRenter and CarOwner

User Scenario 4: Car Renter Makes Payment



1. Car Renter opens an external Paypal window by clicking on the Paypal button on the confirmation page of the UI and signs into his Paypal account to make payment.
2. Through Docker, it will route the request to the API Gateway to send a payment request to the Payment wrapper on Port 7400 to call the external Paypal API. A payment status will be returned and upon successful payment, a confirmation message will be displayed on the UI.
3. Concurrently, the Payment microservice will send the Car Renter ID to the Notification microservice on Port 7600. Notification microservice then sends the respective notification messages to the RabbitMQ broker to route to the queues for the Email and Telegram wrapper services to consume and call the SendGridAPI and Telegram API.

(Micro)Services

Service Name	Operational information	Description of the functionality	Input (if any)	Output (if any)
Payment	[POST] /payment	Send paypal credential to make payment	{username, password}	Return payment status {status}
RabbitMQ	Topic exchange	Send payment notification to Notification microservice	{crID}	
Notification		Send notification via function to Email and Telegram	{crID}	{TelegramID, email_address, email_subject, message}
Email		Send notification to car renter email	{email_address, email_subject, message}	
Telegram		Send notification to car renter Telegram ID	{TelegramID, message}	

Beyond the Labs

1. Utilisation of API Gateway to access the Payment microservice, instead of direct invocation from UI
2. Utilisation of external services: Paypal API for payment transactions, Telegram API and SendGrid API for sending notification to users via telegram and email.
3. Implementation of wrapper microservices such as Payment, Email and Telegram for interactions between Notification Microservice and the external APIs.
4. Creation of composite microservice, Notification, to interact with the wrapper microservices such as Email, Payment and Telegram.

Remaining Beyond the Labs not covered above

1. Deployment of microservices on AWS EC2 instance and decoupling data storage with the usage of RDS. They were shut down with the approval of Instructor Eng Kit after the demo due to AWS free tier limitation and costs.
2. Usage of docker-compose to build and pull multiple images using 1 file instead of manually running each individual docker file.