Technical Solution Document - 1

Author- Alexander Adu-Sarkodie

MSc Telematics MSc Engineering

LinkedIn - <https://www.linkedin.com/in/alexander-adu-sarkodie-063b4b4a/> GitHub - https://github.com/kukuu?tab=repositories

**……………………………………………………………………………………………………………………………………………………………**

**……………………………………………………………………………………………………………………………………………………………**

Penguin Random House

CRM Digital Solution Architecture

June 2018

This new Recipes Management CRM will replace the existing Monolithic platform.

The architectural solution is focused in the following areas:

● Architecturally Significant Requirements

● Systems Architecture (Data and Application)

● Infrastructure Architecture

* Scalability and Availability

………………………………………………………………………………………………………………..

………………………………………………………………………………………………………………..

**Architecturally Significant Requirements**

The architecture will be defined with **modern engineering lean practices to allow auto scaling depending on load**  peaks and spikes as mentioned in the brief.

The architecture will provide **lose coupling and separate the Backend from the Frontend**. The Backend will expose a web based API that the Frontend will consume. Typically, the backend will handle incoming requests and return a JSON or XML encoded response. The Frontend will then be in charge of formatting, styling, and displaying this response to the user. This is “separation of concerns” and applying it at this scale has many benefits.

**Portable Modular Components Architecture**

The backend services for the Management Solution can grow and evolve independent of the Frontend client. New APIs for PRH, if properly versioned, can provide new features and functionality without breaking existing integrations. A single Cookery backend can interface with multiple clients (Penguin, Ebury, Michael Josephs) and the Frontend clients will not be limited to any specific framework or programming language. I will focus on an architecture using AngularJS and REACT - an isomorphic JavaScript library for developing Single Page Web application to support the API. This means that your single backend can work with your multiple clients. For both a web based implementation, your mobile app, Theatre, Podcast, and even with IoT devices.

The Recipes Management solution will;

* Provide the ‘back-end’ for internal websites such as Happyfoodie, Ottolenghi and other external Author websites such as Nadiya Hussain or Melissa Hemsley.
* Support the management and tracking of promotional recipe material to prevent recipe content being permanently used rather than for promotional use.
* Define and create a more developed rights model for recipe content based on use and opportunities and potential integration with other PRH systems.
* The Backend will have an algorithm to check file format
* The Front end UI will have an API that will have a **mongoDB database to store imported files**. This will be available to the SEARCH mechanism that will be integrated as service module. By storing searched recipes for example at the UI level we can cache them for requests. This will effectively improve performance.
* The UI will provide CRUDE (Create, Update, Delete, Edit) features programmatically for searched items. This will be the edit management part of the Management Solution.

I recommend the use of **nodeJS platform and the use of REACT** at the UI level for development. Both architectures are hi-lighted further in this technical brief

The API needs to be protected with a digital security. In addition to protecting our API endpoints (eg. Ebury etc.), you will also have to add specific roles to each of our clients and show how we can give granular access to our recipes API using scopes.

For example, If you are challenged with building 4 different clients (Michael Josephs, Penguin, Ebury user facing and an admin for monitoring) which are linked, rather than building two separate backends for the user facing side and the admin application, you can build and expose a single API with four endpoints. The user facing app will have access to the 3 endpoints (explained above), while the admin app will additionally have access to the 4th endpoint which can potentially be used for moderation, and not publicly accessible. So, two scopes will be applied programmatically.

For implementing **digital security**, json web token (**JWT**) will be used. During implementation, I will add the ability to check if the client has permissions to view the endpoint requested. To do this, I will create a middleware that will look at the decoded JWT and see if it has the token has the correct scope. If it doesn’t I will send an appropriate forbidden message, otherwise will send the data.

This guard middleware will be called on each request and will ensure that the token has the correct scope. If it does, we’ll send the data, otherwise we’ll return a 403 Forbidden status and appropriate message.

Other architectural areas to consider to support development are:

* Cloud Computing (Services & Platforms)
* Code Management
* Code Framework
* Code Repository Structure and Development guidelines
* Components overview
* Coding Standards & Best Practices
* Error Handling
* Source Control

**UX Patterns**

* UX Best Practices - Keep It Simple (KISS)
* Responsive Web Design, Adaptive Designs & Mobile First
* Flexible Grid Systems
* Font icons
* Client and Server side component rendering

**Quality Assurance Approach**

* Testing Environment & Test Coverage, The testing Pyramid Matrix
* Functionality testing through BDD test Automation
* Browser Support
* Browser testing
* Operating Systems
* Mobile Devices

**Cloud Computing Platforms & Service Models**

* Software as a Service (SaaS)
* Platform as a Service (PaaS)
* Infrastructure as a Service (IaaS).

**Integrating Social Media, Email & Campaign Strategies**

**Application Development**

The application will be developed as a Progressive Web Application based in a **Single Page architecture (Single Page Application).**

PWAs leverage sophisticated web capabilities to deliver in-app experiences to mobile website visitors. This includes the ability for visitors to add the website to their mobile home screen, access content while offline, receive mobile notifications from publishers, and more!

It is another term for a website that has been optimised for mobile performance and that utilises newly available Web APIs to deliver features that are similar to a traditional native app, such as push notifications and offline storage.

**PWA** also leverages machine learning to discover important information about visitor browsing habits. It can then adjusts things like menu style and layout to help extend sessions and improve revenue.

PWAs deliver users faster and more dynamic experiences than regular mobile pages.

You can deploy your app as a PWA as well as Native app and take advantage of both channels. Ionic allows you to ship your app to not only the app store, but also deploy to the mobile web as a PWA.

With PWAs, one can offer all the features of an app on their mobile website without changing their CMS, content, or code.

This Single Page Application will built using REACT. REACT is chosen for the UI development because it provides rich interactive features and isomorphic by nature. Versatile library for both on the Server, the Client and great on administering performance.

By virtue of being a library other than a Framework , it can be easily plugged into different platforms. REACT creates virtual DOMs. These alleviates the traditional DOM manipulation conundrum associated with traditional client side processing. All of which slows down application performance.

**Adding Progressive Web App Support**

Adding PWA support to any existing frontend project is easy. Just add an **App Manifest file and configure a service worker:**

*App Manifest*

First, you'll need an App Manifest file (manifest.json) that sits **alongside your index.html** file and provides metadata about your app, such as its name, theme colors, and icons. This information will be used when your app is installed on the home screen, for example.

*Service Worker*

Next, in order to send **push notifications and store data offline**, a Service Worker will enable your web app to **proxy network requests a**nd perform background tasks needed to process and sync data.

Service Workers are powerful, but complicated. Generally, writing them from scratch is not recommended. Instead, take a look at tools like Workbox that provide common Service Worker recipes that you can easily incorporate into your app.