Dublin Business School

Higher Diploma in Science in Computing
(Software Development)
Final Report

Solution Title: thÉireP || ePrescriptions

Student Details:

Cian Walker

10021419@mydbs.ie

Student No: 10021419

Project Supervisor: Dr. Obinna Izima

Submission Date: 06/02/2023

URL: https://theirep.onrender.com

GitHub Repository Link

Word Count (Body): 3936

Abstract:

Having built a basic Python/Flask/SQLite CRUD application for ePrescriptions as part of my final module in Advanced Programming (GitHub Repository here), I wished to deepen my knowledge and skills in full-stack web-application development.

In particular: I wished to emphasise learning in both user authentication and the latest and most relevant development technologies used in the wild today.

As is attested to within the proposal and even within the interim report, I had only a couple of tooling certainties decided upon until quite a late stage in the project lifecycle. I knew I wanted to implement the application in JavaScript and to build it in a Node.js runtime environment. Aside from that, I envisaged serving HTML from a single backend server in much the same way as a Python/Flask/SQLite stack would, albeit with much more dynamic elements controlled by more JavaScript.

It was apparent to me early on that Node.js is used in conjunction with the Express server framework and is increasingly being back-ended by a MongoDB non-relational database. What I stumbled upon quite late – late November, with only approximately one third of the project timeframe remaining – was the React.js frontend framework and the MERN stack that it rounds out in conjunction with the technologies already mentioned above.

With the retrospective knowledge that a React frontend framework is multiple times more complex that the rest of an application stack combined, it was foolhardy on one level to attempt to implement this layer with only one of the three months remaining. Some of the functionalities and interfaces are rough and patchy as a result. On another level, using the MERN stack and gaining an understanding of how these technologies integrate has been a profoundly useful learning experience. The insights and experience I've been forced to glean as regards authentication in addition to application state and properties have been invaluable.

Acknowledgments:

Further to previous acknowledgements in my interim report to Robert Graham and Gonzalo Lorca of IBM, my supervisor Dr. Izima, Claire Caulfield, Jennie Byrne, and Jon Harry: I'd like to especially acknowledge Bob Kalka, global head of Security PreSales Engineering, and Constantinos Stavrou, Security PreSales Engineering lead UK & Ireland of IBM, for facilitating and encouraging my transition to a new technical direction in life. My partner, Gillian Murphy, also kindly and patiently endured endless recitation of the logic behind MERN-stack development so that I could get it straight in my own head. Her generosity took a particular sting out of the process for me.

Contents

List of contents, including tables and figures, with page numbers.

Chapter 1: Introduction

By far the greatest additions and contrasts to be found in this report since its interim counterpart are all related to the React.js frontend framework and the learning paths it has led me down. What I had learned virtually nothing of when building my Python/Flask/SQLite stack was the concept of application state and properties. My learning steered itself in this direction quite inevitably as a consequence of my interest in identity & access management. While user authentication is far from the only aspect of application state, it is an elemental one. The available literature, training tutorials, and materials out there remain quite disjointed and dependent on almost infinite different combinations of technologies. The MERN stack seemed the combination most likely to have consistent and reliable pedagogical materials available to it, in addition to being extremely ubiquitous, current, and relevant. Coming around to the technical meat and bones of the implementation, I would state that the greatest change in my perspective since the interim report is the following. Barely six weeks ago, I conceived of front-end and user interface as synonyms for the same material object. I now realise that user interface is but a subset of front-end development. Application state and properties and their management can be a gargantuan undertaking, and one that is intimately intertwined with the discipline in which I am an apprentice engineer: Identity & Access

management. Following the implementation of my Python/Flask/SQLite stack, I'd imagined a web application as being composed of three layers:

- HTML styled with CSS and dynamicised by JavaScript.
- A middle layer of REST APIs performing HTTP crud functions between HTML pages and a database.
- A relational or non-relational database being read, written to, updated, and deleted.

Reminiscent of how time is thought of as a fourth dimension, I now realise the importance of the consideration of *statefulness*, which I understand to mean the status of a user's session within the application. To reiterate: the combination of motivation to gain skills in identity and access, coupled with the need to choose a technology that had ample learning resources available to it, plus the fact that I had already determined to implement the platform in a Node.js runtime environment, meant that a MERN stack web application was the only realistic direction of travel at that time. Statefulness aside, it was important to learn how to design and implement a single page web application as per the expected industry standard today. React.js, along with Angular.js and Vue.js, is one of the three pre-eminent frameworks in this regard also.

Chapter 2: Background/Literature Review

Around the time of my completion of the Python/Flask/SQLite stack, I had a mentoring session with an IBM colleague, Robert Graham. Robert delivers large Identity & Access projects with our clients post-sale and recommended that I become familiar with Node.js given my particular interest in consumer-facing identity and access, and because that is increasingly what developers are implementing.

As I delved into the JavaScript language generally, I came to realise that it had evolved into something completely unique. I would describe it as having evolved from the user interface — where it was initially just a markup element orchestrator and dynamiciser (Eich, 2011) — back 'down the valve' (for want of a better term) into the server where it came to be used to write APIs. This seems singularly unique for a programming language. Further to this, JavaScript Object Notation (JSON) had become arguably the most common non-SQL data format in use. I began to notice the emerging '*JavaScript Everywhere'* (Rauch, 2012) trend and became keen to write my web application from one end to the other in one programming language. Furthermore, learning JavaScript is somewhat unique compared to other languages in that it's originally designed to orchestrate and renders user interface elements. Where development students may spend a lot of time playing around with Python or C-type languages in a command line-style interface, JavaScript feels 'real' very quickly as youcan quickly and easily run your code in a browser. I've somewhat fallen in love with it as a language. I hope I will receive some credit for being almost completely self-taught in this regard as it was only covered as an as aside and as a fraction of a topic in two modules.

Beyond JavaScript as a mere language, it quickly became apparent that Node.js was a very suitable programming paradigm to use for an application scoped the way *thÉireP* is. In an application where there is minimal processing of data sets, Node's non-blocking in/out event loop is optimised for speed of HTTP request and response (Chitra & Satapathy, 2017). This asynchronicity is a textbook 'hack' that sweats performance from assets without requiring increased compute power or indeed even mbps network performance:

"As an example, let's consider a case where each request to a web server takes 50ms to complete and 45ms of that 50ms is database I/O that can be done asynchronously. Choosing non-blocking asynchronous operations frees up that 45ms per request to handle other requests. This is a significant difference in capacity just by choosing to use non-blocking methods instead of blocking methods." – Node.js Documentation.

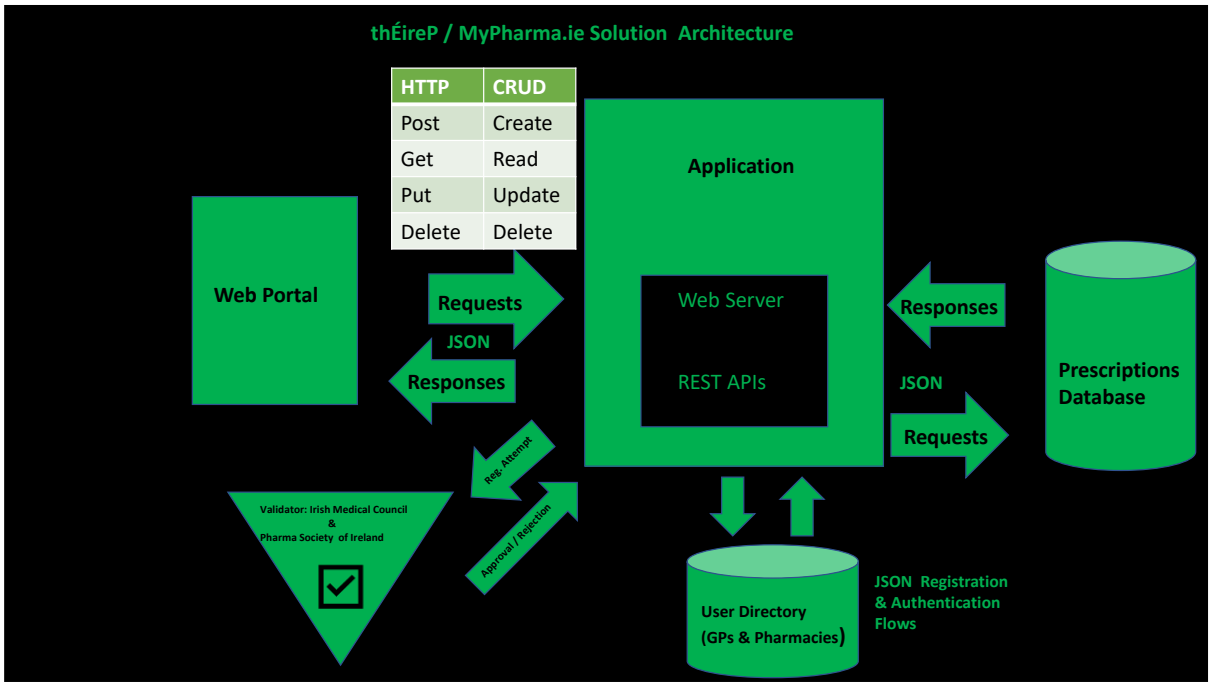
In light of this, I was positive that Node.js was an ideal programming paradigm in which to wite an effective application for registering and authenticating users; and submitting, retrieving, updating, and deleting prescription records.

By the point in time of late November to early December, I had become quite familiar with JavaScript and had built some nice dynamic user interface but was still struggling on the research front in terms of finding frameworks and tutorials for integrating user authentication workflows into a web application. I was particularly adamant of my need to learn to implement an OAuth/OpenID Connect-style authentication flow that uses JSON Web Tokens (JWT) in browser local storage to determine if a user is authenticated or not. I've understood this principle for quite some time now being an Identity & Access solutions engineer but I was extremely anxious to learn the nuts and bolts of *how* this is implemented at a deepr, more practical level.

Chapter 3: Specification and Design

This section is best tackled in terms of the delta between the envisaged solution architecture snapshot in mid November against the present schematic.

This is pulled, unedited, from my interim report:



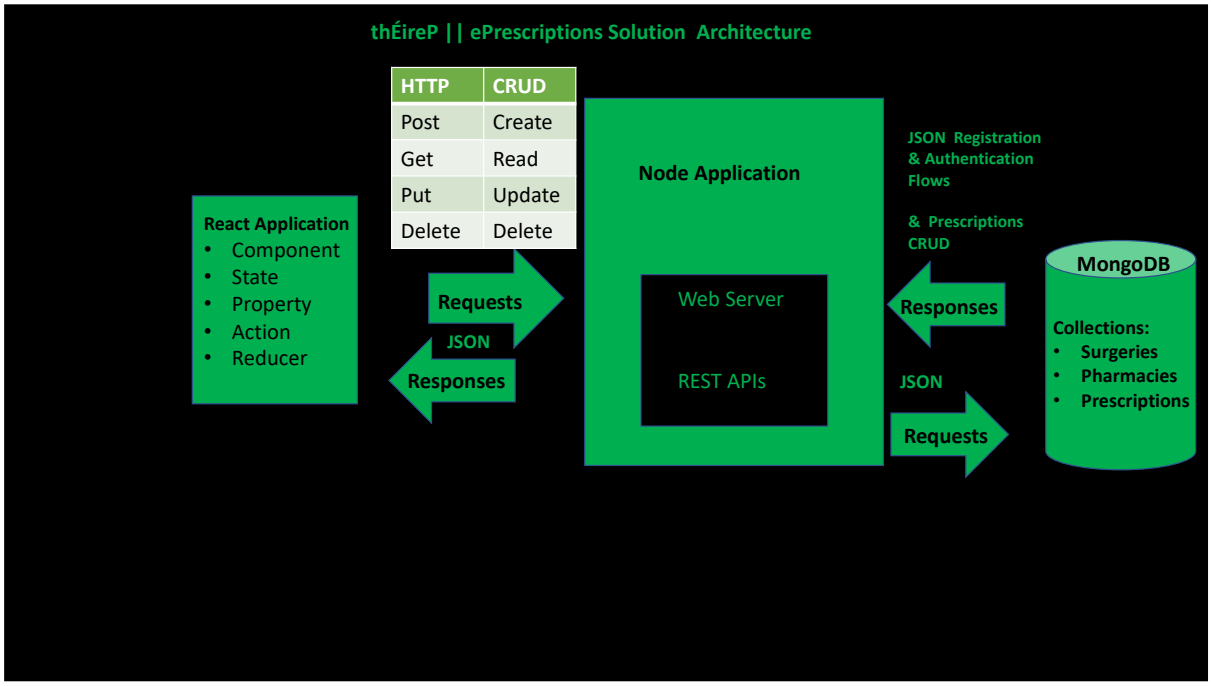
In actuality, the backend was manifestly less complex than I envisaged, with the front end being of far, far greater complexity than I imagined. A simple web portal of styled HTML with JavaScript functions, either inline, or in DOM-manipulating app.js file, became a fully-fledged front-end application unto itself running on a separate server.

React front-end applications use a single-page application paradigm similar to that seen in Angular.js or Vue.js. Components are injected into the single page as a user calls upon them, and action functions, properties, and families of state reducers need to be written in their own files that need to be able to find, call, and respond to one another. It can feel a little like trying to think in five dimensions having to consider components, states, actions, properties, and reducers – and all of this when I'd only given myself three to four weeks to become acquainted with it.

On the backend: the data structures were much more simplistic with there being three 'collections' – which would be MongoDB's closest equivalent to entities or tables – for the two kinds of users, Doctors and Pharmacists, with Prescriptions being the third kind of entity. MongoDB creates JSON documents for the equivalent of each 'row' of a table and the MongoDB/JSON equivalent of an SQL schema is a model. Similar kinds of routes and functions were used to register and authenticate users as were used to submit and retrieve prescriptions.

Another component of the web application that I could not implement before I ran out of time was a Pharmaceutical Society of Ireland / Irish Medical Council number validator similar to the one coded in my Python/Flask/SQLite stack. This infrastructure and functionality is completely absent.

This is a more reflective schematic of the web application at time of submission:



Here are the three MongoDB/JSON models (schemas) that outline the data structure:

Surgeries:

```
    IMCN:{
        type: Number,
        required: true,

    },
    DocEmail:{
        type: String,
        required: true,

    },
    DocPassword:{
        type: String,
        required: true
    },
    DocPasswordConf:{
        type: String,
        required: true
    },
```

```
    DocName:{
        type: String,
        required: true
    },
    DocPhone:{
        type: Number,
        required: true,

    },
    DocAddress:{
        type: String,
        required: true
    },
    avatar:{
        type: String,
    },
    Date:{
        type: Date,
        default: Date.now
    },

});
```

Pharmacies:

```
    PSIN:{
        type: Number,
        required: true,

    },
    PharmaEmail:{
        type: String,
        required: true,

    },
    PharmaPassword:{
        type: String,
        required: true
    },
    PharmaPasswordConf:{
        type: String,
        required: true
    },
    PharmaName:{
        type: String,
        required: true
    },
    PharmaPhone:{
        type: Number,
        required: true,

    },
    PharmaAddress:{
        type: String,
        required: true
    },
    avatar:{
        type: String,
    },
    Date:{
```

```
        type: Date,
        default: Date.now
    },

});
```

And finally, Prescriptions:

```
   PPSN:{
        type: String,
        required: true,
    },
    patientName :{
        type: String,
        required: true,
    },
    presFreQ:{
        type: Number,
        required: true
    },
    prescContents:{
        type: String,
        required: true
    },
    prescDosageMG:{
        type: Number,
        required: true
    },
    prescribedDate:{
        type: Date,
        required: true,
        default: Date.now
    },
    dispensedDate:{
        type: Date,
        default: Date.now
    //    similar to like button?
    },
    isDue:{
        type: Boolean,
    },

});
```

Note: I have taken the decision to slightly alter the suggested format of this report as suggested by the project handbook, covering both implementing and testing in the next chapter, with appraisal, conclusions, and future work in the next and final chapter. This reflects how the work was executed in practice during the process.

Chapter 4: Implementation & Testing

I became acquainted with JetBrains' Integrated Development Environments this time last year during our *Object Orientated Programming* module, during which I used PyCharm. My student subscription included WebStorm, which was my primary tool (version 2022.3.1) while writing this solution. Postman (version 10.6.7) was an invaluable tool for making quick HTTP pings to my backend Node server whilst I was building out the backend infrastructure. While it may seem obvious, it's worth mentioning the incredible power of Google Chrome's developer tools, arguably a perfectly capable IDE in and of itself, it was also essential for monitoring HTTP traffic in the console.

By mid-November, I had completed enough JavaScript training materials to have built out a dynamic form (directory: thisfolder/FormProto.html) with a pop-up confirm table complete with small delete icons where multiple records could be stacked and removed before confirming submission. I had had it in mind as a prototype data entry form. I had also implemented a login-dropdown box (my first ever). This brings us up to the point (GitHub commit link here) where I put 'vanilla' JavaScript aside and launched into acquainting myself with the Node Packet Manager and MongoDB in order to get building the application's backend.

The greatest breakthrough was, on December 6th, I was able to successfully POST user registration data from Postman to the server, have the server write it into a MongoDB document, whilst salting and hashing their password:

```
router.post('/', [
                    check('IMCN', 'Please Enter a valid IMC Number').isNumeric().not().isEmpty()
                    , check('DocEmail', 'Please Enter an Email').isEmail().not().isEmpty(),
                    check('DocPassword', 'Password Should Contain at least 8 characters').isLength({min
                    async (req, res) => {
                    const errors = validationResult(req);
                    if(!errors.isEmpty()){
                    return res.status(400).json({errors: errors.array()});
                    }
                    console.log(req.body);


                    const {IMCN, DocEmail, DocPassword, DocPasswordConf, DocName, DocPhone, Do


                    try{
```

```
let surgery = await Surgery.findOne({DocEmail});

//Check is user exists

if(surgery){
res.status(400).json({errors: [{msg: 'There is already a surgery registered against this
}

//Get Doctor's Avatar
const Docavatar = gravatar.url(DocEmail, {s: '200', d: 'mm'});

surgery = new Surgery({
IMCN,
DocEmail,
DocPassword,
DocPasswordConf,
DocName,
DocPhone,
DocAddress

});

//Salt & Hash Password
const salt = await bcrypt.genSalt(12);
surgery.DocPassword = await bcrypt.hash(DocPassword, salt);
await surgery.save()



//Return JWT









res.send('Surgery Successfully Registered!');
} catch(err){
```

```
                console.error(err.message);
                res.status(500).send("Server Error");


                }









                });
```

The next day, December 7th, I could successfully POST request user registration data to the server and receive back JWTs which could in turn add to private GET request headers in order to pull down data from private routes. The middleware used to makes these routes private (ie requiring an auth token in the header) was very interesting to implement:

```
const jwt =
require('jsonwebtoken');
                const config = require('config');
                module.exports = function(req, res, next){
                // Get token from header
                const token = req.header('x-auth-token');
                // Check if no token
                if(!token) {
                return res.status(401).json({msg: 'You must be logged in to carry out this
                operation'});
                }
                // Verify Token:
                try{
                const decoded = jwt.verify(token, config.get('jwtSecretPharma'));
                req.pharmacy = decoded.pharmacy;
                next();
                } catch(err){
                res.status(401).json({msg: 'Auth Token Invalid!'});
                }
                }
```

As an apprentice IAM engineer, this was all extremely exciting and satisfying but I was still mostly at a complete loss how any of this backend authentication technology would be

rendered or manifested at the front end. I was still searching furiously for training materials and repositories that I could learn these implementation patterns from.

Come December 13th, I had resolved to add what I thought would be a relatively thin layer of user interface orchestration to the solution. I understood at that point that React.js is a single page paradigm that renders components as and when they are called by the user. Where in classic HTML/CSS/JS implementations, the JavaScript was embedded in the HTML (and more recently called on by the document object model from an app.js file) – React.js seemed to embed HTML within JavaScript to make components that were injected into the "root" div. As with many other aspects of React.js, I underestimated how complex even this initial component stage was. React components render a form of Extensible Markup Language known as JSX, which is a form of syntactic sugar that, under the hood, in fact runs as classical createElement functions. There was far more work required than I envisaged in converting classical HTML to JSX components. Many elements would render and many would not since many of the syntactic rules along with directory and filing conventions were different, despite many being the same. It was somewhat akin to trying to extremely quickly switch from Spanish to Italian. In addition: JSX is far less forgiving of coding errors than HTML is, with mistakes crashing the entire front-end application. For instance, this HTML:

```
<div class="navbar">

   <div class="dropdown">
      <button class="dropbtn">RegisterDoc
         <i class="fa fa-caret-down"></i>
      </button>
      <div class="dropdown-content">
         <a href="RegisterSurgery.html">RegisterDoc Surgery</a>
         <a href="RegisterPharmacy.html">RegisterDoc Pharmacy</a>
      </div>
   </div>
</div>

<div class="dropdown">
   <button class="dropbtn">Prescriptions
      <i class="fa fa-caret-down"></i>
   </button>
   <div class="dropdown-content">
      <a href="/FormProto.html">Submit Prescription</a>
      <a href="/FormProto.html">Retrieve Prescription</a>
      <a href="/FormProto.html">Amend Prescription</a>
   </div>
</div>
</div>
```

Becomes this JSX:

```
<>
  <div className="navbar">
    <div className="dropdown">
      <button className="dropbtn">
```

```
        RegisterDoc
        <i className="fa fa-caret-down" />
      </button>
      <div className="dropdown-content">
        <a href="RegisterSurgery.html">RegisterDoc Surgery</a>
        <a href="RegisterPharmacy.html">RegisterDoc Pharmacy</a>
      </div>
    </div>
  </div>
  <div className="dropdown">
    <button className="dropbtn">
      Prescriptions
      <i className="fa fa-caret-down" />
    </button>
    <div className="dropdown-content">
      <a href="/FormProto.html">Submit Prescription</a>
      <a href="/FormProto.html">Retrieve Prescription</a>
      <a href="/FormProto.html">Amend Prescription</a>
    </div>
  </div>
</>
```

At this late stage of the project, I didn't expect to be struggling with moving user interface elements around the viewing port as I had just started to become reasonably comfortable with classic HTML.

Depending on how it's viewed, I 'lost' roughly ten days wrestling with these issues. This question/contrast will be returned to throughout the remainder of this report. Depending on whether we regard learning outcomes or treat the project like a production deadline, the appraisal of whether my decision to explore these avenues could be seen as either foolhardy or studious.

From December 20th onwards saw the most difficult challenges the project faced, and as will be evident in the demonstration, these challenges were not entirely overcome. I had begun to feel comfortable handling HTTP requests and responses of all flavours but the calls and responses from within the React application felt like experiencing different laws of physics. As I will demonstrate in the presentation, GET requests for prescriptions, when made from

Postman directly to the Node server, work perfectly. But MongoDB queries seem to somehow get lost in translation as they're made through layers of React code.

In addition to this, I encountered a Cross-Origin-Resource-Sharing (CORS) issue during these days that, in the past it seems, and according to most training materials, could be circumvented by using a proxy in the configuration (package.json) file. This meant that the backend Node server would not accept HTTP requests from the frontend Node server. A node cors module was able to circumvent this issue, but again; not before it had 'cost' me approximately two days.
.

Chapter 6: Appraisal, Conclusions and Future Work

My appraisal of this project is bittersweet. I am bitterly disappointed at some of the holes in the desired use cases, while at the same time being most pleased with the tour I've taken of a completely new and self-taught programming language. I am also very nervous about whether, taken in totality I have done enough to warrant a passing grade. If I have, I believe it will be the relatively functional node backend server that will merit it.

The obvious fact is that I've been over ambitious and perhaps over-curious which has caused me to trade any depth for too much breadth. The only full use case I can say is barely fully implemented is the user authentication flow that grants tokens on register login, and removes them on logout. This is no small feat but I am also disappointed I could only half implement, as deep down as the front end, identity federation with Google Identity Services (ie it can obtain tokens from Google but I didn't have time to implement an API call to the users collections to either register or authenticate new user accounts).

Curiosity and arguable desperation to know about certain technology under the hood and stick my hands into it in order to understand it enough to better help my (mostly developer) client base caused a sacrifice of functionality for exploration – I hope I can get some credit for this but I am very concerned and disappointed, particularly with inability to finish the GET, PUT, and DELETE prescription use cases.

While these basic functionalities being left unfinished, I have implemented these successfully before. Therefore, the biggest disappointment of all was in not doubling down on my identity & access more to write properly different workflows/clearances for doctors and pharmacists. This would be the one biggest thing I would change if I could, controlling what the different kinds of users can do once authenticated is as important as the authentication itself. Again, this is a result of compromising depth for breadth and fudging too many things, resulting in none of them being thoroughly implemented.

Here is a table of sacrifices and gains caused by my decision to indulge my curiosity rather than play conservatively and implement successful use cases :

Sacrificed	Gained
Pharmacist /Doctor Divergent Role Authorisation	**Intermediate JavaScript competency**
IMC / PSI Number Validator	**Single Page App Dev Understanding**
Polished UI	**Understanding of the async Node backend paradigm.**
Marking Prescriptions 'dispensed' with timestamp.	**In depth understanding of HTTP request and response processes**
GET, POST, DELETE actions.	**User authentication flows.**
Identity Federation that connects to the backend system.	**Understanding of frontend aspects outside of UI such as State, Properties, Actions, and Reducers.**

It would have been perfectly reasonable to claim that the time to start implementing a full MERN architecture was mid to late October instead of early December but I keep coming back to how much better armed I am to be a friend to my developer clients and even get involved in repairing broken integrations (again usually REST APIs) as an advanced solutions engineer such as myself is expected to do as I develop into the role going forward. I spent too much of the early stages on basic JavaScript such that, by the time the React framework got on my radar, it was probably too late.

Finally, some thoughts on conclusions and future learning:

The project has confirmed in me my preference for backend work. It almost seems as if the further forward in the stack I go, the more difficult I find it. It inverse is also true whereby I always find the data layer extremely intuitive. In particular, APIs are my first passion and so I hope to deepen my skills even further in the development of those and perhaps specialise in application integration in future. Frontend development feels like an overwhelming amount of work for a thoroughly underwhelming amount of return.

Two pre-eminent further areas of future learning are:

Containerisation: I am reasonably comfortable with building and destroying virtual machines of all flavours, and I particularly enjoy building virtual Windows or Linux servers. However, my understanding of containers is extremely rudimentary. This project has made me fully appreciate the value in being able to serve and application from one with all of its dependencies tucked tidily inside.

Building and speaking LDAP: as someone who finds the data layer of computing relatively intuitive, I feel it essential for me to build an LDAP (perhaps an Oracle OpenLDAP) on a Linux server and learn to speak it's schematics and queries as fluently as I speak SQL

Appendix:

URL: https://theirep.onrender.com

GitHub Repository Link

Bibliography:

Abeln , M. (2019) *What is the purpose of having two running ports when we working with reactjs and nodejs?*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/57362813/what-is-the-purpose-of-having-two-running-ports-when-we-working-with-reactjs-and (Accessed: December 19, 2022).

Admin (2022) *Redux in ReactJS*, *W3schools*. Available at: https://www.w3schools.blog/redux-reactjs (Accessed: December 20, 2022).

Bosler, F. (2020) *Set up an express.Js app with passport.js and mongodb for password authentication*, *Medium*. The Startup. Available at: https://medium.com/swlh/set-up-an-express-js-app-with-passport-js-and-mongodb-for-password-authentication-6ea05d95335c#94df (Accessed: January 4, 2023).

Chitra, L.P. and Satapathy, R. (2017) "Performance comparison and evaluation of node.js and traditional web server (IIS)," *2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)* [Preprint]. Available at: https://doi.org/10.1109/icammaet.2017.8186633.

Codes, C. (2022) *Google Identity Services Login with react (2022 react Google Login)*, *YouTube*. YouTube. Available at: https://www.youtube.com/watch?v=roxC8SMs7HU&t=944s&ab_channel=CooperCodes (Accessed: January 5, 2023).

Conditional Rendering (no date) *ReactJS*. Available at: https://reactjs.org/docs/conditional-rendering.html#gatsby-focus-wrapper (Accessed: January 3, 2023).

CORS (no date) *Node Package Manager*. Available at: https://www.npmjs.com/package/cors (Accessed: December 22, 2022).

Crockford , D. (2011) *Douglas Crockford: The JSON Saga*, *YouTube*. YUI Library. Available at: https://www.youtube.com/watch?v=-C-JoyNuQJs&ab_channel=YUILibrary (Accessed: December 6, 2023).

Eich, B. and Friedman, L. (2021) "Brendan Eich: JavaScript, Firefox, Mozilla, and Brave | Lex Fridman Podcast #160," *The Lex Friedman Podcast*. Available at:

https://www.youtube.com/watch?v=krB0enBeSiE&ab_channel=LexFridman (Accessed: November 30, 2022).

Erikson, M. (2022) *Redux createstore() is deprecated - cannot get state from getState() in Redux Action*, *Stack Overflow*. Available at: https://stackoverflow.com/a/71947129/19219155 (Accessed: December 20, 2022).

Fatunmbi, T. (2022) *A comparison of cookies and tokens for secure authentication*, *Okta Developer*. Okta Inc. Available at: https://developer.okta.com/blog/2022/02/08/cookies-vs-tokens#:~:text=Cookies%20and%20tokens%20are%20two,characters%20created%20by%20the%20server. (Accessed: October 4th, 2022).

JavaScript Async (no date) *W3 Schools*. Available at: https://www.w3schools.com/js/js_async.asp#:~:text=async%20makes%20a%20function%20return,function%20wait%20for%20a%20Promise (Accessed: October 19, 2022).

JSX in depth (no date) *React*. ReactJS. Available at: https://reactjs.org/docs/jsx-in-depth.htm (Accessed: December 13, 2022).

On accepting passwords at login:

Maaajomaaajo 74955 silver badges1010 bronze badges, K.E. (2020) *What is the Difference Between handlechange vs onchange in which is used in react?*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/62197917/what-is-the-difference-between-handlechange-vs-onchange-in-which-is-used-in-reac (Accessed: December 31, 2022).

Node.js (no date) *Overview of blocking vs Non-Blocking*, *Node.js*. Node.js. Available at: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/ (Accessed: December 13, 2022).

Olawanle, J. (2022) *Post HTTP Request in React*, *Stack Abuse*. Stack Abuse. Available at: https://stackabuse.com/post-http-request-in-react/ (Accessed: December 21, 2022).

Openid Connect (2022) *IBM.com*. International Business Machines Inc. Available at: https://www.ibm.com/docs/en/was-liberty/base?topic=liberty-openid-connect (Accessed: December 19, 2022).

Patadiya, J. (2022) *React vs react native - key difference, features, advantages*, *Radixweb*. Radixweb. Available at: https://radixweb.com/blog/react-vs-react-native (Accessed: December 19, 2022).

Raj, V. (2019) *Running react and node.js in one shot with concurrently!*, *DEV Community*. DEV Community. Available at: https://dev.to/numtostr/running-react-and-node-js-in-one-shot-with-concurrently-2oac (Accessed: December 19, 2022).

Rauch, G. (2012) *Smashing node javascript everywhere*. New York: Wiley.

React ES6 arrow functions (no date) *W3 Schools*. Available at: https://www.w3schools.com/react/react_es6_arrow.asp (Accessed: December 19, 2022).

React ES6 spread operator (no date) *W3 Schools*. Available at: https://www.w3schools.com/react/react_es6_spread.asp (Accessed: December 19, 2022).

Sakimura, N. *et al.* (2022) *OpenID Connect Core 1.0 incorporating errata set 1*, *OpenID Connect Core 1.0* . OpenID Connect. Available at: https://openid.net/specs/openid-connect-core-1_0.html (Accessed: December 22, 2022).

Start using your openid (2012) *OpenID*. Available at: https://openid.net/start-using-your-openid/ (Accessed: November 8, 2022).

Throw, and Try...Catch...Finally (no date) *JavaScript Errors*. W3 Schools. Available at: https://www.w3schools.com/js/js_errors.asp#:~:text=JavaScript%20try%20and%20catch,occurs%20in%20the%20try%20block. (Accessed: November 3, 2023).

Tiwari, V. (2022) *How to fix cors error "no 'access-control-allow-origin' header is present on the requested resource"?*, *codedamn*. Available at: https://codedamn.com/news/backend/how-to-fix-cors-error (Accessed: December 22, 2022).

Traversy, B (2019) *DevConnector_2.0: Social Network for Developers, Built on The MERN stack, GitHub*. GitHub. Available at: https://github.com/bradtraversy/devconnector_2.0 (Accessed: November 26, 2022).

Understanding and Resolving Cors Error (no date) *Contentstack RSS*. Available at: https://www.contentstack.com/docs/developers/how-to-guides/understanding-and-resolving-cors-error/ (Accessed: December 22, 2022).

Using the Effect Hook (no date) *ReactJS*. Available at: https://reactjs.org/docs/hooks-effect.html (Accessed: December 24, 2022).

Using the State Hook (no date) *ReactJS*. Available at: https://reactjs.org/docs/hooks-state.html (Accessed: December 19, 2022).

Why redux toolkit is how to use Redux Today (2022) *ReduxJS*. Available at: https://redux.js.org/introduction/why-rtk-is-redux-today (Accessed: December 20, 2022).