# Term Project Phase I: Car Dealer Backend Service

**Due Date: 11:59 PM-October 15, 2018**

**Objectives**

1. Understand the design, implementation and use of a List ADT using Doubly Linked Lists.
2. Understand client server programming using REST and JSON.
3. Gain experience implementing applications using layers of increasing complexity and fairly complex data structures.
4. Gain further experience with object-oriented programming concepts, specially interfaces and inheritance.

## Term Project Overview

For the term project, you will build a the server-side code, called a *backend*, of an application that keeps track of information in car dealer. The backend will keep collections that help its users:

* Manage the basic information for the customers (Name, last name, emails, phones).
* Manage information about cars (e.g., model, brand, option, price) and car sales.
* Track sales of cars to customers.
* Track appointment dates for car service (e.g., oil change)

In the first phase of the project, you will master the basics of keeping track of car information. In the remaining phases you will add functionality for maintaining information about customers, car sales, service appointments, and so on. In addition, you will experiment with alternative ADTs and data structures to keep track of all this information.

**Phase I Overview**

Modern applications are built based on three major components (shown in Figure 1):

1. User-interface (UI) Layer – the app used the end user to interact. This layer is part of the client-side of the application. This can be a desktop app written in Java, a mobile app written for a phone with Java or Swift, or a web based application built with Javascript and HTML/CSS.
2. Business Logic Layer – the code that implements the core functionality of the application. For example, in a banking application (e.g. ATH Movil) this is the part that cares of validating login information, moving funds from one account to another, and producing a transaction receipt. This layer is part of the backend.
3. Persistence Layer – this is the code that implements all necessary steps to store data on disk, and to keep frequently accessed data in memory. This layer is most frequently implemented with a relational database system. This layer is also part of the backend.

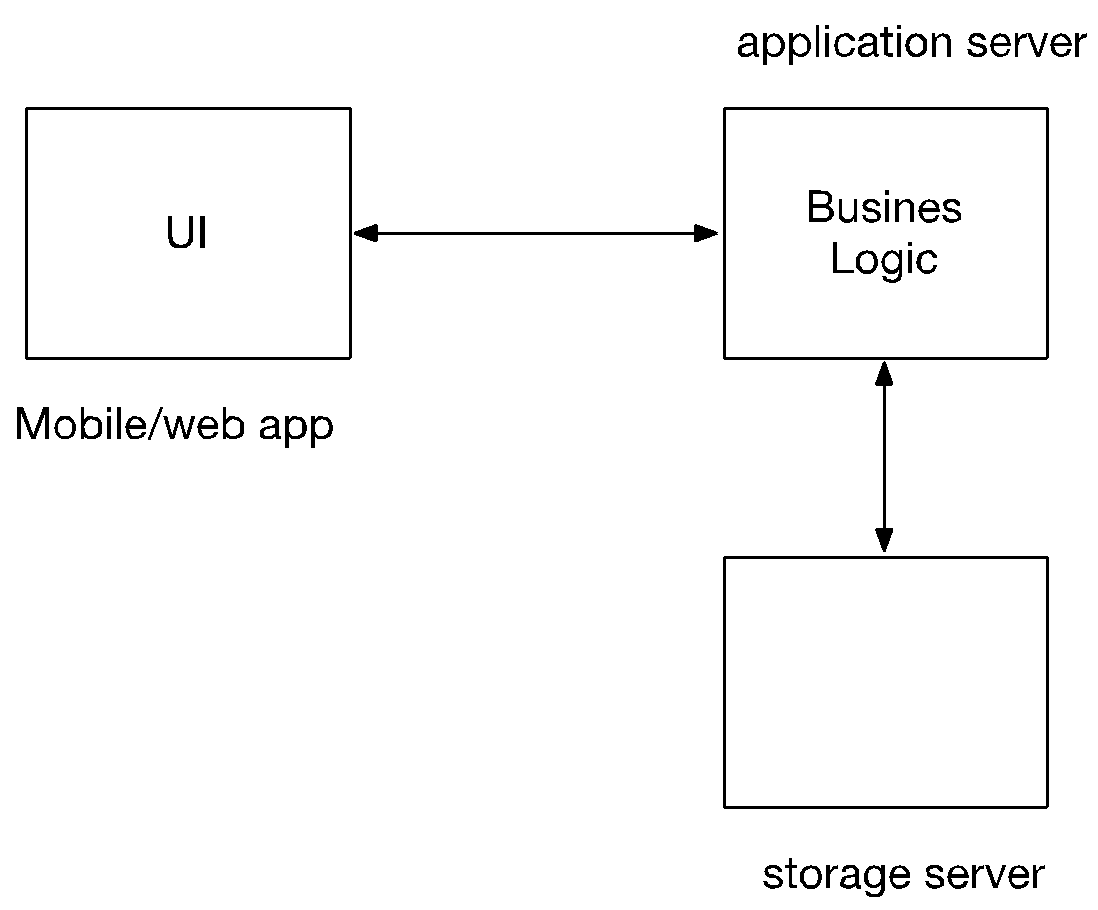


Figure 1: Application Architecture

In this project, you will implement portions of the business logic layer and a persistence layer using the ADTs being discussed in class.

**Specific Tasks**

1. **Task# 1: Building a REST API**

Your backend application will implement REST calls and encoding data with JSON, using Java Jersey. The term REST refers to the concept of **Representational State Transfer**, a common way to represent data on the Web by means of Uniform Resource Identifier (URI) and implementing access operators atop the HTTP protocol. URI are similar to URL, but do not identify a web page, but rather an object or collection of objects managed by an application. Typically, REST is used to manage data objects stored on a remote server. The data can be stored in memory, file, relational database system, NoSQL database, or any other storage system.

**Required REST Tutorial**

Please follow the tutorial on this web site on how to build a REST API in Java using Java Jersey:

<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/basic_grizzly_jersey/jersey-grizzly-json-service.html#section7>

**CRUD Operations**

The data objects are maintained and accessed in terms of four operations, often described with the acronym CRUD:

1. **C**reate – creates and stores a new object into the system. The object gets assigned a unique identifier (Id), which becomes an attribute of the object and serves as a mechanism to tell the object apart from others. This method returns the object created.
2. **R**ead – reads a whole collection of objects, or the contents of specific object based on the id. This method returns either an array of objects or a single object.
3. **U**pdate – updates one or more fields from an existing object (except for the Id) in the system. The Id is used to identify the target. This method returns the updated object.
4. **D**elete – removes an object from the persistence store, based on its Id. This method simply returns a status code indicating whether the object was removed.

Each of the CRUD operations get implemented by means of specific HTTP operations and by using a specific formatted URI. For example, suppose you have a server with address [http://mydealer.com](http://myserver.com), and application name **cardealer**. You will need to store objects of type Car. Each Car instance has a carId that is of type long. In addition, the following fields are part of the Car class: carBrand (String), carModel (String), carModelOption (String), and carPrice (double). Then, we can organize our REST API as follows:

1. Create – must be implemented by calling a HTTP POST operation. The URI to call the operation on the server will be : http://mydealer.com/cardealer/cars/add . Here we use lower case **cars** name to identify the collection of objects to type Car. This operation adds a new car to the collection. The data for the collection must go as the payload of the POST operation. The operation creates a new Car object, adds an Id to it, stores the object, and returns back the newly created instance. Usually client libraries send this request with an object that has the Id as 0, and then expect back the created object with the assigned Id. The data for the object comes in the payload of the POST response, and the HTTP status code should be 201 (Created)
2. Read - must be implemented by calling a HTTP GET operation. The URI to call the operation on the server to get a specific car will be :  [http://mydealer.com/cardealer/cars/Id](about:blank) , where Id is the Id of the Car. Example http://mydealer.com/cardealer/cars/20 would request the object for the car with Id 20. The data for the car comes in the payload of the GET response, and a status code of 200 (OK). If the car is not found, then the GET response shall be 404 (Not Found). To get the list of cars, the URI will be <http://mydealer.com/cardealer/cars>. This will return an array of Car objects.
3. Update - must be implemented by calling a HTTP PUT operation. The URI to call the operation on the server will be  [http://mydealer.com/cardealer/cars/Id](about:blank)/update , where Id is the Id of the Car to update. Example: http://mydealer.com/cardealer/cars/20/update would request an update on the object for the car with Id 20. The data in the request is usually the whole Car object instance with the modified fields (Id won’t be changed). The server can attempt to figure out what changed and only update those fields, or it can simply replace everything but Id with the new data. Sometimes, the updated object instance is returned back to the client. The data for the car comes in the payload of the PUT response, and a status code of 200 (OK). If the car is not found, then the PUT response shall be 404 (Not Found).
4. Delete - must be implemented by calling a HTTP DELETE operation. The URI to call the operation on the server will be:  [http://mydealer.com/cardealer/cars/Id](about:blank)/delete, where Id is the Id of the Car to delete. Example: <http://mydealer.com/cardealer/cars/20/delete> would request a delete on the object for the car with Id 20. No further data is sent on the request. The server erases the object instance. It returns no data in the response and a status code of 200 (OK). If the car is not found, then the DELETE response shall be 404 (Not Found).

For this project you DO NOT need to run the application from a production server. Instead, you will run it locally from Eclipse. In this case, the URI of the address of the server will be as follows:

<http://localhost:8080/cardealer/>

So, for example, the URI (also called “route”) to get a car with id 20 would be:

http://localhost:8080/cardealer/cars/20

**JavaScript Object Notation (JSON)**

Typically, REST applications exchange data by encoding information in either XML or JSON. The latter has become the preferred method. These are strings that represent pairs of attributes and values in the object. Consider the Class class, and suppose it is implemented as a Plain Old Java Object (POJO)[[1]](#footnote-0):

**public** **class** Car {

**private** **long** carId;

**private** String carBrand;

**private** String carModel;

**private** String carModelOption;

**private** **double** carPrice;

// constructors and getters go here but omitted to save space …

}

Then, a JSON representation for a Car object instance representing a Toyota 4Runner XLE priced at $42,000 and with id 3 would be:

{

"carId":3,

"carBrand":"Toyota",

"carModel":"4Runner",

"carModelOption":"XLE",

"carPrice":42000.0

}

The JSON string must start with a { and end with a }. All attributes (keys) must be enclosed with quotes (“”). All values that are strings also go with the quotes. Numbers do not need quotes. Commas (,) separate the key-value pairs, and the colon (:) separate a key and value within a pair. Arrays are presented with the [], and elements in the array get separated with commas:

[

{

"carId":3,

"carBrand":"Toyota",

"carModel":"4Runner",

"carModelOption":"XLE",

"carPrice":42000.0

},

{

"carId":2,

"carBrand":"Toyota",

"carModel":"Rav4",

"carModelOption":"LE",

"carPrice":30000.0

}

]

1. **Task #2: Build a sorted circular doubly linked list**

In this phase, the persistence layer with be implemented with a SortedList<T>. Here is the API for this ADT:

**public** **interface** SortedList<E> {

**public** **boolean** add(E obj);

**public** **int** size();

**public** **boolean** remove(E obj);

**public** **boolean** remove(**int** index);

**public** **int** removeAll(E obj);

**public** E first();

**public** E last();

**public** E get(**int** index);

**public** **void** clear();

**public** **boolean** contains(E e);

**public** **boolean** isEmpty();

**public** **int** firstIndex(E e);

**public** **int** lastIndex(E e);

}

The sorted list keeps its element sorted in increasing order. To compare two elements, X and Y, you need to have class called a **comparator**. The comparator implements the Java interface Comparator<E>, which has one method:

public int compare(E obj1, E obj2)

This method compares obj1 with obj2 and returns either: a) 0 – if they are equal, b) a negative number if obj1 is less than obj2, or c) a positive number if obj1 is greater than obj2.

You will implement the SortedList using a **circular doubly linked list with a dummy header**. As shown in Figure 2:

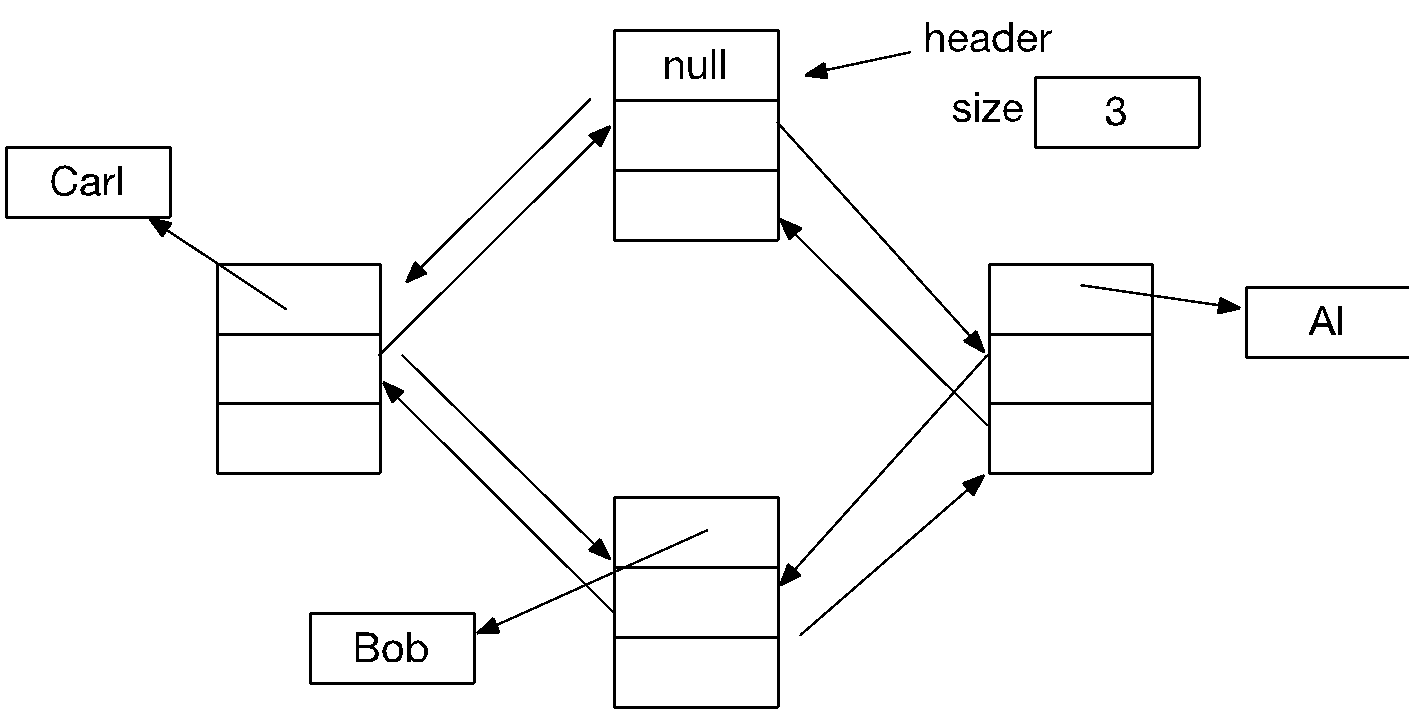


Figure 2: Sorted Circular doubly linked list

This list has a dummy header. Each node has three references: a) element – the object being stored, b) next – a reference to the next node in the chain, and c) prev – reference to the previous node. The elements are kept in increasing sorted order based of the comparator. The list also has a field to keep its current size. **An empty list has size equal to 0, and one dummy header with its previous and next references pointing to itself.**

**Programming Tasks:**

To complete this phase of the project you will need to carry out the following tasks:

1. Download and install Eclipse Enterprise Edition. This version support web development.
2. Clone the gitlab repo for the starter project:

<https://gitlab.com/manuelr417/p1start.git>

1. The project is a Java project base on Apache Maven, a software project management and comprehension tool used to manage projects with external libraries (dependencies). You need to install maven on your computer. <https://maven.apache.org>
2. Rename the project as p1. Erase the .git directory. And then manage the project with your Gitlab account.
3. You only need to add code in the directory src/main/java. **Do not modify anything else**.
4. There is a Car class implemented in the package:

edu.uprm.cse.datastructures.cardealer.model

1. In this package, implement a **CarComparator** class that compares two cars based on a string formed with brand, model and options. For example, ToyotaRav4LE goes before ToyotaRav4SE. And Honda cars go before Toyota cars.
2. The SortedList<E> ADT is located in package edu.uprm.cse.datastructures.cardealer.util.
3. In this package, implement the CircularSortedDoublyLinkedList class.
4. The REST API is implemented in the package edu.uprm.cse.datastructures.cardealer. There is already a class there named Main which is the main program for the application.
5. In this package, implement a class named CarManager that implements the REST API to manage a Car. It must have 5 methods :
   1. A method to read all cars as an array of Car. The URI will be :

http://localhost:8080/cardealer/cars

* 1. A method to read a car with a given id. The URI (route) will be:

http://localhost:8080/cardealer/cars/{id}

where {id} is a number for the car id

* 1. A method to add a new car to the system. The URI will be:

http://localhost:8080/cardealer/add

* 1. A method to update an existing car in the system. The URI will be:

http://localhost:8080/cardealer/cars/{id}/update

* 1. A method to delete an existing car from the system. The URI will be:

http://localhost:8080/cardealer/cars/{id}/delete

**How to run your app**

1. Go to the directory of the app.
2. Run the following command to compile the project:
   1. mvn clean compile
3. Run the following command to run the server:
   1. mvn exec:java

You should now be able to access the following URL from your browser:

http://localhost:8080/cardealer

But you will not see anything unless you implement the REST API and add cars to the system.

**Adding cars to the system**

You can add new cars using the **curl** command. Curl is a tool used to interact with web-based backends. Here a few examples on how to add new cars:

curl -X POST -i -H "Content-Type: application/json" -d '{"carId" : 3, "carBrand" : "Toyota", "carModel" : "4Runner", "carModelOption" : "XLE", "carPrice" : 42000}' http://localhost:8080/cardealer/cars/add

This adds a new Toyota 4Runner to the system. The output should be something like this:

HTTP/1.1 201 Created

Date: Sun, 30 Sep 2018 15:39:46 GMT

Content-Length: 0

Here is another example:

curl -X POST -i -H "Content-Type: application/json" -d '{"carId" : 2, "carBrand" : "Toyota", "carModel" : "Rav4", "carModelOption" : "LE", "carPrice" : 30000}' http://localhost:8080/cardealer/cars/add

Output:

HTTP/1.1 201 Created

Date: Sun, 30 Sep 2018 15:41:57 GMT

Content-Length: 0

**Reading the cars:**

Now that we have cars, we can read them. This command reads all cars:

curl -X GET -i <http://localhost:8080/cardealer/cars>

Output:

HTTP/1.1 200 OK

Content-Type: application/json

Date: Sun, 30 Sep 2018 15:43:45 GMT

Content-Length: 187

[{"carId":3,"carBrand":"Toyota","carModel":"4Runner","carModelOption":"XLE","carPrice":42000.0},{"carId":2,"carBrand":"Toyota","carModel":"Rav4","carModelOption":"LE","carPrice":30000.0}]

Notice that the output of the has diagnostic header OK indicating that the operation worked, and also the JSON with the array of Cars.

Now, lets get a specific car:

curl -X GET -i <http://localhost:8080/cardealer/cars/2>

Output:

HTTP/1.1 200 OK

Content-Type: application/json

Date: Sun, 30 Sep 2018 15:46:51 GMT

Content-Length: 90

{"carId":2,"carBrand":"Toyota","carModel":"Rav4","carModelOption":"LE","carPrice":30000.0}

Now, try to get a car not in the list:

curl -X GET -i http://localhost:8080/cardealer/cars/20

Output:

HTTP/1.1 404 Not Found

Date: Sun, 30 Sep 2018 15:51:21 GMT

Content-Length: 0

**Updating a Car:**

Let us change the price of the 4Runner to $54,000:

curl -X PUT -i -H "Content-Type: application/json" -d '{"carId" : 3, "carBrand" : "Toyota", "carModel" : "4Runner", "carModelOption" : "XLE", "carPrice" : 54000}' http://localhost:8080/cardealer/cars/3/update

Output:

HTTP/1.1 200 OK

Date: Sun, 30 Sep 2018 15:59:30 GMT

Content-Length: 0

**Delete a car:**

Let is now delete a car:

curl -X DELETE -i http://localhost:8080/cardealer/cars/2/delete

Output:

HTTP/1.1 200 OK

Date: Sun, 30 Sep 2018 16:01:53 GMT

Content-Length: 0

**What to submit:**

1. **You will submit your gitlab repo for grading. The last acceptable commit will be at 11:59 pm on October 15, 2018.**

In the next coming days you will be given 3 test scripts with curl operations :

1. test1– First test case. **NOTE: YOU PROGRAM MUST PASS THIS FILE WITHOUT ERRORS IN ORDER TO BE CONSIDERED A RUNNING PROGRAM.**
2. test2 – second test case.
3. test3 – third test cases

By passing all three cases, you will get at score of least 60 pts in the project.

**NOTE: Projects that do not compile will get a score of 0. Projects that do not pass test case test1 will get a score of 10.**

PROJECT DUE DATE: **11:59 PM – October 17, 2018**.

1. During the evolution of Java, people added complexity to objects representing data records to help exchange or store them automatically. However, many developers identified this asn a bad practice. Instead, they advocated for creating plain, simple Java objects to represent data, and create other objects to help store and exchange the simple objects. [↑](#footnote-ref-0)