CRUD is an acronym that stands for Create, Read, Update, and Delete. It represents the four fundamental operations required to manage persistent data in many software applications or database systems. Each operation corresponds to a specific action:

1. **Create:** The Create operation refers to the action of creating new data or records in a database or storage system. It involves inserting new data into the system, typically by providing input values for the different fields or attributes of the data structure.

1. **Read:** The Read operation involves retrieving or reading existing data or records from a database or storage system. It allows you to access and view the stored information based on specific criteria, such as searching for records that match certain conditions or fetching all records in a particular table.

1. Update: The Update operation enables modifying or updating existing data or records in a database or storage system. It allows you to change the values of specific fields or attributes within a record, either for a single record or multiple records that meet certain criteria.

1. Delete: The Delete operation involves removing or deleting existing data or records from a database or storage system. It allows you to eliminate specific records from the system based on certain conditions or criteria, effectively erasing them from the data store.

CRUD operations are fundamental to many applications that rely on persistent data storage, such as web applications, content management systems, e-commerce platforms, and more. These operations provide a standardized way to manage data and interact with databases or storage

systems. By implementing CRUD functionality, developers can create, retrieve, update, and delete data, allowing users to perform essential operations on the application's data.

When we are building APIs, we want our models to provide four basic types of functionality. The model must be able to Create, Read, Update, and Delete resources. Computer scientists often refer to these functions by the acronym CRUD. A model should have the ability to perform at most these four functions in order to be complete. If an action cannot be described by one of these four operations, then it should potentially be a model of its own.

The CRUD paradigm is common in constructing web applications, because it provides a memorable framework for reminding developers of how to construct full, usable models. For example, let's imagine a system to keep track of library books. In this hypothetical library database, we can imagine that there would be a `books` resource, which would store `book` objects. Let's say that the `book` object looks like this:

```
"book": {
  "id": <Integer>,
  "title": <String>,
  "author": <String>,
  "isbn": <Integer>
}
```

To make this library system usable, we would want to make sure there were clear mechanisms for completing the CRUD operations:

Create — This would consist of a function which we would call when a new library book is being added to the catalog. The program calling the function would supply the values for `"title"`, `"author"`, and `"isbn"`. After this function is called, there should be a new entry in the `books` resource corresponding to this new book. Additionally, the new entry is assigned a unique `id`, which can be used to access this resource later.

Read — This would consist of a function which would be called to see all of the books currently in the catalog. This function call would not alter the books in the catalog - it would simply retrieve the resource and display the results. We would also have a function to retrieve a single book, for which we could supply the title, author, or ISBN. Again, this book would not be modified, only retrieved.

Update — There should be a function to call when information about a book must be changed. The program calling the function would supply the new values for `"title"`, `"author"`, and `"isbn"`. After the function call, the corresponding entry in the `books` resource would contain the new fields supplied.

Delete — There should be a function to call to remove a library book from the catalog. The program calling the function would supply one or more values (`"title"`, `"author"`, and/or `"isbn"`) to identify the book, and then this book would be removed from the `books` resource. After this function is called, the `books` resource should contain all of the books it had before, except for the one just deleted.

CRUD and REST

In a REST environment, CRUD often corresponds to the HTTP methods POST, GET, PUT, and DELETE, respectively. These are the fundamental elements of a persistent storage system.

Throughout the rest of the article, we will recommend standards and response codes that are typically followed by developers when creating RESTful applications. Conventions may differ so feel free to experiment with different return values and codes as you become comfortable with the CRUD paradigm.

Imagine we are working with a system that is keeping track of meals and their corresponding prices for a restaurant. Let's look at how we would implement CRUD operations.

Create

To create resources in a REST environment, we most commonly use the HTTP POST method. POST creates a new resource of the specified resource type.

For example, let's imagine that we are adding a new food item to the stored list of dishes for this restaurant, and the `dish` objects are stored in a `dishes` resource. If we wanted to create the new item, we would use a POST request:

Request:

```
POST http://www.myrestaurant.com/dishes/
```

Body -

```
{
  "dish": {
    "name": "Avocado Toast",
    "price": 8
  }
}
```

This creates a new item with a `name` value of `"Avocado Toast"` and a `price` value of 8. Upon successful creation, the server should return a header with a link to the newly-created resource, along with a HTTP response code of 201 (CREATED).

Response:

Status Code - 201 (CREATED)

Body -

```
{
  "dish": {
    "id": 1223,
    "name": "Avocado Toast",
    "price": 8
  }
}
```

From this response, we see that the `dish` with `name` "Avocado Toast" and `price` 8 has been successfully created and added to the `dishes` resource.

Read

To read resources in a REST environment, we use the GET method. Reading a resource should never change any information - it should only retrieve it. If you call GET on the same information 10 times in a row, you should get the same response on the first call that you get on the last call.

GET can be used to read an entire list of items:

Request:

```
GET http://www.myrestaurant.com/dishes/
```

Response: Status Code - 200 (OK)

Body -

```
{
  "dishes": [
    {
      "id": 1,
      "name": “Spring Rolls”,
      "price": 6
    },
    {
      "id": 2,
      "name": “Mozzarella Sticks”,
      "price": 7
    },
    ...
    {
      "id": 1223,
      "name": “Avocado Toast”,
      "price": 8
    },
    {
      "id": 1224,
      "name": “Muesli and Yogurt”,
      "price": 5
    }
  ]
}
```

GET requests can also be used to read a specific item, when its `id` is specified in the request:

Request:

```
GET http://www.myrestaurant.com/dishes/1223
```

Response: Status Code - 200 (OK)

Body -

```
{
  "id": 1223,
  "name": “Avocado Toast”,
  "price": 8
}
```

After this request, no information has been changed in the database. The item with `id` 1223 has been retrieved from the `dishes` resource, and not modified. When there are no errors, GET will return the HTML or JSON of the desired resource, along with a 200 (OK) response code. If there is an error, it most often will return a 404 (NOT FOUND) response code.

Update

PUT is the HTTP method used for the CRUD operation, Update.

For example, if the price of Avocado Toast has gone up, we should go into the database and update that information. We can do this with a PUT request.

Request:

```
PUT http://www.myrestaurant.com/dishes/1223
```

Body -

```
{
  "dish": {
    "name": “Avocado Toast”,
    "price": 10
  }
}
```

This request should change the item with `id` 1223 to have the attributes supplied in the request body. This `dish` with `id` 1223 should now still have the `name` “Avocado Toast”, but the `price` value should now be 10, whereas before it was 8.

Response: Status Code - 200 (OK)

Body -

```
{
  "dish": {
    "name": “Avocado Toast”,
    "price": 10
  }
}
```

The response includes a Status Code of 200 (OK) to signify that the operation was successful. Optionally, the response could use a Status Code of 204 (NO CONTENT) and not include a response body. This decision depends on the context.

Delete

The CRUD operation Delete corresponds to the HTTP method DELETE. It is used to remove a resource from the system.

Let’s say that the world avocado shortage has reached a critical point, and we can no longer afford to serve this modern delicacy at all. We should go into the database and delete the item that corresponds to “Avocado Toast”, which we know has an `id` of 1223.

Request:

```
DELETE http://www.myrestaurant.com/dishes/1223
```

Such a call, if successful, returns a response code of 204 (NO CONTENT), with no response body. The `dishes` resource should no longer contain the `dish` object with `id` 1223.

Response: Status Code - 204 (NO CONTENT)

Body - None

Calling GET on the `dishes` resource after this DELETE call would return the original list of dishes with the `{"id": 1223, "name": "Avocado Toast", "price": 10}` entry removed. All other `dish` objects in the `dishes` resource should remain unchanged. If we tried to call a GET on the item with `id` 1223, which we just deleted, we would receive a 404 (NOT FOUND) response code and the state of the system should remain unchanged.

Calling DELETE on a resource that does not exist should not change the state of the system. The call should return a 404 response code (NOT FOUND) and do nothing.

CRUD Practice

The functions to Create, Read, Update, and Delete resources are fundamental components of a usable storage model. You have now seen a couple of examples for how the CRUD paradigm can help us design systems. Now, try to use CRUD to list out routes for a new example model. Imagine we are trying to design a system that keeps track of workout classes, including the name of each class, who teaches it, and the duration of the class. An example `class` object would look like:

```
{
  "class": {
    "id": 1
    "name": "Pure Strength",
    "trainer": "Bicep Bob",
    "duration": 1.5
   }
}
```

All of the classes are stored in a `classes` resource at `www.musclecademy.com/classes`.

For each CRUD operation, write out answers to the following questions:

- What routes would you need to implement to provide your workout class model with this CRUD functionality and what are their corresponding HTTP verbs?
- What effect would each route have on the database?
- What response body would each route return?
- What response code would each route return?