

Coursework2

by Aliu Djalo

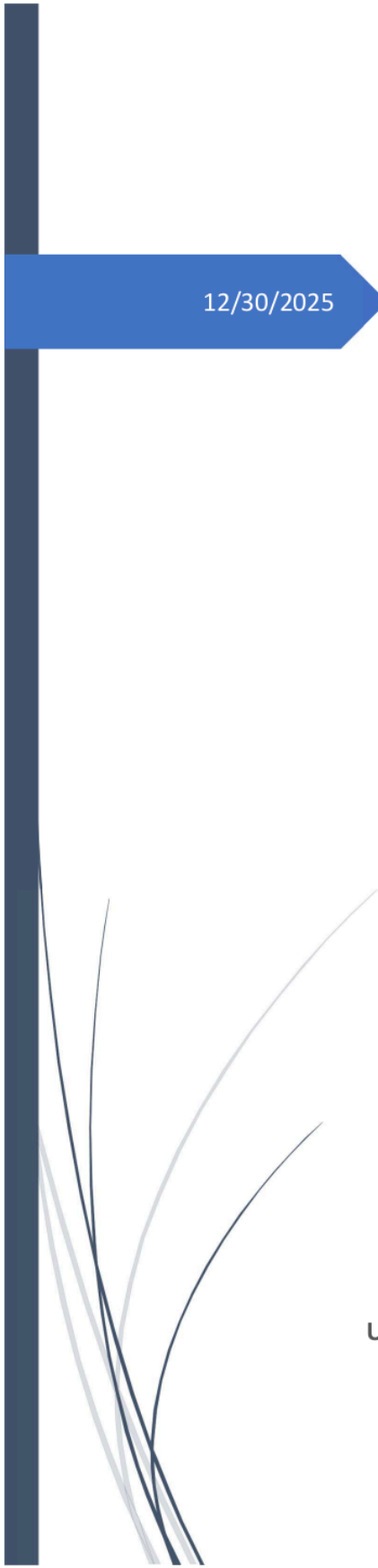
Submission date: 05-Jan-2026 12:08PM (UTC+0000)

Submission ID: 272115123

File name: 34332_Aliu_Djalo_Coursework2_708810_1977023203.docx (330.7K)

Word count: 1973

Character count: 11069



12/30/2025

Cloud Based Web Services (ACCA6024)

Assignment 2: Cloud Based Web Service

Aliu Djalo (2117368)

UNIVERSITY OF WALES TRINITY SAINT DAVID

Table of Contents

1.	Introduction.	2
2.	Organisation and application scope.....	2
3.	User interface design.	3
4.	REST API integration using AJAX (Fetch).	4
5.	Features Implemented.	5
5.1.	Displaying Products and item details.	5
5.2.	Create, update and delete products.	5
5.3.	Cart operations (add, edit quantity and delete).....	6
5.4.	Two stage workflows: Cart and Checkout.	6
5.5.	Search, filtering and polling.....	7
6.	Client-side state and feedback.	7
7.	Testing and evidence.....	7
8.	Implementation summary.....	8
8.1.	How to run the application.....	8
9.	Evaluation and future improvements.	9
10.	Conclusion.....	9

1. Introduction.

This coursework continues from Coursework 1, where a Shopping Cart REST API was built in Node.js/Express. In Coursework 2. The author created a browser-based single-page application (SPA) that uses that API and provides a small shopping experience for users. The back end runs locally at ***http://localhost:3000*** and shows endpoints for products, cart operations, and checkout.

The goal is a single-page interface that loads data from the API, updates the screen dynamically without refreshing the whole page and allows the user to view item details and perform create/edit/delete actions through AJAX calls. [1]

The chosen company is a small grocery shop called “Mini Shop”. Users can browse the catalogue, see item details, add items to a cart, edit quantities, remove items and complete checkout.

Checkout represents the workflow with two stages: first the cart is prepared, then the purchase is confirmed, and the cart is cleared. [2]

The application gives feedback after every request, so it is obvious when the API has changed something.

2. Organisation and application scope.

The solution is simple: One HTML page, one CSS file and one JavaScript. Keeping the structure simple makes it easier to see how SPA behaviour is created mainly through JavaScript (event listeners + AJAX) rather than through many separate pages.

On load, ***app.js*** performs GET /products and GET /cart, then displays both tables into the DOM. User actions trigger REST calls and refresh only the affected section (products or cart). This is what makes the page feel live, the browser replaces table rows and totals instantly rather than reloading a new HTML document.

The core user journey is search the product list, click a product to view details, add it to the cart, adjust quantities, then check out. This reflects a realistic e-commerce pattern and makes the workflow requirement easy to demonstrate.

3. User interface design.

The UI is split into two columns. The left side focuses on the product catalogue: a searchable table plus a form to create (and edit) products. The right side focuses on the cart: a cart table, totals, checkout, and an “Item details” area that updates instantly when the user selects a product.

A header shows an API status badge (online/offline) and a ‘Last update’ timestamp. A temporary on screen notification appears to confirm success and highlight errors (e.g. empty checkout). This feedback matters because AJAX pages do not naturally show progress through reloads.

The dark theme and consistent spacing were chosen to make sections easy to scan. Buttons use different styles for primary actions (e.g., Checkout) versus critical actions *e. g. Delete*. This helps users avoid accidental actions, especially when demonstrating CRUD operations.

The author also kept interaction patterns consistent: actions that change data always trigger a toast message and then a table refresh. For example, creating a product shows a success message, clears the form fields, and refreshes the products table.

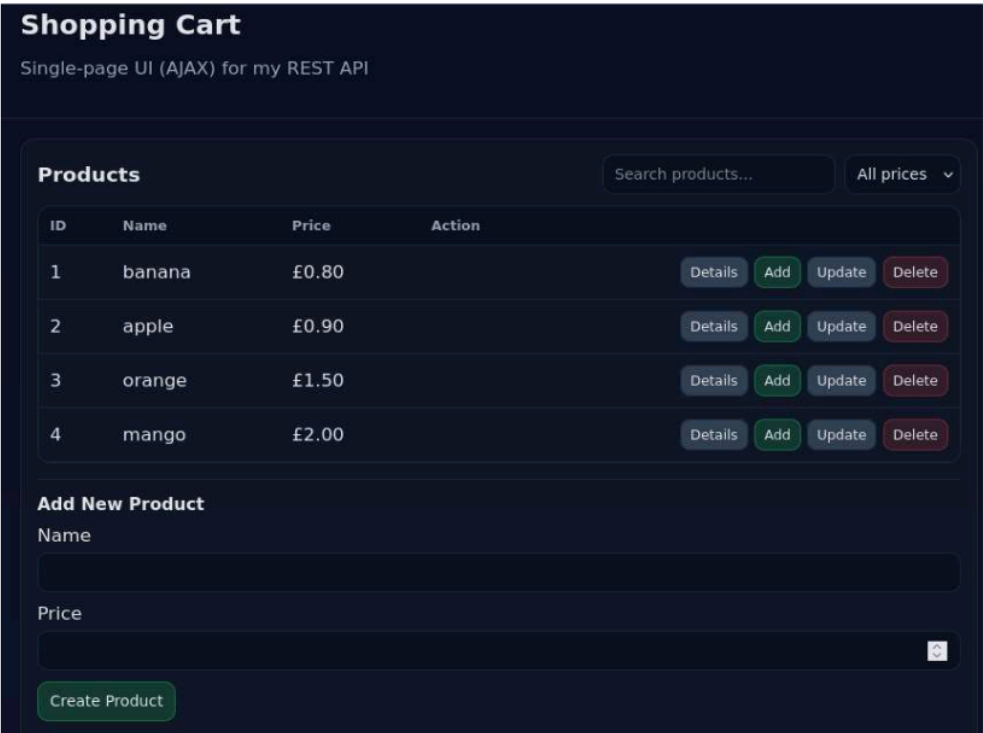


Figure 1: Products loaded from GET /products with action buttons (Details, Add, Update, Delete).

4. REST API integration using AJAX (Fetch).

All data is accessed and modified through a RESTful API. *app.js* uses the Fetch API to send HTTP requests and handle JSON responses [3] For requests with a body, JSON is sent with Content-Type: application/json, for example when creating a product or adding to the cart.

To keep the front end predictable, most actions follow the same pattern: validate input, send the request, check response status, then refresh the relevant table. This keeps the UI consistent and avoids duplicated logic.

The API uses standard HTTP status codes as defined in RFC 9110. [4] Resource creation returns 201 (Created), while successful deletion operations return 204 (No Content). The UI checks status codes and shows clear messages when an operation fails (for example, checkout on an empty cart).

In addition, each request is wrapped in try/catch logic. If the API is offline or the network fails, the UI shows an 'offline' status and a helpful message rather than crashing.

This table summarises the endpoints used by the SPA and shows how each one maps to a UI action.

Method + Route	Purpose	UI trigger
GET /	Health check (status badge)	Load / refresh
GET /products	Load catalogue	Load / refresh / poll
GET /products/:id	Load details	Row click / Details
POST /products	Create product	Create form
DELETE /products/:id	Delete product	Delete button
PUT/PATCH or DELETE+POST	Update product	Update form
GET /cart	Load cart	Load / refresh / poll
POST /cart	Add/increase item	Add buttons
PATCH /cart/:productId	Update quantity	Qty controls
DELETE /cart/:productId	Remove item	Remove button
POST /checkout	Checkout + clear	Checkout

5. Features Implemented.

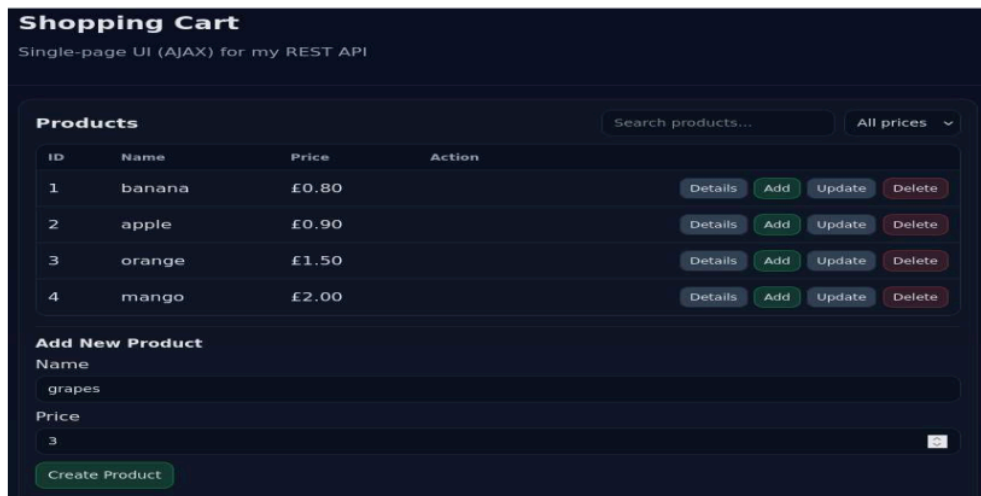
5.1. Displaying Products and item details.

GET /products displays the products table on page load. Selecting a row trigger GET /products/:id and updates the Item details panel without leaving the page. The details panel also provides a quantity input and an 'Add to cart' button for the selected product.

This design satisfies the requirement to show item details via a click mechanism, and it keeps the cart visible while the user browses.

5.2. Create, update and delete products.

Creating products uses POST /products from the form. On 201 Created, the table refreshes and a message confirms success.



The screenshot shows a web application titled "Shopping Cart" with the subtitle "Single-page UI (AJAX) for my REST API". It features a "Products" section with a search bar and a dropdown for "All prices". Below this is a table with columns "ID", "Name", "Price", and "Action". The table lists four products: banana (£0.80), apple (£0.90), orange (£1.50), and mango (£2.00). Each product has four buttons: "Details", "Add", "Update", and "Delete". Below the table is a form to "Add New Product" with fields for "Name" (containing "grapes") and "Price" (containing "3"), and a "Create Product" button.

ID	Name	Price	Action
1	banana	£0.80	Details Add Update Delete
2	apple	£0.90	Details Add Update Delete
3	orange	£1.50	Details Add Update Delete
4	mango	£2.00	Details Add Update Delete

Add New Product

Name: grapes

Price: 3

Create Product

Figure 2: Creating a product through the UI (POST /products) and seeing it appear in the table.

Deleting uses DELETE /products/:id. Updating is supported via an Update button that loads values into the form. Updates can be implemented as PUT/PATCH, or as DELETE then POST when an update route is not available (the final approach is shown in the submitted code).

The update workflow is designed to feel natural: the user clicks Update, edits the fields, and presses Save. Even if the back end uses DELETE+POST, the UI still behaves like a typical edit form, which demonstrates the idea of "modifying data" from the user's perspective.

5.3. Cart operations (add, edit quantity and delete)

Adding to cart uses POST /cart with {productId, quantity}. Editing quantity uses PATCH /cart/:productId (partial update), and removing uses DELETE /cart/:productId. After each operation the cart table and totals are recalculated so the display stays consistent.

Totals are calculated on the front end by multiplying price × quantity for each cart item and then adding the values together across the list, which clearly shows the UI is using live API data.

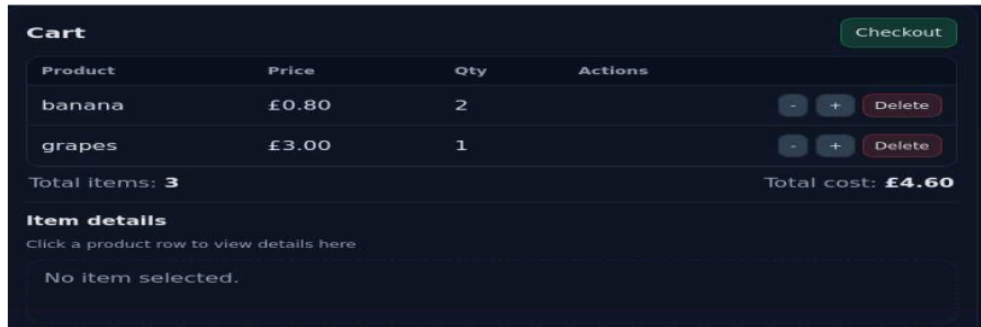


Figure 3: Cart populated; quantities can be edited and items removed (POST/PATCH/DELETE).

5.4. Two stage workflows: Cart and Checkout.

Stage 1 is building the cart (add/remove/edit). Stage 2 is checkout. POST /checkout rejects an empty cart. Otherwise clears the cart and returns a success message. The UI then refreshes GET /cart to show the empty cart, confirming the workflow completed.

After checkout, the details panel is also cleared so the user sees a clean state for the next purchase.

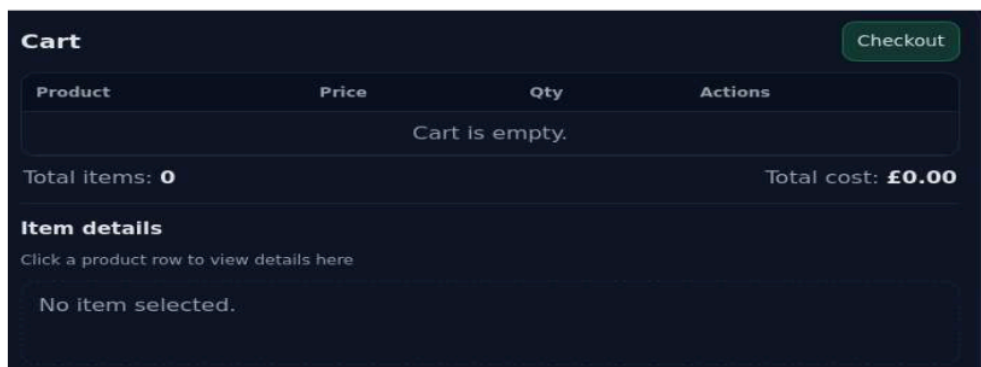


Figure 4 : After checkout succeeds, the cart is cleared and totals reset.

5.5. Search, filtering and polling.

Search filters the products table by name as the user types. A dropdown filter applies basic price bands. Both update the table instantly without a page reload.

The UI also polls the API using `setInterval` to refresh products and cart periodically and update the 'Last update' timestamp. [5] Polling demonstrates a second use of AJAX beyond direct user actions.

6. Client-side state and feedback.

`app.js` keeps a local copy of the latest products list so it can filter instantly, and it tracks the selected product, so the details panel stays consistent. Every operation produces feedback (success/error toasts) plus a table refresh so the user can see the result.

The API status badge is updated by sending a small 'health' request to GET / and switching the badge between online/offline. This makes troubleshooting easier during demonstrations.

Although this coursework is local, real deployments should add validation and security controls. OWASP notes that broken access control and weak validation are common API risks, so production versions should add authentication, authorisation and input checks [6]

7. Testing and evidence.

Testing was functional and focused on user journeys. Postman was used during development to verify request, JSON bodies and status codes before wiring the same calls into the browser UI. [7]

This table summarises key tests. The screenshots in this report provide evidence of core behaviours such as products loading, details updating, cart operations and checkout clearing the cart.

Test	Steps	Expected result
Load products	Open UI	Products table shows items
Show details	Click product row	Details panel updates
Create product	Submit form	New row appears
Cart actions	Add/edit/remove	Cart and totals update
Checkout	Checkout with items	Cart cleared
Search/filter	Type + choose price band	Rows filter live

8. Implementation summary.

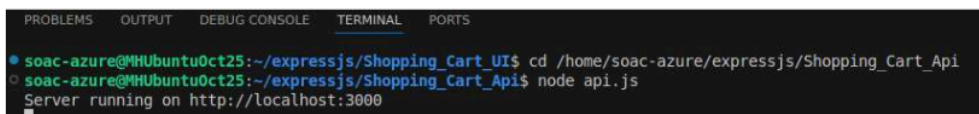
The front end is a static HTML/CSS/JS site. The back end is the existing Node.js/Express API from Coursework 1. Express was used because it provides simple routing and JSON middleware. [8]

The API stores data in memory for simplicity. In a larger system, arrays would be replaced with persistent storage. REST guidance emphasises stable resource design and predictable responses so that clients and servers can evolve independently. [9]

The UI depends only on JSON responses and standard HTTP behaviour, so the visual design can be extended without changing the API contract.

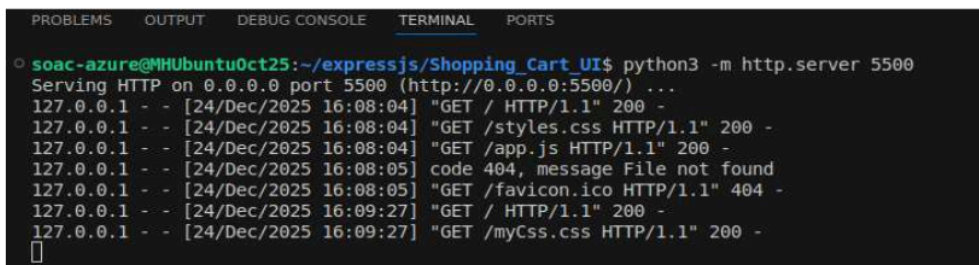
8.1. How to run the application.

- Start the API: in the Shopping_Cart_Api folder, run `npm install` then `node api.js`, confirm it is running on <http://localhost:3000>.
- Serve the UI: in the UI folder run `python3 -m http.server 5500` (or VS Code Live Server) and open <http://localhost:5500>. The API badge should show online.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
soac-azure@MHUbuntu0ct25:~/expressjs/Shopping_Cart_UI$ cd /home/soac-azure/expressjs/Shopping_Cart_Api
soac-azure@MHUbuntu0ct25:~/expressjs/Shopping_Cart_Api$ node api.js
Server running on http://localhost:3000
```

Figure 5: API server running locally on Node.js/Express (localhost:3000).



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
soac-azure@MHUbuntu0ct25:~/expressjs/Shopping_Cart_UI$ python3 -m http.server 5500
Serving HTTP on 0.0.0.0 port 5500 (http://0.0.0.0:5500/) ...
127.0.0.1 - - [24/Dec/2025 16:08:04] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [24/Dec/2025 16:08:04] "GET /styles.css HTTP/1.1" 200 -
127.0.0.1 - - [24/Dec/2025 16:08:04] "GET /app.js HTTP/1.1" 200 -
127.0.0.1 - - [24/Dec/2025 16:08:05] code 404, message File not found
127.0.0.1 - - [24/Dec/2025 16:08:05] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [24/Dec/2025 16:09:27] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [24/Dec/2025 16:09:27] "GET /myCss.css HTTP/1.1" 200 -
```

Figure 6: Serving the UI locally with a static server (`python -m http.server`).

9. Evaluation and future improvements.

Limitations include in-memory data reset on restart and no user accounts. Future improvements include persistence, stronger validation, and authentication. If real-time updates were needed, WebSockets or Server-Sent Events could replace polling.

The project also shows a separation of concerns between front end and API, which is a common recommendation in REST-based systems. [9] The API can remain stable while the UI changes, as long as the JSON contract stays consistent.

10. Conclusion

A complete single-page shopping cart web application was built for the Mini Shop. The Coursework 1 REST API using Fetch/AJAX and updates the page dynamically without a full refresh. Users can browse, search, view details, create/update/delete products, manage a cart, and checkout to complete a workflow. Overall, the result meets the brief and clearly demonstrates core SPA and AJAX techniques.

References

- [1] "MDN Blog RSS." MDN. Accessed: January 1 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
- [2] "MDN Blog RSS." MDN. Accessed: January 1 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Guides/Live_regions
- [3] "MDN Blog RSS." MDN. Accessed: January 1 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- [4] R. Fielding, Ed. Adobe M. Nottingham, Ed. Fastly J. Reschke, Ed. greenbytes, "RFC 9110: HTTP Semantics," June 2022, Accessed: Jan 1, 2026. Available at: <https://www.rfc-editor.org/rfc/rfc9110.html>.
- [5] "MDN Blog RSS." MDN. Accessed: January 1 2026. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/Window/setInterval>
- [6] "OWASP API Security Top 10," Accessed: January 1, 2026. Available at: <https://owasp.org/API-Security/>.
- [7] "Send API requests and get response data in Postman." Postman Docs. Accessed: January 1 2026. [Online]. Available: <https://learning.postman.com/docs/sending-requests/requests/>
- [8] "Express routing." expressjs. Accessed: Jan 1 2026. [Online]. Available: <https://expressjs.com/en/guide/routing.html>
- [9] "Web API Design Best Practices - Azure Architecture Center." Web API Design Best Practices - Azure Architecture Center | Microsoft Learn. Accessed: January 1 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>

FINAL GRADE

GENERAL COMMENTS

72/100

Generic Assessment Criteria	Marks available	Marks awarded
1. Engagement with Literature Skills		
2. Knowledge and Understanding Skills		
3. Cognitive and Intellectual Skills	Assignment 2: 40	
Design and documentation		3
Test documentation and testing		5
Good use of a table for endpoints describing function and UI trigger.		

Understanding demonstrated of the functionality of the SPA and API. Reasonable evidence of testing UI. A PASS / FAIL with evidence would have been desirable for tests. Reasonable set of instructions for running server and accessing UI. A little more detailed instruction here for the documentation would have been desirable.		
4. Practical Application Skills	Assignment 2: 50	
AJAX implementation REST API Nice simple working UI. Good use of files, tidy code. Be careful of folders and sub folders.npm		3 5
5. Transferable Skills for Life and Professional Practice	Assignment 2: 10	
Presentation		2

More detailed understanding of the code and the logical operation of the code (functions etc) would have been desirable.		
---	--	--

Remembering what the functions do would have been beneficial.		
---	--	--

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11
