Workflow Overview:

1. The client(in our case- Postman app) communicates with the backend server via URL.
2. The point of entry to the backend server is the app.js file.
3. The app.js file routes to the appropriate router method based on the request URL
4. Each router handle a particular operation (creating, adding, updating and deleting vehicles).
5. Every router method uses the mongoose framework of MongoDB to carry out the respective database operations.
6. The validations based on the input requirements are implemented at schema level using mongoose schemas.
7. Upon completion of the above operations, the results (data/success/failure messages) are sent back to the client as a json response.

Schema Details:

1. ID (Type: Number, Validation: Unique, not null)
2. Year (Type: Number, Validation: Must be between 1950 and 2050)
3. Make (Type: String, Validation: not null)
4. Model (Type: String, Validation: not null)

Router Details:

1. **vehicle/createVehicle** (request method: **POST**):

Input: Id, Year, Make, Model

Output: Failure/Success Message

Details: The ID from the request json is matched against all the existing IDs in the vehicles collection. If the ID matches an existing ID failure message is sent indicating Duplicate ID. If not, a new vehicle entity is created with the requested details. Quite a few validations are kept on place based on the requirements. These validations are implemented at the schema level (model folder) of the vehicle collection:-

1. The ID is a compulsory field and should be unique.
2. The Year must be between 1950 and 2050.
3. Make and Model fields are required while vehicle creation.

If any of these validation fails, a failure message is generated by the mongoose framework, which subsequently triggers the catch block and returns a failure message to the user. If all the validations are satisfied, a success message is sent to the client.

2. **vehicle/getVehicle**(request method: **GET**):

Input: Id or/and Year or/and Make or/and Model / blank body

Output: List of Vehicles with matching data from the request body.

Details: The records in the vehicle tables are matched against the Input body. If the request body contains multiple fields, the records are filtered on all the criteria using intersection (and) between them. In case a blank body is sent, all the records in the vehicle collection are returned.

3. **vehicle/updateVehicle**(request method: **PUT**):

Input: Id, Year and/or Make and/or Model

Output: Failure/Success Message

Details: The ID from the request json is matched against all the existing IDs in the vehicles collection. If the ID matches an existing ID, the particular vehicle record is updated using the fields mentioned in the request body. All the validations which are applied while creating the vehicles are also applied while updating them. In case the ID does not match any existing ID, the new record is inserted into the vehicles collection and the operation is treated as a create operation. If any of the validation fails, a failure message is returned. If not, a success message is returned.

4. **vehicle/deleteVehicle**(request method: **DELETE**):

Input: Id

Output: Failure/Success Message

Details: The ID from the request json is matched against all the existing IDs in the vehicles collection. If the ID matches an existing ID, the particular vehicle record is deleted. In case the ID is not present the try-catch handler of the router is triggered, indicating the absence of the particular vehicle. If the ID exists, after deleting the particular record a success message is returned.