

Rachel Roche

Student Number: 23309814

Distributed Systems

CA1

Rachel's Shoe Shop

<https://github.com/rachelncirl/retail>

CONTENTS

1	Project Proposal	4
1.1	Rachel's Shoe Shop	4
1.1.1	ListShoes	4
1.1.2	GetPrice	4
1.1.3	ShoppingCart	4
1.1.4	ViewCart	4
1.1.5	Purchase	4
1.1.6	Chat	4
2	Initial Service Definition.....	5
2.1	ListShoes	5
2.2	GetPrice	5
2.3	ShoppingCart	5
2.4	ViewCart	5
2.5	Purchase	5
2.6	Chat	5
3	Initial Shop Proto Definition.....	6
4	Initial User Interface	8
5	Initial Server Side Services	9
6	Initial Client Side Service	11
7	Initial Command Line Execution	14
8	Initial Static Webpage	15
9	Git Hub Integration	16
10	Service Extraction	17
10.1	Refined Services	17
10.1.1	ProductService	17
10.1.2	ShoppingCartService	17
10.1.3	ChatService	17
10.1.4	DiscoveryService	17
10.1.5	DiscountService	17
10.2	Refined Protos.....	18
10.2.1	Product Proto	18
10.2.2	Cart Proto	18
10.2.3	Chat Proto	19
10.2.4	Discovery Proto	19

10.2.5	Discount Proto	19
11	Integrating with Express	20
12	Creating Chat Service	21
13	Integrating Client and Server	26
14	Integrating Product Service.....	27
15	Integrating Cart Service	30
15.1	AddToCart	32
15.2	RemoveFromCart.....	33
15.3	RefreshCart	34
15.4	ClearCart	35
15.5	Rendering the Cart	36
16	Integrating Discount Service.....	40
17	Integrating Purchase Service.....	43
17.1	Pay.....	43
17.2	Order	45
17.3	Making a Purchase.....	46
17.3.1	Submit Event.....	46
17.3.2	Get Cart Items	46
17.3.3	Calculate Total	47
17.3.4	Payment.....	47
17.3.5	Order	48
17.3.6	Clear & Reload	49
18	Integrating Service Discovery.....	50
19	Extended Chat Service.....	53
20	Conclusion	57

1 PROJECT PROPOSAL

Option 4. Retail (Personalized shopping recommendations, automated checkout, inventory tracking)

1.1 RACHEL'S SHOE SHOP

The application will provide an online shopping experience for shoes. It will be made up of a number of services to provide an end-to-end shopping experience:

1.1.1 ListShoes

Returns details on all shoes currently stocked by Rachel's Shoe shop.

1.1.2 GetPrice

Returns the price of a specific shoe.

1.1.3 ShoppingCart

Will add a specific shoe to the user's shopping cart.

1.1.4 ViewCart

Will return the list of shoes in the user's shopping cart.

1.1.5 Purchase

Will purchase the shoes in the user's shopping cart.

1.1.6 Chat

Help facility if the user has any questions.

Each user of the application will need a unique identifier to keep track of items added to their personal shopping cart. When the user wishes to purchase a pair or multiple pairs of shoes, they add each pair to the shopping cart. The user can view the contents of their shopping cart. When the user chooses to Purchase, the items in their shopping cart are processed, and the shopping cart contents are reverted to empty.

2 INITIAL SERVICE DEFINITION

2.1 LISTSHOES

This is a server rpc service. No parameters are passed in the request. It will return a stream of ProductResponse objects, which will represent the shoes stocked by the shop. Each ProductResponse object will return the brand and the price of the shoe.

2.2 GETPRICE

This is a unary rpc service. The user will provide the brand of the shoe which will be passed as the only parameter in the ProductRequest. The service will return a single ProductResponse object which will contain the brand and the price of the shoe.

2.3 SHOPPINGCART

This is a client rpc service which will pass in a stream of ShoppingCartRequest objects. The user will add one or more items to their shopping cart, and the content of the cart will be streamed to the server. Each ShoppingCartRequest object will contain the brand, size, colour and quantity added to the cart. Once all items have been added to the cart, a message will be returned to indicate success, along with a total item count of objects added to the shopping cart.

2.4 VIEWCART

This is a server rpc call. The cartId will be passed in from the client in the ViewCartRequest, and the server will return a stream of all items in the cart in the ViewCartResponse.

2.5 PURCHASE

This is unary rpc call. The cartId will be passed in from the client in the PurchaseRequest, and the server will return the PurchaseResponse which will contain a message with the purchase details. The user's cart will also be cleared after the contents have been purchased.

2.6 CHAT

This is a bidirectional rpc call which will enable the user to engage with the customer service team if any issues are encountered while shopping.

3 INITIAL SHOP PROTO DEFINITION

```
syntax = "proto3";

package shop;

service ShoeShop{

    //Unary Rpc
    rpc GetPrice(ProductRequest) returns (ProductResponse);
    rpc Purchase(PurchaseRequest) returns (PurchaseResponse);

    //Server Rpc
    rpc ListShoes(Empty) returns (stream ProductResponse);
    rpc ViewCart(Empty) returns (stream ViewCartResponse);

    //Client rpc
    rpc      ShoppingCart(stream      ShoppingCartRequest)      returns
    (ShoppingCartResponse);

    //Bidirectional
    rpc Chat(stream ChatMessage) returns (stream ChatMessage);
}

message ProductRequest{
    string brand = 1;
}

message ProductResponse{
    string brand = 1;
    uint32 price = 2;
}

message ViewCartRequest {
    string cartId = 1;
}

message ViewCartResponse {
    string brand = 1;
    uint32 size = 2;
    string color = 3;
    uint32 quantity = 4;
}

message PurchaseRequest {
    string cartId = 1;
}

message PurchaseResponse {
    string message = 1;
}

message Empty{}
```

```
message ShoppingCartRequest {
    string brand = 1;
    string color = 2;
    string size = 3;
    uint32 quantity = 4;
}

message ShoppingCartResponse {
    string message = 1;
}

message ChatMessage{
    string user = 1;
    string message = 2;
}
```

4 INITIAL USER INTERFACE

The user interface will be a single web page that will render all the services outlined in the initial proto definition.

Bootstrap5 will be used to provide the look and feel of the website.

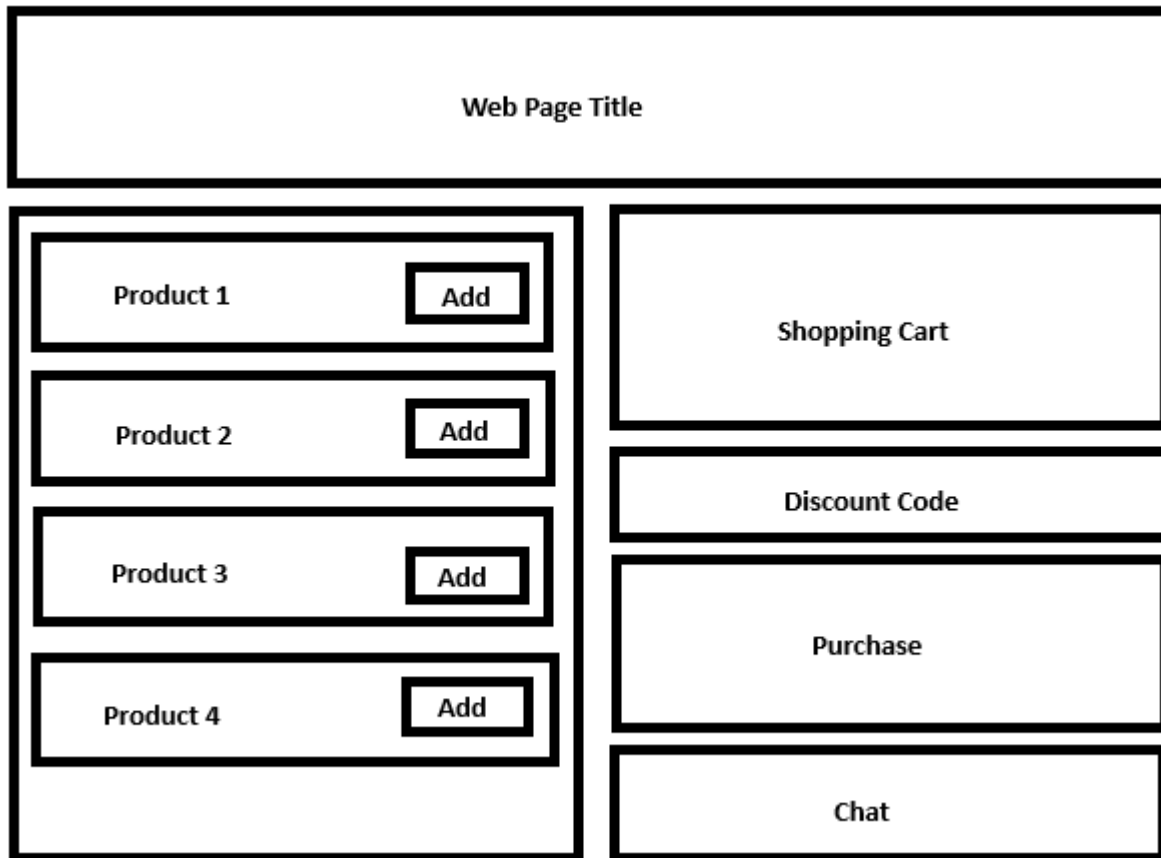
The top of the page will be a title banner for Rachel's shoe shop.

Products will be listed on the left-hand side of the page, with the shopping cart and purchase capability to the right.

It is also proposed to have a chat button which will open a chat page, or a popup modal.

Also, if time permits, there will be a discount service, which will allow the user to apply a discount to their cart if they have a valid code.

A simple wireframe of the web page would be as follows:



The user should be able to scroll through the shoes offered, and click add to put them in the shopping cart.

The user should then be able to purchase the contents of the shopping cart.

5 INITIAL SERVER SIDE SERVICES

The initial server-side services would be similar to the mobile shop example.

As a starting point, I created a server.js which contains all the services outlined in the proposal:

```
// Hard coded product list
const shoes = [
  { brand: "New Balance", price: 120},
  { brand: "Dr Martens", price: 150},
  { brand: "Adidas", price: 100},
  { brand: "Converse", price: 110},
  { brand: "Birkenstock", price: 90}
];

// Shopping cart contents
const cartItems = [];

// Unary
function GetPrice(call, callback){
  const shoe = shoes.find(s => s.brand === call.request.brand);
  if(shoe){
    callback(null, { brand: shoe.brand, price: shoe.price});
  }else{
    callback(null, { brand: shoe.brand, price: 0});
  }
}

// Server streaming
function ListShoes(call){
  shoes.forEach(shoe => call.write(shoe));
  call.end();
}

// Client rpc
function ShoppingCart(call, callback){
  let totalItems = 0;

  call.on("data", (order) => {
    totalItems += order.quantity;
    cartItems.push(order);
    console.log(`${cartItems.length} items in the Cart`)
    cartItems.forEach(item => console.log(item));
  })
}
```

```

    call.on("end", () => {
        callback(null, {message: `${totalItems} items added to Cart`});
    })
}

//server streaming
function ViewCart(call){
    cartItems.forEach(item => call.write(item));
    call.end();
}

//Unary
function Purchase(call, callback){
    let purchasedItems = cartItems.length;
    cartItems.length = 0;
    callback(null, {message: `${purchasedItems} items purchased`});
}

//Bidirectional
function Chat(call) {
    console.log("TODO");
    call.end();
}

const server = new grpc.Server();
server.addService(shopProto.ShoeShop.service, {
    GetPrice, ListShoes, ShoppingCart, ViewCart, Purchase, Chat
});

const PORT = '50051';
server.bindAsync(`0.0.0.0:${PORT}`, grpc.ServerCredentials.createInsecure(), (error, port)=>{
    if(error){
        console.error(error);
        return;
    }
    console.log(`Server running at localhost:${PORT}`);
});

```

6 INITIAL CLIENT SIDE SERVICE

Similarly, the starting point for the client service was to use the example from our lectures.

After modification for Rachel's Shoe shop, it became:

```
// Unary
function getPrice() {
  const brand = readlineSync.question("Enter mobile brand: ");
  console.log(brand);
  client.GetPrice({ brand }, (error, response) => {
    if (error) {
      console.log(error);
    }
    else {
      console.log(`brand: ${response.brand}, Price: ${response.price}`);
    }
  });
}

// Unary
function purchase() {
  let cartId = 1;
  client.Purchase({ cartId }, (error, response) => {
    if (error) {
      console.log(error);
    }
    else {
      console.log(`${response.message}`);
    }
  });
}

// Server rpc
function listShoes() {
  const call = client.ListShoes({});
  call.on("data", (shop) => {
    console.log(`brand: ${shop.brand}, Price: ${shop.price}`);
  });
  call.on("end", () => console.log("End of shoe list"));
}

// Client rpc
function shoppingCart() {
  const call = client.ShoppingCart((error, response) => {
    if (error) console.error(error);
  });
}
```

```

        else console.log(respomse.message);
    });

    let moreItems = true;
    while (moreItems) {
        const brand = readlineSync.question("Enter Shoe brand: ");
        const size = parseInt(readlineSync.question("Enter size: "));
        const color = readlineSync.question("Enter color: ");
        const quantity = parseInt(readlineSync.question("Enter quantity: "));
        call.write({ brand, size, color, quantity });
        moreItems = readlineSync.keyInYN("Add another item to the Shopping Cart?");
    }

    call.end();
}

// Server rpc
function viewCartContents() {
    const call = client.ViewCart({});
    call.on("data", (cart) => {
        console.log(`Brand: ${cart.brand}, Size: ${cart.size}, Color: ${cart.color},
Quantity: ${cart.quantity}`);
    });
    call.on("end", () => console.log("End of Cart Items list"));
}

// Bidirectional
function chat() {
    console.log("TODO");
}

function main() {
    console.log("\n1. Get Shoe Price");
    console.log("2. List Shoes");
    console.log("3. Add to Cart");
    console.log("4. Show Cart Contents");
    console.log("5. Purchase");
    console.log("6. Chat with support");
    const choice = parseInt(readlineSync.question("Choose an option: "))

    switch (choice) {
        case 1:
            getPrice();
            break;
        case 2:
            listShoes();

```

```
        break;
    case 3:
        shoppingCart();
        break;
    case 4:
        viewCartContents();
        break;
    case 5:
        purchase();
        break;
    case 6:
        chat();
        break;
    default:
        console.log("Invalid choice");
    }
}

main();
```

7 INITIAL COMMAND LINE EXECUTION

With the basics of the services in place, it was now possible to run the services from the cli:

Server

```
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src>
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src> cd .\server\
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src\server> node .\server.js
Server running at localhost:50051
```

Client

```
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src> cd .\client\
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src\client> node .\client.js

1. Get Shoe Price
2. List Shoes
3. Add to Cart
4. Show Cart Contents
5. Purchase
6. Chat with support
Choose an option: 1
Enter mobile brand: New Balance
New Balance
brand: New Balance, Price: 120
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src\client> node .\client.js

1. Get Shoe Price
2. List Shoes
3. Add to Cart
4. Show Cart Contents
5. Purchase
6. Chat with support
Choose an option: 2
brand: New Balance, Price: 120
brand: Dr Martens, Price: 150
brand: Adidas, Price: 100
brand: Converse, Price: 110
brand: Birkenstock, Price: 90
End of shoe list
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src\client> node .\client.js

1. Get Shoe Price
2. List Shoes
3. Add to Cart
4. Show Cart Contents
5. Purchase
6. Chat with support
Choose an option: 3
Enter Shoe brand: Adidas
Enter size: 4
Enter color: white
Enter quantity: 1
Add another item to the Shopping Cart? [y/n]: n
1 items added to Cart
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src\client> node .\client.js


1. Get Shoe Price
2. List Shoes
3. Add to Cart
4. Show Cart Contents
5. Purchase
6. Chat with support
Choose an option: 4
Brand: Adidas, Size: 4, Color: white, Quantity: 1
End of Cart Items list
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src\client>
```

8 INITIAL STATIC WEBPAGE

With the basic services in place, I proceeded to develop the user interface as a static proof of concept.

Using simple bootstrap forms, buttons and layouts, the wiremock design from earlier was put in place.


Our Products



Product 1 Name

Description Goes Here


Add to Cart



Product 2 Name

Description Goes Here


Add to Cart



Product 3 Name

Description Goes Here

Add to Cart



Product 4 Name

Description Goes Here

Add to Cart

Rachel's Shoes

Product 1	\$49.99
Product 2	\$29.99
Total	\$79.98

Discount Code

Apply

Payment Information

Name on Card

Card Number

Expiration Date CVV

Complete Purchase

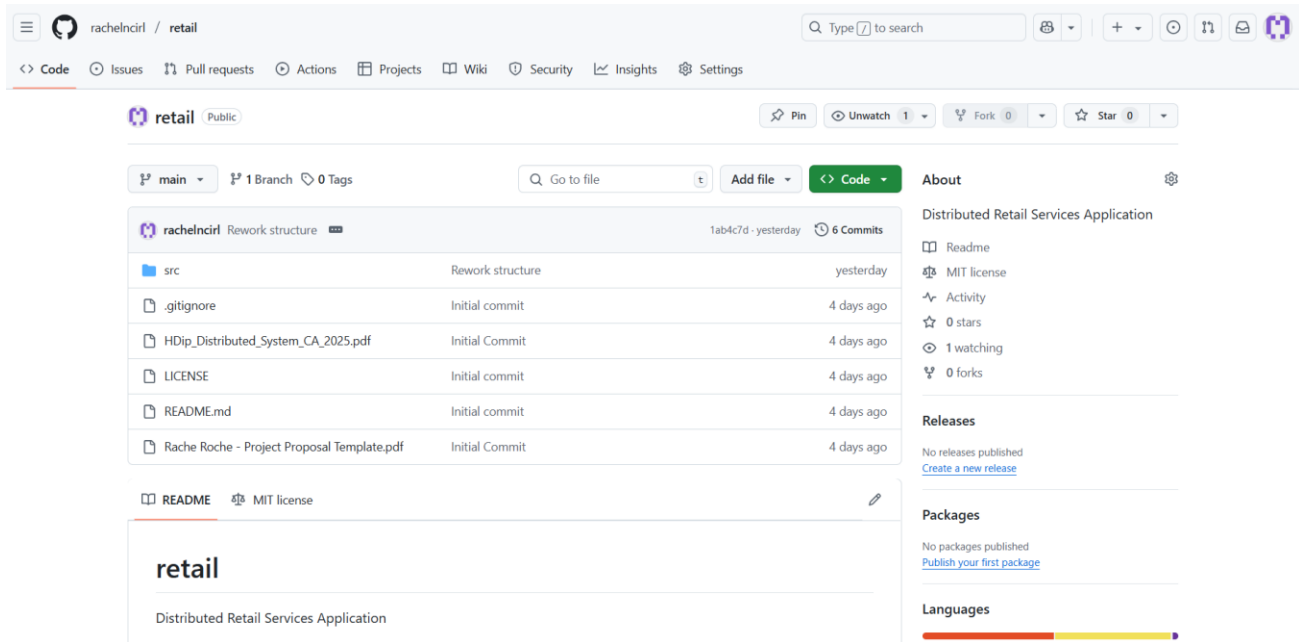
Chat

15

9 GIT HUB INTEGRATION

At this point, the github repository was created for the retail application.

<https://github.com/rachelncirl/retail>



All source code for the application was placed under the src folder, with general project documentation stored at the root level.

10 SERVICE EXTRACTION

Following a meeting with my lecturer, the breakdown of the service became a lot clearer. The single shop proto file needed to be broken down into individual services – rather than a list of functions provided by one service.

10.1 REFINED SERVICES

The following services were identified:

10.1.1 ProductService

ListShoes - would return a list of all products offered by Rachel's Shoe Shop.
GetPrice - would return the price of each individual product.

10.1.2 ShoppingCartService

AddToCart – Products are added to the shopping cart.
ViewCartContents – Display the contents of the shopping cart.
Purchase – Checkout and clear the shopping cart contents.

10.1.3 ChatService

Chat – Allows users to chat about products offered by the shop.

10.1.4 DiscoveryService

Discovery – Allows the services offered by the shop to be discovered.

10.1.5 DiscountService

Discount Service - Apply a code to the shopping cart that would reprice the total. This may be a standalone service, or it may be better suited to integration into the ShoppingCartService.

10.2 REFINED PROTOS

10.2.1 Product Proto

```
syntax = "proto3";

package product;

service ProductService {

    //Unary Rpc
    rpc GetPrice(ProductRequest) returns (ProductResponse);

    //Server Rpc
    rpc ListShoes(Empty) returns (stream ProductResponse);

}

message ProductRequest{
    string brand = 1;
}

message ProductResponse{
    string brand = 1;
    uint32 price = 2;
}
```

10.2.2 Cart Proto

```
syntax = "proto3";

package cart;

service CartService {

    //Unary Rpc
    rpc Purchase(PurchaseRequest) returns (PurchaseResponse);

    //Server Rpc
    rpc ViewCart(Empty) returns (stream ViewCartResponse);

    //Client rpc
    rpc ShoppingCart(stream ShoppingCartRequest) returns (ShoppingCartResponse);

}

message ViewCartResponse {
    string brand = 1;
    uint32 size = 2;
    string color = 3;
    uint32 quantity = 4;
}

message PurchaseRequest {
    string cartId = 1;
```

```

}

message PurchaseResponse {
    string message = 1;
}

message Empty{}

message ShoppingCartRequest {
    string brand = 1;
    string color = 2;
    string size = 3;
    uint32 quantity = 4;
}

message ShoppingCartResponse {
    string message = 1;
}

```

10.2.3 Chat Proto

```

syntax = "proto3";

package chat;

service ChatService {
    rpc sendMessage(stream ChatMessage) returns (stream ChatMessage) {}
}

message ChatMessage {
    string name = 1;
    string message = 2;
}

```

10.2.4 Discovery Proto

// TODO

10.2.5 Discount Proto

// TODO

11 INTEGRATING WITH EXPRESS

The next task was to start a web server and render the static purchase page.

The purchase.html file was moved to a public folder under src, which made it accessible to the express web server.

A simple express app.js was used to start the webserver:

```
const express = require('express');
const http = require('http');
const path = require('path');

const app = express();
const server = http.createServer(app);

app.use(express.static(path.join(__dirname, '../public')));


app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, '../public', 'purchase.html'));
});

server.listen(50050, () => {
  console.log('Homepage is running');
});
```

This allowed the page to be rendered under localhost

```
PS C:\Users\rache\college\Semester3\Distributed\CA1\retail\src\client> node .\app.js
Express server running at http://localhost:50050
```

localhost:50050

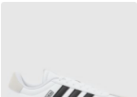


Product 1 Name

Description Goes Here

\$49.99

Add to Cart



Product 2 Name

Description Goes Here

\$39.99

Add to Cart

Shopping Cart

Product 1	\$49.99
Product 2	\$29.99
Total	\$79.98

Discount Code

Apply

Payment Information

20

12 CREATING CHAT SERVICE

The chat service from our lectures would again be the basis for the chat service for Rachels Shoe Shop.

The chat service would also use express, and display the chat.html content from the public folder when the /chat/ url is called.

Extracting the chat server code into its own server.js file:

```
const express = require('express');
const socketIo = require('socket.io');
const http = require('http');
const path = require('path');

const app = express();
const server = http.createServer(app);
const io = socketIo(server);

let userCount = 1;

app.get('/chat/', (req, res) => {
  res.sendFile(path.join(__dirname, '../public', 'chat.html'));
});

io.on('connection', (socket) => {
  console.log('A User is connected');

  const username = 'user' + userCount++;
  socket.emit('set username', username);
  console.log(username);

  socket.on('chat message', (msg) => {
    io.emit('chat message', {username, msg});
  });

  socket.on('disconnect', () => {
    console.log('user disconnected')
  });
});

server.listen(3000, () => {
  console.log('Server is running');
});
```

And the simple chat.html is as defined during class:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title> Chat </title>
    <style>
      body {
        font-family: Arial, Helvetica, sans-serif;
        margin: 0;
        padding: 0px;
        background-color: white;
      }
      #chat-container {
        width: 400px;
        margin: 0 auto;
        padding: 10px;
        background-color: white;
        border-radius: 5px;
        box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
      }
      #messages {
        height: 300px;
        overflow-y: scroll;
        border-bottom: 1px solid;
        padding: 10px;
        margin-bottom: 10px;
      }
      #message-input {
        width: calc(100% - 20px);
        padding: 10px;
        font-size: 16px;
        margin-top: 10px;
      }
      #send-button {
        padding: 10px, 15px;
        color: white;
        border: none;
        cursor: pointer;
        background-color: green;
      }
    </style>
  </head>
  <body>
    <div id="chat-container">
```

```

    <div id="messages"></div>
    <input type="text" id="messages-input" placeholder="Type your
message..." />
    <button id="send-button">Send</button>
</div>

<script src="/socket.io/socket.io.js"></script>
<script>
    var socket = io();
    var username = '';

    var messages = document.getElementById('messages');
    var input = document.getElementById('messages-input');
    var sendButton = document.getElementById('send-button');

    socket.on('set username', function(name) {
        username = name;
        console.log('Your username is : ', username);
    });

    socket.on('chat message', function(data) {
        var messageElement = document.createElement('div');
        messageElement.textContent= data.username + ': ' + data.msg;
        messages.appendChild(messageElement);
        messages.scrollTop = messages.scrollHeight;
    });

    sendButton.addEventListener('click', function() {
        var message = input.value;
        if (message.trim()) {
            socket.emit('chat message', message);
            input.value = '';
        }
    });
</script>
</body>
</html>

```

To link the chat service into the purchase page, the chat button opens a modal that contains the chat.html in an embedded iFrame

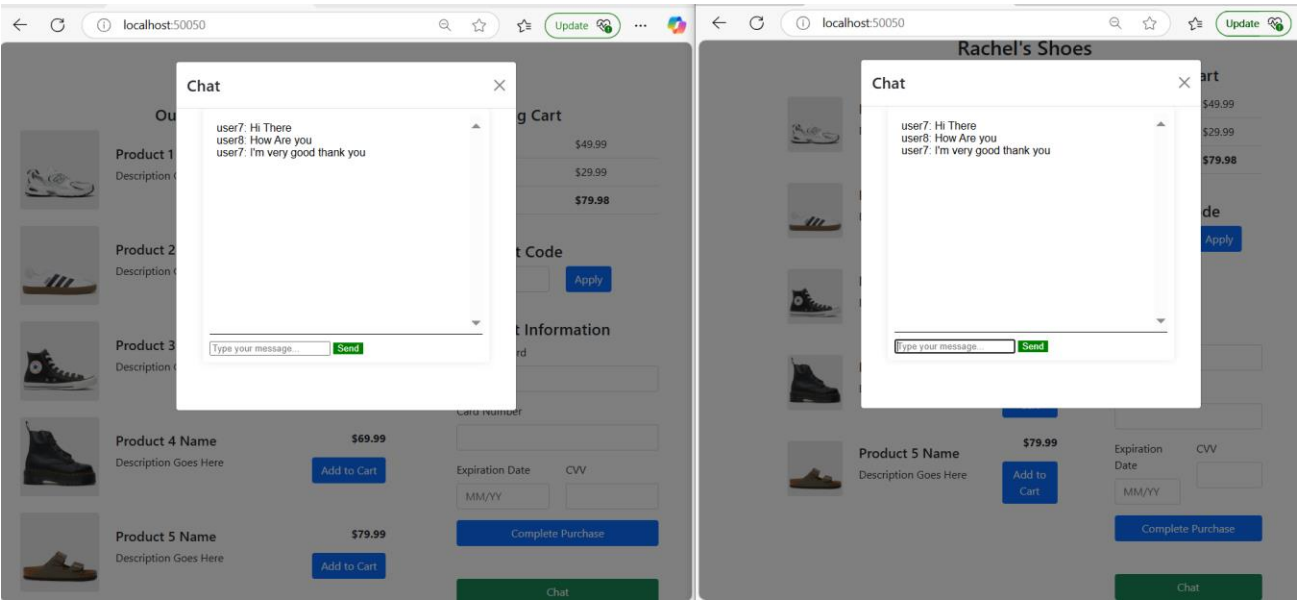
```
<button type="button" class="btn btn-success w-100" data-bs-toggle="modal" data-bs-
target="#chatModal">
  Chat
</button>

<!-- Chat Modal -->
<div class="modal fade" id="chatModal" tabindex="-1" aria-
labelledby="chatModallabel"
  aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">

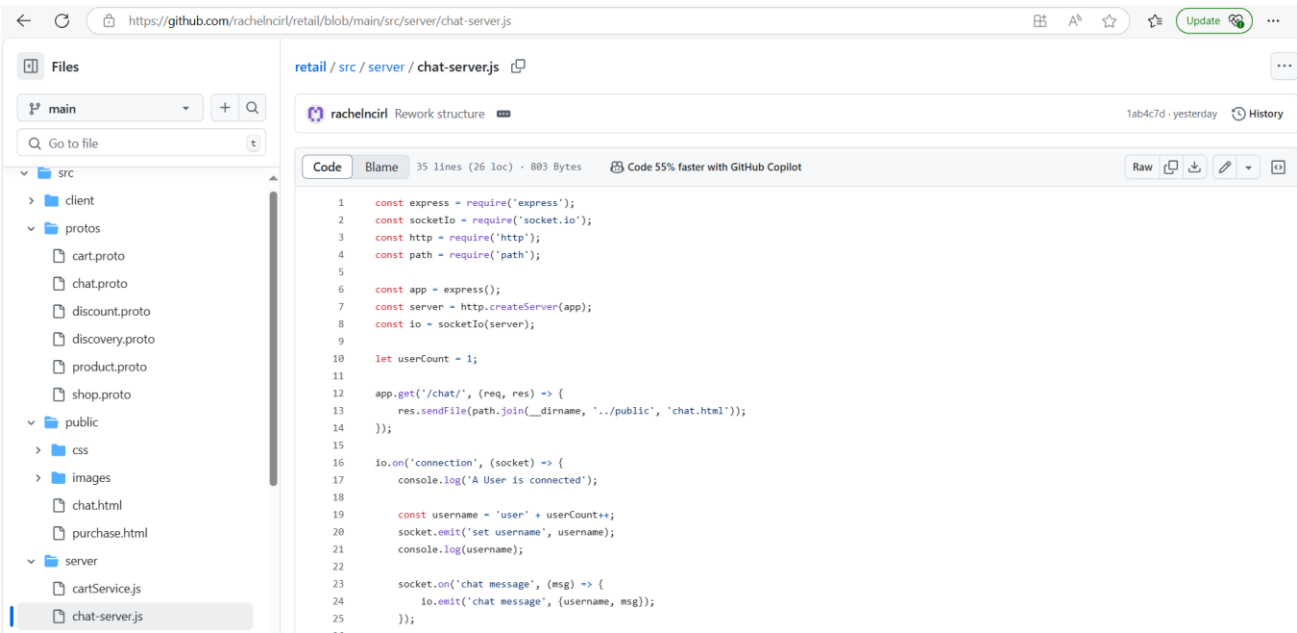
      <!-- Modal Header -->
      <div class="modal-header">
        <h4 class="modal-title" id="chatModallabel">Chat</h4>
        <button type="button" class="btn-close" data-bs-dismiss="modal"
          aria-label="Close"></button>
      </div>

      <!-- Modal body -->
      <div class="modal-body">
        <div id="messages">
          <div class="row">
            <iframe id="responsive-iframe"
src="http://localhost:3000/chat"></iframe>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```


Displaying the chat modals on the website:

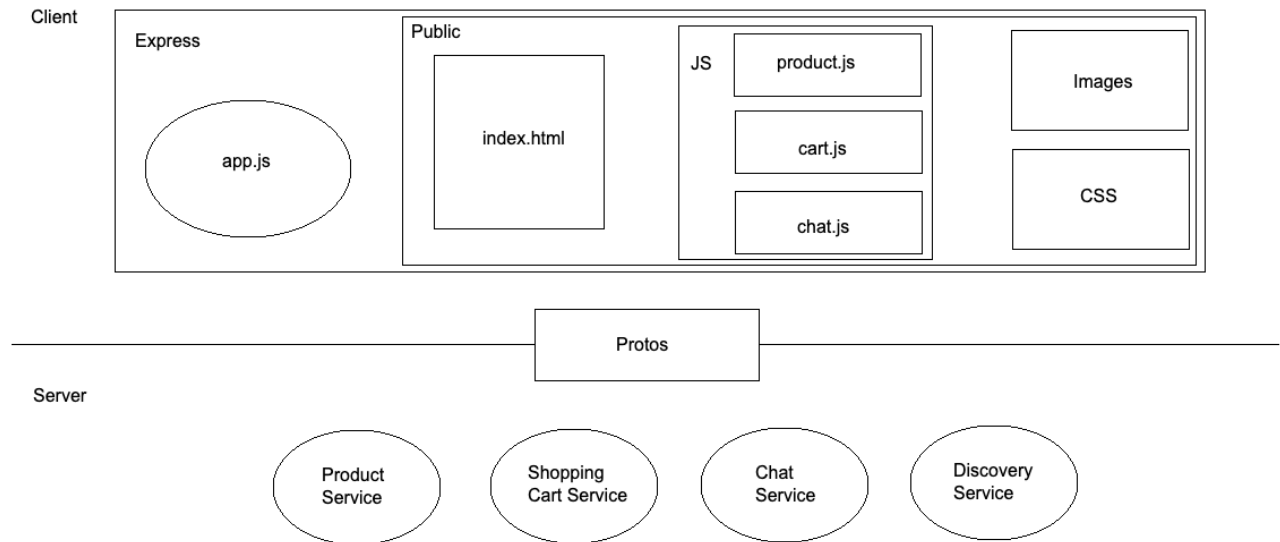


The structure of the project is also becoming more refined.



13 INTEGRATING CLIENT AND SERVER

At this point I needed to draw up the structure of the application to better understand the flow from the web page to the services and back.



Previously created `purchase.html` is now moved to `index.html` in the express web server.

`Index.html` will render static content, with the dynamic content rendered by the individual js files.

The js files will call the client services defined in the `app.js`

`App.js` will call the server-side services.

The gRPC `protos` define the communication between the `app.js` and the server-side services.

14 INTEGRATING PRODUCT SERVICE

Rather than presenting a static product list on the webpage, the product list will instead be loaded from the server-side product service as defined in the protos.

The static list was removed from the index.html, and replaced with a placeholder html object.

```
<!-- Products -->
<h4 class="text-center">Our Products</h4>

<div class="row">
  <!-- Load the products from the server Node Js -->
  <table>
    <tbody id="productList"></tbody>
  </table>
</div>
<script src="/js/products.js"></script>
```

The placeholder productList object is overwritten on page load by the products.js content. When the page loads, the products.js will call the list api to fetch all the products:

```
// Retrieve Product List from Server on Page Load
document.addEventListener('DOMContentLoaded', function () {

  // Make an HTTP request to the Express server
  fetch(`/list`)
    .then(response => response.json())
    .then(data => {

      // Get the table body element where rows will be inserted
      const tableBody = document.getElementById("productList");

      // Loop through the JSON data and create rows
      data.forEach(shoe => {

        // HTML Code for each product will go here
      });
    })
    .catch(error => {
      console.error('Error:', error);
    });
});
```

The list api is defined in the express server's app.js

```
// Define an endpoint for listing shoes
app.get('/list', (req, res) => {

  // Create an array for gathering the shoe list
  let shoes = [];

  const call = products.ListShoes({});

  call.on("data", (shoe) => {

    // Add the shoe to the list
    shoes.push({
      id: shoe.id,
      brand: shoe.brand,
      price: shoe.price,
      description: shoe.description,
      image: shoe.image
    });
  });

  call.on("end", () => {
    // Return the list of shoes as a JSON response
    res.json(shoes);
  });

  call.on("error", (error) => {
    console.error(error);
    res.status(500).send('Error occurred while fetching shoes');
  });
});
```

Products.proto will define the contract for the ListShoes call.

```
//Server Rpc
rpc ListShoes(Empty) returns (stream ProductResponse);

message ProductResponse{
  string id = 1;
  string brand = 2;
  uint32 price = 3;
  string description = 4;
  string image = 5;
}
```

And the server-side products.js will implement the backend functionality.

```
// Hardcoded product list
const shoes = [
  { id: "NB1", brand: "New Balance", price: 120, description: "530", image:
"/images/newbalance.jpg" },
  { id: "DR1", brand: "Dr Martens", price: 200, description: "1460 Pascal", image:
"/images/drmartens.jpg" },
  { id: "AD1", brand: "Adidas", price: 100, description: "VL Court 3.0", image:
"/images/adidas.jpg" },
  { id: "CV1", brand: "Converse", price: 75, description: "All Star Hi", image:
"/images/converse.jpg" },
  { id: "BK1", brand: "Birkenstock", price: 90, description: "Arizona", image:
"/images/birkenstock.jpg" },
  { id: "CR1", brand: "Crocs", price: 90, description: "Classic Clog", image:
"/images/crocs.jpg" }
];

// Server streaming - List all available shoes
function ListShoes(call) {
  shoes.forEach(shoe => call.write(shoe));
  call.end();
}
```

The Unary call to get the price of an individual product will be implemented using a similar flow, and will use the same ProductResponse object.

```
// Unary - Get the price of a shoe by its brand
function GetPrice(call, callback) {
  const shoe = shoes.find(s => s.brand === call.request.brand);
  if (shoe) {
    callback(null, { brand: shoe.brand, price: shoe.price });
  } else {
    callback(null, { brand: "", price: 0 });
  }
}

//Unary Rpc
rpc GetPrice(ProductRequest) returns (ProductResponse);

message ProductRequest{
  string brand = 1;
}
```

15 INTEGRATING CART SERVICE

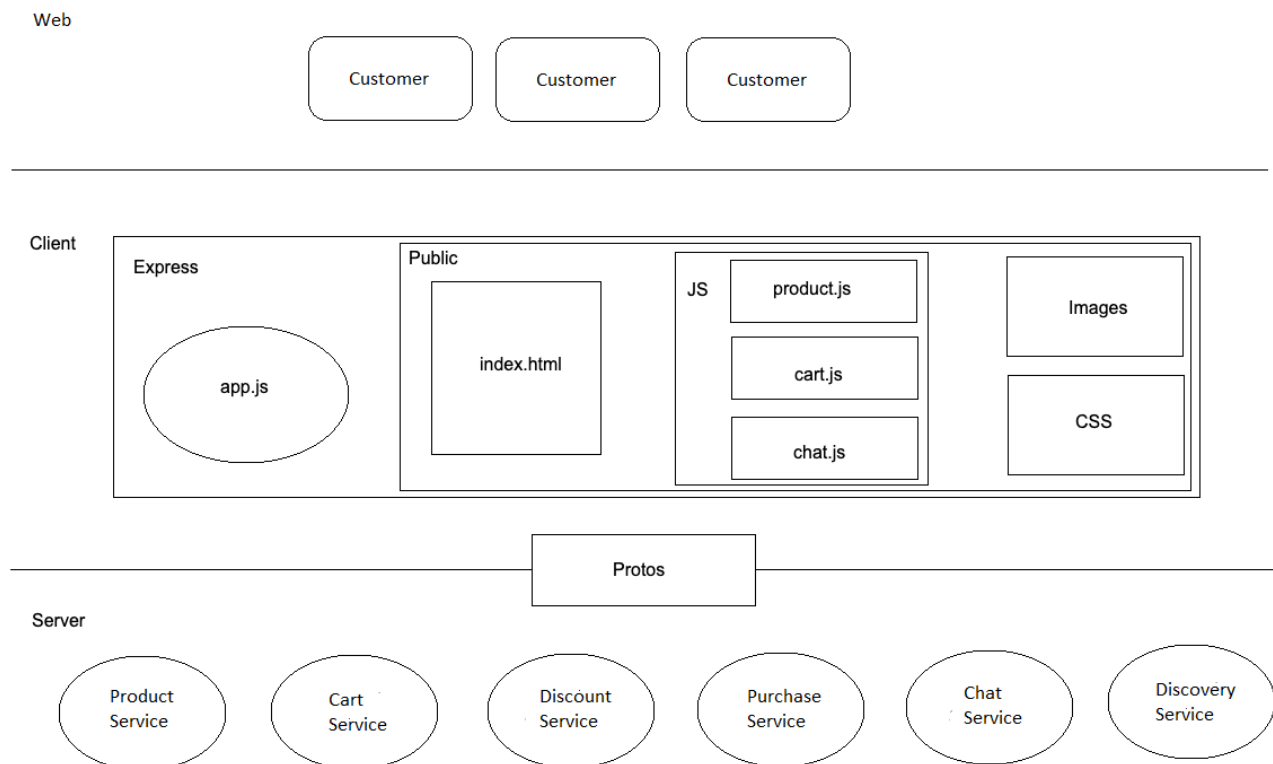
The Shopping Cart is designed to resemble the Bulk Order functionality from our lectures. The customer can build up a collection of products which they wish to order. When they want to place the order, the purchase button is clicked.

However, the cart needs to be functional before it can be used by the Purchase Service to place an Order.

The shopping cart presents a further layer of complexity.

- The product service just presents a static list to the shopper. However, the shopping cart is dynamic. It will start out as empty, and change as person adds, removes, clears or completes the purchase.
- The cart contents are also specific to each person on the website, so a simple array to contain the cart contents will not work for multiple shoppers.

The overall design of the application is tweaked a little at this point, with the Cart Service solely responsible for building the collection of objects that will be used by the Purchase Service. There will be an additional Discount Service to allow customers to enter a discount code and avail of reduced prices.



Similar to how a chat identifier was defined for each chat window, a unique user id will be assigned to each person shopping on the website.

When index.html is loaded, a script will run on the page to generate a unique user id for the shopper. This will be UI specific, so refreshing the page will result in a new user id and the contents of the cart will no longer be available to the shopper.

```
<script>
  function generateUserId() {

    // Random 9-character alphanumeric string
    return Math.random().toString(36).substr(2, 9);
  }

  window.userId = generateUserId();
  console.log("Generated User ID:", userId);
</script>
```

This userid can then be sent to server when adding items in the cart.

The cart will be stored on the server side using a Map, with the userId as the key, and an array of products as the value for each entry.

```
// Map to hold user carts: key is userId, value is an array of cart items
const userCarts = new Map();
```

Four operations will be defined for the Cart Service

- AddToCart – Server Side Streaming as the request will send one item to the basket, but the server will return the updated cart contents for the UI to display
- RemoveFromCart – also Server Side Streaming as the request will send one item to be removed from the basket, and the server will return the updated cart contents for the UI to display with the item removed.
- GetCartContents – also Server Side Streaming. The userId will be sent and the contents of the cart will be streamed back to the UI
- EmptyCart – Unary RPC – removes all the contents of the cart based on the userId.

Initial draft of the Cart Proto Services:

```
service CartService {

  //Unary Rpc
  rpc EmptyCart(CartRequest) returns (CartResponse);

  //Server Rpc
  rpc GetCartContents(CartRequest) returns (stream ProductResponse);
  rpc AddToCart(UpdateCartRequest) returns (stream ProductResponse);
```

```

    rpc RemoveFromCart (UpdateCartRequest) returns (stream ProductResponse);
}

```

15.1 ADDTOCART

On the server side, the incoming UpdateCartRequest will have the shoe id. The server will first check that the customer has a cart in the map, and if not an entry will be assigned. The server then pushes the shoe id in the request to the cart, and streams back the full contents of the cart for rendering in the UI.

```

// Server streaming - Add an item to the cart
function AddToCart(call) {
  const shoe = shoes.find(s => s.id === call.request.id);
  if (shoe) {
    const userId = call.request.userId;
    if (!userCarts.has(userId)) {
      userCarts.set(userId, []);
    }
    userCarts.get(userId).push(shoe);
    console.log(`${userId} added ${shoe.brand} to their cart. Cart size:
    ${userCarts.get(userId).length}`);

    // Return the contents of the cart after adding the shoe
    userCarts.get(userId).forEach(shoe => call.write(shoe));
    call.end();
  } else {
    console.log("Shoe not found");
    call.end();
  }
}

```

On the client side, the api is defined in the client.js. As the products are returned to the client they are gathered into an array, and when the full list has been populated, the result is returned as a json object to the UI call.

```

// Define an endpoint for adding shoes to cart
app.get('/addToCart', (req, res) => {
  const id = req.query.id;
  const userId = req.query.userId;

  let cart = [];

  console.log("Product ID requested: " + id + " for User: " + userId);
  const call = cartClient.AddToCart({ id, userId });

```



```

    call.on("data", (shoe) => {
      cart.push({
        id: shoe.id,
        brand: shoe.brand,
        price: shoe.price
      });
    });

    call.on("end", () => {
      res.json(cart);
    });

    call.on("error", (error) => {
      console.error(error);
      res.status(500).send('Error occurred while adding to cart');
    });
  });
}

```

15.2 REMOVEFROMCART

On the server side, the incoming UpdateCartRequest will also have the shoe id. The server will get the user's cart from the map of user carts, and then remove the specified shoe from the array holding the cart contents. All products remaining in the array will be streamed back for UI rendering.

```

// Server streaming - Remove an item from the cart
function RemoveFromCart(call) {
  const userId = call.request.userId;
  if (userCarts.has(userId)) {
    const cart = userCarts.get(userId);
    const shoeIndex = cart.findIndex(s => s.id === call.request.id);
    if (shoeIndex !== -1) {
      cart.splice(shoeIndex, 1);
      console.log(`Removed ${call.request.id} from ${userId}'s cart.`);
    }

    // Return the contents of the cart after removing the shoe
    userCarts.get(userId).forEach(shoe => call.write(shoe));
  }
  call.end();
}

```

Similar to add, on the client side, the api is defined in the client.js. As the products are returned to the client they are gathered into an array, and when the full list has been populated, the result is returned as a json object to the UI call.

```

// Define an endpoint for removing shoes from the cart
app.get('/removeFromCart', (req, res) => {
  const id = req.query.id;
  const userId = req.query.userId;

  let cart = [];

  console.log("Remove Product ID: " + id + " for User: " + userId);
  const call = cartClient.RemoveFromCart({ id, userId });

  call.on("data", (shoe) => {
    cart.push({
      id: shoe.id,
      brand: shoe.brand,
      price: shoe.price
    });
  });

  call.on("end", () => {
    res.json(cart);
  });

  call.on("error", (error) => {
    console.error(error);
    res.status(500).send('Error occurred while removing from the cart');
  });
});

```

15.3 REFRESH CART

Unlike the previous two methods, this does not modify the content of the customer's cart. It is used to retrieve the cart contents at any time, and streams the content back to the caller similar to the add and remove functionality.

```

// Server streaming - Get the contents of the user's shopping cart
function GetCartContents(call) {
  const userId = call.request.userId;
  if (userCarts.has(userId)) {

    // Return the contents of the cart after removing the shoe
    userCarts.get(userId).forEach(shoe => call.write(shoe));
  }
  call.end();
}

```

Again, similar to add and remove on the client side, the api is defined in the client.js. As the products are returned to the client they are gathered into an array, and when the full list has been populated, the result is returned as a json object to the UI call.

```
// Define an endpoint for removing shoes from the cart
app.get('/refreshCart', (req, res) => {
  const userId = req.query.userId;

  let cart = [];

  console.log("Refresh Cart for User: " + userId);
  const call = cartClient.GetCartContents({ userId });

  call.on("data", (shoe) => {
    cart.push({
      id: shoe.id,
      brand: shoe.brand,
      price: shoe.price
    });
  });

  call.on("end", () => {
    res.json(cart);
  });

  call.on("error", (error) => {
    console.error(error);
    res.status(500).send('Error occurred while refreshing cart');
  });
});
```

15.4 CLEARCART

Clearing the cart is a unary call, that takes in a user id and removes all items from their cart. It simply returns a success or failure upon completion.

```
// Unary - Empty the customer's shopping cart
function EmptyCart(call, callback) {
  const userId = call.request.userId;
  if (userCarts.has(userId)) {
    userCarts.get(userId).splice(0, userCarts.get(userId).length);
    callback(null, { message: "Shopping Cart Emptied" });
  } else {
    callback(null, { message: "Shopping Cart Not Found" });
  }
}
```

```

    }
  }
}

```

The client side for clearing the cart is more straightforward, with an indication of success or failure returned from the clear cart API call.

```

// Define an endpoint for clearing the contents of the shopping cart
app.get('/clearCart', (req, res) => {
  const userId = req.query.userId;

  cartClient.EmptyCart({ userId }, (error, response) => {
    if (error) {
      console.error('Error:', error);
      res.status(500).send({
        message: 'An error occurred while emptying the cart',
        details: error.details,
      });
    } else {
      res.status(200).send({
        message: response.message,
      });
    }
  });
});
});

```

15.5 RENDERING THE CART

Similar to the product rendering, the html is simple, with a placeholder for the cart contents to be populated dynamically.

The refresh and clear functionality is executed via bootstrap buttons

```

<!-- Cart Contents -->

<h4>Shopping Cart</h4>

<div class="row">
  <!-- Load the Cart contents from the server Node Js -->
  <table>
    <tbody id="cartContents"></tbody>
  </table>
  <div class="row">
    <div class="col-md-6 mb-3">
      <button type="button" class="btn btn-warning w-100"
        id="refreshCart">Refresh</button>
    </div>
    <div class="col-md-6 mb-3">
      <button type="button" class="btn btn-danger w-100"

```

```

        id="clearCart">Clear</button>
    </div>
</div>
</div>

```

The dynamic content will be provided via cart.js, with event handling for adding, removing and clicking the buttons. Adding items will be done from the product list, but removing will be done from the cart.

```

// Creates an empty Shopping Cart on Page Load
document.addEventListener('DOMContentLoaded', function () {
    refresh();
});

// Add product to Shopping Cart
document.getElementById('productList').addEventListener('click', function (event) {
    add(event);
});

// Remove item from the Shopping Cart
document.getElementById('cartContents').addEventListener('click', function (event) {
    remove(event);
});

// Reload the Shopping Cart from the Server
document.getElementById('refreshCart').addEventListener('click', () => {
    refresh();
});

// Empty the Shopping Cart
document.getElementById('clearCart').addEventListener('click', () => {
    empty();
    refresh();
});

```

These events will call functions to invoke the API endpoints outlined above:

```

function add(event) {
    if (event.target && event.target.matches('button[id^="addToCart-"]')) {
        const productId = event.target.name;
        const userId = window.sessionId;
        console.log('Adding Product:', productId);

        fetch(`/addToCart?id=${productId}&userId=${userId}`)
            .then(response => response.json())
            .then(data => {
                console.log(data);
            });
    }
}

```

```

        renderCartData(data);
    })
    .catch(error => {
        console.error('Error:', error);
    });
}
}

```

And a shared function will be used to render the content of the shopping cart based on the collection of products streamed back from the API call.

```

// Function to render the cart data
function renderCartData(cartItems) {
    const cartContentsDiv = document.getElementById('cartContents');
    cartContentsDiv.innerHTML = '';

    const cartTable = document.createElement('table');
    cartTable.classList.add('table');

    const tbody = document.createElement('tbody');
    let total = 0;

    cartItems.forEach(item => {
        // Render each product in the cart on its own row
    });

    // Add total row
    const totalRow = document.createElement('tr');
    const totalLabelCell = document.createElement('td');
    totalLabelCell.textContent = 'Total';
    totalLabelCell.style.fontWeight = 'bold';
    totalRow.appendChild(totalLabelCell);

    const totalAmountCell = document.createElement('td');
    totalAmountCell.textContent = `$$${total.toFixed(2)}`;
    totalAmountCell.style.fontWeight = 'bold';
    totalRow.appendChild(totalAmountCell);

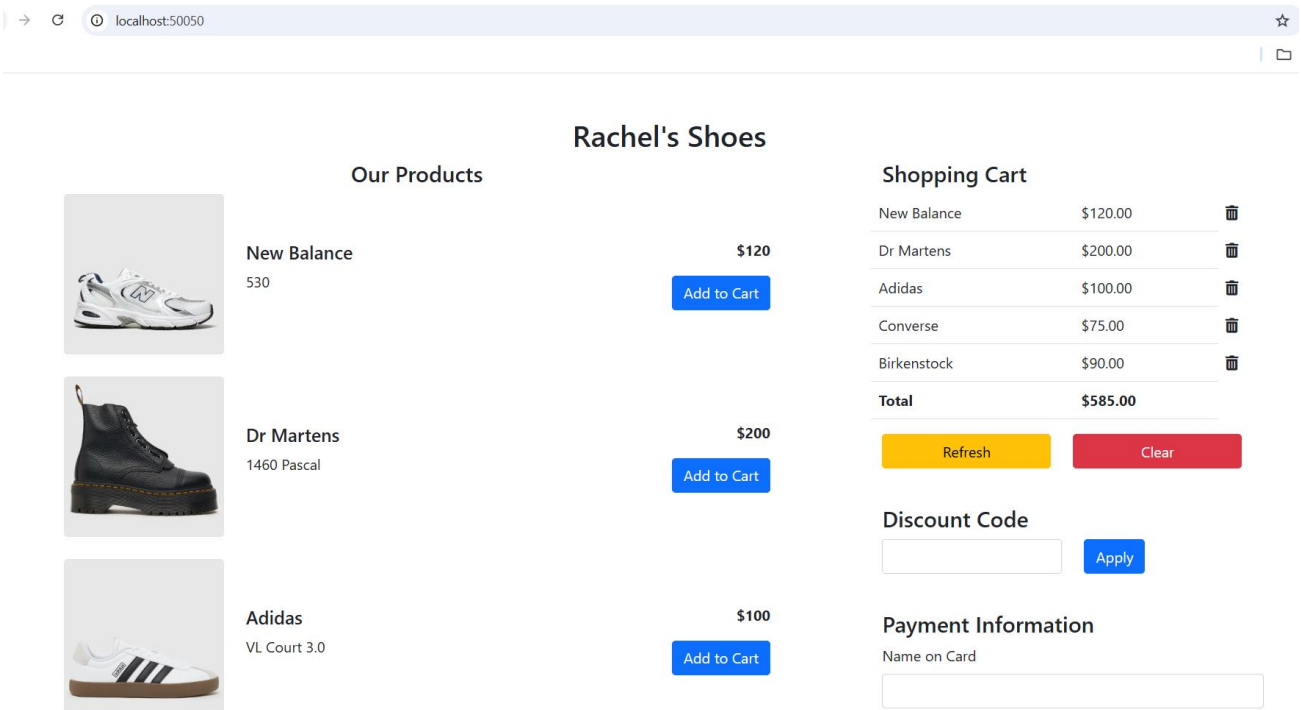
    tbody.appendChild(totalRow);

    cartTable.appendChild(tbody);

    // Append the table to the cart contents div
    cartContentsDiv.appendChild(cartTable);
}

```

The shopping cart can now build up a list of products to make a purchase.



16 INTEGRATING DISCOUNT SERVICE

Most online shopping sites use promotional discount codes to reward shoppers for their business.

Rachel's Shoe Shop will offer a simple discount service via unary call to return a percentage if the shopper enters a valid discount code. If the code is not valid the server will simply return zero.

```
service DiscountService {  
  
    //Unary Rpc  
    rpc GetDiscount(DiscountRequest) returns (DiscountResponse);  
  
}  
  
message DiscountRequest {  
    string code = 1;  
}  
  
message DiscountResponse {  
    uint32 percentage = 1;  
}
```

The server will hold a valid code, along with a percentage to return if the code is matched.

```
// Hard code discount percentage for now  
const discountPercentage = 20;  
const discountCode = "NEW20";
```

When the call is received, the server validates the code and returns the percentage if the code is matched. If not, it returns a zero.

```
// Unary - Send back a percentage if the shopper enters a valid code  
function GetDiscount(call, callback) {  
    if (call.request.code == discountCode) {  
        callback(null, { percentage: discountPercentage});  
    } else {  
        callback(null, { percentage: 0 });  
    }  
}
```

The Discount API is defined on the client node js code to implement the grpc call:

```
// Client Side API implementation to call the Discount Service  
app.get('/discount', (req, res) => {  
    const code = req.query.code;  
  
    discountClient.GetDiscount({ code }, (error, response) => {  
        if (error) {
```



```

        console.error('Error:', error);
        res.status(500).send({
            message: 'An error occurred while applying the discount',
            details: error.details,
        });
    } else {
        console.log('Discount Amount: ' + response.percentage)
        res.status(200).send({
            percentage: response.percentage,
        });
    }
});
});
});

```

A discount javascript file will define the interceptor for the button click on the web page and call the API. It will then call the refresh() function on the shopping cart to re-render the cart with the discount if a value of greater than zero has been returned.

```

// Apply a discount code to the user's session
document.getElementById('applyDiscount').addEventListener('click', () => {
    applyDiscount();
});

function applyDiscount() {
    const code = document.getElementById('discountCode').value;

    console.log("Discount Code Entered: " + code);

    // Make an HTTP request to the Express server
    fetch(`/discount?code=${code}`)
        .then(response => response.json())
        .then(data => {
            console.log(data);
            discount = data.percentage;
            refresh();
        })
        .catch(error => {
            console.error('Error:', error);
        });
}

```

The refresh() function in the cart will be modified to add an additional for the discount if its greater than zero.

```

// Ensure the discount is applied only if it's greater than 0
if (discount > 0) {

```

```

console.log('Applying Discount of : ' + discount);

// Add Discount Row
const discountRow = document.createElement('tr');

const discountLabelCell = document.createElement('td');
discountLabelCell.textContent = 'Discount (' + discount + '%)';
discountLabelCell.style.fontWeight = 'bold';
discountLabelCell.style.color = 'red';
discountRow.appendChild(discountLabelCell);

let discountAmount = ((total/100) * discount);
total = total - discountAmount;

const discountAmountCell = document.createElement('td');
discountAmountCell.textContent = `-$${discountAmount.toFixed(2)}`;
discountAmountCell.style.fontWeight = 'bold';
discountAmountCell.style.color = 'red';
discountRow.appendChild(discountAmountCell);


tbody.appendChild(discountRow);
}

```

Successful application of the code results in the Discount being applied to the cart.

Rachel's Shoes


Our Products



New Balance
530

\$120

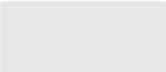
Add to Cart



Dr Martens
1460 Pascal

\$200

Add to Cart



Adidas

\$100

Shopping Cart

New Balance	\$120.00	🗑
Dr Martens	\$200.00	🗑
Adidas	\$100.00	🗑
Birkenstock	\$90.00	🗑
Birkenstock	\$90.00	🗑
Discount (20%)	-\$120.00	
Total	\$480.00	

Refresh

Clear

Discount Code

NEW20

Apply

17 INTEGRATING PURCHASE SERVICE

The Purchase consists of a number of operations.

Two new specific calls will be added to facilitate the Purchase – Pay and Order.

```
service PurchaseService {  
  
    //Unary Rpc  
    rpc Pay (PaymentRequest) returns (PaymentResponse);  
  
    //Client rpc  
    rpc Order(stream PurchaseRequest) returns (PurchaseResponse);  
  
}  
  
message PurchaseRequest {  
    string id = 1;  
    string brand = 2;  
    uint32 price = 3;  
    string description = 4;  
}  
  
message PurchaseResponse {  
    string message = 1;  
}  
  
message PaymentRequest {  
    string cardName = 1;  
    string cardNumber = 2;  
    string exp = 3;  
    string cvv = 4;  
    uint32 amount = 5;  
}  
  
message PaymentResponse {  
    string message = 1;  
    bool success = 2;  
}
```

17.1PAY

This a unary call that will provide the Customer Name, Card Number, Expiry Date, CVV and Total Amount to make a payment.

As the application does not have the ability to process a payment, the server side will be a placeholder for integrating with an external payment service. For the shoe shop it will simply return a successful response for all payment calls.

```
// Unary - Pay - Payment processing would be done here  
function Pay(call, callback) {  
    callback(null, { message: "Payment Successful", success: true});  
}
```

The API on the Express server will provide all the necessary details to make a payment when called. As a form is used for the payment details, this API will be a post API, with the payment details contained in the body of the request.

```
// Payment API call that will call the server to process the payment
app.post('/pay', (req, res) => {
  const paymentRequest = req.body
  console.log(paymentRequest);

  purchaseClient.Pay(paymentRequest, (error, response) => {
    if (error) {
      console.error('Error:', error);
      res.status(500).send({
        message: 'An error occurred while emptying the cart',
        details: error.details,
      });
    } else {
      console.log('Payment Response: ' + response.message)
      res.status(200).send({
        message: response.message,
        success: response.success
      });
    }
  });
});
});
```

Skipping over the purchase event for now, the form to input the payment details will be defined in the index.html

```
<form>
  <div class="mb-3">
    <label for="card-name" class="form-label">Name on Card</label>
    <input type="text" class="form-control" id="cardName" required>
  </div>
  <div class="mb-3">
    <label for="card-number" class="form-label">Card Number</label>
    <input type="text" class="form-control" id="cardNumber" required>
  </div>
  <div class="row">
    <div class="col-md-6 mb-3">
      <label for="exp-date" class="form-label">Expiration Date</label>
      <input type="text" class="form-control" id="exp" placeholder="MM/YY"
required>
    </div>
    <div class="col-md-6 mb-3">
      <label for="cvv" class="form-label">CVV</label>
      <input type="text" class="form-control" id="cvv" required>
    </div>
  </div>
</form>
```

```

    </div>
  </div>
  <button id="purchase" type="submit" class="btn btn-primary w-100">Complete
Purchase</button>
</form>

```

17.2 ORDER

This is a Client Streaming call that will send the list of products from the customer's Shopping Cart to place the Order. On the server side, each item will be passed in via the stream and added to the userOrder. When the stream is ended the server places the order and sends back a message to the caller.

```

// Client rpc - Handle the Order Request
function Order(call, callback) {

  let userOrder = [];

  call.on("data", (orderItem) => {
    userOrder.push(orderItem);
    console.log(`Order has ${userOrder.length} items.`);
  });

  call.on("end", () => {
    callback(null, { message: `Order has been placed for ${userOrder.length} items`
  });
  });
}

```

On the express api implementation, the post request will contain a json payload with the list of items in the order. The API calls the purchaseClient to open the stream, and then loops through the content of the order, adding each individual item from the order. When all items have been written to the stream, the stream is ended and the api call is completed.

```

// Order API that will be called to place an order
app.post('/order', (req, res) => {

  const order = req.body

  // Open the stream
  const call = purchaseClient.Order((error, response) => {
    if (error) console.error(error);
    else console.log(response.message);
  });

  // Write the items to the stream

```

```

    for (let item of order) {
      const { id, brand, price } = item;
      call.write({ id, brand, price });
    }

    // Close the stream
    call.end();
  });

```

17.3 MAKING A PURCHASE

These two new calls need to be combined with existing calls to make a purchase. Upon clicking the Purchase button, the full Purchase flow will be as follows:

17.3.1 Submit Event

The submit event is picked up by the purchase.js, and all form data is retrieved, along with validating that the fields have been filled.

```

event.preventDefault();

// Get values from the form
const cardName = document.getElementById('cardName').value;
const cardNumber = document.getElementById('cardNumber').value;
const expDate = document.getElementById('exp').value;
const cvv = document.getElementById('cvv').value;

// Ensure all form data is provided
if (!cardName || !cardNumber || !expDate || !cvv) {
  alert('Please fill in all the payment fields.');
```

17.3.2 Get Cart Items

The Shopping Cart contents are retrieved from the server. These will be streamed to the Purchase Service when placing the Order.

```

// Get the cart contents
let cartContents
const userId = window.sessionId;
fetch(`/refreshCart?userId=${userId}`)
  .then(response => response.json())

```

```
.then(data => {  
    cartContents = data;  
})
```

17.3.3 Calculate Total

The total cost of the items in the cart is calculated, including any discount applied.


```
// calculate amount  
console.log("Purchase Discount is: " + discount);  
amount = 0;  
for (let item of cartContents) {  
    const { id, brand, price } = item;  
    if (discount > 0) {  
        let discountedPrice = price - ((price * discount) / 100);  
        amount += discountedPrice  
    } else {  
        amount += price  
    }  
}  
console.log("Total Purchase Price: " + amount);
```

17.3.4 Payment

The Payment is made using the Card Details entered in the Purchase form and the total net price calculated.

```
const paymentData = {  
    cardName: cardName,  
    cardNumber: cardNumber,  
    expDate: expDate,  
    cvv: cvv,  
    amount: amount  
};  
  
// Send payment data to the server  
fetch('/pay', {  
    method: 'POST',  
    headers: {  
        'Content-Type': 'application/json',  
    },  
    body: JSON.stringify(paymentData),  
})
```

Our Products




New Balance

530

\$120

Add to Cart




Dr Martens

1460 Pascal

\$200

Add to Cart




Adidas

VL Court 3.0

\$100

Add to Cart





Converse

All Star Hi

\$75

Add to Cart

Shopping Cart

New Balance	\$120.00	
Converse	\$75.00	
Discount (20%)	-\$39.00	
Total	\$156.00	

Refresh

Clear

Discount Code

Apply

Payment Information

Name on Card

Card Number

Expiration Date

CVV

Complete Purchase

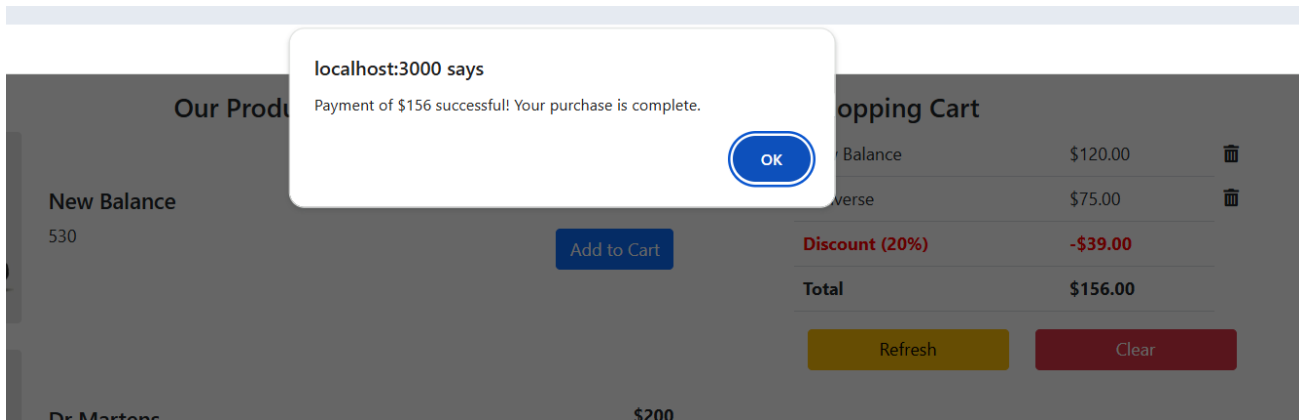
17.3.5 Order

If the payment is successful, the cart contents are sent to the order api to create the Order.

```
// Now Place the Order
fetch('/order', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(cartContents),
})
```

A simple pop up alert is displayed to indicate success or failure of the purchase.

```
// Simple pop up alert to indicate successful payment
alert('Payment of $' + amount + ' successful! Your purchase is complete.');
```

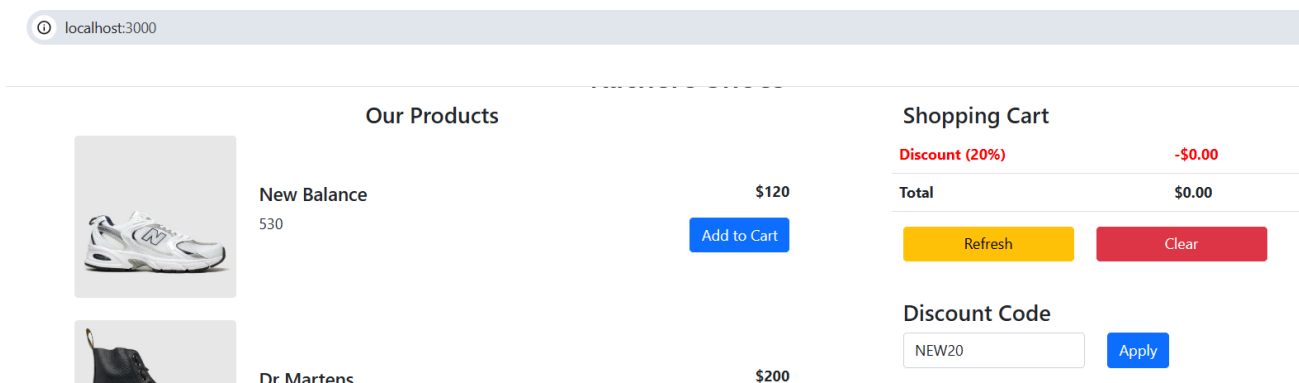



17.3.6 Clear & Reload

The Shopping Cart is then cleared of the items purchased. A reload of the shopping cart is then performed to re-render the cleared-out Shopping Cart.

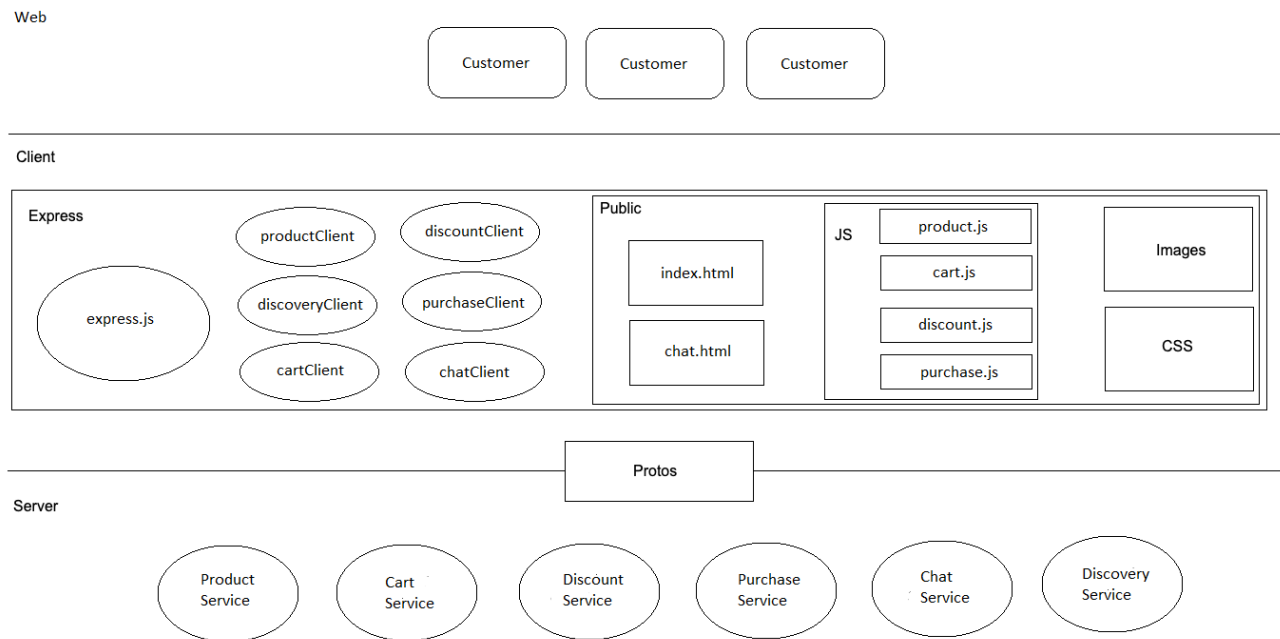
```
// Clear the Cart Contents after Purchase
empty();
```

```
// Reload the Cart after clearing
refresh();
```



18 INTEGRATING SERVICE DISCOVERY

Currently, all protos are loaded in the app.js. After reviewing our lectures on service discovery, it became clear that to facilitate service discovery, each client-side service needed to be extracted into its own client node js file, with the module exporting the features of the service. The app.js file is also renamed to express.js to indicate that it's the web server.



The proto file for the Discovery service is as per our lectures, with the service name sent in the request, and the location of the service returned:

```
syntax = "proto3";  
package discovery;  
service DiscoveryService {  
    rpc Discover(ServiceRequest) returns (ServiceResponse);  
}  
message ServiceRequest {  
    string serviceName = 1;  
}  
message ServiceResponse {  
    string address = 1;  
}
```

The server side holds a mapping of the services to their location:

```
// List of Discoverable Services
const services = {
  "productService": "localhost:50051",
  "cartService": "localhost:50052",
  "discountService": "localhost:50053",
  "purchaseService": "localhost:50054",
  "chatService": "localhost:50055"
};
```

When the Discover function is called, the address of the service is returned based on the service name passed in:

```
function Discover(call, callback) {
  const serviceName = call.request.serviceName;
  const address = services[serviceName];

  if (address) {
    callback(null, { address });
  } else {
    callback({
      code: grpc.status.NOT_FOUND,
      details: "service not found"
    });
  }
}
```

The client-side node js is extracted from the web server node js into its own discoveryClient file, which calls the server side Discover function.

```
// Function to discover service information
const discoverService = (serviceName, callback) => {

  discoveryClient.Discover ({
    serviceName: serviceName
  }, (err, response) => {
    if (err) {
      console.log("Error discovering service", err);
      callback(null);
    } else {
      callback(response);
    }
  });
};
```

The client side functionality is then exported to allow it to be used by the express.js web server

```
// Export the functions
module.exports = {
  discoverService
}
```

The same client-side extraction is performed for the product, cart, discount, purchase and chat services. The exported modules can now be used by the express.js web server:

```
const product = require('./productClient');
const cart = require('./cartClient');
const discount = require('./discountClient');
const purchase = require('./purchaseClient');
const chat = require('./chatClient');
const discovery = require('./discoveryClient');
```

All endpoints supported by express.js are now updated to discover the service providing the functionality before calling the function on the specific client:

```
app.get('/price', (req, res) => {

  // Discover product service and then get the price of the product
  discovery.discoverService('productService', (productService) => {
    if (!productService) {
      res.status(500).send("Product Service not found");
      return;
    }

    product.getPrice(req, (err, priceData) => {
      if (err) {
        res.status(500).send("Error getting the price");
        return;
      }
      res.status(200).send(priceData);
    });
  });
});
```

19 EXTENDED CHAT SERVICE

While the initial chat service was functional, it was based mainly on the client side, and did not use any protos or bi-directional streaming. It needed to be extended to use the proto.

```
syntax = "proto3";

package chat;

service ChatService {
    // Bidirectional rpc
    rpc Chat(stream ChatMessage) returns (stream ChatMessage) {}
}

message ChatMessage {
    string name = 1;
    string message = 2;
}
```

The server side was created to hold a list of connected clients. When an initial chat message is sent, the shopper is added to the list of connected clients. When a message is sent by one shopper it is written to all active clients.

```
var clients = {}

function Chat(call){
    call.on('data', function(chatMessage){

        console.log(chatMessage);

        if(!(chatMessage.name in clients)){
            clients[chatMessage.name] = {
                name: chatMessage.name,
                call: call
            }
        }

        for(var client in clients){
            clients[client].call.write(
                {
                    name: chatMessage.name,
                    message: chatMessage.message
                }
            )
        }
    });
}
```

```

call.on('end', function(){
  call.end();
});

call.on('error', function(e){
  console.log(e)
});
}

```

On the client side, when the socket connection is established, the server-side Chat function is called. This establishes the bi-directional streaming of messages between multiple shoppers on the site.

```

function chat(io) {
  io.on('connection', (socket) => {
    console.log('Client connected to chat');

    const call = chatClient.Chat();

    // Receive messages from gRPC stream and forward to frontend
    call.on('data', (data) => {
      console.log("gRPC Message:", data.user, data.message);
      socket.emit('chat message', data);
    });

    // Receive messages from frontend and send to gRPC
    socket.on('chat message', (data) => {
      console.log("Frontend Message:", data.user, data.message);
      call.write({
        name: data.user,
        message: data.message,
        timestamp: Date.now()
      });
    });

    socket.on('disconnect', () => {
      console.log('Client disconnected from chat');
      call.end();
    });
  });
}

```

Similar to the other services, the chat client is exported from the chatClient:

```

module.exports = chat;

```

And the chat function is called from the express.js web server:

```
const socketIo = require('socket.io');
const http = require('http');

const app = express();
const server = http.createServer(app);
const io = socketIo(server);

// Start Chat Socket
chat(io);
```

The initial UI design is retained, with the chat button the webpage opening a modal, which is populated by the chat.html and embedded java script functionality:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io();
  var username = '';

  var messages = document.getElementById('messages');
  var input = document.getElementById('messages-input');
  var sendButton = document.getElementById('send-button');

  socket.on('chat message', function (data) {
    console.log(data);
    var messageElement = document.createElement('div');
    messageElement.textContent = data.name + ': ' + data.message;
    messages.appendChild(messageElement);
    messages.scrollTop = messages.scrollHeight;
  });

  sendButton.addEventListener('click', function () {
    var message = input.value;
    if (message.trim()) {
      var data = { user: window.chatId, message }
      console.log("User: " + data.user + ", Message: " + data.message);

      // This sends the message to the express server
      socket.emit('chat message', data);
      input.value = '';
    }
  });
</script>
```

One small UI enhancement was to create more human friendly usernames for guests on the site:

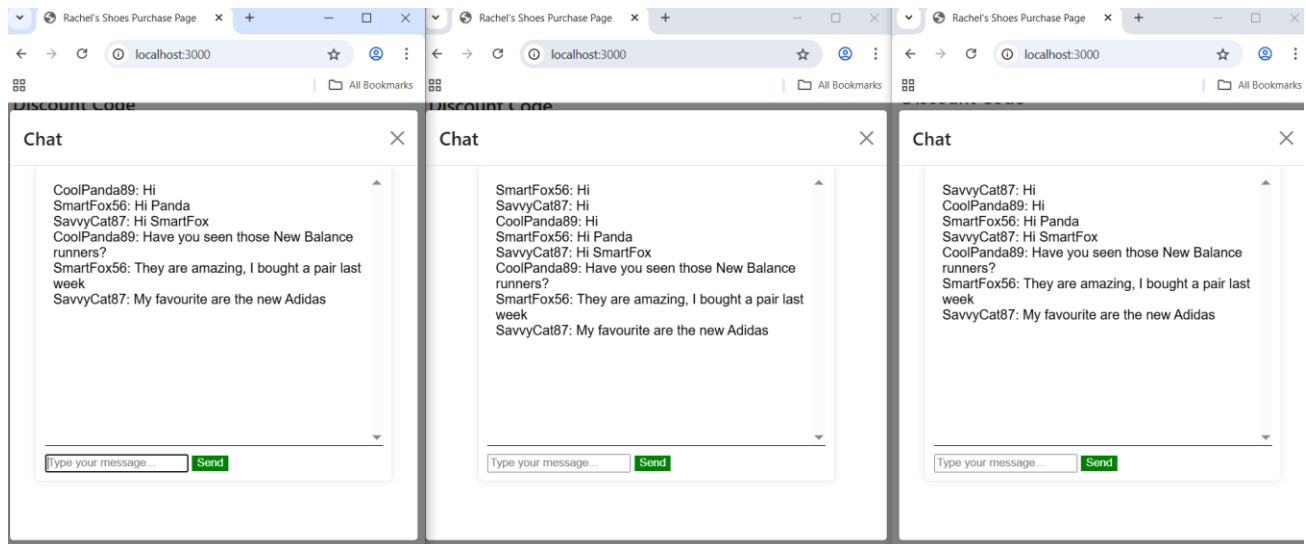
```
<script>
  function generateUsername() {
    const adjectives = ["Cool", "Happy", "Trendy", "Smart", "Hip", "Savvy"];
    const animals = ["Cat", "Panda", "Eagle", "Shark", "Wolf", "Fox"];
    const number = Math.floor(Math.random() * 100);

    const adjective = adjectives[Math.floor(Math.random() * adjectives.length)];
    const animal = animals[Math.floor(Math.random() * animals.length)];

    return `${adjective}${animal}${number}`;
  }

  window.chatId = generateUsername();
  console.log("Generated Chat ID:", chatId);
</script>
```

Shoppers can now chat about their purchases and tips using bi-directional streaming.



20 CONCLUSION

The application has evolved throughout its development, but it has still largely aligned with the initial design in the proposal. The final application contains six backend distributed services, using Unary, Client RPC, Server RPC and Bi-Directional RPC communication.

- Product Service
- Cart Service
- Discount Service
- Purchase Service
- Chat Service
- Discovery Service

Further services could be added in the future to provide Inventory Management, Dynamic Sale Pricing, Recommended Products based on Cart contents.