Corey Sokol - SWDV-691 - Week 3

Autumn Hills Service Layers

For the Autumn Hills web application, multiple service layers will be utilized in order to moderate the interaction between our user interface and our database. By utilizing these services, we are able to separate concerns within the application and focus on certain tasks being handled by their specific layers.

I'd like to start by looking at the overall large layer of the program. Since we are developing a web application, the Autumn Hills website will be rendering typical HTML, CSS and JavaScript to the visitors of the website. Under the hood however, things are not quite that simple. Since we are interacting with a database in our application the standard HTML/JavaScript approach isn't enough. In order to connect the user interface with the database, a server side JavaScript library is needed to act as the larger service for the application. Because of this, I have decided to build my app on Node.Js which will utilize Express to act as the web application framework. I have also included a mysql dependency in the app so that Express can query data to the database and display the resulting data in pages set up through routes. Finally, since this application is a server side application, a JavaScript templating engine is needed in order to render the HTML pages for a user to interact with. I've decided to utilize Pug, which will not only allow us to pass data from our templates to our database, but will also render the content in HTML on the server.

As for services within the Autumn Hills app, there will be a few different services that will be utilized:

dbconnection-service - service that creates the database connection. Stores the connection details for the database and will attempt to make the connection.

create-user-service - this is the service that will be called when a user signs up for a new account. This service will handle the data that is passed in through an input form from the user, and will then query that data with the database.

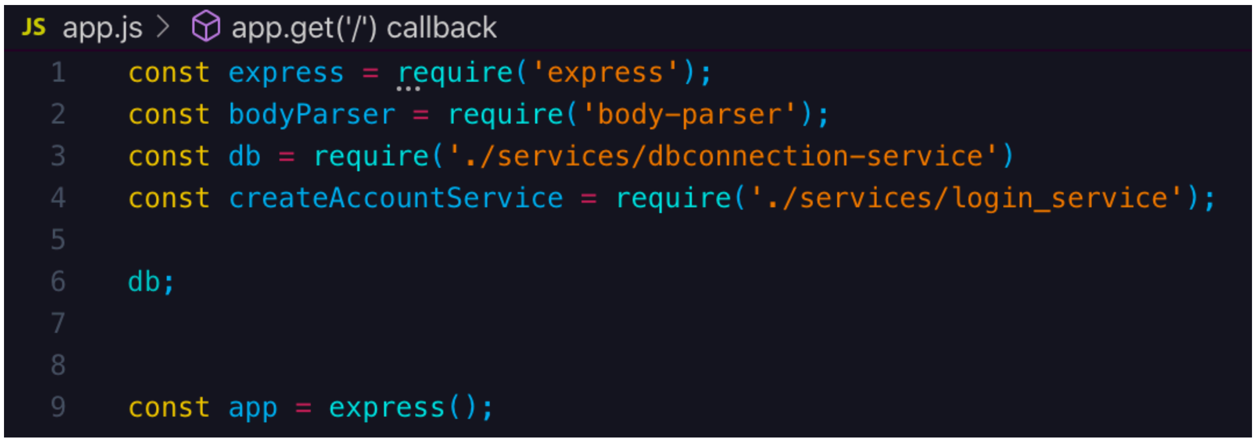
login-service - this service will focus on handling the input from the user when they attempt to log in. The input is passed through the service and queried against user information in the database. If the input matches a database result then the user will be able to log in.

reserve-tee-time-service - this service will allow the user to reserve a tee time at Autumn Hills country club. The user will input the date and time in which they want to book a tee time and from there the service will handle the user input and insert a new reservation into the database.

Example

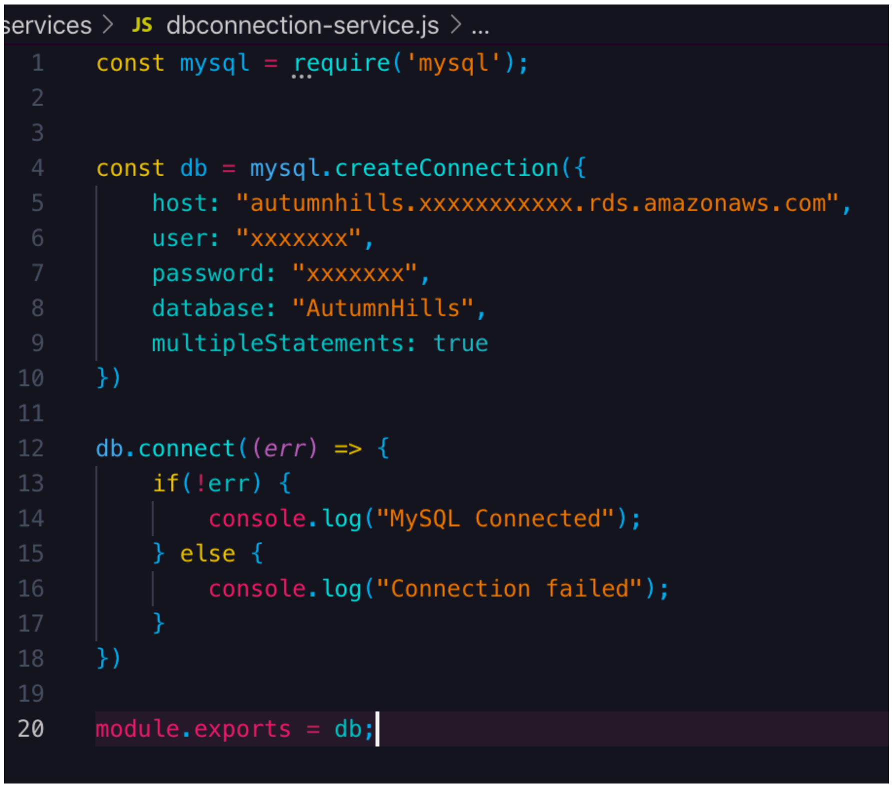
As an example, let’s look at the dbconnecction-service.

app.js (entry point)



```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const db = require('./services/dbconnection-service')
4  const createAccountService = require('./services/login_service');
5
6  db;
7
8
9  const app = express();
```

dbconnection-service



```
1   const mysql = require('mysql');
2
3
4   const db = mysql.createConnection({
5       host: "autumnhills.xxxxxxxxxxx.rds.amazonaws.com",
6       user: "xxxxxxx",
7       password: "xxxxxxx",
8       database: "AutumnHills",
9       multipleStatements: true
10  })
11
12  db.connect((err) => {
13      if(!err) {
14          console.log("MySQL Connected");
15      } else {
16          console.log("Connection failed");
17      }
18  })
19
20  module.exports = db;
```

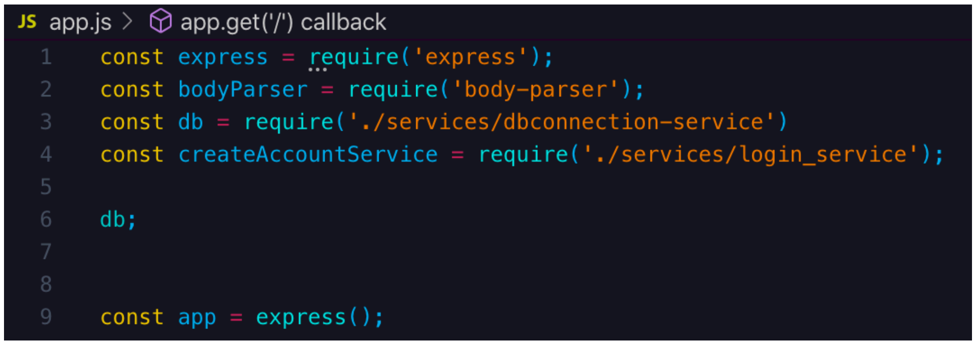
When looking at app.js, which is the main file the web application starts with, you can see that we are requiring the ‘db’ service on line 3. The db service is shown in the dbconnecction-servicce screenshot. In this file, the database credentials are declared and the connection is being established. This service handles one thing, and one thing only; to connect to the database. It is then exported, where it then gets imported into app.js. On line 6 of app.js we simply call 'db' to start the service. As the other services that were listed above are built out, they will be handled the same way.

When appropriate, the service layers are using some form of RESTful principles since our server is transferring the data state from the client to the server. By utilizing the Express library within the Autumn Hills application, I am able to make GET and POST HTTP requests within the application. This was actually one of the reasons I opted to build my application using a JavaScript framework versus in PHP. While PHP can handle GET and POST requests, JavaScript offers more flexibility in managing the state that is transferred throughout the application. With the JavaScript application, the app.js file will initially call a GET method on the routes that are being set up in Express. When data is requested, and the request is successful, data is displayed on these routes after a POST is called. If a request is unsuccessful then we will catch the error through Express' error handler and a message will be sent to the user.

Accessing the Endpoints

Below is a diagram showing which pages will utilize each service endpoint. But first, let's look at one more example showing how a POST request is accessing one of the services:

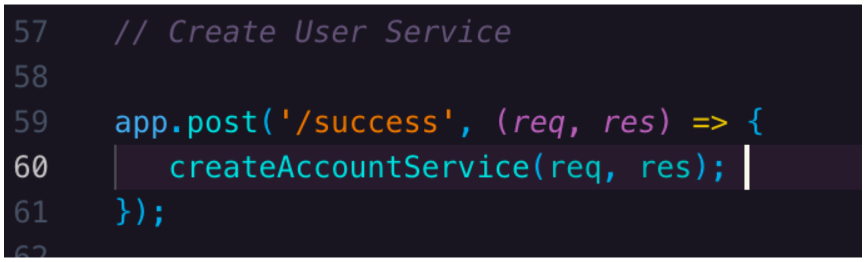
app.js



app.js > app.get('/') callback

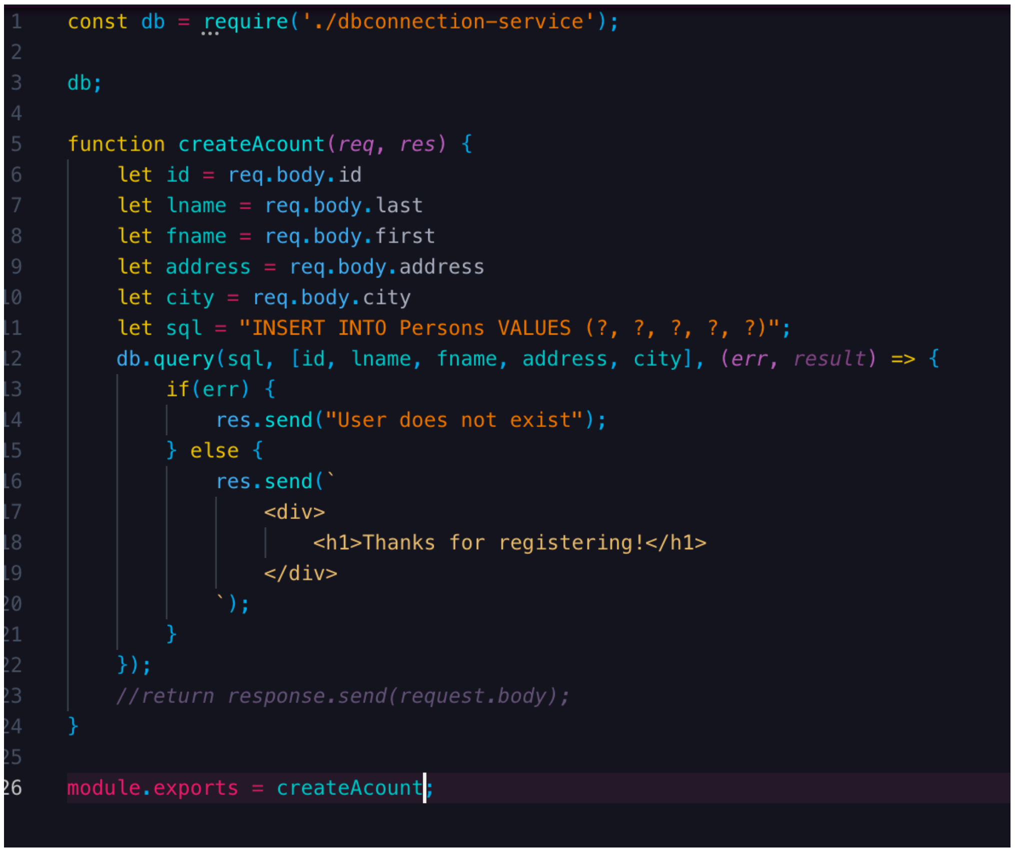
```
const express = require('express');
const bodyParser = require('body-parser');
const db = require('./services/dbconnection-service')
const createAccountService = require('./services/login_service');

db;


const app = express();
```

further down in app.js



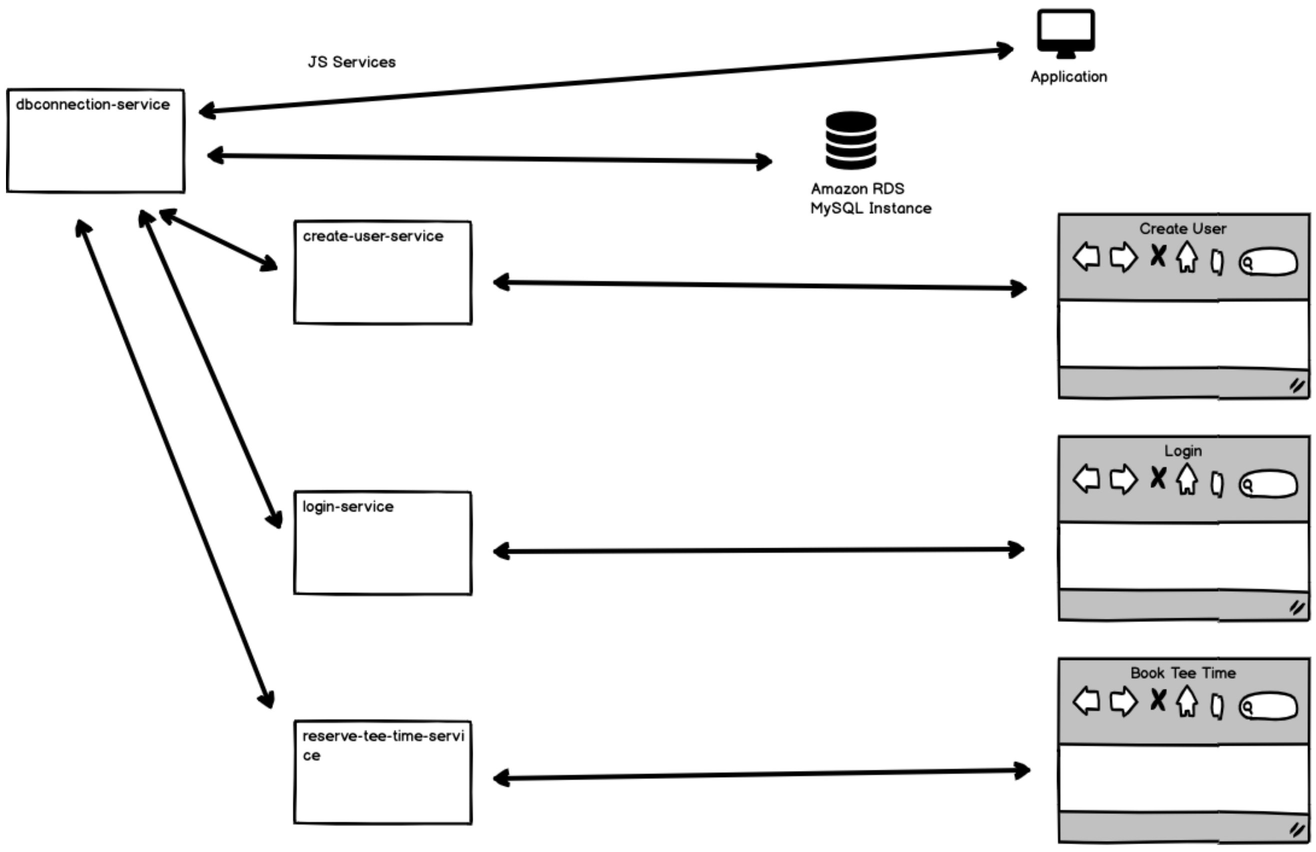
```
// Create User Service

app.post('/success', (req, res) => {
    createAccountService(req, res);
});
```

create-user-service



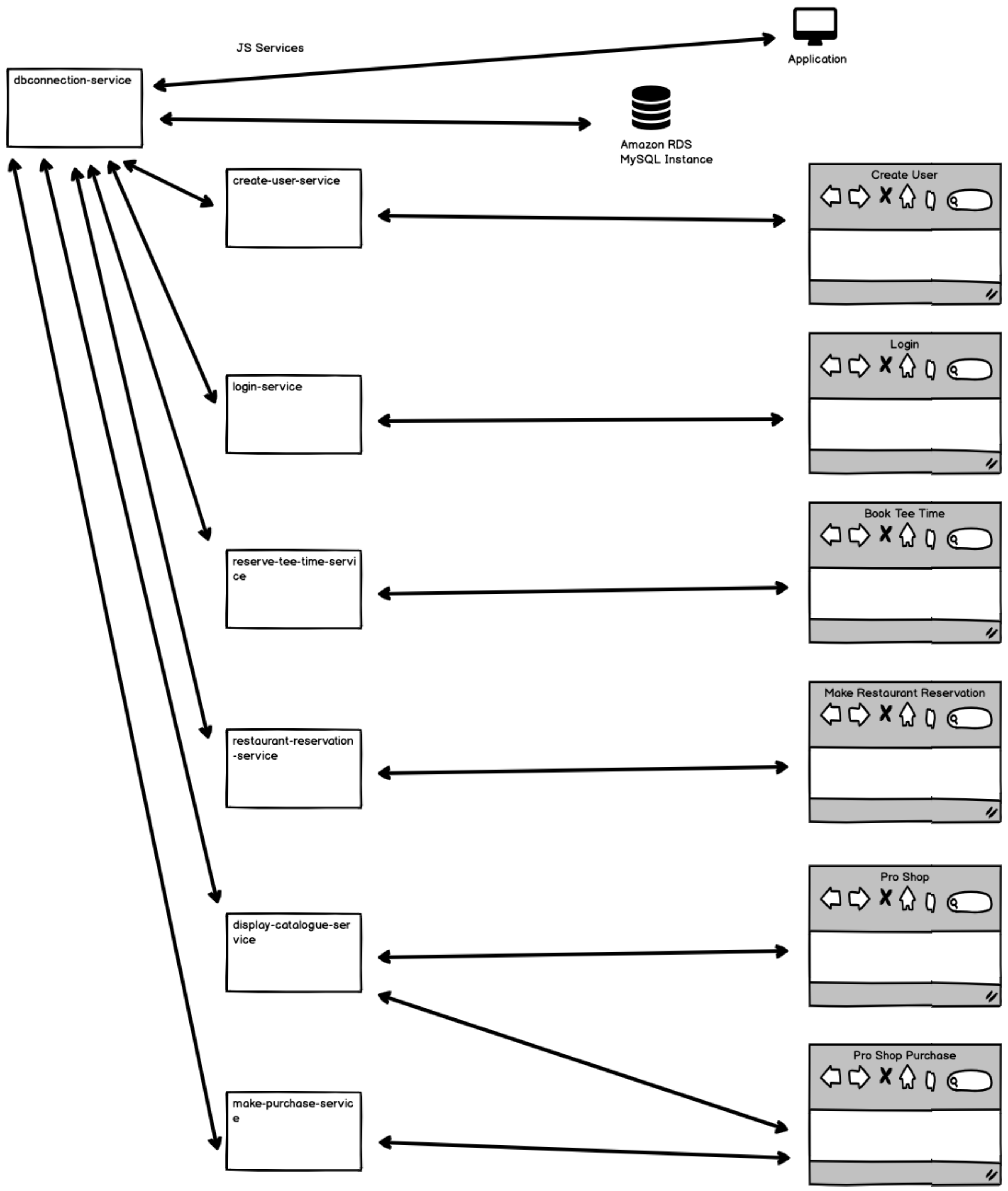
```
const db = require('./dbconnection-service');

db;

function createAcount(req, res) {
    let id = req.body.id
    let lname = req.body.last
    let fname = req.body.first
    let address = req.body.address
    let city = req.body.city
    let sql = "INSERT INTO Persons VALUES (?, ?, ?, ?, ?)";
    db.query(sql, [id, lname, fname, address, city], (err, result) => {
        if(err) {
            res.send("User does not exist");
        } else {
            res.send(`
                <div>
                    <h1>Thanks for registering!</h1>
                </div>
            `);
        }
    });
    //return response.send(request.body);
}

module.exports = createAcount;
```

Again, app.js is our app's starting point. At the top of app.js we are requiring the create-user-service. Further down in app.js on line 59 a POST request is being made and within that POST request the create-user-service is being called. The create-user-service file processes the data passed into it and inserts a new user into the database. Here are examples of how some pages will interact with the different services:



As we look at the diagram above, we can see that the application will first connect with the database through the dbconnection-service. Depending on the page a user is interacting with, its respective service will be called. As we can see, since all of the other services will need a database connection, they will need run through the dbconnection-service first in order to continue their task.

As the system expands, the next feature we would look to implement would be the ability for users to make a dinner reservation at the country club. Once that feature is implemented, we would then incorporate another feature that launches an online pro shop where users can purchase apparel and equipment to use when golfing. Each of these features will continue to interact with the database that has already been established, however new services will need to be created to send data between the user interface and the database. The diagram on the next page shows how services will interact between the database and the user interface when new stretch features are implemented.


JS Services
Application
dbconnection-service
Amazon RDS
MySQL Instance
create-user-service
Create User
login-service
Login
reserve-tee-time-service
Book Tee Time
restaurant-reservation-service
Make Restaurant Reservation
display-catalogue-service
Pro Shop
make-purchase-service
Pro Shop Purchase

The diagram above builds upon the original MVP service layer diagram earlier in this document. The three new stretch features that will be implemented in the future are “Make Restaurant Reservation”, “Pro Shop”, and “Pro Shop Purchase”. With these features come three new services:

restaurant-reservation-service - this service will allow the user to make a dinner reservation at Autumn Hills country club. The user will input the date, time, number of guests, and an optional message for the staff and from there the service will handle the user input and insert a new reservation into the database.

display-catalogue-service - this service will query the database and display pro shop inventory on the page. The inventory is stored within the database, so in order to serve it to the page that the user is interacting with, the service will query the database and display items depending on the category of items a user is looking at.

make-purchase-service - this service will hand the process of storing user purchase data when they buy an item from the pro shop. When the user is on the “Pro Shop Purchase” page, there are actually two services in play; the display-catalogue-service will display the item on the page that the user wants to buy, and when the user decides to make a purchase, then the make-purchase-service handles the process of storing the user purchase data in the database.