Datative

Technical Guide

Camilla Boyle - 16732949
Emily McGivern - 16305506
Supervisor: Ray Walshe
Date of Completion: 16/05/2020

Contents

Abstract

Datative is a web-based analytics application. This application is a rich, powerful analytics tool which allows users to create interactive, personalised dashboards which can be exported as a PDF report. It provides the user numerous ways to intuitively visualise and represent their data. This application is designed to allow users to gain insights into their data without the need for a deep understanding of data analytics. A selection of components along with their underlying algorithms offer a way for anyone with data from any field to gain insights and deeper understanding of what their data represents. The application is easy to use and facilitates the quick design of creative dashboards which can then be exported as a PDF report to harness the power of the analytics outside of the application.

1. Introduction

1.1 Overview

Datative is a web application developed with the Spring MVC framework. To use the application, the user must register an account using a unique email address and password. Once the user is logged into the application, they have access to all of the application functionalities. To create a dashboard, the user must have a dataset they wish to analyse. The user can upload a new dataset to the application in the form of a .csv, .txt or .xlsx file. The dataset is completely cleaned and processed to ensure it is ready for analysis, and a table is created in the database containing all the data from the cleaned and processed dataset. This means that the user can perform analytics on the same dataset again and again until they decide to delete the dataset from their account. The creation of the dashboard allows the user full control over its design. Users can add a selection of components to the dashboard, choosing which data they would like to display in each component. They can also customise the component visually, changing colours and font style within the component.

Once the user is happy with their dashboard, they can export it as a PDF report. This report is downloaded to the users local machine, and a copy is also stored in their individual Cloud Storage Bucket. Users can access their previously generated PDF reports through their profile, meaning that they will never lose their analytics. Each user has an individual profile where they can view their previously uploaded datasets and previously generated PDF reports. The application was designed to be easy to use and intuitive to all users. Pages are uncluttered and emphasis was put on ensuring that the user understood the functionality of all page elements through their style and content. There is no prior knowledge of data analytics or creating dashboards expected. The user is free to experiment with different ways to display their data on their dashboard and different combinations of components.

1.2 Motivation

In recent years, we have entered the Age Of Analytics. The importance of data analytics is now being recognised and people are reaping the benefits. In modern industry, there is huge emphasis on using insights from data to make better business decisions. It can be extremely hard to notice trends and make informed decisions based on raw data. By visualising the results of analytics algorithms performed on data, individuals can spend less time deciphering their data, and more time improving their processes and making better decisions. The choice to build an application like this was down to our interest in data analytics and the power of insights into data. Datative was designed to be an easy to use solution to data analytics which offers quick and easy insights into data without the need for a connection to a data source or a deep understanding of data analysis. It serves to provide easy to understand insights which apply to many areas. The goal was to facilitate any industry, large or small, to analyse their data and be able to use the results of this analysis in a meaningful way. Possible uses of this application exist in enterprises with large volumes of sales, education and sport to name a few. As we had never developed an application using a Java framework, we were interested in using this as an opportunity to explore and learn Java application development. We also explored new technologies which we had not experienced yet, while ensuring that we were developing an application which is useful and well designed.

1.3 Research

There was significant research required throughout all stages of the development process. The research process involved analysis of many areas of the application functionality in order to make the best decision in terms of technologies and algorithms for implementation. The following are some of the areas of research which were conducted which had a significant impact on the application development.

1.3.1 Data Cleaning and Processing

To ensure that the analytics performed on the dataset produced reliable and accurate results, it was important that any dataset uploaded to the application was cleaned and processed. Data cleaning ensures that there is no inconsistent or incorrect data in the dataset which could hinder the analytics algorithms. To plan what algorithms would be necessary for data cleaning, an understanding of how data is cleaned and how incorrect or corrupt data is dealt with was required. It was important to ensure that the algorithms implemented would deal with inconsistent and incorrect data while also ensuring that the reliability of the dataset was not compromised. The research into the data cleaning process concluded that we would need to implement the following algorithms:

- Replace missing data of different types with data which would not skew the results of the analytics.
- Remove any columns which are empty.
- Remove any columns which are missing more than 30% of their data.

- Ensure all columns have the correct type based on their data.
- Remove duplicate rows from the table.

1.3.2 Data Storage

Data storage was an extremely important area of research for the development of the application. The database instance needed to be set up very early in the development process, which meant that decisions on data storage needed to be made early. As the user uploads their dataset in the form of a file to application, the implementation of a permanent data storage solution was important. The application also needed to facilitate the storage of file objects to act as a destination for file uploads, and also store the users generated PDF documents. Originally, Google Cloud Storage was considered as the only storage solution in the application. However, as our research progressed, it was clear that a MySQL database was better suited. The conversion of a dataset into a MySQL table offered a permanent table structure representing the dataset and facilitated querying the data with ease. To satisfy all data storage needs of the application, a cloud-based storage service was deemed to be the best solution. Google Cloud Platform was chosen as it offered Cloud SQL to store the created database tables and Cloud Storage to store the file objects. These solutions facilitated the storage of all necessary data within the application.

1.3.3 Dashboard Implementation

One of the defining features of the application is the customisable dashboard. The dashboard needed to offer movable components to ensure that the user had full control when creating their dashboard. This meant that research had to be performed in the area of drag-and-drop functionality and creating components which would be movable around the dashboard area. Gridstack.js was chosen to implement the dashboard area as it offered all the required functionalities needed for users to create their dashboards. It also offers a Bootstrap-friendly layout which integrated seamlessly into our front-end design. Gridstack offers drag-and-drop functionality and also the creation of resizable and draggable widgets which act as the dashboard components.

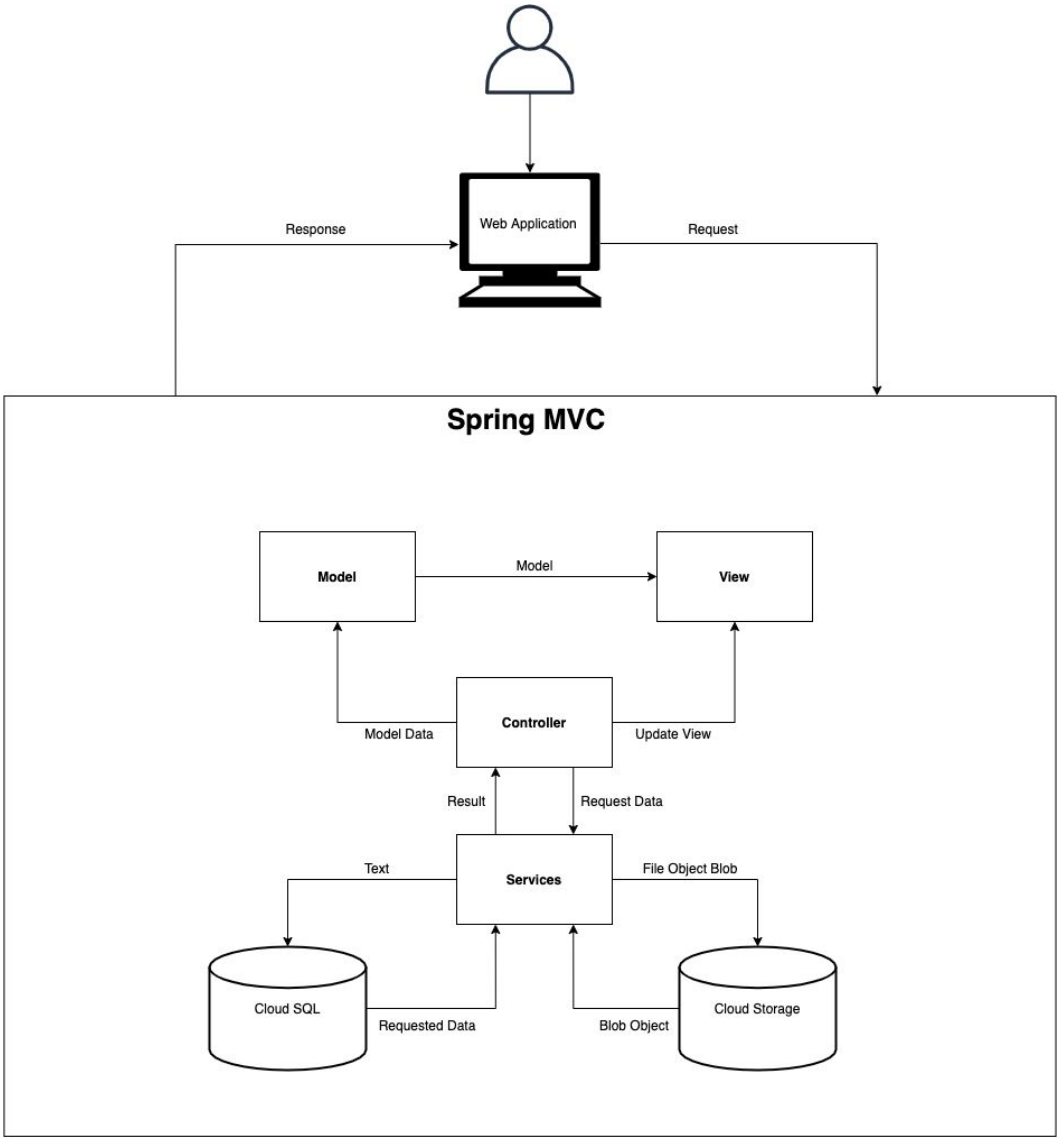
1.4 Glossary

1. **Spring MVC -** Java framework for building web applications which follows the Model-View-Controller design pattern.
2. **Spring Security -** Authentication and access-control framework which is standard for securing Spring-based Applications.
3. **Google Cloud Platform -** Suite of cloud computing services running on Google's infrastructure
4. **Google Cloud Storage -** Online, cloud-based file storage system for storing and accessing objects on the Google Cloud Platform.
5. **Google Cloud SQL -** A managed database service to manage, maintain and administer a MySQL database in the cloud.
6. **Google Cloud SQL Admin API -** REST API for administration of instances programmatically.

7. **Tablesaw -** Dataframe and visualisation library which can load, transform, filter and summarize data.
8. **JOOQ -** Java database-mapping library
9. **Gridstack.js -** JavaScript library to create responsive, draggable and resizable grids which are Bootstrap friendly.
10. **Bootstrap** - CSS framework for responsive front-end development
11. **Chart.js -** HTML5 javascript-based charts to create visualisations
12. **PivotTable.js -** JavaScript library for building Pivot Tables or Crosstabs in Javascript
13. **Spark SQL -** Spark module which provides an abstraction called DataFrames to facilitate data processing
14. **jsPDF -** A client-side library for generating PDF documents

2. Design

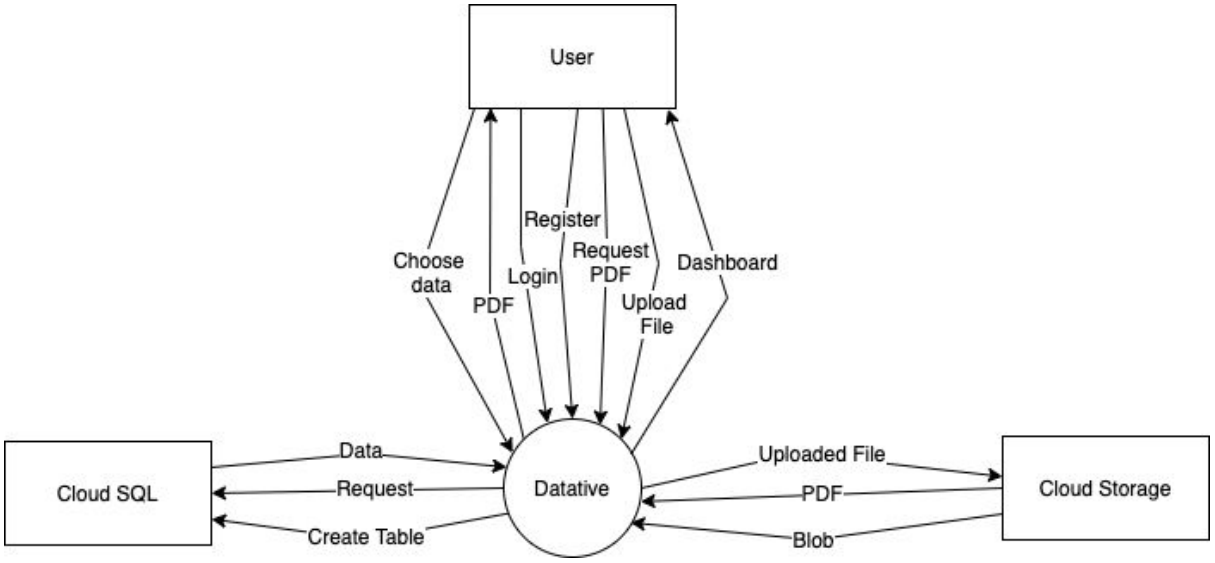
2.1 System Architecture Diagram



2.1.1 System Architecture Overview

- **Web Application -** Represents the front-end interface of the application. The user can interact with the application through the front-end interface.
- **Model -** Stores the application data which can be displayed in the HTML template.
- **View -** Renders the model data and generates the HTML output which appears in the browser
- **Controller -** Contains mapped methods and detects request mappings to handle mapping requests
- **Services -** Represents the Class files which contain business logic in the service layer of the application.
- **Cloud SQL -** Represents the database used to store data in the application
- **Cloud Storage -** Represents the cloud storage platform used to store file objects in the application

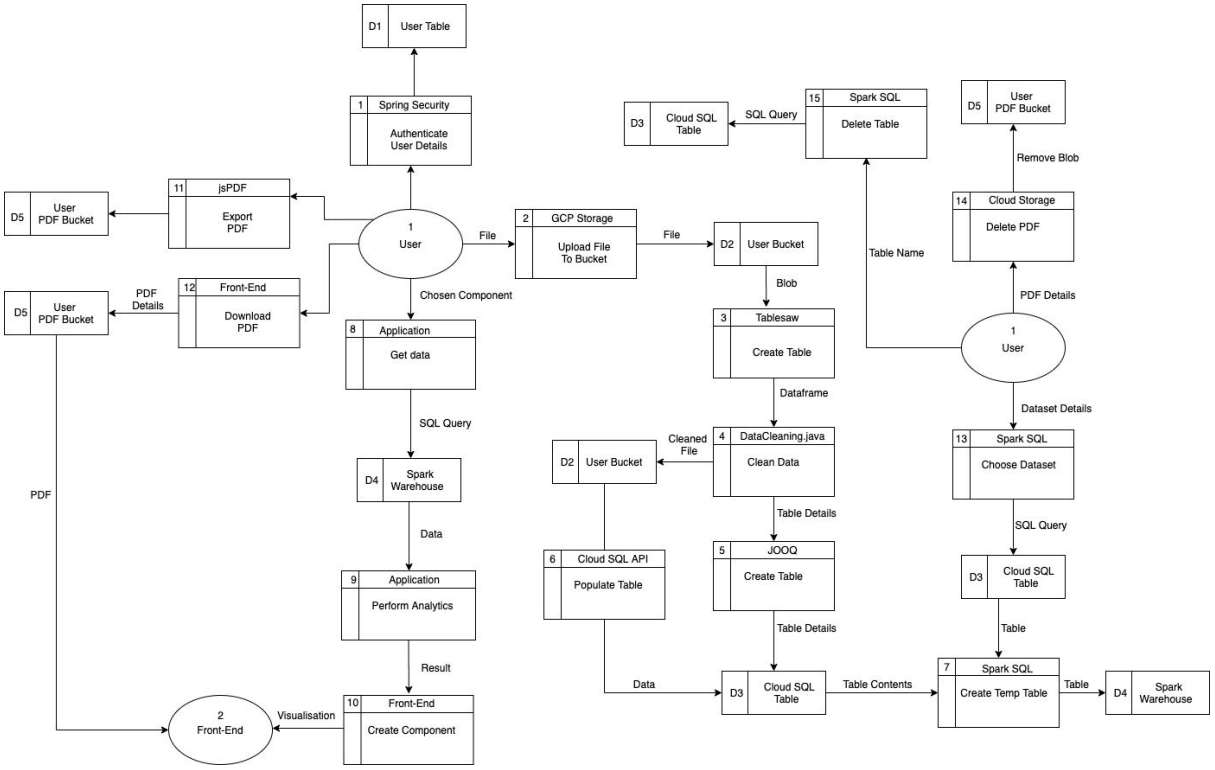
2.2 Context Diagram



2.2.1 Context Diagram Overview

User - Represents the person who is interacting with the application
Cloud SQL - Represents the Cloud SQL database which stores the data from the application
Cloud Storage - Represents the Cloud Storage instance which stores the file objects from the application.

2.3 Data Flow Diagram



3. Implementation

Datative was implemented using the Spring MVC framework. It is a flexible framework which implements all the features of the core Spring framework. It also implements the DispatcherServlet pattern which handles incoming HTTP Requests and dispatches them to the correct handler. This framework was suitable for the development of this application as it is robust and modular. The application is deployed on a Google App Engine flexible environment. This environment offers automatic scaling and load balancing. It also required no deployment configurations and server management. The application also uses instances of Google Cloud SQL and Google Cloud Storage, meaning that the application harnesses the power of Google Cloud Platform in many ways. The following section discusses implementation details of specific application features.

3.1 Registration and Login

The registration and login are implemented using Spring Security. To store the user details and user role details, User and Role entities were created with getter and setter methods for saving and fetching specific data for each entity. To secure the application, form-based authentication was implemented using Spring Security. The configuration for Spring Security for this application is located in the SecurityConfiguration configuration class. When a user registers with the application the details entered into the form are validated by the UserRegistrationDto, a data transfer object which is injected by the “user” model. The controller mapped to the “/registration” URL validates the form using the UserRegistrationDto and checks that the user is not already registered to the application. A field match validator is used to ensure that the two passwords entered by the user match. The new User object is saved to the user table in the UserServiceImpl class where the UserRegistrationDto object fields are added to a new User object and saved.

```
@Controller
@RequestMapping("/registration")
public class UserRegistrationController {

    @Autowired
    private UserService userService;

    @ModelAttribute("user")
    public UserRegistrationDto userRegistrationDto() {
        return new UserRegistrationDto();
    }

    @GetMapping
    public String showRegistrationForm() {
        return "registration";
    }

    @PostMapping
    public String registerUserAccount(@ModelAttribute("user") @Valid UserRegistrationDto userDto, BindingResult result){
        User existing = userService.findByEmail(userDto.getEmail());
        if (existing != null){
            result.rejectValue("email", null, "There is already an account registered with that email");
        }

        if (result.hasErrors()){
            return "registration";
        }

        userService.save(userDto);
        return "redirect:/registration?success";
    }

}
```

Registration Controller

```
public User save(UserRegistrationDto registration){
    User user = new User();
    user.setFirstName(registration.getFirstName());
    user.setLastName(registration.getLastName());
    user.setEmail(registration.getEmail());
    user.setPassword(passwordEncoder.encode(registration.getPassword()));
    user.setRoles(Arrays.asList(new Role("ROLE_USER")));
    return userRepository.save(user);
}
```

Saving new User object

A Spring JPA Repository was also implemented as a data access layer for a user's details. The following code shows the implementation of a CRUD function to find a user by their email.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.example.datative.security.model.User;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmail(String email);
}
```

Find User object by email

3.2 Homepage

The application homepage was built using Bootstrap. The current authenticated users email is displayed on the homepage using the sec:authentication attribute, a Thymeleaf attribute which prints the authenticated users username.

```
<div class="welcome-container">
    <div id="welcome">
        <h3>Welcome <span sec:authentication="principal.username"></span>!</h3>
    </div>
</div>
```

***sec:authentication* attribute**

The homepage layout is extended from the base.html template. This template defines the basic structure for the majority of the application pages and includes the top navigation bar and the placement of the main page content. The individual page content is inserted as fragments into the base.html layout. This was done to prevent repetition of common template layouts throughout the application.

3.3 Uploading a dataset

To add a dataset to the application, the user uploads a file in the format .csv, .txt or .xlsx. The file upload is implemented using the Spring MultipartFile interface and facilitated by the form on the upload page. The form has an input tag with the type set to “file” to set the input to a file upload. The form also contains an input tag which accepts text. The user enters their unique table name in this input box. Once the user has added their file to the form and entered their table name, they can submit the form. The controller mapped to “/uploadFile” receives the file as a MultipartFile object variable and the table name which is attached to the request. Before the input is processed, it is checked to see if it is valid. The file is checked to see if it is empty and if it is, the user is redirected back to the upload page where an alert displaying an error message is shown. The method checkForTable() in the FileService class is called to check if the table name exists. If the table name already exists in the database, the user is redirected back to the upload page where an alert displaying an error message is shown. If the table name is unique for that user's tables and the file is not empty, the file is then passed into the uploadFile() method in the FileService class to be processed.

```
@GetMapping("/upload")
public String upload() {
    return "upload";
}

@PostMapping("/uploadFile")
public String uploadFile(@RequestParam("file") MultipartFile file, RedirectAttributes redirs, HttpServletRequest request) throws SQLException {
    if (file.isEmpty()) { //Validating file to ensure it is not empty
        redirs.addFlashAttribute("message", "File is empty, please select another file to upload");
        return "redirect:upload";
    }

    String[] name = request.getParameterValues("tableName");

    //Checking if table name exists already in DB
    ResultSet results = fileService.checkForTable(name[0]);

    if (results.next() == true) {
        redirs.addFlashAttribute("message", "Table with this name already exists, please choose another!");
        return "redirect:upload";
    }

    String tableName = fileService.uploadFile(file, name[0]);


    List<Map<String, String>> columnNames = dashboardService.getDataSet(tableName);
    redirs.addFlashAttribute("columnNames", columnNames);

    return "redirect:/dashboard";
}
```

FileController Controller Class with Controller Methods

In the uploadFile() method, the file extension is checked to ensure the file is the correct type to be processed. If the file is not the correct type, then a custom FileTypeException is thrown to let the user know their file could not be uploaded. As other file types cannot be processed, it was important to have client and server-side validation of the file extensions. This check also returns an error message to the user specificing what file type is not accepted so that there is no confusion around why the error has been thrown. If a file is accepted, then the file is processed further. It is uploaded to the users unique Cloud Storage Bucket, where it is converted into an InputStream streaming object and used to created a Tablesaw table object. This is so the dataset can be cleaned and processed in-place. The data cleaning process is discussed further in Section 3.4. After the table object has been cleaned, it is then converted back to a file object and uploaded to the users unique Cloud Storage Bucket. The

Cloud SQL Admin API executes a request which populates the Cloud SQL database table with the data from the cleaned file.

```
public String uploadFile(MultipartFile file, String userTableName) {
    List<String> acceptedExtensions = Arrays.asList("csv", "xls", "xlsx", "txt");
    String extension = FilenameUtils.getExtension(file.getOriginalFilename());

    if (!acceptedExtensions.contains(extension)) {
        throw new FileTypeException("Cannot upload file of type ." + extension + ", please try again!");
    }

    try {
         //Uploading file and creating stream
         String originalFileName = file.getOriginalFilename();
         Path copyLocation = Paths.get(uploadDir + File.separator + StringUtils.cleanPath(originalFileName));
         Files.copy(file.getInputStream(), copyLocation, StandardCopyOption.REPLACE_EXISTING);
         InputStream fileInputStream = file.getInputStream();
         String tableName = addFileToBucket(userTableName, originalFileName, fileInputStream);

         return tableName;


     } catch (Exception e) {
         e.printStackTrace();
         throw new FileStorageException("Could not store file " + file.getOriginalFilename()
             + ". Please try again!");
     }

}
```

uploadFile() method which accepted file with correct extension for processing

```
@ControllerAdvice
public class AppExceptionHandler {

    @ExceptionHandler(FileStorageException.class)
    public String handleException(FileStorageException exception, RedirectAttributes redirectAttributes) {

        String message = exception.getMsg();
        redirectAttributes.addFlashAttribute("message", message);
        return "redirect:upload"; //Redirecting back to upload and flashing failure message
    }

    @ExceptionHandler(FileTypeException.class)
    public String handleFileTypeException(FileTypeException exception, RedirectAttributes redirectAttributes) {

        String message = exception.getMsg();
        redirectAttributes.addFlashAttribute("message", message);
        return "redirect:upload"; //Redirecting back to upload and flashing failure message
    }


}
```

Custom Exception Handling for Uploads and File Processing

3.4 Data Cleaning

The data cleaning is performed by a number of data cleaning algorithms. These algorithms are located in the DataCleaning class. The following section discusses these algorithms and their effects on the dataset.

Missing data must be handled to ensure that the results of the analytics are not affected. If a column is entirely empty, it is removed. If a column is missing 30% or more of its data it is removed as it is presumed it will have a negative effect on the algorithm results. If the column is missing less than 30% of its data, then the missing values are replaced. If an Integer or Double column value is missing, it is replaced with the average value of that column. If a String column is missing values, the values are replaced with the String “None”. However, if the “None” values make up 30% or more of the column, then it is removed. The following code snippets demonstrate these algorithms.

```
public static void manageMissingData(Table dataSet) {
    checkColumnTypes(dataSet);
    List<String> columnNames = dataSet.columnNames();
    for (int i = 0; i < columnNames.size(); i++) {
        String currName = columnNames.get(i);
        Column<?> curr = dataSet.column(currName);
        int missing = curr.countMissing();
        int colSize = curr.size();
        double percentageColSize = colSize * 0.50;

        if (colSize == missing) {
            dataSet.removeColumns(curr);   //If the column is empty, it is removed
        }
        else if (missing >= percentageColSize) {
            dataSet.removeColumns(curr);
        }

        else {
            replaceMissingValues(curr, currName, dataSet, colSize);
        }
    }

}
```

```
@SuppressWarnings({ "rawtypes", "unchecked" })
public static void replaceMissingValues(Column curr, String currName, Table dataSet, int colSize) {
    double percentageColSize = colSize * 0.30;
    ColumnType colType = curr.type();
    String cString  = colType.toString();

    if (cString == "INTEGER") {
        IntColumn c = dataSet.intColumn(currName);
        int avg = (int) Math.round(c.mean());
        curr.setMissingTo(avg);
    }

    else if (cString == "STRING") {
        curr.setMissingTo("None");
        StringColumn strCol = dataSet.stringColumn(currName);
        int numEmpty = strCol.countOccurrences("None");
        if (numEmpty >= percentageColSize) {
            dataSet.removeColumns(currName);
        }
    }

    else if (cString == "DOUBLE") {
        DoubleColumn c = dataSet.doubleColumn(currName);
        double avg = Math.round(c.mean());
        curr.set(curr.isMissing(), avg);

    }
}
```

Rows which are duplicates of each other are also removed. This is performed using the dropDuplicateRows() function in the Tablesaw library. The following code snippet demonstrates this.

```
public static Table removeDuplicateRows(Table dataSet) {
    Table result = dataSet.dropDuplicateRows();
    return result;
}
```

There is also an algorithm checkColumnTypes() which checks the content of the column to ensure the column is assigned the correct type. It does this by checking the class type of the

column values and using regular expressions to establish if the value matches the pattern specified. For example, a regular expression is used to evaluate if a string is a date. If it is a date, a date count is incremented. If this count is greater than the count of strings which are not in a date format, the column is assigned a LocalDate type and the values are converted to LocalDate objects. This ensures that all assigned column types are correct based on the column content.

3.5 Profile

The profile page allows the authenticated user to view their previously uploaded datasets and previously generated pdf reports. The use of Bootstrap allowed for the implementation of tabs which allows the user to seamlessly switch between the list of their datasets and list of their PDF documents. The following section discusses the functionalities associated with the Datasets tab and the PDFs tab.

3.5.1 Datasets tab

This tab allows a user to see all their previously uploaded datasets. Each dataset has a corresponding table in the database. The name of the file and the table name is displayed so that they can be easily identified. For each dataset entry, there are two buttons: “Select” and “Delete”. The “Select” button allows the user to create a new dashboard using this table. When the user clicks this button, the table name is posted to the controller method mapped to “/dashboard” which accepts the post request with the parameter “select” attached. The following code snippet shows this controller method.

```
@RequestMapping(value = "/dashboard", method = RequestMethod.POST, params = {"select"})
public String dashboard(@RequestParam(name = "select") String tableName, Model model) {
     List<Map<String, String>> columnNames = dashboardService.getDataSet(tableName);
     model.addAttribute("columnNames", columnNames);
     return "dashboard";
}
```

The table name is passed to the method getDataSet() in the DashboardService class. This method creates a new SparkSession object which facilitates a connection to the application’s Cloud SQL instance. The corresponding table is retrieved from the database through the created SparkSession connection, and a dataframe is created representing the table. This dataframe is temporarily stored in a warehouse directory within the application. This method returns a Java Map object containing each column name mapped to its type. This Map object is added to the model and the dashboard view is displayed. The following code snippet shows this method.

```
public List<Map<String, String>> getDataSet(String tableName) {
    SparkSession spark = getSparkSession();
    String tableOption = "datative." + tableName;
    Map<String, String> options = new HashMap<>(); //Creating HashMap of options to be passed to the spark session to build the DataFrame
    options.put("url", "jdbc:mysql:///datative?cloudSqlInstance=datative:europe-west1:datative&socketFactory=com.google.cloud.sql.mysql.SocketFactory
    options.put("dbtable", tableOption);
    Dataset<Row> df = spark.read().format("jdbc").options(options).load().cache(); //Creating dataframe from current table
    df.createOrReplaceTempView("tempTable"); //Creates a new temporary view using a dataframe
    Tuple2<String, String>[] columnTypes = df.dtypes(); //Creating tuple of column name and column type to be added to model
    List<Map<String, String>> columnTypePairs = new ArrayList<Map<String, String>>();
    Map<String, String> map1 = new HashMap<String, String>();

    //Adding tuple items to a map which will be added to a list
    for (Tuple2<String, String> item : columnTypes) {
        map1.put(item._1(), item._2());
    }

    columnTypePairs.add(map1);

    return columnTypePairs;

}
```

The “Delete” button allows the user to delete the dataset from the application. When the user clicks this button, the table name is posted to the controller method mapped to “/dashboard” which accepts post request with the parameter “delete” attached. The following code snippet shows this controller method. This controller method redirects the user back to their profile page after the table has been deleted.

```
@RequestMapping(value = "/dashboard", method = RequestMethod.POST, params = {"delete"})
public String delete(@RequestParam(name = "delete") String tableName) throws SQLException {
    dashboardService.deleteTable(tableName);
    return "redirect:profile";
}
```

The table name is passed to the method deleteTable() in the DashboardService class. This method creates a new Connection object which allows the execution of SQL statements against the Cloud SQL instance. Statements for deleting the table and the corresponding entry in the user_file table are constructed and executed in a batch. This removes the table and any record of it from the application and cannot be undone. The following code snippet shows the implementation of this method.

```
public void deleteTable(String tableName) throws SQLException {
    Connection con = FileService.createDbConnection();
    Statement statement = con.createStatement();
    statement.addBatch("drop table " + tableName);
    statement.addBatch("DELETE FROM user_file WHERE table_name = '" + tableName + "';");
    statement.executeBatch();
}
```

3.5.2 PDFs tab

This tab allows a user to see all their previously generated PDF reports. The name of the PDF and the date it was generated is displayed. This makes it easier for users to recognise what a PDF document is and also ensures a unique name for each due to the inclusion of a LocalDateTime object in the PDF name. For each PDF entry, there are two buttons: “Download” and “Delete”. The “Download” button allows the user to download the PDF report to their machine's local storage. This is done using the download attribute specified in the download button hyperlink. The thymeleaf href attribute contains the media link URL to each PDF object in the users Cloud Storage Bucket. When the button is clicked, the PDF is downloaded from this media link.

```
<form method="POST" th:action="@{/deletePdf}">
    <th:block th:each="entry : ${pdfs}">
        <th:block th:with="pdfDetails=${#strings.arraySplit(entry, ',')}">
            <div class="list-group-item">
                <div class="d-flex">
                    <div class="p-2 bd-highlight">
                        <span class="d-block" th:text="${pdfDetails[0]}"></span>
                    </div>
                    <div class="ml-auto">
                        <a class="btn btn-outline-success btn-sm download-button" th:href="${pdfDetails[1]}" download="pdf-doc">Download</a>
                        <button type="submit" id="dataset-submit" class="btn btn-outline-danger btn-sm" name="delete" th:value="${entry}">Delete</button>
                    </div>
                </div>
            </div>
            <hr />
        </th:block>
    </th:block>
</form>
```

The “Delete” button allows the user to delete a PDF file from the application. When the user clicks the delete button, the PDF file name is posted to the controller method mapped to “/deletePdf” which accepts post request with the parameter “delete” attached. The following code snippet shows this controller method. This controller method redirects the user back to their profile page after the PDF file has been deleted.

```
@RequestMapping(value = "/deletePdf", method = RequestMethod.POST, params = {"delete"})
public String deletePdf(@RequestParam(name = "delete") String pdfName) {
    dashboardService.deletePdf(pdfName);
    return "redirect:profile";
}
```

The PDF file name is passed to the method deletePdf() in the DashboardService class. This method calls the getUserBucket() method in the DashboardService class to get the user's unique bucket name. It then creates a Storage object for the Cloud Storage instance and executes a delete operation on this object, removing the file that has the specified file name from the users bucket. The following code snippet shows this method.

```
public void deletePdf(String pdfName) {
    String pdfBucket = getUserBucket();
    String[] pdfStr = pdfName.split(","); //Getting name of file without media link
    Storage storage = StorageOptions.getDefaultInstance().getService();
    storage.delete(pdfBucket, pdfStr[0]);
}
```

```
public List<String> getUserPdfs() {
    String pdfBucket = getUserBucket();
    Storage storage = StorageOptions.getDefaultInstance().getService();
    Bucket userBucket = storage.get(pdfBucket);
    Page<Blob> blobs = userBucket.list();
    List<String> userPdfs = new ArrayList<String>(); //List of user pdf names and links
    for (Blob blob : blobs.iterateAll()) {
        userPdfs.add(blob.getName() + "," + blob.getMediaLink()); //Creating string which contains blob name and blob media link to download pdf client side
    }

    return userPdfs;
}
```

3.6 Dashboard

When a user has uploaded a new file, or chosen a file from their profile page, they are then brought to the dashboard page. Here, they can create a custom dashboard with components displaying results of analytics on their data. The dashboard.html page is built using Bootstrap. The dashboard area is a movable grid which is implemented using Gridstack.js.

With Gridstack, components are added to the dashboard in the form of movable, responsive and resizable widgets. These widgets are highly customisable and allow for enabling and disabling resize handles, changing size, and removal from the grid area. The user can drag their component anywhere in the dashboard area, but will be alerted if there is no room left on the dashboard for any more components. The following code snippet shows the creation of a Gridstack grid where the components will be added. The grid is initialised to accept new widgets which will be the dashboard components, always show resize handles on widgets which have them enabled and allow widgets to be placed anywhere on the dashboard through the float property being enabled.

```
var grid = GridStack.init({
    alwaysShowResizeHandle: /Android|webOS|iPhone|iPad|iPod|BlackBerry|IEMobile|Opera Mini/i.test(
        navigator.userAgent
    ),
    disableOneColumnMode: true,
    maxRow:14,
    float: true,
    /* Enable resize handles on edges  */
    resizable: {
        handles: 'e, se, s, sw, w'
    },
    removeTimeout: 100,
    acceptWidgets: '.newWidget'
});
```

The Gridstack grid is appended to a div in the body of the dashboard template. The following code snippet shows the div. The Gridstack grid is always appended to a div of the class “grid-stack”.

```
<div class="col-md-8">
  <div class="grid-stack" data-gs-animate="yes"></div>
</div>
```

The user can choose the name of their dashboard. This name is the name applied to the PDF report generated of the dashboard. It cannot contain any newlines. To prevent the user from entering newlines, an event handler for the dashboard name div is bound to the “keypress” JavaScript event. When a key is pressed, the event is checked to see if it is the enter key for the insertion of a newline. If it is not that key the event is fired but if it is that key it is not fired and the insertion of the newline is prevented.

```
$("#dashboardName").keypress(function(event){
    return event.which != 13; //Adding handler for newline event to prevent user adding new line to dashboard name
});
```

When a file is uploaded by the user and processed, the uploadFile() controller method in the FileController controller class redirects the user to the dashboard page where they can begin creating their dashboard. As the user has uploaded the dataset at that moment, the table is automatically loaded from the database for them to begin building their dashboard. The following code snippet shows how the redirect occurs.

```
@PostMapping("/uploadFile")
public String uploadFile(@RequestParam("file") MultipartFile file, RedirectAttributes redirs, HttpServletReq
    if (file.isEmpty()) { //Validating file to ensure it is not empty
        redirs.addFlashAttribute("message", "File is empty, please select another file to upload");
        return "redirect:upload";
    }

    String[] name = request.getParameterValues("tableName");

    //Checking if table name exists already in DB
    ResultSet results = fileService.checkForTable(name[0]);

    if (results.next() == true) {
        redirs.addFlashAttribute("message", "Table with this name already exists, please choose another!");
        return "redirect:upload";
    }

    String tableName = fileService.uploadFile(file, name[0]);


    List<Map<String, String>> columnNames = dashboardService.getDataSet(tableName);
    redirs.addFlashAttribute("columnNames", columnNames);

    return "redirect:/dashboard";
}
```

If the user chooses a dataset to create a dashboard with that has previously been uploaded, then the table is fetched from the database based on the table name of the chosen dataset. This is discussed in Section 3.5.1.

3.7 Adding a component

To add a component, the user chooses a component from the dashboard sidebar. A modal opens which contains all the details needed for creating the component. The following section discusses the implementation of the various dashboard components.

3.7.1 KPI Tiles

The KPI Tile component allows the user to create a tile visualisation of chosen columns in their data. The user can choose from 3 different types of KPI Tile to display on their dashboard:

1. Numeric Tile

This tile displays the result of a chosen algorithm performed on a column chosen by the user. It is named the Numeric Tile as the results of the algorithm are numeric type. The three algorithms which the user can choose from are total, average and median. The total algorithm displays a sum of the data in the column. The average algorithm displays the average of the data in the column. The median algorithm displays the median of the data in the column. When the user chooses the KPI Tile component, a modal appears for them to make their selections of algorithm and enter other details for the tile. Once the user has entered the details in the form and made their selection, they click the submit button. The chosen details are posted by ajax request to the related component controller. The controller passes the parameters into the algorithm in the service file, and the result is returned as a response to the ajax request. The response is passed into a javascript function which creates the KPI Tile to be displayed and renders it on the dashboard.

```
function sendKpiForm(e) {
    e.preventDefault();
    var token = $("meta[name='_csrf']").attr("content"); // Specifying CSRF tokens
    var kpiType = $("#kpiType").val();
    var target = $("#kpiTarget").val();
    var title = $("#kpiTitle").val();
    var tester = $("#tester").text();

    //  Ajax request to post to component controller
    //  Choosing where to post data based on users choice of KPI tile

    var params = {
            type: 'post',
            headers: {"X-CSRF-TOKEN": token},
            success: function(response) {
                if (kpiType === 'num') {
                    addNewKpiNumTile(response, target, title); //Passing result of post request to function
                }
                else if (kpiType === 'trend') {
                    addNewKpiTrendTile(response, title); //Passing result of post request to function
                }
                else if (kpiType === 'target') {
                    addNewKpiTargetTile(response, target, title);
                }
            },
            data: $("#kpiForm").serializeArray()
        };

        if (kpiType == 'num' || kpiType == 'target') {
            params.url ='/kpiNumTarget';
        }

        else if (kpiType == 'trend') {
            params.url ='/kpiTrend';
        }

    $.ajax(params);
    document.getElementById('kpiForm').reset(); //Resetting form so that another tile can be added

    //Hiding divs after form has been submitted so it is reset to original view
    $('#choose-column').hide();
    $('#choose-column-trend-date').hide();
    $('#kpi-op').hide();
    $('#choose-target').hide();
    $('#choose-name').hide();

}
```

AJAX Request for KPI Numeric Tile

```
function addNewKpiNumTile(result, target, title) {

    //Creates a widget of cube shape
    var node = {width: 2, height: 3, autoPosition: true, noResize: true};
    var color;
    var tileId = Math.random();

    /*
    The following checks if the user has entered a target value, and if
    the total value is greater than or less than the target value to
    determine the color of the integer on the tile
    */

    if (target != "") {
        if (result < target) {
            color = 'red';
        }

        else {
            color = 'green';
        }
    }
    else {
        color = 'black';
    }

    if (grid.willItFit(null, null, 2, 3, true)) {
        grid.addWidget('<div id=tile' + tileId + '><div class="grid-stack-item-content"><div class="tileHeader"><div class="widgetTitle">'
    }
    else {
      alert('Not enough free space to place the widget');
    }


};
```

Creating a KPI Numeric Tile Widget

```
public Double getKpiIntegerResult(Map<String, String> params) {
    SparkSession spark = DashboardService.getSparkSession();
    Dataset<Row> columnContent = spark.sql("SELECT `" + params.get("columnName") + "` FROM tempTable"); //Select chosen column from table
    List<Row> columnList = columnContent.collectAsList(); //Collecting rows from previous query into a list
    double result = performNumOps(columnList, params.get("kpiOp"));
    return result;
}


public Double performNumOps(List<Row> columnList, String opName) {
    double result = 0;
    Double[] numsList = new Double[columnList.size()]; //Creating an array with size of length of list of columns

    //Iterating through list of rows and adding first value to integer list so that operations can be performed on list
    for (int i = 0; i < columnList.size(); i++) {
        Row item = columnList.get(i); //Get corresponding row
        if (item.get(0) instanceof Integer) {
            int v = (int)item.get(0);
            double doubleV = v;
            numsList[i] = doubleV;
        }
        else {
            double v = (double)item.get(0);
            numsList[i] = v;
        }
    }

    //User selects total operation to be performed
    if (opName.equals("total")) {
        double sum = 0;
        for (int i = 0; i < numsList.length; i++)
            sum += numsList[i];
        result = sum;
    }

    //User selects average operation to be performed
    else if (opName.equals("average")) {
        if (numsList.length == 0) {
            result = 0;
        }
        else {
            double total = 0;
            for (int i = 0; i < numsList.length; i++)
                total += numsList[i];
                double average = total / numsList.length;
            result = average;
        }

    }

    //User selects median operation to be performed
    else if (opName.equals("med")) {
        double med = 0;
        Arrays.sort(numsList);

        if (numsList.length % 2 == 0)
            med = (numsList[(numsList.length / 2) - 1] + numsList[numsList.length / 2]) / 2;
        else
            med = numsList[(numsList.length -1) / 2];

        result = med;
    }

    return Math.round(result * 100.0) / 100.0; //Rounding results to 2 decimal places

}
```

KPI Numeric Tile Algorithms

2. Trend Tile

This tile allows users to pick a numeric column and a date column. The numeric column results are then displayed in a time-series mini-chart in accordance with their corresponding value in the date column. This allows users to see a representation of their data over time in a small, but meaningful way.

```
@SuppressWarnings("rawtypes")
public Map<String, List> getDateTotals(List<Row> columnList) {
    Map<Date, List<Row>> groupByDate = columnList.stream().collect(Collectors.groupingBy(i -> (Date) i.get(1)));
    Map<Date, List<Row>> sortedDateMap = new TreeMap<>(groupByDate);

    List<Date> dateLabels = new ArrayList<Date>(sortedDateMap.keySet()); //Getting each individual date
    List<Double> dateSums = new ArrayList<Double>();

    //Getting sum of values for each date to show trend over time
    for (Entry<Date, List<Row>> entry : sortedDateMap.entrySet()) {
        List<Row> dateEntries = entry.getValue();
        double total = 0;
        for (Row item : dateEntries) {
            if (item.get(0) instanceof Integer) {
                int v = (int) item.get(0);
                double numVal = v;
                total += numVal;
            }
            else {
                double numVal = (double)item.get(0);
                total += numVal;
            }
        }

        dateSums.add(Math.round(total * 100.0) / 100.0);
    }


    Map<String, List> listMap = new HashMap<String, List>();
    listMap.put("dates", dateLabels);
    listMap.put("dateSums", dateSums);

    return listMap;
}
```

3. Target Tile

This tile offers the same algorithms as the KPI Numeric Tile, but displays the results in a different way. The results are displayed in a small donut chart, allowing the user to visualise how close they are to hitting their target.

3.7.2 Text Box

The text box component allows users to add a text box to their dashboard which they can add their own text to. It can be customised like all the other components. The following code snippet shows how the text box is implemented. The fit_content() function ensures that the textarea resizes automatically to the text input depending on the number of lines the user adds.

```
/* Add new text box widget when user clicks button */
$('#add-new-text-box').click(function() {
        var node = {width: 4, height: 1, autoPosition: true};
        var textId = Math.random();
        if (grid.willItFit(null, null, 4, 1, true)) {
            grid.addWidget('<div id=textbox' + textId + '><div class='
        }
        else {
            $("#dialog").dialog("open");
        }
    }
);
```

Creating a new textbox

3.7.3 Crosstab

The crosstab component allows the user to display their data in a table format, showing the relationship between two columns. There are two types of result which can be shown in the crosstab - count and sum. The count operation counts the number of occurrences of a row within a column. The sum operation sums the row values for each column. The algorithm performed depends on the choice of operation. The crosstab front-end code is implemented with the JavaScript library PivotTable.js.

```
public ArrayList<HashMap<String, Object>> createCrosstab(Map<String, String> requestParams) {
    SparkSession spark = DashboardService.getSparkSession();
    if (requestParams.get("opName").equals("count")) {
        Dataset<Row> columns = spark.sql("SELECT `" + requestParams.get("columnName") + "`, `" + requestParams.get("rowName") + "` FROM tempTable");
        List<Row> columnList = columns.collectAsList();
        ArrayList<HashMap<String, Object>> crosstabCount = new ArrayList<HashMap<String, Object>>();
        for (Row item : columnList) {
            HashMap<String, Object> valuesMap = new HashMap<String, Object>(); //Creating new HashMap which will contain each set of row values
            valuesMap.put(requestParams.get("columnName"), item.get(0));
            valuesMap.put(requestParams.get("rowName"), item.get(1));
            crosstabCount.add(valuesMap); //Adding HashMap of values to ArrayList
        }

        return crosstabCount;
    }

    else {
        Dataset<Row> columns = spark.sql("SELECT `" + requestParams.get("columnName") + "`, `" + requestParams.get("rowName") + "`, `" + requestParams.get("aggregateName")
        List<Row> columnList = columns.collectAsList();
        ArrayList<HashMap<String, Object>> crosstabSum = new ArrayList<HashMap<String, Object>>();
        for (Row item : columnList) {
            HashMap<String, Object> valuesMap = new HashMap<String, Object>(); //Creating new HashMap which will contain each set of row values
            valuesMap.put(requestParams.get("columnName"), item.get(0));
            valuesMap.put(requestParams.get("rowName"), item.get(1));
            valuesMap.put("value", item.get(2));
            crosstabSum.add(valuesMap); //Adding HashMap of values to ArrayList
        }

        return crosstabSum;
    }
}
```

Generating Values List for Crosstab

3.7.4 Bar Chart

The Bar Chart component allows users to visualise their data in the form of a bar chart. The user chooses two columns to create their barchart, one column of data to display and another which acts as a dimension for the data. The corresponding rows of the table are added to a sorted map, with the dimension as the key and the corresponding data for each dimension as the value. This data is returned as a response in an AJAX request, where it is passed to the addNewBarChart() function. In this function, the bar chart chart.js object is created and appended to the gridstack widget which will be placed on the dashboard area.

```
public Map<String, List> createBarchart(Map<String, String> barParams) {
    SparkSession spark = DashboardService.getSparkSession();
    Dataset<Row> columns = spark.sql("SELECT `" + barParams.get("columnName") + "`, `" + barParams.get("columnNameDimensionCompBar") + "` FROM tempTable");
    List<Row> columnList = columns.collectAsList(); //Creating list of relevant rows returned from SQL statement
    return getDimensionTotals(columnList);

}
```

```
public Map<String, List> getDimensionTotals(List<Row> columnList) {
    Map<Object, List<Row>> groupByDimension = columnList.stream().collect(Collectors.groupingBy(i -> i.get(1)));
    Map<Object, List<Row>> sortedDimensionMap = new TreeMap<>(groupByDimension);

    List<Object> dimensionLabels = new ArrayList<Object>(sortedDimensionMap.keySet());
    List<Double> dimensionSums = new ArrayList<Double>();

    //Getting sum of values for each date to show trend over time
    for (Map.Entry<Object, List<Row>> entry : sortedDimensionMap.entrySet()) {
        List<Row> dimensionEntries = entry.getValue();
        double total = 0;
        for (Row item : dimensionEntries) {
            if (item.get(0) instanceof Integer) {
                int v = (int) item.get(0);
                double numVal = v;
                total += numVal;
            }
            else {
                double numVal = (double)item.get(0);
                total += numVal;
            }
        }

        dimensionSums.add(Math.round(total * 100.0) / 100.0);
    }

    Map<String, List> listMap = new HashMap<String, List>();
    listMap.put("dimensions", dimensionLabels);
    listMap.put("dimensionSums", dimensionSums);

    return listMap;
}
```

```
function addNewBarChart(response, title) {

    const datarr = [];      // constant for data
    const xlabels = [];     // constant for labels

    const data = response.dimensionSums;
    for (i = 0; i < data.length; i++) {
        datarr.push(data[i]);
        xlabels.push(response.dimensions[i]);
    }

    const columnName = $("input[name='columnName']:checked").val(); //Getting value of chosen radio button

    //Position and dimensions of widget added
    const node = {width: 5, height: 5, autoPosition: true};
    const divId = Math.random();
    const barId = Math.random();

    if (grid.willItFit(null, null, 4, 4, true)) {
        grid.addWidget('<div id=barId' + barId + '><div class="grid-stack-item-content"><div class="barHead
    }
    else {
        $("#dialog").dialog("open");
    }

    const ctx = document.getElementById("myChart_" + divId).getContext('2d');

    const myChart = new Chart(ctx, {
          type: 'bar',
          data: {
            labels: xlabels,
            datasets: [{
                label: columnName,
                data: datarr,
                fill: false,
                backgroundColor: getBarColour(response.dimensionSums.length),
                borderColor: "#3e95cd",
                borderWidth: 1
              }
            ]
          },
          options: {
            tooltips: {enabled: false},
            hover: {mode: null},
            legend: {display: false},
            responsive: true,
            maintainAspectRatio: true,
            scales: {
              xAxes: [{
                ticks: {
                  maxRotation: 90,
                  minRotation: 80
                }
              }],
              yAxes: [{
                ticks: {
                  beginAtZero: true
                }
              }]
            }
          }
        });
};
```

3.7.5 Pie Chart

The Pie Chart component allows users to visualise their data in the form of a pie chart. The pie chart displays values of a String or Date Type columns in terms of their percentage count of the overall number of values in the column. This allows the user to visualise the distribution of their data within a column. The user chooses a column, and enters a title for the Pie Chart tile if they wish. When the modal form is submitted, the data is sent by ajax request to the controller method mapped to “/piechart”. The percentage values for each entry type in the column is computed and returned in the form of a map, with the column values in a list as the key, and the percentage result as the value. This map is returned as a response to the ajax request, where it is passed to addNewPieChart(). This function creates a new pie chart and adds it to the gridstack widget which will be displayed on the dashboard area.

```
function sendPiechartForm(e) {
    e.preventDefault();
    var token = $("meta[name='_csrf']").attr("content"); // Specifying CSRF tokens
    var title = $("#pieTitle").val();

    var params = {
            type: 'post',
            headers: {"X-CSRF-TOKEN": token},
            data: $("#piechartForm").serialize(),
            url: '/piechart',
            success: function(response) {
                document.getElementById('piechartForm').reset();
                addNewPieChart(response, title);
            },
            async: true,
        };

    $.ajax(params);
}
```

```
function addNewPieChart(response, title) {

    columnName = $("input[name='columnName']:checked").val(); //Getting value of chosen radio button

    //Position and dimensions of widget added
    node = {width: 5, height: 5, autoPosition: true};
    divId = Math.random();
    pieId = Math.random();

    if (grid.willItFit(null, null, 4, 4, true)) {
        grid.addWidget('<div id=pieId' + pieId + '><div class="grid-stack-item-content"><div class="pi
    }
    else {
        $("#dialog").dialog("open");
    }

    const ctx = document.getElementById("myChart_" + divId).getContext('2d');

    const myChart = new Chart(ctx, {
          type: 'pie',
          data: {
            labels: response.countLabels,
            datasets: [{
                label: columnName,
                data: response.counts,
                fill: false,
                backgroundColor: getPieColour(response.counts.length),
              }
            ]
          },
          options: {
            tooltips: {enabled: false},
            hover: {mode: null},
            responsive: true,
            maintainAspectRatio: true,
          }
        });
};
```

```java
@Service
public class Piechart {

    @Autowired
    DashboardService dashboardService;


    public Map<String, List> createPiechart(Map<String, String> barParams) {
        SparkSession spark = DashboardService.getSparkSession();
        Dataset<Row> columns = spark.sql("SELECT `" + barParams.get("columnName") + "` FROM tempTable");
        List<Row> columnList = columns.collectAsList(); //Creating list of relevant rows returned from SQL statement
        return getColumnValueCounts(columnList);
    }

    public Map<String, List> getColumnValueCounts(List<Row> columnList) {
        Map<Object, Long> valueCounts = columnList.stream().collect(Collectors.groupingBy(e -> e, Collectors.counting()));
        List<Object> countNames = new ArrayList<Object>(valueCounts.keySet());
        List<Long> counts = new ArrayList<Long>();
        List<String> countLabels = new ArrayList<String>();

        for (Object item : countNames) {
            String countLabel = item.toString();
            countLabel = countLabel.substring(1, countLabel.length() - 1);
            countLabels.add(countLabel);
        }

        for (Map.Entry<Object, Long> entry : valueCounts.entrySet()) {
            Long countValue = entry.getValue();
            Integer columnListLength = columnList.size();
            Long percentageValue = countValue * 100 / columnListLength;
            counts.add(percentageValue);
        }

        Map<String, List> listMap = new HashMap<String, List>();
        listMap.put("countLabels", countLabels);
        listMap.put("counts", counts);

        return listMap;
    }

}
```

3.8 Customising a component

The user can also further customise their components by clicking on them. When they click on a component, a sidebar slides in from the right-hand side. In this sidebar, they can customise the component title by changing the font style and colour. They can also customise the component background colour. There are two buttons in this sidebar. One button clears the style the user has applied to the component, and the other deletes the component from the dashboard. The following code snippet shows the implementation of the template for the dashboard sidebar.

```html
<div class="sidebar-group">
    <div id="font-header">Font</div>
    <select class="form-control" id="fontFamily">
        <option value="Arial">Arial</option>
        <option value="Times New Roman">Times New Roman</option>
        <option value="Georgia">Georgia</option>
        <option value="Trebuchet MS">Trebuchet MS</option>
        <option value="Verdana">Verdana</option>
        <option value="Courier New">Courier New</option>
    </select>
</div>

<div class="sidebar-group">
    <div class="textButtons">
        <button id="bold" class="textButton btn btn-outline-dark" onclick="boldText()"><i class="fa fa-bold"></i></button>
        <button id="italic" class="textButton btn btn-outline-dark" onclick="italicText()"><i class="fa fa-italic"></i></button>
        <button id="underline" class="textButton btn btn-outline-dark" onclick="underlineText()"><i class="fa fa-underline"></i></button>
    </div>
</div>

<hr/>

<div class="sidebar-group">
    <div id="font-colour-options">
        <div id="font-colour">Font Colour</div>
        <input id="color-picker-header" value='#010101'/>
    </div>
</div>

<div class="sidebar-group">
    <div id="font-background-options">
        <div id="background-colour">Background Colour</div>
        <input id="color-picker-background" value='#010101'/>
    </div>
</div>

<hr/>

<div class="sidebar-group sidebar-buttons">
    <button type="button" class="btn btn-danger sidebar-button" id="delete-component">Delete Component</button>
    <button type="button" class="btn btn-primary sidebar-button" id="clear-style">Clear Chosen Style</button>
</div>
```

3.8.1 Customise component

The implementation of customisation options for individual components was extremely important to ensure that the user had full control of their dashboard style to suit their needs and branding. JavaScript and jQuery were used to implement the component customisation options. Within the customisation sidebar, the user can choose the title font-family, font-style, font-colour and component background-colour. Choosing the background and font-colour were implemented using the Spectrum Colour Picker jQuery library. To implement other styling options, the jQuery toggleClass() function was used. When the user chooses an option in the customisation sidebar, the toggleClass() function toggles a class on for the target div. This applies the customisation to the target element directly and in real-time. By using toggleClass(), if a style is changed, then the class is updated to reflect the new style to be applied. The following code snippet shows the implementation of the toggleClass() function on target divs.

```
function headerColour(colour) {
    currentSelection.find('.widgetTitle').css('color', colour);
}

function boldText() {
    currentSelection.find('.widgetTitle').toggleClass('bold');
}

function italicText() {
    currentSelection.find('.widgetTitle').toggleClass('italic');
}

function underlineText() {
    currentSelection.find('.widgetTitle').toggleClass('underline');
}

function fontFamily(font) {
    currentSelection.find('.widgetTitle').css('font-family', font);
}

function widgetBackgroundColour(colour) {
    currentSelection.find('.grid-stack-item-content').css('background-color', colour);
}
```

The customisation options chosen for each component needed to be stored so that for each widget the user clicked, the customisation sidebar reflected their customisation choices. This was handled using JavaScript localStorage to store the chosen customisation options for each component using the components unique id. When the component is clicked on, if the component id is found in localStorage, the customisation sidebar div is populated with the customisation details already chosen. If it is not found, the customisation sidebar elements are set to the default options. The following code snippets demonstrate this functionality.

```
function storeStyleState(prevDiv) {
    if (prevDiv != "") {
        var fontColour = prevDiv.find('.widgetTitle').css('color');
        var backgroundColour = prevDiv.find('.grid-stack-item-content').css('background-color');
    }
    var clickedButtons = $('.textButton.clicked').map(function () {return this.id;}).get();
    var styleValues = JSON.stringify({ //Adding json string representation of user chosen values
        font:document.getElementById('fontFamily').value,
        fontcolour:fontColour,
        bgcolour:backgroundColour,
        clickedButtons:clickedButtons
    });

    localStorage.setItem(prevId, styleValues);
}

function populateDiv(prevId, prevDiv) {
    resetStyleOptions(prevDiv);
    var styleValues = JSON.parse(localStorage.getItem(prevId));
    if (styleValues != null) {
        $('#fontFamily').val(styleValues.font).prop('selected', true);
        for (var i = 0; i < (styleValues.clickedButtons).length; i++){
            var currButton = styleValues.clickedButtons[i]
            $('#' + currButton).toggleClass('clicked');
        }
        $("#color-picker-header").spectrum("set", styleValues.fontcolour);
        $("#color-picker-background").spectrum("set", styleValues.bgcolour);
    }
}
```

3.8.2 Clear current style

Within the customisation sidebar, the user can clear the current customisation options which have been selected and applied to the div. By clicking the “Clear Chosen Style” button, the customisation sidebar is reset to default values, and the classes which have been added to the divs in the component are removed or reset to default. This is handled using jQuery which targets the specific divs, and the removeClass() and css() functions which removes the specified class and updates the target divs css. The following code snippet demonstrates this functionality.

```
function resetStyleOptions() {
    $('#bold').removeClass('clicked');
    $('#italic').removeClass('clicked');
    $('#underline').removeClass('clicked');
    $('#fontFamily').val('Arial').prop('selected', true);
    $("#color-picker-header").spectrum("set", "#010101");
    $("#color-picker-background").spectrum("set", "#fafafa");
}

$('#clear-style').on('click', function() {
    resetStyleOptions(); //Resets all style option inputs
    prevDiv.find('.widgetTitle').css('color', '#2C3E50');
    prevDiv.find('.widgetTitle').removeClass('bold');
    prevDiv.find('.widgetTitle').removeClass('italic');
    prevDiv.find('.widgetTitle').removeClass('underline');
    prevDiv.find('.widgetTitle').css('font-family', 'Arial');
    prevDiv.find('.grid-stack-item-content .text-area-bg').css('background-color', '#FFFFFF');
});
```

3.8.3 Delete component

Within the customisation sidebar, the user can delete the current chosen component. By clicking on the “Delete Component” button, the component is removed from the dashboard by calling the Gridstack removeWidget() function on the grid, passing the id of the div to be deleted to the function. The following code snippet shows this functionality and how it is implemented. When the delete button is clicked, the customisation sidebar then slides back to the right and hidden from the page.

3.9 Exporting a PDF report

Once the user is happy with the dashboard they have created, they can export it in PDF format as a report. They can do this by clicking the “Export PDF” button in the top right corner. The PDF is downloaded to the users local machine. The PDF export is implemented using jsPDF, a client-side PDF generation library. The following code shows the implementation of the PDF export. Each component on the dashboard area is individually added to the PDF document by getting its position on the current dashboard area and converting it to a png image.

```
function createPDF() {

    var pdf = new jsPDF('l', 'pt', 'a4'); //Creating PDF object
    pdf.internal.scaleFactor = 1.33;
    var items = $(".grid-stack .grid-stack-item"); //Getting all items from grid
    var deferreds = [];

    var pdfName = document.getElementById('dashboardName').innerText + '.pdf'; //Sets name of pdf to b

    //Iterating through items which have been added to the dashboard
    for (i = 0; i < items.length; i++) {

        var pos = $(items[i]).position();
        var numberGraphs = $(items[i]).find("#chartContent").length; //Checking if there are graph div

        if (numberGraphs > 0){
            var chartId = $(items[i]).find('canvas')[0].id; //Getting individual chart id of current c
            addImage(pdf, pos, chartId);
        }
        else {
            var widgetId = $(items[i])[0].id;
            var widget = document.getElementById(widgetId); //Getting chart by id
            var widgetWidth = widget.getBoundingClientRect().width * 0.75; //Multiply by 0.75 to conve
            var widgetHeight = widget.getBoundingClientRect().height * 0.75; //Multiply by 0.75 to con
            var deferred = $.Deferred(); //Creating deferred object
            deferreds.push(deferred.promise());
            generateCanvasWidget(widgetId, pdf, deferred, widgetWidth, widgetHeight, pos);
        }
    }

    $.when.apply($, deferreds).then(function () { // Executes after all images have been added
        pdf.save(pdfName);
        uploadPDF(pdf, pdfName);
    });

}


function generateCanvasWidget(widgetId, pdf, deferred, widgetWidth, widgetHeight, pos){
    html2canvas(document.getElementById(widgetId), {
        onrendered: function (canvas) {
            var img = canvas.toDataURL("image/png", 1.0);
            pdf.addImage(img, 'JPEG', pos.left * 0.75, pos.top * 0.75, widgetWidth, widgetHeight); //A
            deferred.resolve(); //Resolving the deferred
         }
    });
};


function addImage(pdf, pos, chartId) {
    var canvas = document.getElementById(chartId); //Getting chart by id
    var canvasImg = canvas.toDataURL("image/png", 1.0); //Converting chart to an image
    var imgWidth = canvas.getBoundingClientRect().width * 0.75; //Multiply by 0.75 to convert to pt
    var imgHeight = canvas.getBoundingClientRect().height * 0.75; //Multiply by 0.75 to convert to pt
    /* (x, y) co-ordinates of top left corner of image is (pos.left * 0.75, pos.right * 0.75)
     and multiplying the position px by 0.75 to convert to pt */
    pdf.addImage(canvasImg, 'JPEG', pos.left * 0.75, pos.top * 0.75, imgWidth, imgHeight); //Adding im
};
```

A copy of the PDF report that is generated is also uploaded to the users unique Google Cloud Storage bucket. This allows the user to access their PDF documents in the future. When the PDF is generated, the PDF object string is passed to the uploadPdf() controller method, where it is converted into a byte array. This byte array is then added to the users bucket as a Blob object. The following code snippets show the implementation of adding the PDF byte array to the PDF bucket.

```
@PostMapping("/uploadPDF")
@ResponseBody
public void uploadPdf(@RequestParam(name = "pdf") String pdf, @RequestParam(name = "pdfName") String pdfName) {
    String base64Pdf = pdf.split(",")[1]; //Splitting Base64 string to get PDF data and remove metadata
    byte[] pdfBytes = javax.xml.bind.DatatypeConverter.parseBase64Binary(base64Pdf); //Decoding Base64 string to byte array
    dashboardService.uploadPdfToBucket(pdfBytes, pdfName); //Passing byte array to upload function

}
```

```
public void uploadPdfToBucket(byte[] pdfBytes, String pdfName) {
    String pdfBucket = getUserBucket();
    Storage storage = StorageOptions.getDefaultInstance().getService();
    Bucket newBucket = storage.get(pdfBucket);
    LocalDateTime date = LocalDateTime.now();
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
    String textDate = date.format(formatter);
    String fileName = pdfName + " " + textDate;
    newBucket.create(fileName, pdfBytes , "application/pdf");
    BlobId blobId = BlobId.of(pdfBucket, fileName);
    storage.createAcl(blobId, Acl.of(Acl.User.ofAllUsers(), Acl.Role.READER));
}
```

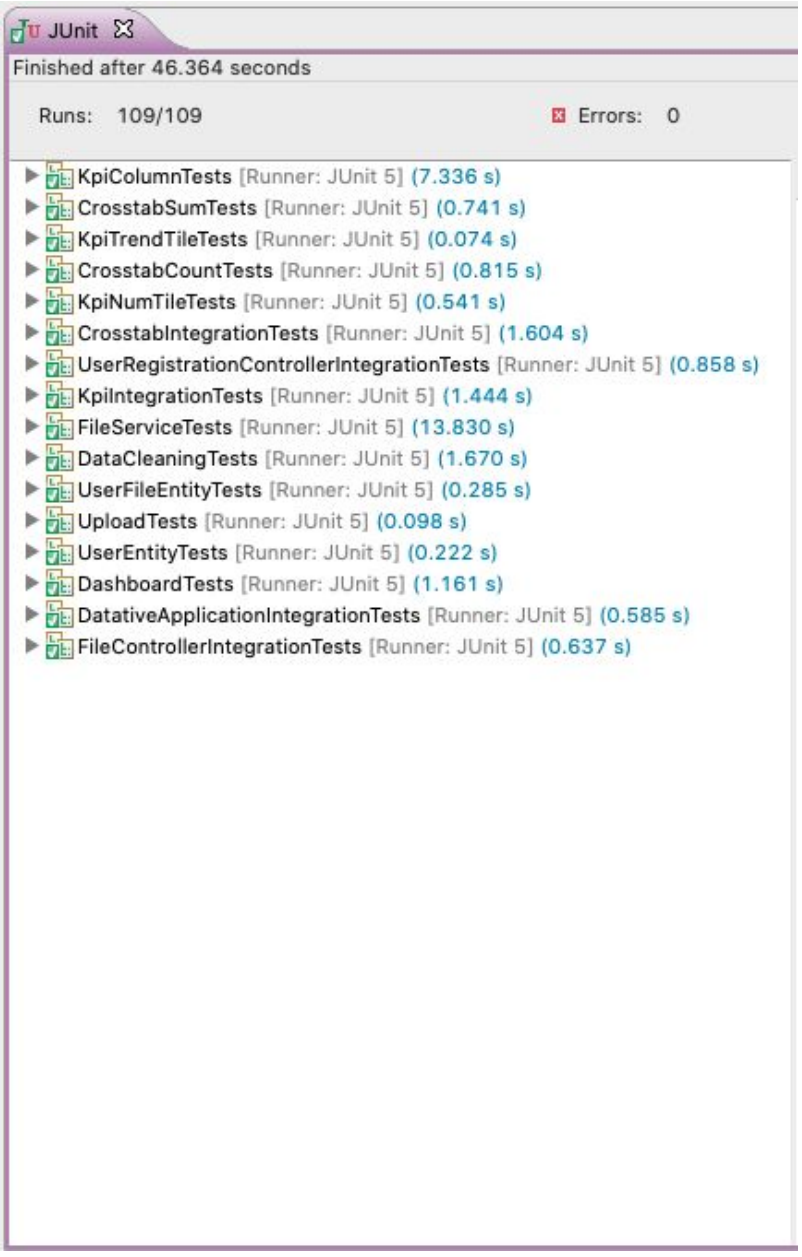
3.10 Logout

The logout button is located in the top navigation bar of the application. When the user clicks the logout button, they are logged out of the application. The logout functionality is also handled by Spring Security. If the request path is matched to “/logout”, then the user authentication is cleared and the user is redirected to the logout successful url which is the login url with the logout parameter passed.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
            .authorizeRequests()
                .antMatchers(
                        "/registration**",
                        "/js/**",
                        "/css/**",
                        "/img/**",
                        "/webjars/**").permitAll()
                .anyRequest().authenticated()
            .and()
                .formLogin()
                    .loginPage("/login")
                        .permitAll()
            .and()
                .logout()
                    .invalidateHttpSession(true)
                    .clearAuthentication(true)
                    .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
                    .logoutSuccessUrl("/login?logout")
            .permitAll();
}
```

4. Testing and Validation

To ensure expected behaviour and an enjoyable user experience, a large emphasis was put on testing and validation of Datative. The following section discusses the types of testing performed on Datative. A total of 109 unit and integration tests were written for the application. Test coverage was used as a general guide to ensure that enough testing was performed on the application, but was not the only metric. Instead, emphasis was placed on ensuring that as many edge-cases as possible were considered, along with all possible response types. The tests are located under src/main/test directory. The tests were run through Eclipse IDE or the command line using the command *mvn test.* The following image shows the result of running all of the application test cases in Eclipse.



4.1 Unit Testing

Unit testing was performed to test the main components of the application. The testing was carried out using JUnit. The goal of the unit tests was to ensure that the application components were working as expected. To ensure that as many possible scenarios were handled, tests were run with expected and unexpected inputs. As the application processed files uploaded by the user, it was important to ensure that the expected results occurred for all accepted file inputs. Sample datasets were added as resources to the application which facilitated the unit testing of the applications behaviour on real files. By specifying the expected output from these sample datasets, this ensured that the file was processed as expected. The following code snippets show examples of unit tests written for the application.

```
@Test //Testing that table name that already exists returns true
@WithUserDetails("testeraccount@mail.com")
public void testTableNameExists_AssertTrueIfExists() throws SQLException {
    ResultSet results = fileService.checkForTable("test");
    assertTrue(results.next());
}
```

```
@Test
public void testCleaningFullTableFromFileMissingData_PassIfEqual() throws IOException {
    ClassLoader classLoader = getClass().getClassLoader();
    URL testResource = classLoader.getResource("datacleaning-test.csv");
    URL resultsResource = classLoader.getResource("cleaned.csv");
    Table table = Table.read().csv(CsvReadOptions.builder(testResource).tableName("Tester table"));
    Table expectedResult = Table.read().csv(CsvReadOptions.builder(resultsResource).tableName("Tester table"));
    Table result = DataCleaning.cleanData(table);
    assertEquals(result.toString() , expectedResult.toString());
}
```

```
@SuppressWarnings({ "rawtypes", "serial" })
@Test //Short list of values to check that expected value returned for 2 dimensions
void checkCountyThreeDimensionsWhole_PassIfEqual() {
    Map<String, List> result = kpiTile.getDimensionTotals(valuesListCountyWhole);
    List<Object> testDimensionLabels = Arrays.asList("Cork", "Dublin", "Waterford");
    List<Double> testSums = Arrays.asList(24.0, 243.0, 5.0);

    Map<String, List> expectedResults  = new HashMap<String, List>() {{
        put("dimensions", testDimensionLabels);
        put("dimensionSums", testSums);
    }};

    assertEquals(result, expectedResults);
}
```

```
@Test //Normal positive list - Whole nums
void checkPosWholeNumOpsTotal_PassIfEqual() {
    double result = kpiTile.performNumOps(positiveWhole, "total");
    assertEquals(result, 2022, 0);
}
```

4.2 Integration Testing

Integration testing was extremely important to verify the end-to-end behaviour of our system. It ensured that controllers functioned as expected and the integration of modules within the application. MockMVC was used to perform integration testing of the application. It allowed us to test the Spring Controllers without having to explicitly start a Servlet container. The Mock MVC class is part of the Spring Test framework. The WebApplicationContext object is passed to the MockMvc builder to mock Spring's WebApplicationContext for the integration tests. MockMVC mocked calls to controllers to ensure that the correct responses were being received and the correct data was being passed. It also tested that the correct views were returned for different mappings and page content of the returned view was correct. The following code snippets show some of the integration tests which were implemented.

```
@Test //Checking the correct view is returned when registration is called
public void givenRegistrationPageUri_whenMockMVC_thenReturnsRegistrationView() throws Exception {
    mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    mockMvc.perform(get("/registration")).andExpect(status().isOk()).andExpect(view().name("registration"));
}
```

```
@Test //Check that redirect occur when file is empty
public void shouldHaveStatus302ForRedirect() throws Exception {
    mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    mockMvc.perform(MockMvcRequestBuilders.multipart("/uploadFile")
            .file("file", multipartEmptyFile.getBytes())
            .characterEncoding("UTF-8"))
            .andExpect(status().is(302));
}
```

```
@Test//Checking content of Login view is correct
public void testLoginViewContent_PassIfCorrect() throws Exception {
    mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    mockMvc.perform(get("/login"))
            .andExpect(content().string(containsString("Login")));
}

@Test(expected = AssertionError.class)//Checking content of Login view is correct
public void testIndexViewContent_PassIfIncorrect() throws Exception {
    mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    mockMvc.perform(get("/"))
            .andExpect(content().string(containsString("This is not on this page")));
}
```

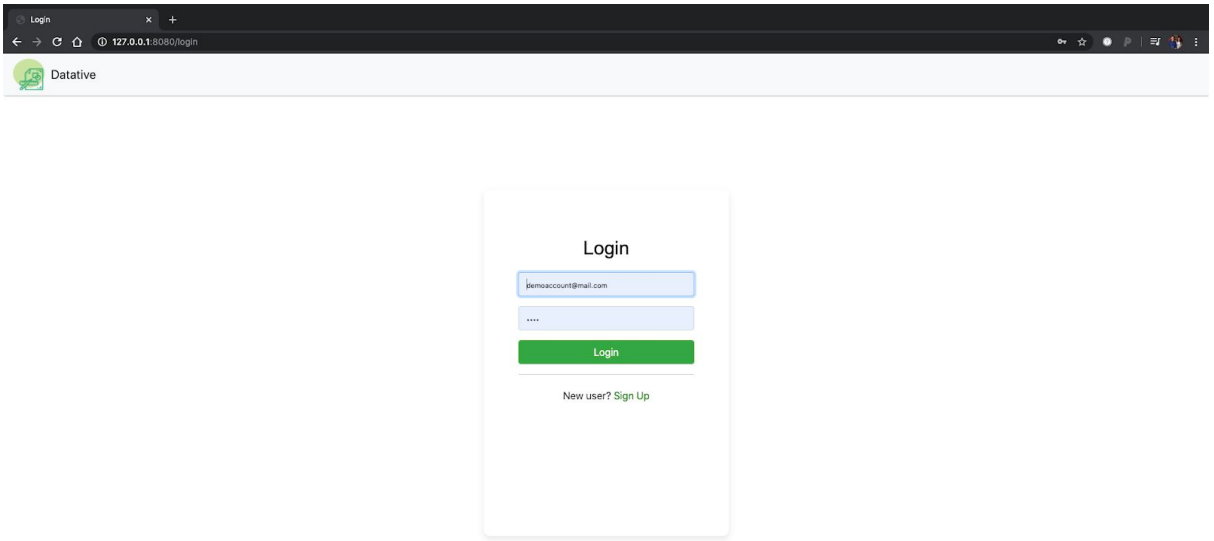
4.3 Ad Hoc Testing

Throughout the development process, a significant amount of Ad Hoc testing was performed. This was informal testing which was carried out to ensure that the application behaved as expected and correct results were produced. The file upload functionality was tested using different file types and datasets downloaded from Kaggle. Using these datasets ensured that data was processed correctly and that database tables were created correctly from the data. These datasets also allowed us to test the visualisations and ensure that they displayed the expected results based on the data. Although formal unit and integration testing was performed, ad hoc testing through the development process ensured that the

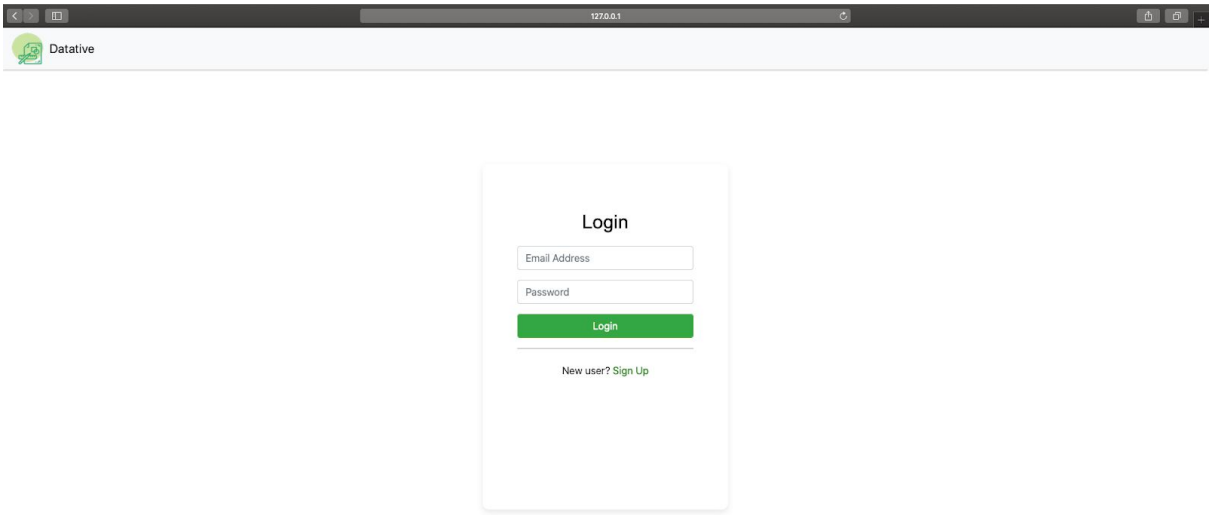
implemented functionalities were producing the output that would be expected by the user when using the application.

4.4 Browser Testing

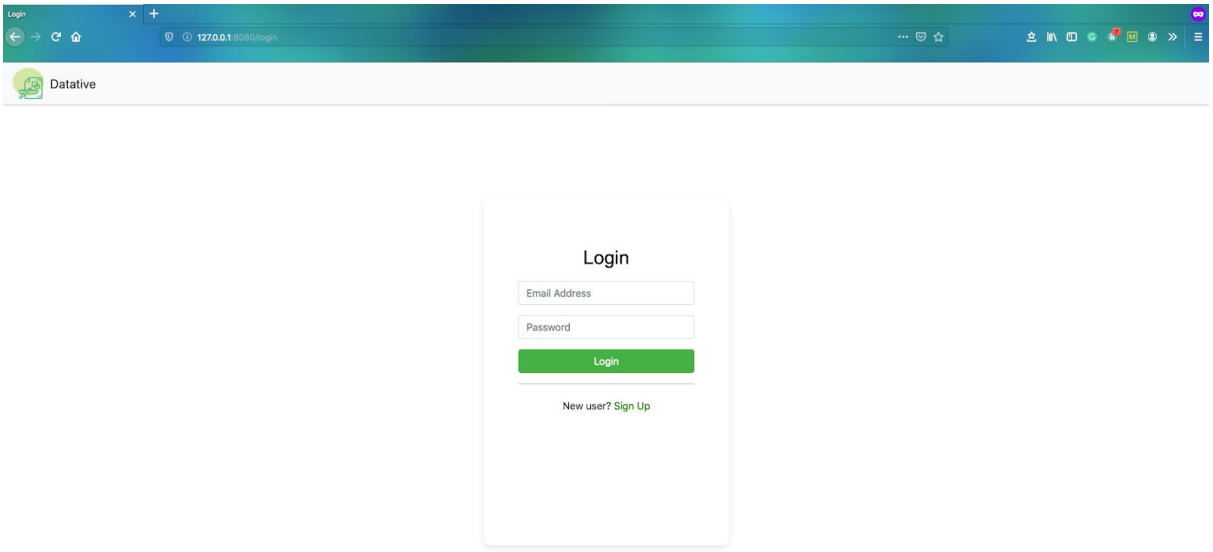
To ensure a seamless and consistent user experience, testing across multiple browsers was required. The application was run and viewed in Google Chrome, Safari and Firefox to ensure functionality and design across multiple browsers. The following images show the application in each browser.



Chrome



Safari



Firefox

5. Problems and Resolutions

5.1 Table header row

When creating a table in the database, the column names are first added when the table structure is created. However, when populating the table with the data from the users dataset, the header row would be added as a row into the table. This row could not remain because it was not related to the data in the column and was just a description of the data. The header row could not be skipped when using the Cloud SQL Admin API, so another solution had to be implemented. To solve this problem, a method was added to the FileService class called deleteHeaderRow(). This method takes the table name and the database connection object and executes a SQL statement on the table which always removes the first row of the table only. This ensured that only the header row would be removed each time. The following code snippet shows this method.

```
public static void deleteHeaderRow(String tableName, Connection con) throws SQLException {
    String deleteStatement = "DELETE FROM `" + tableName + "` LIMIT 1"; //Deleting first row from the table
    PreparedStatement stm = con.prepareStatement(deleteStatement);
    stm.executeUpdate();
}
```

5.2 Remove header row executes too early

After the Cloud SQL Admin API executes the request to populate the database the table, the deleteHeaderRow() method is called. However, as the request had not finished executing, the deleteHeaderRow() method was executing before the data had been added to the table which meant that the header row was not being removed. To solve this problem, a sleep of 5

seconds was added to ensure that the table could be populated before the execution of the method. The user is unaware of this sleep as while the file is uploading, there is a loader gif displayed which shows the user the upload is currently taking place.

5.3 Keeping customisation stored for each component

A highlight of the application's functionality is the ability of a user to be able to customise each component on the dashboard. When a user customises a component, it was important to ensure that the customisation settings were saved for each component and the customisation sidebar reflected the current state of the chosen component. As there is no set amount of components added to the dashboard, storing the component settings was a difficult task. To implement this, the JavaScript localStorage object was used. This stores data directly in the browser and does not expire until it is cleared. After a component is customised, and the user exits the customisation sidebar or chooses another component, a JSON object string is created which contains all the chosen options. The options are then stored in localStorage, with the component id as the key. This means that a set of options will belong to a unique component id so all customisation options can be stored without interference. When a user clicks on a component, localStorage is checked to see if its id is contained there. If it is, the customisation sidebar div is populated with the chosen options from localStorage. If it is not, this means that the component has not been customised yet and the sidebar div is populated with the default options.

```
function storeStyleState(prevDiv) {
    if (prevDiv != "") {
        var fontColour = prevDiv.find('.widgetTitle').css('color');
        var backgroundColour = prevDiv.find('.grid-stack-item-content').css('background-color');
    }
    var clickedButtons = $('.textButton.clicked').map(function () {return this.id;}).get();
    var styleValues = JSON.stringify({
        font:document.getElementById('fontFamily').value,
        fontcolour:fontColour,
        bgcolour:backgroundColour,
        clickedButtons:clickedButtons
    });

    localStorage.setItem(prevId, styleValues);
}
```

5.4 Components going outside dashboard area

As the dashboard area is constructed from Bootstrap Columns, it is flexible. This meant that when new components were added to a dashboard which was full, the component would be displayed outside the dashboard area. To solve this, the GridStack.js function willItFit() was called on the grid area. The function takes the dimensions of the item to be added. It checks if there is sufficient space on the dashboard area to add a component of this dimension. If there is not, an alert is displayed telling the user that there is no space left on the dashboard.

6. Future Work

There are many possibilities for future work with this application. One avenue of future work would be to save the state of dashboards in progress so that users can return to dashboards that they were previously working on. Also, dashboards could be shared with other users so that collaboration could occur in their creation. There is also scope for the addition of more components, as there are endless possibilities of components which could be added to the dashboard that would be of value to the user. Work to improve the UI could also be carried out. There is a lot of free space on each of the application pages, and although we feel they are well designed and it is easy to focus on the content of each page, further work could be done to reduce the number of pages and display the majority of functionalities on a main profile page. The quality of the PDF report produced also needs to be improved, as nesting the content in the items on the Gridstack component causes the quality of the component to be slightly degraded.

7. References

1. https://spring.io/projects/spring-security
2. https://spring.io/
3. https://cloud.google.com/storage/docs
4. https://cloud.google.com/sql/docs
5. https://github.com/gridstack/gridstack.js/tree/develop/doc
6. https://jtablesaw.github.io/tablesaw/
7. https://getbootstrap.com
8. https://cloud.google.com/sql/docs/mysql/admin-api