SOLUTION

1. Design a good architecture to present your solution.

The proposed solution implements the **Business Logic**[1] over a RESTful web service. This provides an uniform interface which allows performance, scalability, and modifiability. Figure 1 shows the proposed solution where the stakeholders can interact through the Business RESTful web services.

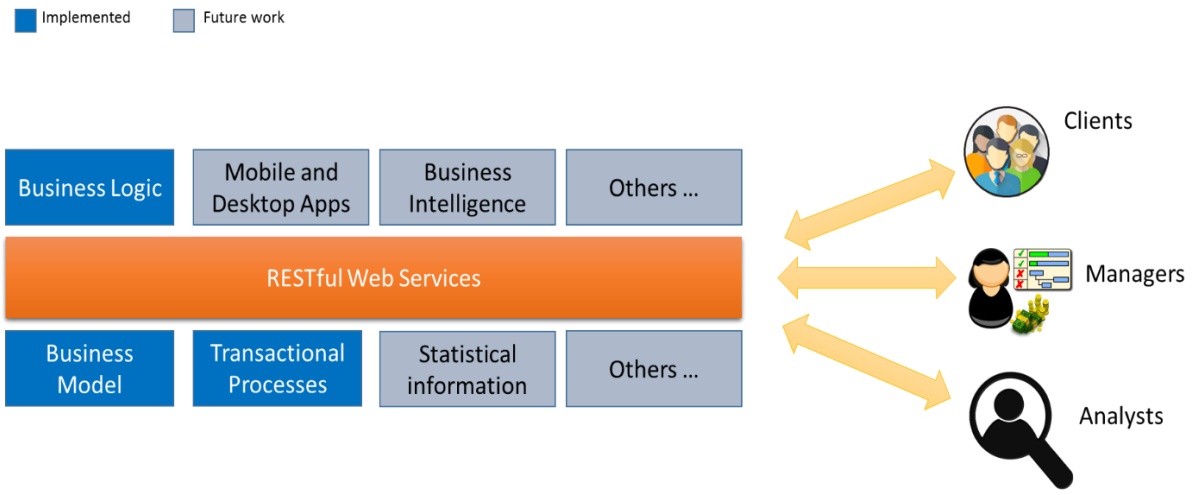

Figure 1. Business solution architecture

Business RESTful web services.

This architecture allows a simple, standard, lightweight, and fast solution, and its main principles propitiates a scalable development, here its principles:

- **Resource identification through URI**: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery.
- **Uniform interface**: Resources are manipulated using a fixed set of four create, read, update, delete operations.
- **Self-descriptive messages**: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.
- **Stateful interactions through hyperlinks**: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI

[1] Business Logic: Methods, routines, definitions and others that meets the Problem Statement. In this particular case, the renting Business.

rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.[2]

1.Web service application

This solution opens a scenery where it is possible to exchange information between several applications, systems and possibly other associated business. At the moment, the implemented solution is a two-tier architecture, therefore the business logic, the databases for the Business Model and Transactional processes are embedded in the same server. However, in my approach it can be easily adapted to a Three-Tier architecture, where the Client layer, Business layer and Data layer can be differentiated and stay separated.

I define a transaction as the process where an **input JSON string** [3] is a request for renting a car. This transaction creates an **output JSON string** [4] according to the Business Logic.

Following the Business Logic, here the implemented web services:

Sr.No.	HTTP Method	URI	Operation	Operation Type
1	**GET**	/resources/cars	Get list of available cars (XML Style)	Read Only
2	**GET**	/resources/jcars	Get list of available cars (JSON Style)	Read Only
3	**GET**	/resources/cars/{model}	Get car with Id model	Read Only
4	**POST**	/resources/cars/{model,type}	Create car with model and type	N/A
5	**DELETE**	/resources/cars/{model}	Delete car with Id Model	Idempotent
6	**OPTIONS**	/resources/cars	List the supported operations in web service	Read Only
7	**GET**	/resources/rents	Get list of executed transactions	Read Only
8	**POST**	/resources/rents/{Json Input String}	Create a transactions using the Business Logic	N/A

[2] From: https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html

[3] Input JSON string syntax: "{"rentDates":["2017-11-19T05:00:00.000Z","2017-11 20T05:00:00.000Z",…],"car":{"model":"Cherato","type":"sport"},"membership":false,"age":24}"

[4] Output JSON string syntax: "{"subtotal":350,"insuranceTotal":53, "discountPercentage":22.5, "totalPayment":324.25}"

Examples of use:

- To see the statement problem:
  - http://localhost:8080/WebService/default

- To see the available cars (XML format):
  - http://localhost:8080/WebService/resources/cars

- To see the available cars (JSON format):
  - http://localhost:8080/WebService/resources/jcars

- To see the executed transactions:
  - http://localhost:8080/WebService/resources/rents

2. Make good use of version control tools (like Git for GitHub) when presenting your solution.

   https://github.com/Borreguin/CarRentalWS/commits/master

3. Keep it simple, you won't need to over decorate your solution.
4. Production quality and maintainable code.

   The code is separated in the following folders:

business_logic.	Implement the Business Logic
Classes.	Contains the main classes for the abstraction of the Business Logic.
dataHandlers.	Deals with Data Sources
Wbservices.	Implements all web services
Settings.	Configurations about the Business and Application
Resources.	Contains any required HTML resource
TestModule.	Contains a routine for testing the code
MainPage.java.	The Main Page of the WebService
MainApp.java.	Allow to run the web service application

5. Follow the practices for good object oriented design and its patterns.

   See folder: "classes"

6. The solution must be idiomatic according to the language you've chosen to present your solution. (Use the programming language you feel most proficient with)
7. Well tested code, using Test Driven Development practices.

   See folder: "testModule"

2.Test Module

The module application runs as a web client and communicates with the GlassFish web server. It uses a Jersey Client to simulate user web forms to fill the data and commit the requests.

This module tests:

- **Web services for cars**
- **Insert the initial Cars:** If they are not yet in the business DB.
- **Transactions from the client**: Request for renting a car according to the input JSON string. It reads a JSON file as an input[5], and then the Business Logic is applied. The executed transactions[6] can be requested on "/resources/rents" service.

To run this project run first the GlassFish 5.0 web server and then the test module.

[5] In <PROJECT DIRECTORY> /src/TestModule/input.json

[6] In <GLASSFISH DIRECTORY> /domains/domain1/config/transactions.dat