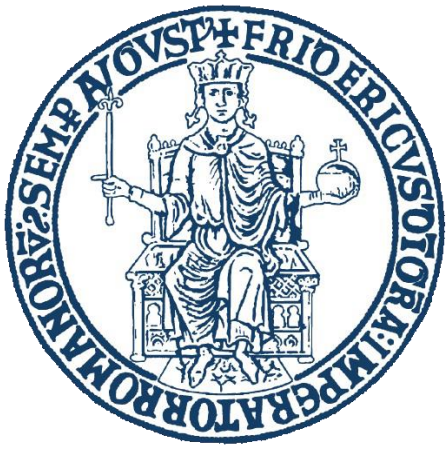

UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

AUTOMAZIONE DEI PROCESSI DI BUSINESS

Parking system

By Guillermo Arce

0001/16313

Summary

The following project is focused on discovering the field of the automation of business processes. For that purpose, one of the multiple workflow engines (AWS Step Functions) is studied and applied into a possible use case.

The use case built is about a parking system, in which cars can be registered and removed from the system. In order to achieve this objective, a workflow is designed.

Table of Contents

Introduction

In the business world, an efficient, effective and automated management of the business processes can really make the difference. When we are talking about the automation of processes, we are talking about their coordination in such a way that we have a fluent and robust system which helps organizations save time and ensure best practices are implemented.

BPA (Business Process Automation) makes use of **workflows** in order to represent the different business processes. A workflow consists of an orchestrated and repeatable pattern of business activities enabled by the systematic organization of resources into processes, represented by a **sequence of operations**.

That succession of operations can be seen as a a sequence of services each one evolving a specific stateless operation. That is why process automation discipline, takes us to the use of **services**. Services or a SOA (Service Oriented Architecture) are the perfect solution for the application of a **workflow**.

However, services need to be orchestrated and for that purpose, **workflow engines** are used. A workflow engine manages and monitors the state of activities in a workflow, such as the processing and approval of a loan application form, and determines which new activity to transition to according to defined processes.

All these concepts are studied and applied in the following project.

Objective

The main objective of this project is to study and **understand the management of workflows**. For that purpose, we will be working with some services from AWS (Amazon Web Services) which provide the necessary functionalities.

In order to solve a “real-life” problem and work with workflows at the same time, the idea has been creating a **parking system**. The intention of the service is that the parking can register and remove cars and their owners from the system. For that, a workflow management is used.

In addition, the system follows a **serverless** architecture. A serverless architecture (also known as serverless computing or function as a service, FaaS) is a software design pattern where applications are hosted by a third-party service, eliminating the need for server software and hardware management by the developer.

Applications are broken up into individual functions (Lambda Functions in our case) that can be invoked and scaled individually. At that point it is where the difference from PaaS (like AWS Elastic Beanstalk) is noted. In FaaS, each function is hosted and automatically scaled by the provider, while in PaaS the whole application is deployed as a single unit and scaling is done at the entire application level.

Cloud Platform

In order to achieve our goal of creating a serverless system based on orchestrated services, we need a cloud platform that can provide the functionalities we need.

For this project, **Amazon Web Services** (AWS) has been chosen; however, there are plenty of others like Microsoft Azure or Google Cloud.

AWS provides a free layer of services for the period of a year that we have taken approach in order to build our system. Those services will be explained in detail in the next chapters.

Serverless Framework

Following a serverless architecture facilitates the scalability and the infrastructure management, however, it entails some difficulties. The complexity of these systems is to integrate everything in the good way. Dealing with the console provided by the cloud platform usually doesn't help a lot.

For that reason, it would be useful to use something that can join everything and help you organize and automate some initial configurations which may be difficult to understand. In addition, the deployment is also usually a problematic procedure.

For example, lambda removes the pain of managing servers and orchestrating complex container clusters to handle load. API Gateway takes away the need to worry about load balancing and throttling HTTP requests. AWS (and other cloud providers) have done a great job of providing such powerful service primitives to enable a serverless architecture. But if **developers still need to write a lot of code/config to wire these services** together into an app and deploy them, then that's just unnecessary friction.

In order to fix this difficulty, the solution is a simple one: use a serverless-focused deployment framework which has already solved these problems for you.

In this project, the **Serverless Framework** has been used. As a brief review of how it is composed we can highlight the following:

The framework normally makes use of a control file **serverless.yml** where the definitions of the service, functions, configuration, resources, permissions, etc. are detailed.

A **Service** is the Framework's unit of organization. We can think of it as a project file, though we can have multiple services for a single application. It's where we define our Functions, the Events that trigger them, and the Resources our Functions use, all in one file entitled serverless.yml

An example of a serverless yml file is shown in *Figure 1*.

```
# serverless.yml

service: users

functions: # Your "Functions"
  usersCreate:
    events: # The "Events" that trigger this function
      - http: post users/create
  usersDelete:
    events:
      - http: delete users/delete

resources: # The "Resources" your "Functions" use.  Raw AWS CloudFormation goes in here.
```

Figure 1

Services and Workflow

Now that the cloud provider and the framework are introduced, we can focus on the services to use and their orchestration.

For this project, four main services from AWS have been used: Step Functions, Lambda Functions, API Gateway and DynamoDB. Also, other services such as CloudFront or S3 Bucket have been used but are not so relevant for this context.

Step Functions

As introduced at the beginning of the document, the main objective of this project is to understand and handle workflow management by using a workflow engine. For that reason, first of all we have selected our worflow engine, which in this case is Step Functions from AWS. However, other workflow engines such as Conductor from Netflix could have been also used.

Step Functions is an orchestration service that allows us to **model workflows as state machines**. The design of the state machine is achieved using a JSON-based specification language (Amazon-State-Language). However, we can also define it from the serverless.yml from the serverless framework.

Step Functions provides quite a bit of convenient functionality: automatic retry handling, triggering and tracking for each workflow step, and ensuring steps are executed in the correct order. Although that list might at first seem unspectacular, it turns out it's not at all trivial to ensure all these things happen correctly in workflows that contain dozens of steps and hundreds of parallel executions. Step Functions takes on a lot of the work that previously had to be done in our application.

Step Functions allow us to quickly create complex sequences of tasks while taking on the error handling and retry logic and allowing us to decouple our application's business logic from its orchestration logic.

Some of its main benefits are enumerated below:

Quickly create complex sequences of tasks.

Manage state between executions of various stateless functions: For many serverless workflows, setting up queues and databases for communication between all the services can be time-consuming and error-prone.

Decouple application workflow logic from business logic: Here we have another best practice of Serverless development. Adding workflow logic to applications that should only handle business logic increases the complexity of the applications and can easily generate issues.

More efficient workflows with parallel executions: Step Functions allows us to have many parallel executions of a workflow at the same time, making sure that performance scales with our application's load.

Step Function on the Parking system

In relation to our project, in order to create a correct workflow the first thing was to think about all the necessary tasks that were needed. Once all the tasks are known, we can think about their relationships and their order.

It is important to remark that we are talking about services, so they are stateless. In addition, the handle of input and output of the services should be also taken into account, so that they communicate correctly.

As a result of this approach, the following state machine workflow has been created:

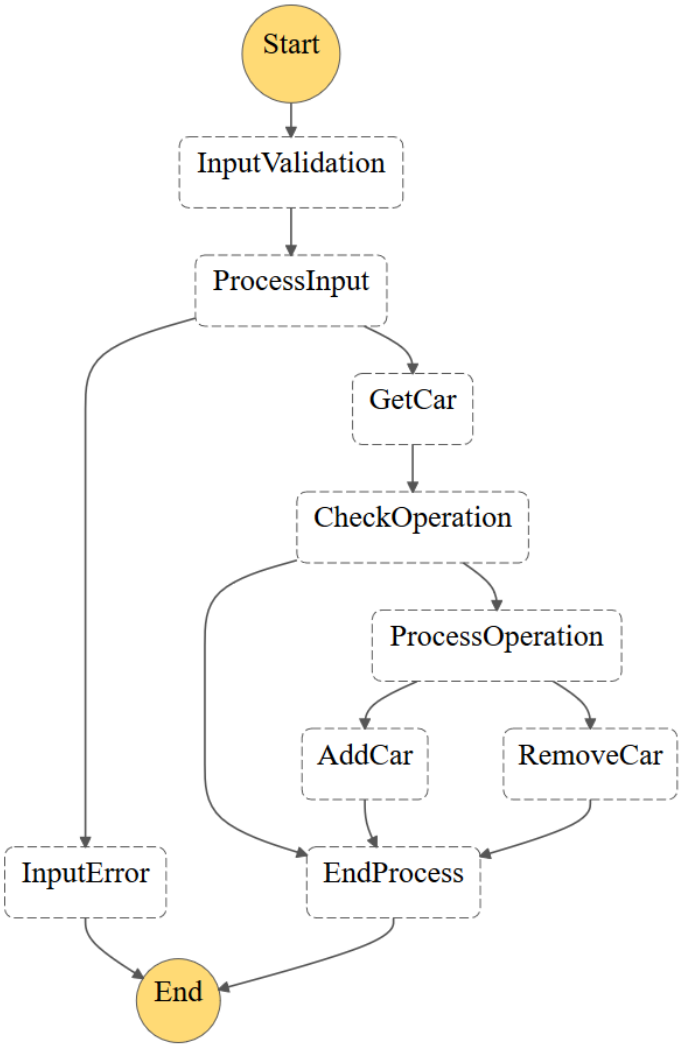

Figure 2

This is the workflow that each operation from the parking system should follow in order to be completed. First of all, the input collected from the parking system is validated in **InputValidation** *task*. In case the input is correct the next operation is GetCar, otherwise it is InputError. These two possibilities are decided on **ProcessInput** *choice* operation.

If there is an error on the input, the **InputError** *fail* operation would be raised and the step function will be interpreted as a failed one.

Instead, **GetCar** retrieves the car requested from the database. In such a way, if the operation requested from the parking is "add" and the car is already registered, the operation is not done and the workflow continues to EndProcess *pass* operation. In case the operation is "remove" and the car is not on the database, it would be the same result.

Otherwise, the **ProcessOperation** *choice* is executed and redirects the workflow to AddCar or RemoveCar *task* depending on the type of operation requested from the parking system.

In case of **AddCar** *task*, the car extracted from the input is added to the database. If instead the *task* selected is **RemoveCar**, the car from the input is deleted from the database.

Finally, **EndProcess** *pass* operation would arrive; it just gets the input and return the output. It is needed because *choice* operations cannot be the final ones. The workflow would be considered succesfull if the end is reached through EndProcess *pass* operation.

All the *tasks* operations are, in this workflow, Lambda functions. However, they could be message or notification operations (with Amazon SQS, for example) , or inserting or retrieving information from a database (like DynamoDB), for example. In the case of DynamoDB operations, even there is the possibility of doing it directly, we have decided to handle the database operations through Lambda functions (for simplicity).

One of the most interesting functionalities used during the development of the project is the ability of the step functions of showing **how the workflow is being executed** and where it may fail. This help us to find errors or bugs but also to better understand the general behaviour of the process; an example is shown on *Figure 3*.

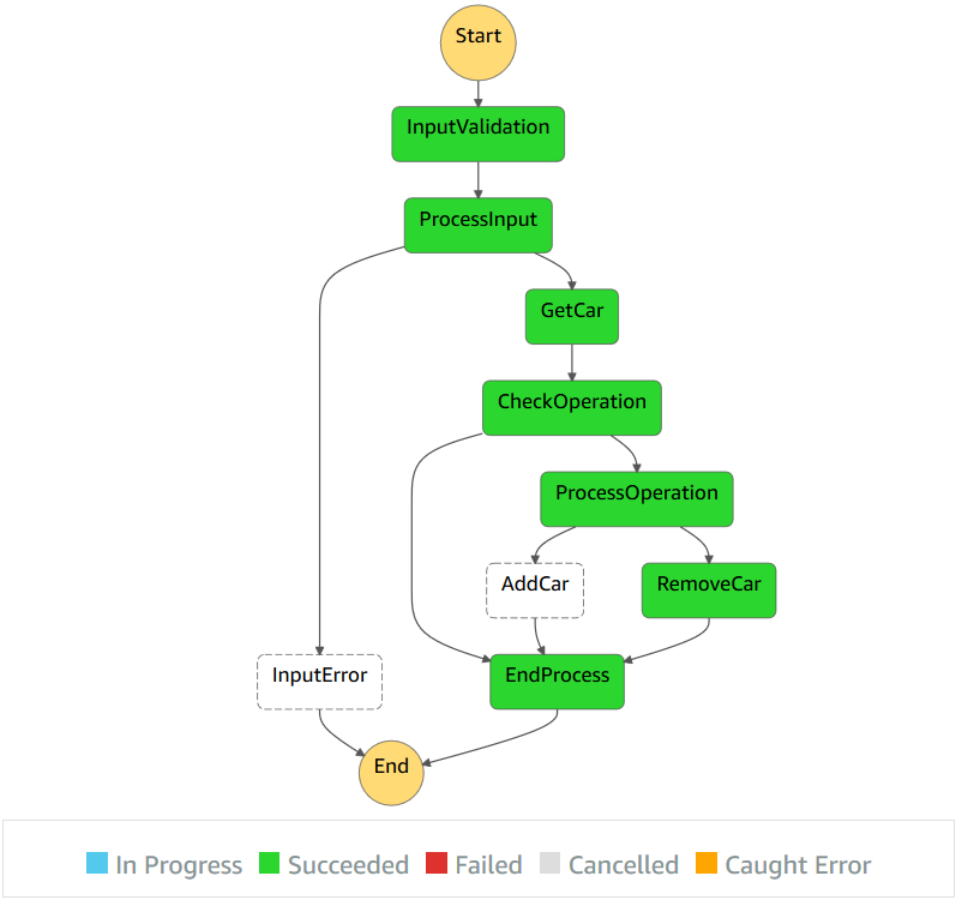

Figure 3 – Succesful removal of car operation

Another example, but from a failed (invalid input) execution is shown on *Figure 4*.

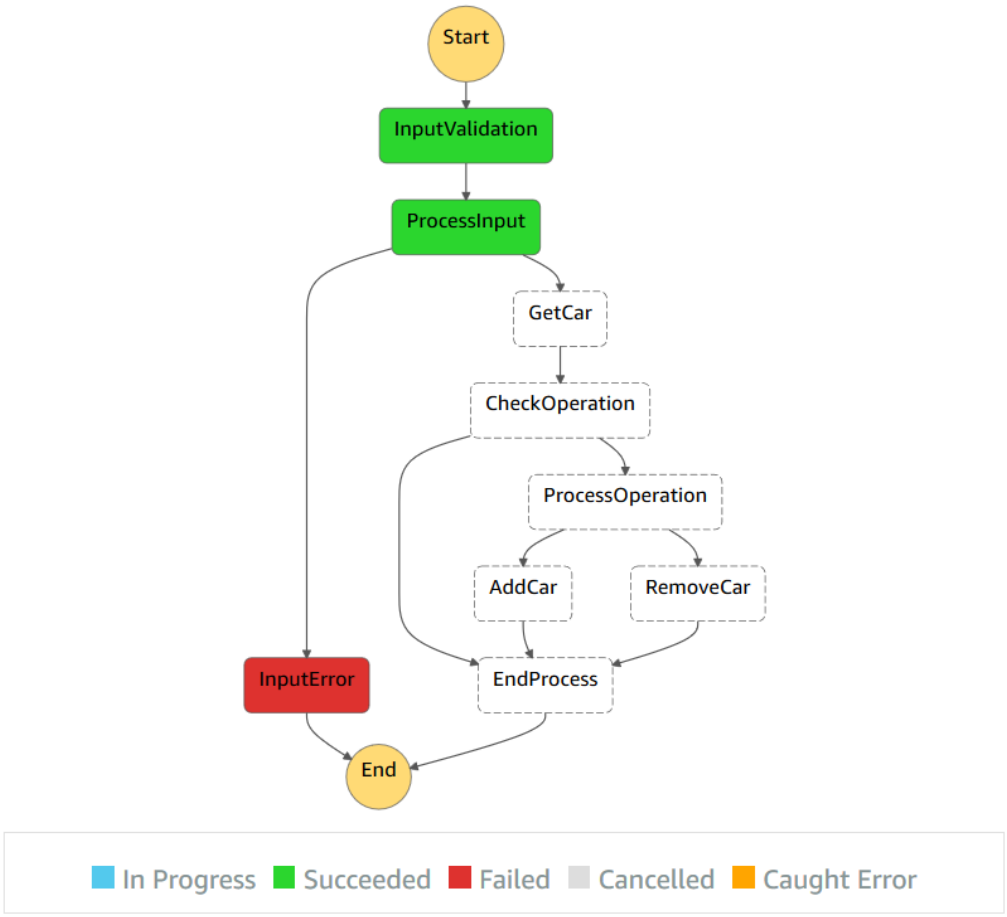

Figure 4

In addition, in the AWS Console, we can see a list (check *Figure 5*) of all the different executions of the workflow. It includes the result of the workflow, as well as the time of the execution and a link to the execution details (to check where has an error raised or which were the input and output of different tasks, for example).

Name	Status	Started	End Time
b458eca8-c708-f92a-9c33-cefca90e000e	Failed	May 21, 2020 07:48:24.291 PM	May 21, 2020 07:48:24.543 PM
8156d743-1e18-65ac-0a6b-013c2fafd13f	Succeeded	May 21, 2020 07:48:00.271 PM	May 21, 2020 07:48:01.692 PM
cfe89e87-445e-3925-e718-b123e80a6d67	Succeeded	May 21, 2020 07:47:31.038 PM	May 21, 2020 07:47:31.544 PM

Figure 5

Lambda Functions

Lambda Functions are a very representative part of a Step Function workflow. By representative it is intended that they are not something very specific from this project but the purest representation of what a service inside a workflow can do. For that reason, they have been used on this project.

A Lambda function is an independent unit of deployment, like a microservice. It's merely **code, deployed in the cloud**, that is most often written to perform a single job such as:

- Saving a user to the database
- Processing a file in a database
- Performing a scheduled task

As mentioned on the previous chapter, all the *tasks* present on the Step Functions are Lambda functions. That is, all the different services orchestrated by the workflow are independent units of code.

Lambda Functions on the Parking system

In this project, they have been writen with JavaScript, but they can be writen in other languages. They are all located in the *handler.js* file relative to our serverless framework project, however they are all linked with their correspondent *task* on the Step Function.

An example of one of the used Lambda functions on the project is presented in *Figure 6*.

```
//Lambda Function -> Process Remove of Car
module.exports.removeCar = (event, context, callback) => {
  console.log('removeCar was called');

  if (event.exists === "TRUE") {
    databaseManager.removeCar(event.carId).then(response => {
      console.log(response);
      callback(null, response)
    });
    callback(null, event);
  }
}
```

Figure 6

Also, an additional Lambda function will be used in order to call the execution of the Step Function. This will be explained on the API Gateway chapter.

DynamoDB

Another important service used during this project is the database DynamoDB.

It is a fully managed **NoSQL database** service that provides fast and predictable performance with seamless scalability.

DynamoDB on the Parking system

Useful for this project because there is the need of storing data during the workflow process. Concretely, it is used because the cars need to be registered persistently, so with the use of a Lambda function the database is accessed and updated or consulted.

Cars added on the Parking system could be checked from the AWS Console on the DynamoDB service; shown in *Figure 7*.

carId	carModel	clientName
KQ186SH	Seat Leon TDI	Janet Molloy
ME614DA	Fiat Panda	Mollie Clemons
MV621AD	Volkswagen Golf GTI	Ravinder Curry

Figure 7

API Gateway

Finally, an API Gateway service has been used in order to provide a clear entry point into our Parking system.

An API Gateway is an **intermediary system that provides a REST** (in our case) or WebSocket API **interface** to act as a router from a single entry point, to a group of defined third-party microservices and / or APIs. It interacts as a gateway.

Essentially, it **unifies or decouples the interface that customers see** (in this case, API consumers that could be mobile applications, web, etc.) from the implementation of the microservices and/or APIs. If we simplify its purpose, it is nothing more than a reverse proxy, optimized for authentication and access control against microservices and/or APIs.

API Gateway on the Parking system

As we can see during deployment (*Figure 8*), the only endpoint from our system is done through a GET request to a specific URL, which is the API Gateway.

```
Serverless: Stack update finished...
Service Information
service: step-function
stage: dev
region: eu-west-3
stack: step-function-dev
resources: 29
api keys:
  None
endpoints:
  GET - https://3s2o8s03v2.execute-api.eu-west-3.amazonaws.com/dev/run
functions:
  executeStepFunction: step-function-dev-executeStepFunction
  inputValidation: step-function-dev-inputValidation
  getCar: step-function-dev-getCar
  removeCar: step-function-dev-removeCar
  addCar: step-function-dev-addCar
  getCars: step-function-dev-getCars
layers:
  None
```

Figure 8

This endpoint, doesn't call directly the Step Function but a Lambda Function that calls it. We have added another layer (lambda function) that hides the real call of the Step function in order

to avoid the leak of information (avoid the need of the client that calls the API Gateway to know the exact function to call). This is done by the creation of a lambda function (known by the client through the API Gateway) which calls the Step function. We have basically added an intermediator between the API Gateway and the Step Function.

So, in the diagram shown in *Figure 9* a Lamba Function will be in the middle of the Amazon API Gateway and the Step Function.

Figure 9

Use Example

To conclude with the documentation of the project an example of the two possible operations is presented.

In order to make the project more visual and to ease the interaction with the services (avoid just using a REST client) a frontend for the application has also been developed using HTML, CSS and JavaScript. It just provides two different forms for the two possible operations (register and remove car) which call the API Gateway.

Frontend shown in *Figure 10*.

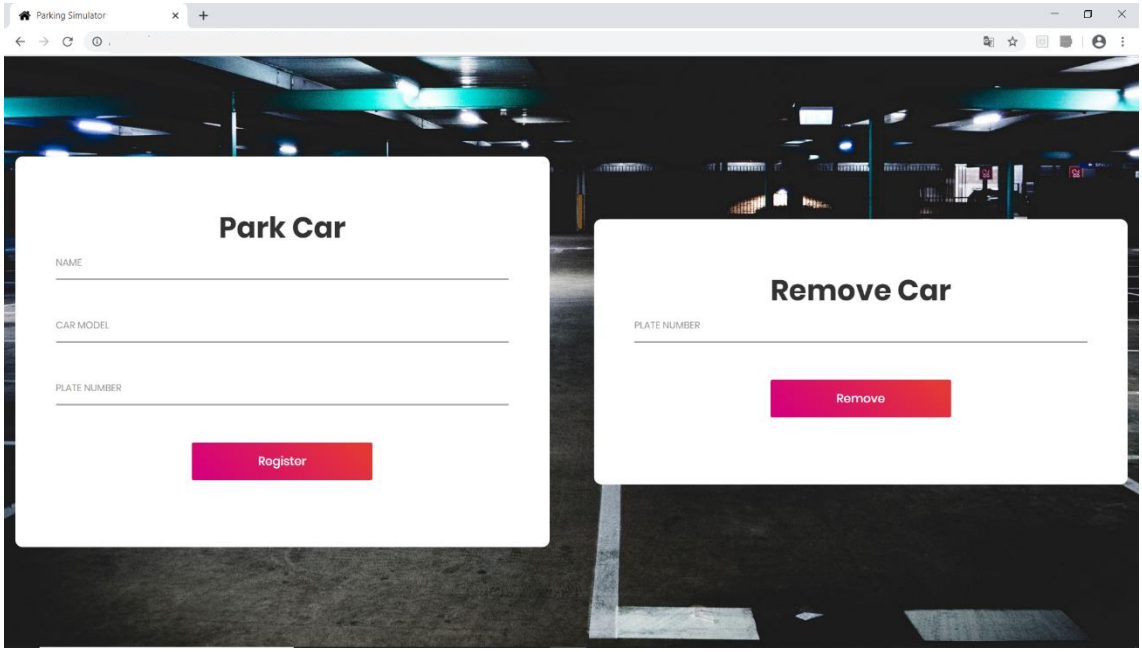

Figure 10

Below, an example of a registration of a car into the parking system (*Figure 11*)

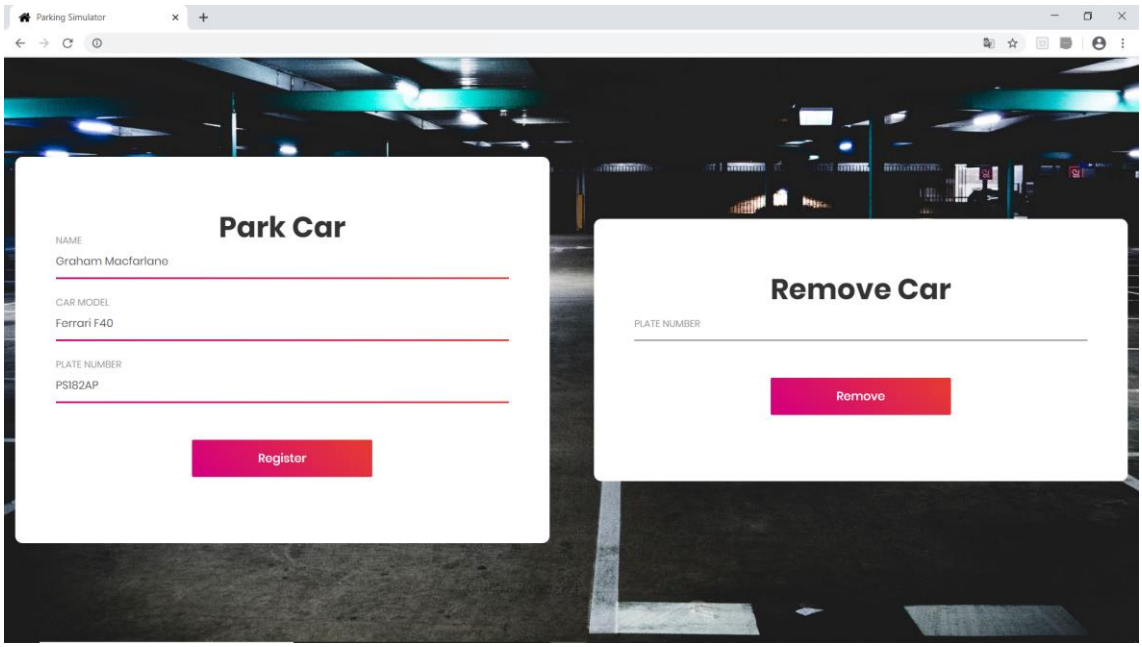

Figure 11

After pressing *Register*, the situation on the database will be the following:

carId	carModel	clientName
KQ186SH	Seat Leon TDI	Janet Molloy
ME614DA	Fiat Panda	Mollie Clemons
MV621AD	Volkswagen Golf GTI	Ravinder Curry
PS182AP	Ferrari F40	Graham Macfarlane

Figure 12

Finally, another example on the removal of a car from the system (*Figure 13*).

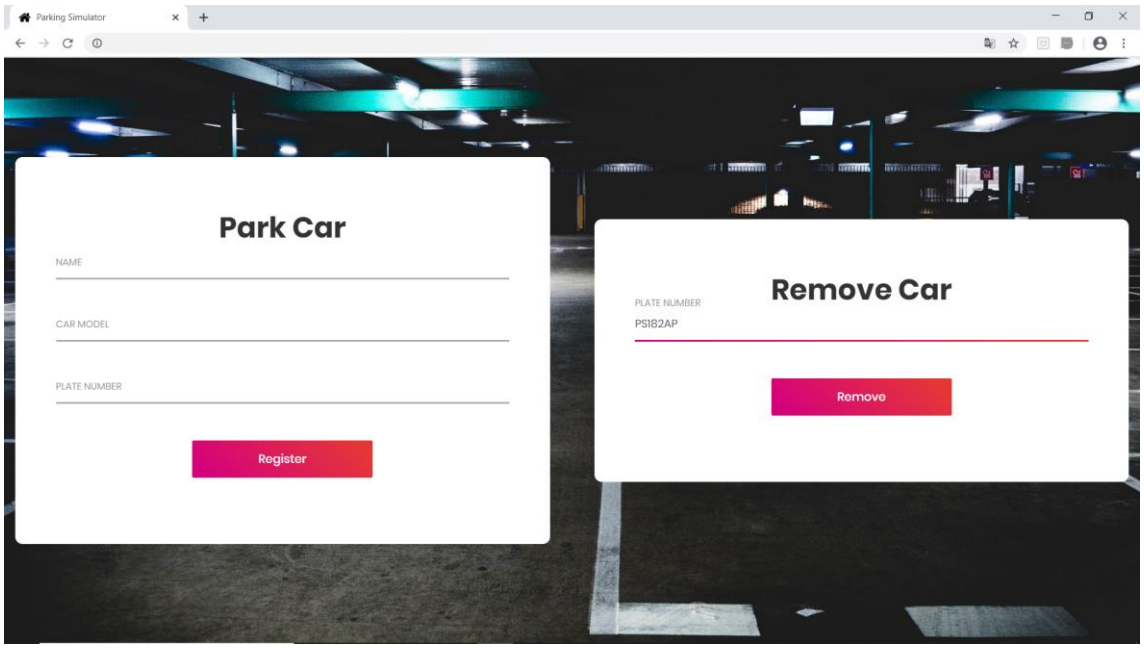

Figure 13

After pressing *Remove*, the situation on the database will be the following:

carId	carModel	clientName
KQ186SH	Seat Leon TDI	Janet Molloy
ME614DA	Fiat Panda	Mollie Clemons
MV621AD	Volkswagen Golf GTI	Ravinder Curry

Figure 14

Appendix

Serverless Framework

serverless.yml

```
service: step-function

plugins:
  - serverless-step-functions
  - serverless-pseudo-parameters

custom:
  settings:
    PARKING_TABLE_NAME: parkingTable

provider:
  name: aws
  runtime: nodejs12.x
  region: eu-west-3
  profile: default
  environment: ${self:custom.settings}

  iamRoleStatements:
  - Effect: "Allow"
    Action:
      - "states:ListStateMachines"
      - "states:StartExecution"
      - "dynamodb:DeleteItem"
      - "dynamodb:PutItem"
      - "dynamodb:GetItem"
    Resource: "arn:aws:states:#{AWS::Region}:#{AWS::AccountId}:*"

resources:
    Resources:
      parkingTable:
        Type: AWS::DynamoDB::Table
        Properties:
          TableName: ${self:custom.settings.PARKING_TABLE_NAME}
          AttributeDefinitions:
            - AttributeName: carId
              AttributeType: S
          KeySchema:
            - AttributeName: carId
              KeyType: HASH
          ProvisionedThroughput:
            ReadCapacityUnits: 1
            WriteCapacityUnits: 1

functions:
  executeStepFunction:
    handler: handler.executeStepFunction
    events:
      - http:
          path: run
          method: get
  inputValidation:
    handler: handler.inputValidation
  getCar:
```

```
    handler: handler.getCar
  removeCar:
    handler: handler.removeCar
  addCar:
    handler: handler.addCar
  getCars:
    handler: handler.getCars


stepFunctions:
  stateMachines:
    testingStateMachine:
      name: TestingStateMachine
      definition:
        StartAt: InputValidation
        States:
          InputValidation:
            Type: Task
            Resource:
arn:aws:lambda:#{AWS::Region}:#{AWS::AccountId}:function:${self:servic
e}-${opt:stage}-inputValidation
            Next: ProcessInput
          ProcessInput:
            Type: Choice
            Choices :
              - Variable: "$.operationType"
                StringEquals: "INVALID"
                Next: InputError
            Default: GetCar
          InputError:
            Type: Fail
            Cause: "Wrong Inputs"
          GetCar:
            Type: Task
            Resource:
arn:aws:lambda:#{AWS::Region}:#{AWS::AccountId}:function:${self:servic
e}-${opt:stage}-getCar
            Next: CheckOperation
          CheckOperation:
            Type: Choice
            Choices:
              - And:
                  - Variable: $.operationType
                    StringEquals: ADD
                  - Variable: $.exists
                    StringEquals: 'TRUE'
                Next: EndProcess
              - And:
                  - Variable: $.operationType
                    StringEquals: REMOVE
                  - Variable: $.exists
                    StringEquals: 'FALSE'
                Next: EndProcess
            Default: ProcessOperation
          ProcessOperation:
            Type: Choice
            Choices :
              - Variable: "$.operationType"
                StringEquals: "ADD"
                Next: AddCar
              - Variable: "$.operationType"
```

```
                StringEquals: "REMOVE"
                Next: RemoveCar
          AddCar:
            Type: Task
            Resource:
arn:aws:lambda:#{AWS::Region}:#{AWS::AccountId}:function:${self:servic
e}-${opt:stage}-addCar
            Next: EndProcess
          RemoveCar:
            Type: Task
            Resource:
arn:aws:lambda:#{AWS::Region}:#{AWS::AccountId}:function:${self:servic
e}-${opt:stage}-removeCar
            Next: EndProcess
          EndProcess:
            Type: Pass
            End: true
```

handler.js

```
'use strict';

const AWS = require('aws-sdk');
const stepfunctions = new AWS.StepFunctions();
const databaseManager = require('./databaseManager');

//Lambda Function -> Init Step Function
module.exports.executeStepFunction = (event, context, callback) => {

  var operationType = event.queryStringParameters.operationType;
  var clientName = event.queryStringParameters.clientName;
  var carModel = event.queryStringParameters.carModel;
  var carId = event.queryStringParameters.carId;



callStepFunction(operationType,clientName,carModel,carId).then(result
=> {
    let message = 'Step function is executing';
    if (!result) {
      message = 'Step function is not executing';
    }

    const response = {
      statusCode: 200,
      headers: {

        'Access-Control-Allow-Origin': '*',

        'Access-Control-Allow-Credentials': true,

      },
      body: JSON.stringify({ message })
    };

    callback(null, response);
  });
};

//Lambda Function -> InputValidation
module.exports.inputValidation = (event, context, callback) => {
```

```
  console.log('inputValidation was called');

  var operationType = event.operationType;
  var clientName = event.clientName;
  var carModel = event.carModel;
  var carId = event.carId;

  var validOp = false;
  var validClName = false;
  var validCMod = false;
  var validCPla = false;

  if (operationType === "ADD" || operationType === "REMOVE")
    validOp = true;
  if (operationType === "ADD") {
    if (/^([\w\s]{3,40})$/.test(clientName))
      validClName = true;
    if (/^([\w\s]{3,40})$/.test(carModel))
      validCMod = true;
    if (/[a-zA-Z]{2}[0-9]{3}[a-zA-Z]{2}$/.test(carId))
      validCPla = true;
  }
  if (operationType === "REMOVE") {
    if (/[a-zA-Z]{2}[0-9]{3}[a-zA-Z]{2}$/.test(carId))
      validCPla = true;
    validCMod = true;
    validClName = true;
  }


  if (!validOp || !validClName || !validCMod || !validCPla)
    operationType = "INVALID";

  callback(null, { operationType, clientName, carModel, carId });
};

//Lambda Function -> Get Car
module.exports.getCar = (event, context, callback) => {
  console.log('getCar was called');

  databaseManager.getCar(event.carId).then(response => {
    console.log(response);
    if (response == null) {
      event.exists = "FALSE";
      callback(null, event);
    }
    else {
      event.exists = "TRUE";
      callback(null, event);
    }

  });
};

//Lambda Function -> Process Remove of Car
module.exports.removeCar = (event, context, callback) => {
  console.log('removeCar was called');

  if (event.exists === "TRUE") {
    databaseManager.removeCar(event.carId).then(response => {
      console.log(response);
```

```
      callback(null, response)
    });
    callback(null, event);
  }
}

//Lambda Function -> Process Addition of Car
module.exports.addCar = (event, context, callback) => {
  console.log('addCar was called');

  var clientName = event.clientName;
  var carModel = event.carModel;
  var carId = event.carId;

  var car = { carId, carModel, clientName };
  if (event.exists === "FALSE") {
    databaseManager.addCar(car).then(response => {
      console.log(response);
      callback(null, response)
    });
  }
};

function callStepFunction(operationType, clientName, carModel, carId) {
  console.log('callStepFunction');

  const stateMachineName = 'TestingStateMachine'; // The name of the
step function we defined in the serverless.yml
  console.log('Fetching the list of available workflows');

  return stepfunctions
    .listStateMachines({})
    .promise()
    .then(listStateMachines => {
      console.log('Searching for the step function', listStateMachines);

      for (var i = 0; i < listStateMachines.stateMachines.length; i++)
{
        const item = listStateMachines.stateMachines[i];

        if (item.name.indexOf(stateMachineName) >= 0) {
          console.log('Found the step function', item);

          var params = {
            stateMachineArn: item.stateMachineArn,
            input:                                    JSON.stringify({
operationType:operationType,clientName: clientName, carModel: carModel,
carId: carId })
          };

          console.log('Start execution');
          return stepfunctions.startExecution(params).promise().then(()
=> {
            return true;
          });
        }
      }
    })
    .catch(error => {
      return false;
    });
```

```
};
```

AWS Step Function definition

```
{
  "StartAt": "InputValidation",
  "States": {
    "InputValidation": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-3:499428548934:function:step-function-dev-inputValidation",
      "Next": "ProcessInput"
    },
    "ProcessInput": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.operationType",
          "StringEquals": "INVALID",
          "Next": "InputError"
        }
      ],
      "Default": "GetCar"
    },
    "InputError": {
      "Type": "Fail",
      "Cause": "Wrong Inputs"
    },
    "GetCar": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-3:499428548934:function:step-function-dev-getCar",
      "Next": "CheckOperation"
    },
    "CheckOperation": {
      "Type": "Choice",
      "Choices": [
        {
          "And": [
            {
              "Variable": "$.operationType",
              "StringEquals": "ADD"
            },
            {
              "Variable": "$.exists",
              "StringEquals": "TRUE"
            }
          ],
          "Next": "EndProcess"
        },
        {
          "And": [
            {
              "Variable": "$.operationType",
              "StringEquals": "REMOVE"
            },
            {
              "Variable": "$.exists",
              "StringEquals": "FALSE"
            }
```

```
          ],
          "Next": "EndProcess"
        }
      ],
      "Default": "ProcessOperation"
    },
    "ProcessOperation": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.operationType",
          "StringEquals": "ADD",
          "Next": "AddCar"
        },
        {
          "Variable": "$.operationType",
          "StringEquals": "REMOVE",
          "Next": "RemoveCar"
        }
      ]
    },
    "AddCar": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-3:499428548934:function:step-
function-dev-addCar",
      "Next": "EndProcess"
    },
    "RemoveCar": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:eu-west-3:499428548934:function:step-
function-dev-removeCar",
      "Next": "EndProcess"
    },
    "EndProcess": {
      "Type": "Pass",
      "End": true
    }
  }
}
```