
Taxi Booking System Design Consideration and Deployment

1.Implementation

I have used **.NET Core** which is a **cross-platform** version of .NET for building websites, **services**, and console apps (**.Net Core 3.1** with **C#** as the development language)

<https://dotnet.microsoft.com/download>

I have implemented and exposed **REST endpoints** using .Net core **Web API** which helps in building **RESTful** web **services**.

2.Design Constraints

APIs

service is running on PORT **8080**.

- **As requested** , the service **does not** implement any form **persistent storage**. The service uses **In memory** data structures to process and handle the requests.
- **As requested**, the **service does not** handle **concurrent API calls**/data races , as The APIs will be triggered **serially**.

3. Algorithm and data structures

I have created a ***User defined In Memory data structure*** to hold the details of the API operation and sub sequent information

A Taxi class holds the information related to a single taxi.

```
namespace TaxiBookingAPI.Model
{
    /// <summary>
    /// In Memory Model for the Taxi , Which Holds the three taxi details
    /// </summary>
    public class Taxi
    {
        /// <summary>
        /// car id either 1 , 2 , 3
        /// </summary>
        public int CarId { get; set; }

        /// <summary>
        /// current location of the car
        /// </summary>
        public LocationCoordinates Location { get; set; }

        /// <summary>
        /// Is car already booked
        /// </summary>
        public bool IsBooked { get; set; }

        /// <summary>
        /// if car is booked , BookedUntilTime will tell you until what time it is booked
        /// </summary>
        public int BookedUntilTime { get; set; }
    }
}
```

And a LocationCoordinates for holding X and Y coordinates of the taxi.

```

namespace TaxiBookingAPI.Model
{
    /// <summary>
    /// Location Model for the JSON
    /// </summary>
    public class Location
    {
        /// <summary>
        /// Passenger's initial Position
        /// </summary>
        [Required]
        public LocationCoordinates Source { get; set; }

        /// <summary>
        /// Passenger's destination Position
        /// </summary>
        [Required]
        public LocationCoordinates Destination { get; set; }
    }

    /// <summary>
    /// Location Coordinates X,Y
    /// </summary>
    public class LocationCoordinates
    {
        /// <summary>
        /// X coordinate of the Location
        /// </summary>
        [Required]
        public int X { get; set; }

        /// <summary>
        /// Y coordinate of the Location
        /// </summary>
        [Required]
        public int Y { get; set; }
    }
}

```

3.1 Manhattan Distance

The distance between two points $x = (a,b)$ and $y = (c,d)$ is calculated using

$$\text{Manhattan Distance} = |a - c| + |b - d|$$

3.2 Algorithm

Time to travel between point $x = (a,b)$ to point $y = (c,d)$ is $|a-c| + |b-d|$

Main algorithm is divided in two steps and which is in **FindNearestTaxiToCustomer** function,

Step 1: Loop through the current position of all taxi's from the customer's initial position and also calculate the time for each car to reach to the customer. **$O(n)$**

It will look something like this after the step 1. I am holding this information in a dictionary for the faster retrieval (performance for the **$O(1)$** retrieval)

```
// Dictionary to hold the car ID and Time to travel to customer
//
// car ID      | Time required to reach Customer
// -----
// 1           | 3
// 2           | 6
// 3           | 8
```

Step 2: Sort the dictionary based on the 'nearest car to the customer' and the 'smallest car id; **$O(n \log n)$** in worst case.

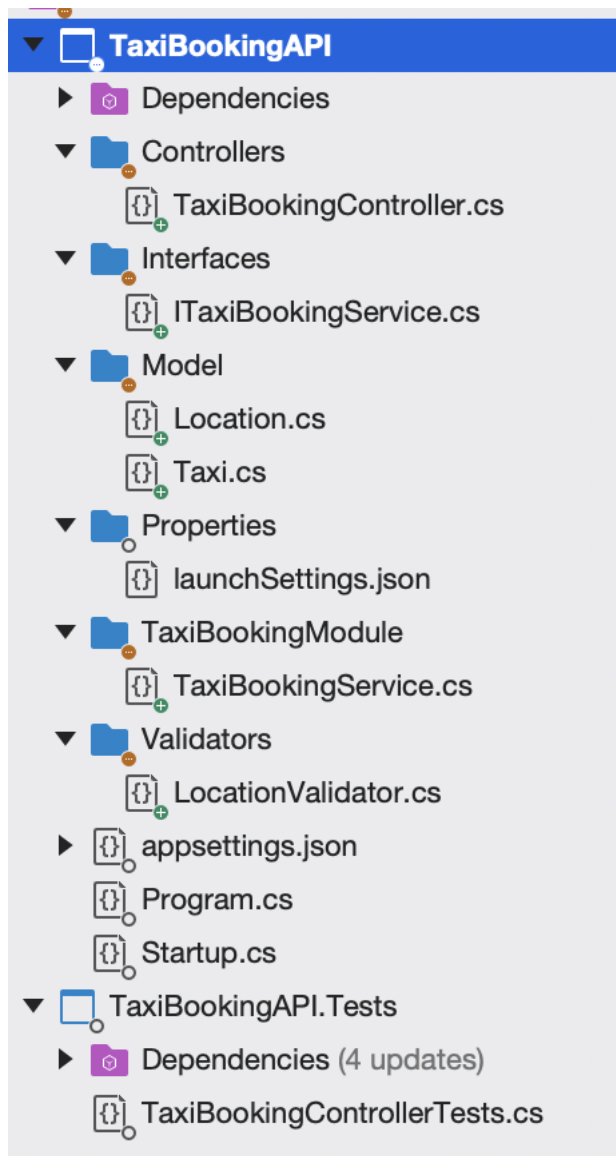
Overall time complexity of the algorithm is **$O(n \log n)$**

4. Software design and engineering practices

TaxiBooking API service tries to capture the best design and engineering practices. I have kept in mind the basics of **SOLID Principles** and other architecture constraints to make system extensible.

a. Separation of concerns

Clear separation of building blocks to make sure different concerns are separated (**Separation of Concerns** which gives us a **Highly cohesive** and **loosely coupled** systems)



b. Interface segregation and Open/Close Principles

I have created an **Interface** for TaxiBooking API, which is **extensible** and *new functionalities can be easily added later in future without modifying and changing existing architecture which makes the system extensible (open for new extension)*

```

namespace TaxiBookingAPI.Interfaces
{
    /// <summary>
    /// Interface for the Taxi Booking Service API
    /// </summary>
    public interface ITaxiBookingService
    {
        /// <summary>
        ///
        /// </summary>
        public void ResetTaxiBookingSystem();

        /// <summary>
        ///
        /// </summary>
        public void IncrementServiceTimeStamp();

        /// <summary>
        ///
        /// </summary>
        /// <param name="location"></param>
        /// <returns></returns>
        public (int, int) BookTaxi(Location location);
    }
}

```

c. Dependency Injection (Inversion of Control)

ASP.NET Core framework includes built-in **IoC container** for automatic **dependency injection**. The built-in IoC container is a simple yet effective container.

I have Register and injected the **TaxiBookingService** in the IOC Container, which takes care of the object creation and the object life cycle. I have register as a **singleton** instance as I wanted to make sure I have only single instance across different requests

```

// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddMvc().AddFluentValidation();

    services.AddTransient<IValidator<Location>, LocationValidator>();
    // Inject the Taxi Booking Service in Default IOC Container Service Collection
    services.AddSingleton<ITaxiBookingService, TaxiBookingService>();
}

```

5. Build and Deployment

Clone the repo from

<https://github.com/ShahUjval/TaxiBookingSystem.git>

Search or jump to... Pull requests Issues Marketplace Explore

ShahUjval / TaxiBookingSystem

Unwatch 1 Star 0 Fork 0

<> Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security 0 Insights Settings

No description, website, or topics provided. Edit

Manage topics

2 commits 1 branch 0 packages 0 releases 2 contributors

Branch: master New pull request Create new file Upload files Find file Clone or download

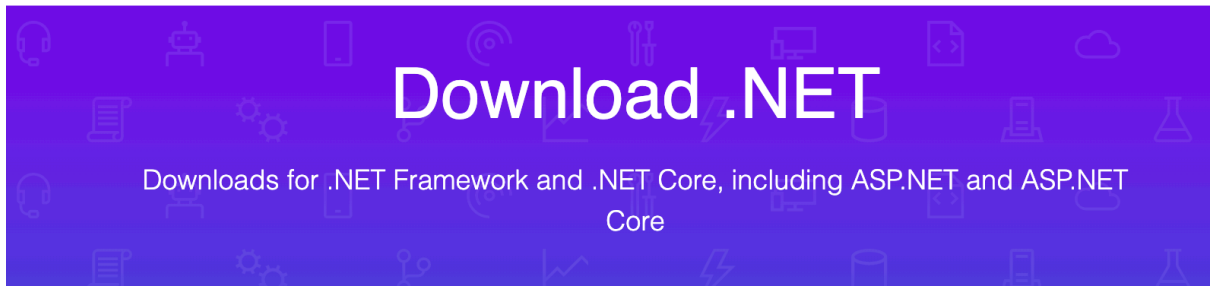
ShahUjval Create README.md Latest commit 52c5def 18 minutes ago

.idea	initial commit	28 minutes ago
.vs/TaxiBookingAPI/xs	initial commit	28 minutes ago
TaxiBookingAPI.Tests	initial commit	28 minutes ago
TaxiBookingAPI	initial commit	28 minutes ago
published	initial commit	28 minutes ago
README.md	Create README.md	18 minutes ago
TaxiBookingAPI.sln	initial commit	28 minutes ago
TaxiBookingAPI.sln.DotSettings.user	initial commit	28 minutes ago
basic_solution_checker.py	initial commit	28 minutes ago

There are several ways to run the application, I will list down few.

Prerequisite:

Download and install the latest version of the .Net Core SDK for your Linux machine.



❓ Not sure where to start? See the [Hello World in 10 minutes tutorial](#) to install .NET and build your first app.

Windows

Linux

macOS

Docker

.NET
Core

.NET Core 3.1

.NET Core is a cross-platform version of .NET for building websites, services, and console apps.

Build Apps ⓘ

[Download .NET Core SDK](#)

a. Using '**dotnet run**' command (command line)

1. Open the Terminal/Console and Navigate to '**TaxiBookingAPI**' folder
2. Run '**dotnet run**'
3. You can see 'Now listening on: <http://localhost:8080>'
4. Run 'python basic_solution_checker.py'

```
TravelodminsMBP:TaxiBookingAPI ujval.shah$ python basic_solution_checker.py
success - expected: {'total_time': 2, 'car_id': 1}, actual: {u'total_time': 2, u'car_id': 1}
success - expected: {'total_time': 10, 'car_id': 2}, actual: {u'total_time': 10, u'car_id': 2}
success - expected: {'total_time': 17, 'car_id': 3}, actual: {u'total_time': 17, u'car_id': 3}
```

b. Using '**dotnet publish**' command

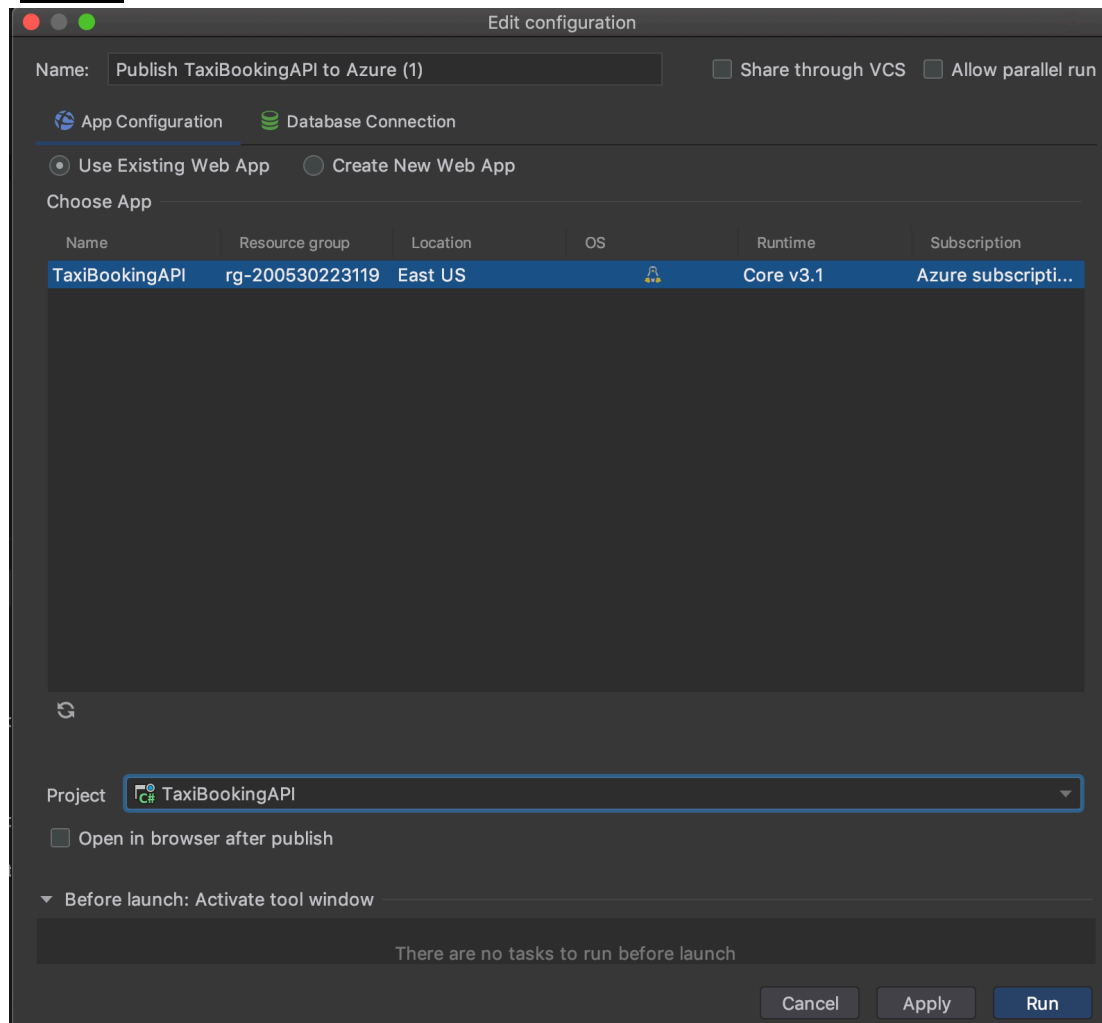
On your local machine, you can prepare the application for deployment by running "dotnet publish". This builds the application artifacts, does any minification and so forth.

1. Open the Terminal/Console and Navigate to '**TaxiBookingAPI**' folder
2. Run command '**dotnet publish -c Release -o published**' – this publish the application to 'published' folder (I have published one for example)
3. 'Cd published'
4. Run 'dotnet TaxiBookingAPI.dll'
5. You can see 'Now listening on: <http://localhost:8080>'
6. Run 'python basic_solution_checker.py'

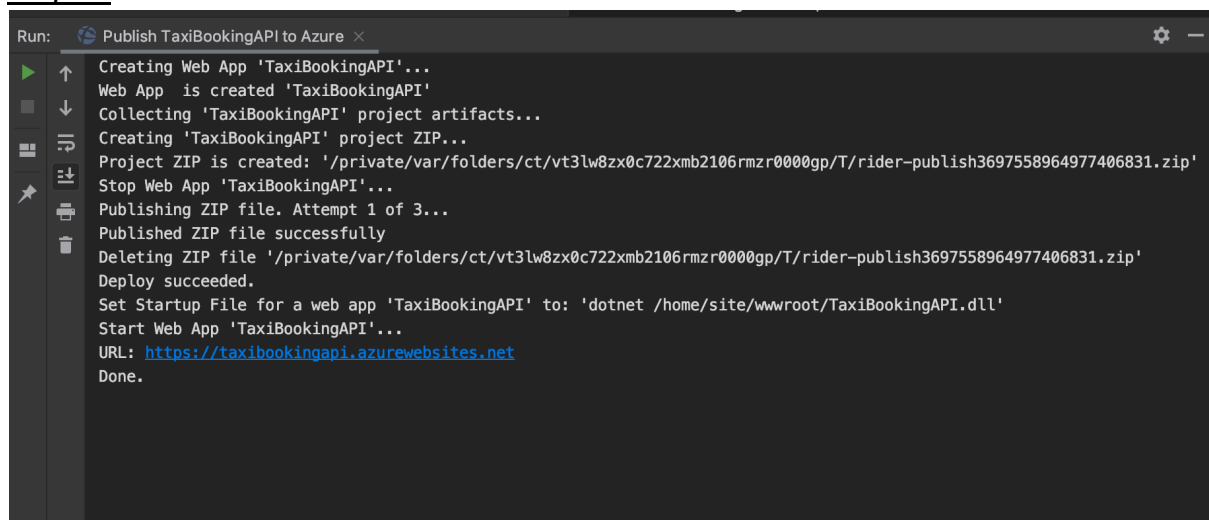
```
TravelodminsMBP:TaxiBookingAPI ujval.shah$ python basic_solution_checker.py
success - expected: {'total_time': 2, 'car_id': 1}, actual: {u'total_time': 2, u'car_id': 1}
success - expected: {'total_time': 10, 'car_id': 2}, actual: {u'total_time': 10, u'car_id': 2}
success - expected: {'total_time': 17, 'car_id': 3}, actual: {u'total_time': 17, u'car_id': 3}
```

C. Using Azure Web Apps to publish our service to Azure cloud

Step 1:



Step 2:



Step 3:

The screenshot shows the Microsoft Azure portal interface. The browser address bar displays the URL: `portal.azure.com/?quickstart=true#@df529e6b-23e5-4ae7-8fd1-030e7cfab423/resource/subsc...`. The page title is "TaxiBookingAPI - Microsoft Azure". The user is logged in as "shah.ujval@outlook.com" with the role "DEFAULT DIRECTORY".

The main content area shows the "TaxiBookingAPI" App Service overview. The left sidebar contains a navigation menu with the following items:

- Overview (selected)
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Security
- Events
- Deployment
 - Quickstart
 - Deployment slots
 - Deployment Center

The main content area displays the following information:

- Resource group:** [rg-200530223119](#) (change)
- Status:** Running
- Location:** East US
- Subscription:** [Azure subscription 1](#) (change)
- Subscription ID:** 357735aa-0f5a-4147-91af-0d367424b13b
- URL:** <https://taxibookingapi.azurewebsites.net>
- App Service Plan:** [appsp-200530223119 \(B1: 1\)](#)
- FTP/deployment username:** No FTP/deployment user set
- FTP hostname:** <ftp://waws-prod-blu-167.ftp.azurewebsites.win...>
- FTPS hostname:** <https://waws-prod-blu-167.ftp.azurewebsites.wi...>

A notification banner at the top of the main content area states: "The free trial for the plan 'appsp-200530223119' will expire on 6/29/2020".

Run <https://taxibookingapi.azurewebsites.net>

Swagger UI

taxibookingapi.azurewebsites.net/index.html

Swagger.
Supported by SMARTBEAR

Select a definition

TaxiBooking API V1

TaxiBooking API v1 OAS3

/swagger/v1/swagger.json

a simple taxi booking system in a 2D grid II using ASP.NET Core Web API

[Ujval Shah - Website](#)
[Send email to Ujval Shah](#)
Use under MIT

TaxiBooking

POST

/api/book POST /api/book - returns the nearest available car to the customer location and returns the total time taken to travel

POST

/api/tick /api/tick REST endpoint, advances your service time stamp by 1 time unit

PUT

/api/reset /api/reset REST endpoint, will reset all cars data back to the initial state

Schemas

LocationCoordinates >

Location >

6. Unit Testing and Execution

1. Using '**dotnet test**' command (command line)
2. Open the Terminal/Console and Navigate to '**TaxiBookingAPI**' folder
3. Run '**dotnet test**'

Output

Starting test execution, please wait...

A total of 1 test files matched the specified pattern.

Test Run Successful.

Total tests: 3

Passed: 3

Total time: 1.4735 Seconds

Also,

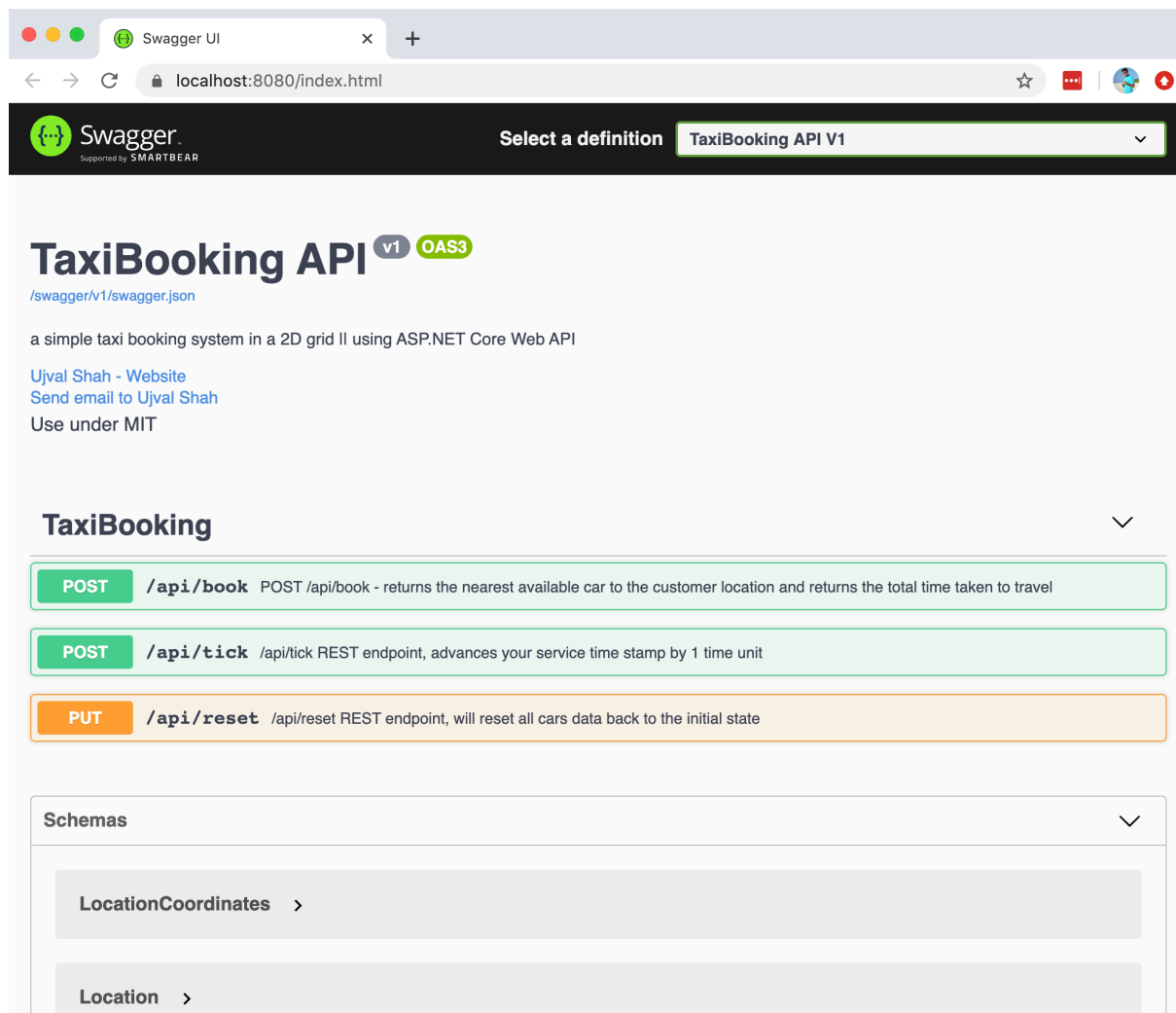
Run '**python basic_solution_checker.py**'

```
TravelodminsMBP:TaxiBookingAPI ujal.shah$ python basic_solution_checker.py
success - expected: {'total_time': 2, 'car_id': 1}, actual: {u'total_time': 2, u'car_id': 1}
success - expected: {'total_time': 10, 'car_id': 2}, actual: {u'total_time': 10, u'car_id': 2}
success - expected: {'total_time': 17, 'car_id': 3}, actual: {u'total_time': 17, u'car_id': 3}
```

7.API Documentation

For API Documentation I have used **Swagger**, which reads your API's structure, and automatically builds beautiful and interactive API documentation.

And I am using **Swagger UI** to generate **interactive API documentation** that lets you try out the API calls directly in the browser (no need of any third party REST Client like Postman)



You can try different option and test the API directly from here.

a. POST api/book

Swagger UI

localhost:8080/index.html

☆

...

TaxiBooking

POST

/api/book

POST /api/book - returns the nearest available car to the customer location and returns the total time taken to travel

Sample request:

```
POST /api/book
{
  "source": {
    "x": x1,
    "y": y1
  },
  "destination": {
    "x": x2,
    "y": y2
  }
}
```

Sample response:

```
{
  "car_id": id,
  "total_time": t
}
```

Parameters

Try it out

No parameters

Request body

application/json

Example Value | Schema

```
{
  "source": {
    "x": 1,
    "y": 0
  },
  "destination": {
    "x": 1,
    "y": 1
  }
}
```


Response

Responses

Curl

```
curl -X POST "http://localhost:8080/api/book" -H "accept: */*" -H "Content-Type: application/json" -d "{\nsource\":{\n\"x\":1,\n\"y\":0},\n\"destination\":{\n\"x\":1,\n\"y\":1}}"
```

Request URL

```
http://localhost:8080/api/book
```

Server response

Code	Details
201	<div><div>Response body</div><div><pre>{\n \"car_id\": 1,\n \"total_time\": 2\n}</pre></div><div>Download</div></div> <div><div>Response headers</div><div><pre>content-type: application/json; charset=utf-8\ndate: Sun, 31 May 2020 16:24:51 GMT\nserver: Kestrel\ntransfer-encoding: chunked</pre></div></div>

Responses

Code	Description	Links
201	Returns the car_id with total_time	No links

b. POST api/tick

The image shows a Swagger UI interface in a web browser. The browser tab is titled "Swagger UI" and the address bar shows "localhost:8080/index.html". The main content area displays the details for the **POST /api/tick** endpoint, described as "/api/tick REST endpoint, advances your service time stamp by 1 time unit".

Parameters

No parameters

Execute

Responses

Curl

```
curl -X POST "http://localhost:8080/api/tick" -H "accept: application/json"
```

Request URL

```
http://localhost:8080/api/tick
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "version": { "major": 1, "minor": 1, "build": -1, "revision": -1, "majorRevision": -1, "minorRevision": -1 }, "content": null, "statusCode": 200, "reasonPhrase": "OK", "headers": [], "trailingHeaders": [], "requestMessage": null, "isSuccessStatusCode": true }</pre>

C. PUT api/reset

The image shows the Swagger UI for a REST API. The browser tab is titled 'Swagger UI' and the address bar shows 'localhost:8080/index.html'. The endpoint being viewed is 'PUT /api/reset', with a description: '/api/reset REST endpoint, will reset all cars data back to the initial state'. Under the 'Parameters' section, it states 'No parameters'. There is an 'Execute' button. The 'Responses' section shows a '200' status code, which is underlined in green. The 'Request URL' section, also outlined in green, shows 'http://localhost:8080/api/reset'. The 'Server response' section shows a JSON body with the following structure:

```
{
  "version": {
    "major": 1,
    "minor": 1,
    "build": -1,
    "revision": -1,
    "majorRevision": -1,
    "minorRevision": -1
  },
  "content": null,
  "statusCode": 200,
  "reasonPhrase": "OK",
  "headers": [],
  "trailingHeaders": [],
  "requestMessage": null,
  "isSuccessStatusCode": true
}
```

8.Future Work

1. Using **persistent** memory storage like RDBMS like **PostgreSQL**, AWS **Aurora**, **DynamoDB**
2. Adding the **concurrency**
3. Implement **micro service**-based architecture to scale the application
4. Adding either the **InMemory** Cache or **distributed** Cache to increase the performance
5. Use **OAuth** 2.0 for the **authentication**
6. Securing a RESTful web services by using **HTTPS** instead of **HTTP**