Endpoint Specification

DE BOER, Lucas (K2023556)

Kingston University

1 Introduction

A RESTful API is being developed using the Express.js framework. This framework enables the creation of a backend that communicates with a MySQL database. The API supports standard functions such as GET, POST, PUT, and DELETE to retrieve resources from the database. A set of endpoints must be created for the web-based front end to use this RESTful API. This document will outline the different endpoint syntaxes used to allow the system to support the full range of user stories and requirements. Specifying the syntaxes ensures that the implementation of endpoints is consistent throughout the system.

2 RESTful APIs

An API or Application Programming Interface enables an application or service to access a resource within another application or service. The “client” refers to the application requesting access, while the “server” is the application or service containing the resources.

REST or Representational State Transfer imposes a set of guidelines or conditions on how an API should work. APIs which follow the REST architecture style are called RESTful APIs. The REST architecture style includes principles such as statelessness and uniform interfaces.

Statelessness implies the server will not store anything about the client's latest HTTP request. It will execute every client request independently of all previous requests. Thus, the client can request data at any time in any order, and the server will be able to fulfil these requests.

A uniform interface for interacting with resources should be adhered to across the system. This ensures functional communication via HTTP requests to perform standard database functions like creating, reading, updating, and deleting records within a resource.

3 Endpoint Syntax Formatting

The following endpoints have adhered to a couple of best design pattern practices to ensure they are consistent and easy to understand. All URLs include nouns and do not include verbs. These nouns are always plural for consistency throughout the system. Furthermore, while not considered best practice, all endpoint syntaxes will follow a pattern of defining the object type that wants to be returned as the first noun in the endpoint following /API. Thus, an endpoint to return all boats would look like “/api/boats”.

An endpoint has two parts: the path parameter and the query parameter. Figure 3.0 shows the endpoint sections. Query parameters are attached to the end of a URL and are found following the "?". The query parameter defines operations such as sort, pagination, and filtering. The user input is also passed as a variable. Multiple queries can be passed using the "&" symbol to join individual queries. The Section before the "?" is the path parameter that defines the location of the desired resource.

```
/api/path_parameter?query_parameter
```

Figure 3.0 Showing the Path and Query Parameters of an Endpoint

3.1 Lists

This endpoint will fetch all the objects of a specified type from the database to be displayed by the front end in a list. The syntax for these endpoints is the pluralised name of the object, which must be returned. Figure 3.1 shows examples of this syntax being used in the system.

```
/api/boats
/api/employees
/api/bookings
/api/items
```

Figure 3.1 Showing the "List" Type Endpoints

3.2 Add a new object to the list

The "POST" endpoint will be identical to the "lists" endpoints in section 3.1. The system can differentiate between the two. The "POST" will have a JSON object attached to the request, while the list "GET" will not.

```
/api/boats
/api/employees
/api/bookings
/api/items
```

Figure 3.2 Showing the "POST" Type Endpoints

3.3 View a single object's details

These endpoints are used when a user clicks on a card in a list to view its details. This could be for user requirements such as "view watercraft details", "view Employee profile", or "view Booking Details". Figure 3.3 shows these endpoints.

```
/api/boats/details/:id
/api/employees/profiles/:empid
/api/bookings/details/:bid
/api/items/details/:id
```

Figure 3.3 Showing the "View Details" Type Endpoints

The /details after the initial /objectType help describe to the end user what the endpoint is retrieving. The /: id, /:empid or /: bid allows a request parameter to be passed through the endpoint URL. This parameter, for example, can later be extracted to use in a "WHERE" clause to fetch the object with the specified ID from the database.

3.4 Edit object

These "PUT" endpoints will allow for the updating of objects. The ID of the object that needs to be updated is sent as a request parameter. This is shown by the /: id in Figure 3.4.

```
/api/boats/:id

/api/employees/:id

/api/bookings/:id

/api/items/:id
```

Figure 3.4 Showing the "PUT" Type Endpoints

The /: id syntax was chosen because it was easy to implement. It did not require any parsing to extract the ID from the URL. Instead, it is attached to the request object and can be directly retrieved through the param function.

3.5 Filtering

The system will follow two filtering syntax structures depending on whether the attribute being filtered is present in the same table as the object that must be displayed. In other words, if the filtering attribute is a foreign key, the endpoint will follow a different syntax than if a native, non-foreign attribute was used. To help understand this, look at the UML class diagrams in Figures 3.5.1 and 3.5.2.

The status attribute in Figure 3.5.1 cannot be filtered directly as the status of a boat is a foreign key to the "BoatStatus" table. Due to two tables being required, the syntax shown in Figure 3.5.3 is used to express this explicitly. Where the object type retrieved is the leftmost noun and the attribute filtered by is the rightmost. The "Status_ID" is attached to the request as a parameter /: id. With this, the system can support stories such as "Get all Watercraft Currently out on the water" or even "Get all employees who are currently not on a charter", as employee status is set up similarly to boat status in the database.

Figure 3.5.2 shows the boat "Models" table. If the system was required to filter the models by "Weight Limit", then the endpoint syntax shown in Figure 3.5.4 will be used. The syntax shown would result in all the models with a weight limit greater than or equal to 2000 but less than or equal to 4000. To reiterate, this syntax is only used when the "filter by" attribute is not a foreign key and is instead a local, native attribute in the same table as the object that will be returned.

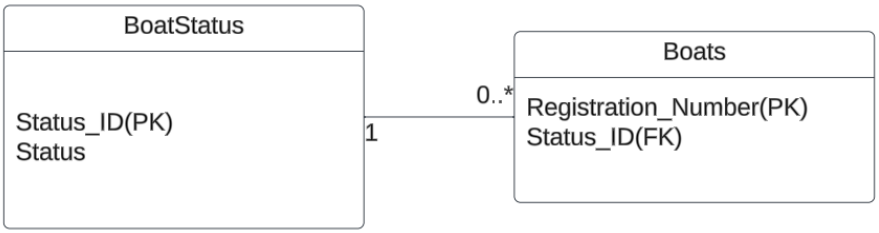

Figure 3.5.1 UML Class Diagram showing "Boats" and "BoatStatus" Tables

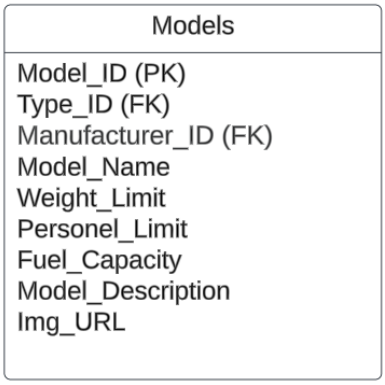

Figure 3.5.2 UML Class Diagram showing "Models" Table.

```
/api/boats/status/:id

/api/employees/status/:id
```

Figure 3.5.3 Showing Endpoint Syntax when two tables are required (Foreign key situations)

```
/api/models?weight_limit=gte:2000&weight_limit=lte:4000
```

Figure 3.5.4 Showing Endpoint Syntax when a Filtering Attribute is native to the table

3.6 Searching

All the list endpoints will require searching capabilities to support "search" oriented user stories such as "search for employee by name".

The main advantage of using the syntax shown in Figure 3.6 is that almost no parsing is required on the backend. A disadvantage, however, is that the inputs can be passed directly to the database if not adequately handled by the backend.

```
/api/employees?name=john

/api/item?name=lifejacket
```

Figure 3.6 Showing the Search Endpoint Syntax

3.7 Pagination

Without pagination, a search could return millions or even billions of results, causing unnecessary network traffic. Pagination allows for retrieving data from the database in smaller, specified chunks. In the context of this system, the list view for watercraft, employees, bookings, and equipment currently fetches and shows every object from their respective tables on one page. This is inefficient and makes browsing the list view impossible, as the user could have to scroll through thousands of results.

Using watercraft as an example, the syntax shown in Figure 3.7 would allow only twenty watercraft to be fetched from the database and displayed on a page at a time. When the user clicks "next page", an offset of twenty is added to the endpoint, indicating the database to start from the

twentieth row to fetch the next twenty watercraft. This is repeated for every “next page” click. Each page will fetch only what must be displayed, reducing the network traffic.

```
/api/boats?limit=20

/api/boats?limit=20&offset=20

/api/boats?limit=20&offset=40

…
```

Figure 3.7 Showing the Pagination Endpoint Syntax with Watercraft(boats)

3.8 Sorting

The endpoint sorting syntax allows the backend to receive data from the database in a specified order. The syntax shown in Figure 3.8.1 will enable the system to support “sorting” type requirements such as “Sort Booking by Date” or “Sort Employees by Name”. Furthermore, the order can be ascending or descending using the syntax shown in Figure 3.8.2.

```
/api/bookings?sort_by=date

/api/employees?sort_by=name

/api/items?sort_by=name
```

Figure 3.8.1 Showing the Sorting Endpoint Syntax

```
/api/bookings?sort_by=date&order_by=desc

/api/employees?sort_by=name&order_by=asc

/api/items?sort_by=name&order_by=desc
```

Figure 3.8.1 Showing the Sorting Endpoint Syntax with OrderBy

4 API and Endpoint Validation

Validation of the endpoints was conducted to ensure the specified syntaxes are functional and meet the system's requirements. Following the guidelines set in section three, endpoints were created for each primary database entity, such that filtering, sorting, pagination, listing and editing endpoints were made for each and tested individually.

A table was created specifying the expected and actual results of an endpoint. These results are the JSON objects with the necessary data to accomplish specific requirements. As suggested, the predicted result was based purely on what fields are required to perform user stories. The actual result, however, was obtained after implementing the endpoint and using Visual Studio Code’s “Postman” extension, which allows for the testing of endpoints as it can communicate with the backend database by sending or receiving JSON objects and displaying them in a console. “Postman” uses the endpoint syntax specified during the implementation, as shown in Figure 4.1. If the JSON received from “Postman” matches the expected result, the test is marked as “Passed” in the outcome column. Table 4.2 shows only a couple of rows of the validation table, while the comprehensive table can be found in Appendix A.

Figure 4.1 Showing the "Get all Watercraft" endpoint syntax when testing in Postman.

Type	Endpoint	Body	Expected result	Actual result	Outcome
GET	/api/boats	N/A	One or more JSON objects of format { "Synthetic_key", "Registration", "Boat Img", "Model name", "Manufacturer_name", "Img URL", "Model_Description", "Weight Limit lbs", "Fuel Capacity Gal", "Type", "Status", "Model_ID", "Status ID" }	{ "Synthetic_Key", "Registration", "Boat_Img", "Model Name": "Aspire 100", "Manufacturer_Name": "Wilderness", "Img URL": " ", "Model Description": "High performance", "Weight Limit lbs": 300, "Seating": 1, "Fuel_Capacity_Gal": null, "Type": "Kayak", "Status": "Available", "Model_ID": 4, "Status ID": 1 }	Passed
POST	/api/employees	{ "Employee_ID": 12345, "Job_ID": 1, "Employee_Status_ID": 2, "Employee_Name": "John, "Start_Date": null, "NIN": 88898, "Phone_Number": "3330099", "Email": "john@gmail.com" }	The employee is added to the database and shown on the screen.	The employee is added to the database and displayed on the screen.	Passed

Table 4.2 Showing a couple of rows from the "API and Endpoint Validation" table.

5 Conclusion

This document provides a clear and concise outline of the syntax used in various endpoint situations, such as filtering, sorting, pagination, listing, and editing. Most of the syntaxes provided follow best practices; however, as outlined in section 3.5, filtering has two syntax structures which could be followed depending on the relationship of entities. By following the syntax structure outlined, the consistent implantation of endpoints can be achieved. Endpoint syntaxes were validated to ensure they were correct and supported the necessary system requirements. The tests for these validations could only be conducted after the back-end implantation of the endpoints, creating a situation in which not all endpoints could be tested and instead must be tested during development.

Appendix

Appendix – A

Where if outcome and actual result is left null/blank assume the endpoint has not been implemented as yet and is not ready for testing.

API and Endpoint Validation

Type	Endpoint	Body	Expected result	Actual result	Outcome
GET	/api/boats	N/A	One or more JSON objects of format { “Synthetic_key”, “Registration”, “Boat_Img”, “Model_name”, “Manufacturer_name”, “Img_URL”, “Model_Description”, “Weight_Limit_lbs”, “Fuel_Capacity_Gal”, “Type”, “Status”, “Model_ID”, “Status_ID” }	{ "Synthetic_Key", "Registration", "Boat_Img", "Model_Name": "Aspire 100", "Manufacturer_Name": "Wilderness", "Img_URL": " ", "Model_Description": "High performance", "Weight_Limit_lbs": 300, "Seating": 1, "Fuel_Capacity_Gal": null, "Type": "Kayak", "Status": "Available", "Model_ID": 4, "Status_ID": 1 }	Passed

GET	/api/boats?limit=20	N/A	{
			“Synthetic_key”,
			“Registration”,
			“Boat_Img”,
			“Model_name”,
			“Manufacturer_name”,
			“Img_URL”,
			“Model_Description”,
			“Weight_Limit_lbs”,
			“Fuel_Capacity_Gal”,
			“Type”, “Status”,
			“Model_ID”,
			“Status_ID”
			}		

GET	/api/boats/details/3	N/A	One JSON object of format
			{
			“Synthetic_key”,
			“Registration”,
			“Boat_Img”,
			“Model_name”,
			“Manufacturer_name”,
			“Img_URL”,
			“Model_Description”,
			“Weight_Limit_lbs”,
			“Fuel_Capacity_Gal”,
			“Type”, “Status”,
			“Model_ID”,
			“Status_ID”
			}
				{
				“Synthetic_key”:3,
				“Registration”:
				“152488”,
				“Boat_Img”,
				“Model_name”,
				“Manufacturer_name”,
				“Img_URL”,
				“Model_Description”,
				“Weight_Limit_lbs”,
				“Fuel_Capacity_Gal”,
				“Type”, “Status”,
				“Model_ID”,
				“Status_ID”
				}
					Passed

GET	/api/boats/status/4	N/A	One JSON object of format { “Synthetic_key”, “Registration”, “Boat_Img”, “Model_name”,	All boats with the status ID of 4 were shown. { “Synthetic_key”:3, “Registration”: “152488”,	Passed

			“Manufacturer_name”, “Img_URL”, “Model_Description”, “Weight_Limit_lbs”, “Fuel_Capacity_Gal”, “Type”, “Status”, “Model_ID”, “Status_ID” } With “Status_ID”: 4	“Boat_Img”, “Model_name”, “Manufacturer_name”, “Img_URL”, “Model_Description”, “Weight_Limit_lbs”, “Fuel_Capacity_Gal”, “Type”, “Status”, “Model_ID”, “Status_ID”: 4 }	

GET	/api/boats?registration=“152488”		A JSON object of format
			{
			“Synthetic_key”,
			“Registration”,
			“Boat_Img”,
			“Model_name”,
			“Manufacturer_name”,
			“Img_URL”,
			“Model_Description”,
			“Weight_Limit_lbs”,
			“Fuel_Capacity_Gal”,
			“Type”, “Status”,
			“Model_ID”,
			“Status_ID”
			}
			With “Registration”: “152488”
				All boats with the status ID of 4 were shown.
				{
				“Synthetic_key”:3,
				“Registration”: “152488”,
				“Boat_Img”,
				“Model_name”,
				“Manufacturer_name”,
				“Img_URL”,
				“Model_Description”,
				“Weight_Limit_lbs”,
				“Fuel_Capacity_Gal”,
				“Type”, “Status”,
				“Model_ID”,
				“Status_ID”
				}
					Passed

POST	/api/boats	{ "Synthetic_Key": 1, "Registration": "TEST1920980", "Boat_Img": null, "Model_ID": 2, "Status_ID": 1	The boat is added to the database and shown on the screen.	The boat is added to the database and displayed on the screen.	Passed

		}			

PUT	/api/boats/:1	{
		"Synthetic_Key": 1,
		"Registration":
		"UPDATED1920980",
		"Boat_Img": null,
		"Model_ID": 3,
		"Status_ID": 2
		}
			The boat is updated with the new info	The boat is updated with the latest info	Passed

GET	/api/employees	N/A	One or more JSON objects of format
			{
			“Employee_ID”,
			“Job_Name”,
			“Employee_Status”,
			“Employee_Name”,
			}
				{
				"Employee_ID": 10987,
				"Employee_Name":
				"Amelia Thompson",
				"Employee_Status":
				"Available",
				"Job_Name":
				"Deckhand"
				},
					Passed

GET	/api/employees/profiles/12345	N/A	A JSON object of format
			{
			“Employee_ID”,
			“Job_Name”,
			“Employee_Status”,
			“Employee_Name”,
			“Start_Date”,
			“NIN”,
			“Phone_Number”,
			“Email”
			}
			With “Employee_ID”: 12345,		

GET	/api/employees?name=John	N/A	A JSON object of format {		

			“Employee_ID”, “Job_Name”, “Employee_Status”, “Employee_Name”, “Start_Date”, “NIN”, “Phone_Number”, “Email” } With “Employee_Name”: “John”,		

POST	/api/employees	{
		“Employee_ID”: 12345,
		“Job_ID”: 1,
		“Employee_Status_ID”: 2,
		“Employee_Name”:
		“John,
		“Start_Date”: null,
		“NIN”: 88898,
		“Phone_Number”:
		“3330099”,
		“Email”:
		“john@gmail.com”
		}
			The employee is added to the database and shown on the screen.	The employee is added to the database and displayed on the screen.	Passed

PUT	/api/employees/12345	{ “Employee_ID”: 12345, “Job_ID”: 1, “Employee_Status_ID”: 1, “Employee_Name”: “John, “Start_Date”: null, “NIN”: 88898, “Phone_Number”: “3330099”,	The employee is updated with the new info.	The employee is updated with the latest info.	Passed

		"Email": "johndsafd@gmail.com" }			
GET	/api/bookings	N/A	One or more JSON objects of format { "Booking_Number", "Charter_Type", "Date", "Duration", "Principal_Guest", "Registration" }	{ "Booking_Number": null, "Charter_Type": null, "Date": null, "Duration": null, "Principal_Guest": null, "Registration": null }	Passed
GET	/api/bookings/details/:bid	N/A	A JSON object of format { "Booking_Number", "Charter_Type", "Date", "Duration", "Booking_Notes", "Principal_Guest", "Registration" }	{ "Booking_Number": null, "Charter_Type": null, "Date": null, "Duration": null, Booking_Notes": null, "Principal_Guest": null, "Registration": null }	Passed
GET	/api/bookings?booking=1	N/A	A JSON object of format { "Booking_Number", "Charter_Type", "Date", "Duration", "Principal_Guest", "Registration" } With "Booking_Number": 1,		

GET	/api/bookings?sort_by=date	N/A	One or more JSON objects of format
			{
			“Booking_Number”,
			“Charter_Type”,
			“Date”, “Duration”,
			“Principal_Guest”,
			“Registration”
			}
			
			The returned bookings are in ascending order of date.		

POST	/api/bookings	A JSON object of format
		{
		“Booking_Number”,
		“Charter_Type”,
		“Date”, “Duration”,
		”Booking_Notes”,
		“Principal_Guest”,
		“Registration”
		}
			The booking is added to the database and shown on the screen.		

PUT	/api/bookings/1	A JSON object of format
		{
		“Booking_Number”: 1,
		“Charter_Type”,
		“Date”, “Duration”,
		”Booking_Notes”,
		“Principal_Guest”,
		“Registration”
		}
			The booking is updated with the new info.		

GET	/api/items	N/A	One or more JSON objects of format { “Item_ID”,		

			"Item_Name", "Item_Type", "Item_Status" }		

GET	/api/items/details/:bid	N/A	A JSON object of format
			{
			"Item_ID",
			"Item_Name",
			"Item_Type",
			"Type_Description",
			"Item_Status"
			}		

GET	/api/items?name=lifejacket	N/A	One or more JSON objects of format
			{
			"Item_ID",
			"Item_Name",
			"Item_Type",
			"Item_Status"
			}		

POST	/api/items	{
		"Item_ID": 00002,
		"Item_Name":
		"Wakeboard",
		"Item_Type_ID": 2,
		"Item_Status_ID": 1
		}
			The item is added to the database and shown on the screen.		

PUT	/api/items/:id	{
		"Item_ID": 00002,
		"Item_Name":
		"Wakeboard",
		"Item_Type_ID": 2,
		"Item_Status_ID": 1
		}
			The item is updated with the new info.		

K2023556 Lucas de Boer
