**Population Health Management UI Walkthrough –**

**Offboarding Document**

***Shafiq Mohammed – March 2024 – Offboarding Document***

***This document pertains to all aspects of the application, and serves as the official off-boarding material that was promised to Healthmaps personnel by Shafiq Mohammed, the core developer of the application.***

The code base for this project consists of the following components/packages:

1. *Resources/application.properties*
2. *Configuration*
3. *Security*
4. *Views*
5. *Data*

**App Flow Diagram**

A diagram of a service

Description automatically generated with low confidence

## Resources/application.properties

Here is where the following is configured:

* Server Name
* Server Port
* Database configuration –
  + url
  + user
  + database
  + credentials

Configuration

1. SpringAsyncConfig
   1. Threadpool configuration for running the actions in async mode.
   2. Currently this is used for executing the stored procedure without the need to wait due to the async nature.
2. PersistanceDS1AutoConfiguration : First database to connect to. What is configured here is the data source, the entity manager, and the transaction manager for the database. All the entity classes inside ds1 package are also configured to use this database.
3. PersistanceDS2AutoConfiguration : similar to the above but this configuration uses the second database and configures the ds2 package entity classes

Security

The spring boot application security is configured here

1. SecurityConfiguration : adds configuration to authenticate the API URLs
2. UserDetailsServiceImpl : Authenticates the username and password entered in the login page. Currently only configured to work with username – admin and password –admin. This is just to POC and we can add functionality here to authenticate users from a DB table
3. AuthenticatedUser : Keeps user logged in and redirects to login page if user is not authenticated – otherwise the internal pages if the user is logged in.

Views  
The package consists of individual page views for the routes.

1. MainLayout : This is the main layout which renders the left side navigation bar and main view
2. EnrichmentNavigationView : This consists of multiple components:
   1. Header : The header of the page
   2. Filters : Renders the text boxes to add filter to the grid data
   3. Grid : Paginated tabular view of the records in the database.
3. OtherViews : similar to the above
4. LoginView : Renders the login page and authenticates the user.

Data

This contains the entity, repository, and service classes. Since the application is connecting to two databases, we have two packages, ds1 and ds2, which connect to the respective databases.

1. Entity: The model class mapping to database columns
2. Repository: This contains the methods to perform CRUD operation on the respective table and execute the stored procedures.
3. Service: Connects the Views with Repository to perform the CRUD operations mentioned. Additionally here we add validations, exception handling etc.

**Code Walkthrough**

**Configuration**

## PersistenceDS1AutoConfiguration

This represents a Java configuration class for configuring and setting up the persistence layer for the ds1 data source in a Spring Boot application.

* The class is annotated with @Configuration, indicating that it is a configuration class for the Spring application context.
* The @PropertySource annotation is used to specify the location of the application.properties file, which contains the configuration properties for the data source.
* The @EnableJpaRepositories annotation is used to enable JPA repositories for the ds1 data source. It specifies the base package for repository scanning, as well as the references to the EntityManager and TransactionManager beans.
* The class includes various bean definitions and configuration methods for setting up the data source, entity manager, and transaction manager.

The different methods and their functionality:

* The ds1DataSource method is annotated with @Bean and @ConfigurationProperties. It creates and configures a DataSource bean using the properties prefixed with "spring.datasource" from the application.properties file.
* The ds1EntityManager method is annotated with @Bean and creates a LocalContainerEntityManagerFactoryBean bean. It configures the entity manager for the ds1 data source by setting the data source, packages to scan for entity classes, JPA vendor adapter (using Hibernate), and additional JPA properties such as naming strategy and dialect.
* The ds1TransactionManager method is annotated with @Bean and creates a PlatformTransactionManager bean. It configures the transaction manager for the ds1 data source by setting the entity manager factory.
* The @Primary annotation is used on certain beans to indicate that they should be considered as primary beans in case of multiple beans of the same type.

In summary, the PersistenceDS1AutoConfiguration class serves as a configuration class for the ds1 data source in a Spring Boot application. It configures the data source, entity manager, and transaction manager beans using properties from the application.properties file and sets up JPA repositories for the ds1 data source.

## Security Configuration

This class is responsible for configuring security settings for a Vaadin web application.

Here's an overview of the configuration:

* @EnableWebSecurity: This annotation is used to enable Spring Security's web security features in the application.
* @Configuration: This annotation indicates that the class is a configuration class.
* public class SecurityConfiguration extends VaadinWebSecurity: The SecurityConfiguration class extends the VaadinWebSecurity class, which is a convenience class provided by Vaadin for integrating Spring Security with Vaadin applications.
* @Bean: This annotation is used to declare a bean of type PasswordEncoder in the Spring application context. The PasswordEncoder bean is used for password encoding and decoding.
* public PasswordEncoder passwordEncoder(): This method defines and returns a BCryptPasswordEncoder instance as the password encoder.
* protected void configure(HttpSecurity http) throws Exception: This method is overridden from the VaadinWebSecurity class and is used to configure security settings for the application.
  + http.authorizeRequests().requestMatchers(new AntPathRequestMatcher("/images/\*.png")).permitAll(): This line of code allows access to any request matching the AntPathRequestMatcher pattern /images/\*.png without authentication. This means that any image file with the .png extension in the /images directory will be accessible without requiring authentication.
  + super.configure(http): This line invokes the superclass's configure() method to perform the default security configuration provided by VaadinWebSecurity.
  + setLoginView(http, LoginView.class): This line sets the login view for the application. It specifies the LoginView class as the login view to be displayed when authentication is required.

Overall, this configuration class enables web security, sets up a password encoder bean, configures security rules, and sets the login view for the application

Views

## EnrichmentNavigation View

1. **Class Overview**: The main class for this view is EnrichmentNavigationView, which extends the Vaadin Div component. It serves as the view for the enrichment navigation functionality. Let's explore its structure and functionalities.
2. **Annotations**:  
   @PageTitle("Enrichment Navigation"): This specifies the page title for the view.  
   @Route(value = "enrichment-navigation", layout = MainLayout.class): This maps the view to the URL path "/enrichment-navigation" and associates it with the MainLayout layout class.  
   @PermitAll: This allows unrestricted access to the view.
3. **Constructor**:  
   EnrichmentNavigationView(EnrichmentNavigationService enrichmentService): This constructs the view by initializing the enrichmentService and setting up the layout components. It also defines the event handler for filter changes (Filters class).
4. **Layout and Components**:  
   VerticalLayout layout: Represents the main layout for the view, containing the header, mobile filters, filters, and the grid.
5. H3 sectionHeader: Displays the section header for the view.  
   HorizontalLayout createMobileFilters(): Creates a horizontal layout for the mobile version of the filters.  
   Component createGrid(): Creates the main grid component using PaginatedGrid for displaying enrichment navigation records.  
   Renderer<EnrichmentNavigation> getExecuteStoredProcedureRenderer(): Defines a custom renderer for the "Execute SP" button column in the grid.  
   Renderer<EnrichmentNavigation> getStatusColumnRenderer(): Defines a custom renderer for the "Status" button column in the grid.  
   void refreshGrid(): Refreshes the grid by querying the enrichment service with the updated filters.
6. **Filters**:  
   static class Filters extends Div implements Specification<EnrichmentNavigation>: This represents the filter section of the view. It includes components such as ComboBox for client and enrichment category filters. The filter values are used to construct predicates for querying the enrichment navigation records.The toPredicate method is implemented to generate the filtering logic based on the selected filter values.
7. **EnrichmentNavigationDetailsRenderer**:  
   ComponentRenderer<EnrichmentNavigationDetailsFormLayout, EnrichmentNavigation> createEnrichmentNavigationDetailsRenderer():  
   Creates a custom renderer for displaying detailed information about an enrichment navigation record.  
   Uses EnrichmentNavigationDetailsFormLayout, a custom FormLayout to present the details of a specific record.
8. **EnrichmentNavigationDetailsFormLayout**:  
   static class EnrichmentNavigationDetailsFormLayout extends FormLayout: This represents a custom form layout for displaying detailed information about an enrichment navigation record.  
   It contains read-only text fields for displaying various attributes of the record, such as client, enrichment category, subcategories, etc.

*Overall, the provided code implements a Vaadin-based user interface for managing and displaying enrichment navigation records. It includes filter options, a grid view with custom renderers for executing stored procedures and displaying status details, and a detailed view of each record. The code demonstrates the usage of various Vaadin components and functionalities to create a responsive and interactive UI for managing enrichment navigation data.*

## Login View

The given code is an implementation of a login view in a Vaadin application. Let's break down the code and provide an overview of its functionality:

* The code is located in the package com.populationhealth.app.views.login.
* The class LoginView extends VerticalLayout and implements the BeforeEnterObserver interface, which allows it to intercept navigation events before entering the view.
* The LoginView is annotated with @Route(value = "login"), which indicates that this view is accessible through the URL path "/login".
* The LoginView is also annotated with @PageTitle("Login"), specifying the page title for this view.
* The @AnonymousAllowed annotation allows anonymous access to this view, meaning that users don't need to be authenticated to access it.
* The constructor of LoginView takes an AuthenticatedUser parameter, which presumably handles authentication and user-related operations.
* The LoginForm component is instantiated as loginForm.
* The loginForm action is set using the setAction method, specifying the route path for the current context and class.
* An LoginI18n object is created and customized to set the internationalization properties for the login form.
* A Header component is created, and an image resource is loaded from the "/images/healthmap-logo.png" path. The image is added to the header.
* A Footer component is created, and a Span containing copyright and usage information is added to it.
* The forgotPasswordButtonVisible property of the loginForm is set to true, enabling the "Forgot Password" button.
* The loginForm is added to the LoginView along with the header and footer components using the add method.
* The expand method is called on the loginForm to make it take up all available vertical space.
* The size of LoginView is set to full using setSizeFull.
* CSS class names are added to the LoginView using addClassNames to apply styling.
* The beforeEnter method is overridden from the BeforeEnterObserver interface to perform actions before entering the view.
* If the user is already logged in (checked by the authenticatedUser.get().isPresent() call), the user is forwarded to the main application page by calling event.forwardTo("").
* If there is an "error" parameter in the query parameters of the current location, indicating a login error, the setError method of the loginForm is called to display an error message.

*In summary, the LoginView class represents a login page in a Vaadin application. It provides a login form, handles authentication, and includes a header and footer with additional content. The view allows anonymous access and redirects authenticated users to the main application page.*

**MainLayout View**

This represents the main layout of a Vaadin application.

* The class MainLayout extends AppLayout, which is a layout component for creating the main application layout with responsive design.
* The constructor of MainLayout takes two parameters: authenticatedUser of type AuthenticatedUser and accessChecker of type AccessAnnotationChecker.
* In the constructor, the primary section of the layout is set to Section.DRAWER, indicating that the main content will be displayed in the main area, and the navigation drawer will be used for additional content.
* The addHeaderContent method is called to add the header content to the layout.
  + **Navigation**
    - A DrawerToggle component is created to toggle the visibility of the navigation drawer.
  + **Title**
    - An H2 component called viewTitle is instantiated.
  + **Header Elements**
    - A HorizontalLayout called layout is created to hold the header elements.
  + **Logo**
    - An image resource is loaded from the "/images/healthmap-logo.png" path and added to the layout.
  + **Application Name Header**
    - An H1 component with the application name is created and added to the layout.
  + **NavBar**
    - The toggle and layout components are added to the navigation bar using the addToNavbar method.
* The addDrawerContent method is called to add the content to the navigation drawer.
  + A Header component is created with an H1 component for the application name.
  + A Scroller component is created to hold the navigation items created in the createNavigation method.
  + The Header, Scroller, and footer created in the createFooter method are added to the navigation drawer using the addToDrawer method.
* The createNavigation method is called to create the navigation items for the application.
  + An AppNav component is created, which represents the navigation menu.
  + Navigation items are created based on the user's access permissions, and each item is added to the AppNav component.
* The createFooter method is called to create the footer of the layout.
  + If the user is authenticated (checked using authenticatedUser.get().isPresent()), an Avatar component and a MenuBar component are created for the user.
  + If the user is not authenticated, a login link is created using the Anchor component.
  + The user menu or login link is added to the footer.
* The afterNavigation method is overridden to update the viewTitle based on the current page title.
* The getCurrentPageTitle method retrieves the PageTitle annotation value from the current content and returns it as the current page title.

**Custom component Renderers**

#### getExecuteStoredProcedureRenderer

* The private method getExecuteStoredProcedureRenderer() returns a Renderer<EnrichmentNavigation> object. This renderer is used in Vaadin applications to render a specific component for each item in a grid or list.

In this case, the renderer is created using the ComponentRenderer class, which allows you to define a custom component for rendering each item in the grid or list. The constructor of ComponentRenderer takes two arguments: a component supplier and an item updater.

1. Component Supplier: Button::new
   1. The component supplier is a lambda expression that creates a new instance of a Button component for each item.
   2. Button::new is a method reference that represents the constructor of the Button class. It is used to create a new button component for each item.
2. Item Updater: (button, enrichmentNavigation) -> { ... }
   1. The item updater is a lambda expression that defines how the button component should be updated based on the item's data.
   2. It takes two parameters: the button component (button) and the current item (enrichmentNavigation).

Inside the item updater, the button component is customized using various methods:

* button.addThemeVariants(ButtonVariant.LUMO\_ICON, ButtonVariant.LUMO\_SUCCESS, ButtonVariant.LUMO\_TERTIARY);
  + This adds multiple theme variants to the button, giving it a specific visual style.
  + The theme variants used here are ButtonVariant.LUMO\_ICON, ButtonVariant.LUMO\_SUCCESS, and ButtonVariant.LUMO\_TERTIARY.
* button.addClickListener(e -> { ... });
  + This adds a click listener to the button. When the button is clicked, the specified code block inside the lambda expression is executed.
* button.setIcon(new Icon(VaadinIcon.PLAY));This sets the button's icon to a "PLAY" icon from the VaadinIcon enumeration.
* button.setText("Trigger");This sets the button's text to "Trigger".

Within the click listener, the following actions are performed:

* Long auditId = clientService.getAuditId();
  + This retrieves the audit ID by invoking the getAuditId() method on the clientService.
* enrichmentNavigation.setAuditId(auditId);
  + This sets the retrieved audit ID on the enrichmentNavigation object.
* clientService.triggerClientStoredProcedure(enrichmentNavigation.getClient(), enrichmentNavigation.getId());
  + This invokes the triggerClientStoredProcedure() method of the clientService, passing the client name and the ID of the current enrichmentNavigation object.
* Notification notification = Notification.show("Stored Procedure for " + enrichmentNavigation.getClient() + " triggered successfully.", 3000, Notification.Position.BOTTOM\_CENTER);
  + This displays a notification message indicating that the stored procedure was triggered successfully. The message includes the client name from the enrichmentNavigation object.
* notification.addThemeVariants(NotificationVariant.LUMO\_SUCCESS);
  + This adds a theme variant to the notification, giving it a success visual style.

*The Renderer<EnrichmentNavigation> object returned by this method can be used in a grid or list to render the items using the custom button component defined in the renderer.*

*GetStatusColumnRenderer*

The constructor of ComponentRenderer takes two arguments: a component supplier and an item updater.

Within the click listener, the following actions are performed:

* Long auditId = enrichmentNavigation.getAuditId();  
  It retrieves the audit ID from the current enrichmentNavigation object.
* if (Objects.isNull(auditId)) { ... }  
  It checks if the audit ID is null. If it is null, an error message is displayed in a confirmation dialog, and the method execution is returned.
* List<AuditInfo> auditResult = clientService.getAuditResult(auditId);  
  It invokes the getAuditResult() method of the clientService, passing the audit ID, to fetch the audit result.
* A Grid<AuditInfo> component is created and customized to display the audit result.
* An information icon is created and styled.
* A confirmation dialog is created and opened, displaying the grid with the audit result.
* If an exception occurs during the process, an error message is displayed in a confirmation dialog.

The Renderer<EnrichmentNavigation> object returned by this method can be used in a grid or list to render the items using the custom button component defined in the renderer. When the button is clicked, it displays the audit status in a confirmation dialog based on the audit ID retrieved from the current item.

In summary, the MainLayout class represents the main layout of a Vaadin application. It includes a responsive design with a navigation drawer, header, and footer. The navigation items in the drawer are dynamically created based on the user's access permissions (**This will allow us to restrict content and views based on different users)**. The layout also handles user authentication and displays the current page title.

## **Repository**

## EnrichmentNavigationRepository

1. The interface extends two Spring Data JPA interfaces: JpaRepository<EnrichmentNavigation, String> and JpaSpecificationExecutor<EnrichmentNavigation>. By extending JpaRepository, the interface inherits basic CRUD (Create, Read, Update, Delete) operations for the EnrichmentNavigation entity, while JpaSpecificationExecutor allows executing dynamic queries using specifications.
2. The repository interface defines two custom query methods using the @Query annotation:
   1. findDistinctByClient(): This method retrieves a list of distinct client values from the EnrichmentNavigation entity. The query selects distinct values from the client column in the EnrichmentNavigation table.
   2. findDistinctByEnrichmentCategory(): This method retrieves a list of distinct enrichment category values from the EnrichmentNavigation entity. The query selects distinct values from the enrichmentCategory column in the EnrichmentNavigation table.
3. Both methods return List<String>, indicating that they return a list of string values.

In summary, the EnrichmentNavigationRepository interface extends Spring Data JPA repositories and provides custom query methods to retrieve distinct values for the client and enrichmentCategory columns from the EnrichmentNavigation entity. This repository interface allows convenient access to data and provides additional flexibility for querying the EnrichmentNavigation table.

## ClientRepository

1. The interface is annotated with @Repository, indicating that it is a Spring repository component.
2. The repository interface extends two Spring Data JPA interfaces: JpaRepository<Client, String> and JpaSpecificationExecutor<Client>. By extending JpaRepository, the interface inherits basic CRUD (Create, Read, Update, Delete) operations for the Client entity, while JpaSpecificationExecutor allows executing dynamic queries using specifications.
3. The repository interface defines several methods:
   1. findByClientNameIgnoreCase(String clientName):
      1. This method is a custom query method that retrieves a Client entity by its clientName attribute. The query is case-insensitive, as indicated by the IgnoreCase keyword.
   2. callParameterizedSetInterventionClosingMedSandboxSP(String clientId, Integer enrichmentId):
      1. This method is annotated with @Procedure and calls a stored procedure named "Parameterized\_set\_intervention\_closing\_med\_sandbox". It passes the clientId and enrichmentId as parameters to the stored procedure.
   3. getAuditId():
      1. This method is annotated with @Procedure and calls a stored procedure named "GetAuditId". It returns a Long value.
   4. getAuditInfo(Long auditId):
      1. This method is annotated with @Procedure and calls a stored procedure named "getAuditInfo". It passes auditId as a parameter to the stored procedure and returns a List<List<Object>>. The outer list represents rows, and the inner list represents columns of the result set returned by the stored procedure.

In summary, the ClientRepository interface extends Spring Data JPA repositories and provides custom query methods and stored procedure invocations. It allows convenient access to data and provides flexibility for querying and executing stored procedures related to the Client entity.

## **Service**

## EnrichmentNavigationService

* The class is annotated with @Service, indicating that it's a Spring service component.
* The service has a dependency on the EnrichmentNavigationRepository interface, which is injected via constructor-based dependency injection.
* The constructor injection ensures that an instance of the repository is available for the service to interact with the underlying data source.

The service provides the following methods:

* get(String id):
  + Retrieves an EnrichmentNavigation entity from the repository based on the provided ID. It returns an Optional that may contain the entity if found.
* update(EnrichmentNavigation entity):
  + Saves or updates the given EnrichmentNavigation entity in the repository and returns the updated entity.
* delete(String id):
  + Deletes the EnrichmentNavigation entity with the specified ID from the repository.
* list(Pageable pageable):
  + Retrieves a page of EnrichmentNavigation entities from the repository based on the provided Pageable configuration.
* list(Pageable pageable, Specification<EnrichmentNavigation> filter):
  + Retrieves a page of EnrichmentNavigation entities from the repository based on the provided Pageable configuration and the given filter specification.
* count():
  + Returns the total number of EnrichmentNavigation entities in the repository.
* findDistinctByClient():
  + Retrieves a list of distinct client values from the repository.
* findDistinctByEnrichmentCategory():
  + Retrieves a list of distinct enrichment category values from the repository.

*Overall, this service class acts as a bridge between the repository and other components in the application, providing methods to access and manipulate EnrichmentNavigation entities. It encapsulates the business logic related to the EnrichmentNavigation domain and delegates the actual data access operations to the repository.*

## ClientService

* This class is annotated with @Service, indicating that it's a Spring service component.
* It also includes the @RequiredArgsConstructor annotation, which generates a constructor that injects dependencies based on the final fields in the class.
* The class utilizes the @Slf4j annotation, which sets up logging using the Simple Logging Facade for Java (SLF4J) framework.

This service has a dependency on the ClientRepository interface, which is injected via constructor-based dependency injection. The service provides the following methods:

* triggerClientStoredProcedure(String clientName, Integer enrichmentId):
  + This method triggers a stored procedure for a client.
  + It retrieves the Client entity with the specified clientName from the repository using the clientRepository.findByClientNameIgnoreCase() method.
  + If the client is not found, it logs an error and throws a RuntimeException.
  + If the enrichmentId is null, it logs an error and throws a RuntimeException.
  + Otherwise, it triggers the stored procedure using the clientRepository.callParameterizedSetInterventionClosingMedSandboxSP() method.
* getAuditId():
  + This method retrieves the audit ID using the clientRepository.getAuditId() method and returns it.
* getAuditResult(Long auditId):
  + This method retrieves audit information for the specified audit ID. It uses the clientRepository.getAuditInfo() method to obtain a list of lists of objects. It then converts each list of objects into an AuditInfo object and returns a list of AuditInfo objects.

The service class encapsulates the business logic related to clients and interacts with the ClientRepository for data access operations. It provides methods to trigger stored procedures, retrieve audit information, and handle the corresponding data.

**Roadblock : SQL Server Authentication – Last Few Weeks**

Understanding Windows Authentication from Linux – In-depth Analysis of Windows Authentication Challeges (what I was focused on my last few weeks)

The integration of Windows Authentication for MSSQL database connections from a Linux-based application presents a complex set of challenges that stem from the fundamental differences between the Windows and Linux operating systems. At the heart of these challenges is the fact that Windows Authentication, particularly when leveraging Active Directory, is deeply rooted in the Windows ecosystem. It employs proprietary protocols such as NTLM and Kerberos, which, while supported to varying degrees on Linux, are not as seamlessly integrated as they are within a pure Windows environment.

**The problem that I was trying to solve was deploying this application, which was built to use Windows Authentication, onto an Elastic Beanstalk Instance which was Linux-based. I had several conversations with the Devops Engineers here who instructed me to utilize Windows LDAP Authentication to work in Beanstalk. Hence began my research.**

My research began with an exploration of the theoretical underpinnings and practical applications of these authentication protocols in cross-platform contexts. This included a deep dive into the workings of Kerberos, a protocol that relies on a trusted third-party for secure authentication across a network. While Linux can support Kerberos, the configuration and maintenance required to facilitate this form of authentication with a Windows-based MSSQL server are non-trivial. It involves synchronizing time services, ensuring proper domain membership, and configuring service principals and keytab files. These tasks, while feasible, introduce a level of complexity that could potentially lead to vulnerabilities if not executed with precision. **Hence, my initial judgment was that this would be a long, arduous path if this is the road we want to go down.**

Moreover, my investigation into NTLM revealed a protocol less suited to modern security standards, raising concerns about its appropriateness for securing sensitive database connections, especially given the nature of the data we utilize at Healthmaps which should be kept extremely secure.   
  
  
  
Despite these challenges, I experimented with various drivers and libraries, including ODBC and JDBC, which offer some level of support for these authentication methods. However, the inconsistent experiences and the high overhead of ensuring secure and reliable connections underscored the need for a different approach.

At each step of the process, I experimented with the application and deployment into the AWS ecosystem. I was given a generic user/pass that should work, however, **these still employed the use of Windows Authentication. Since these were our constraints, I tried several different changes to the application to get it to work with the new authentication that was provided to me. Furthermore, I tested many of these iterations by deploying it in the cloud, spending time to check the logs, go through them, and try to figure out how to get the authentication to go through. You can check the AWS beanstalk deployments, there were several of them.**

**Sources used to code the Windows Authentication Portion:**

Primary:

<https://learn.microsoft.com/en-us/sql/connect/jdbc/setting-the-connection-properties?view=sql-server-2017>

Secondary:

<https://stackoverflow.com/questions/16497998/jdbc-connection-to-mssql-server-in-windows-authentication-mode>

Given these findings, it became evident that leveraging Windows Authentication from a Linux-based application to connect to an MSSQL database was filled with challenges. The technical complexity, potential security risks, and the administrative overhead involved in configuring and maintaining such a setup **led me to conclude that this approach was not viable for our project**. It necessitated a shift in strategy to ensure that our application could securely and reliably connect to the database without compromising on performance or security.

**Exploring MSSQL Connectivity Options: A Shift Towards SQL Server Authentication**

The realization that Windows Authentication posed significant challenges for our Linux-hosted Management UI prompted a shift in focus towards SQL Server Authentication. This method of database connection is universally supported across platforms and does not depend on the complex cross-platform authentication protocols required by Windows Authentication. There were several stackoverflow articles that lured me away from utilizing Windows Authentication towards SQL Server Authentication.

Sources relied upon:

https://stackoverflow.com/questions/37835929/connect-to-sql-server-with-windows-authentication-from-a-linux-machine-through-j

SQL Server Authentication works by requiring a username and password for database access. This method is inherently simpler and more straightforward to implement in a cross-platform environment, eliminating the need for Kerberos configuration, domain membership, and other complexities associated with Windows Authentication. My research into SQL Server Authentication highlighted its broad support across various drivers and libraries, making it a robust and secure choice for our application's database connectivity needs.

However, adopting SQL Server Authentication was not without its considerations. Security is paramount in any authentication strategy, and SQL Server Authentication requires careful management of credentials to prevent unauthorized access. This led me to explore best practices around credential storage, encryption, and access management, ensuring that our implementation would not compromise the security of our database connections.

The transition to SQL Server Authentication necessitated a thorough review and reconfiguration of our application's database connection logic. **The biggest roadblock, however, was the Devops team, who, time and time again, were steering me away from using SQL Server authentication, and instead urged me to utilize Windows Authentication.**

**What they did was they created windows user and provided the permissions (this took several days as well to obtain permissions).**

**The connection works only if the host is in a Windows domain. The only way for it to work for our application is if beanstalk joins the windows domain (which is very difficult and nobody chooses this option from what I researched). Therefore, our application on AWS Beanstalk cannot connect to the DB right now.**

**There were several discussions where they urged me to try out the credentials via a database management tool called Dbeaver. Their claim was, that, if Dbeaver can connect to the SQL Server Database using Windows Authentication, then my application should be able to as well in our AWS Environment.  
  
I spent a lot of time trying to configure Dbeaver. I utilized their instructions, I also utilized this guide here:**[**https://stackoverflow.com/a/55579900**](https://stackoverflow.com/a/55579900)

**But the way that Dbeaver works is that it is utilizing my local Active Directory credentials. I had proved this to the devops team by showing them that, even if I utilize incorrect credentials, Dbeaver is still able to connect to the SQL Server database because, again, it is utilizing my local LDAP connection.**

The final verdict was to transition to a SQL Server based approach. Thus, I began my research here. **Unfortunately, this is where my contract ended, so the application needs a SQL Server implementation.**

**Lastly – Vulnerability Patches**

Updated application to use latest Log4J (2.17.x) and removed any reference to commons-text, which were vulnerabilities that were detected by Healthmap Personnel and I was instructed to remove them.

Furthermore, vaading designer plugin of intellij was also showing up as a vulnerability, which our application does not require, hence it was removed.

Things to note:

* The current application will only work if it is on a VPN connection. It needs to access the two datasources disclosed throughout the document associated with credentials, which are:
* spring.datasource.jdbcUrl=jdbc:sqlserver://pilot-db.cdxx0j28ltxl.us-east-1.rds.amazonaws.com;databaseName=HMAPProtoType  
  spring.datasource.username=sMohammed  
  spring.datasource.password=Pass4Pil0t!  
  spring.datasource.driverClassName=com.microsoft.sqlserver.jdbc.SQLServerDriver  
  spring.second.datasource.jdbcUrl=jdbc:sqlserver://HMAP-CODS-D001.healthmapoffice.com;databaseName=SANDBOX\_INNOVATION;integratedSecurity=true  
  spring.second.datasource.driverClassName=com.microsoft.sqlserver.jdbc.SQLServerDriver
* Note that my username/pass may not work anymore as I am no longer a resource at Healthmaps.
* Note that the second datasource relies on LDAP authentication, so the windows machine must be connected to VPN in order to connect to the datasources needed
* If there’s any questions please reach out to me at [shafiq@techiesolutions.dev](mailto:shafiq@techiesolutions.dev)

Thank you and I wish you the best of luck,  
Shafiq Mohammed