Coursework Three

6048COMP Innovations in Software Development

Weighting: 35%

Joshua Collins (726682) & Michael Grant (725401)

Liverpool John Moores University



Contents

[Figures 2](#_Toc509411581)

[Abstract 3](#_Toc509411582)

[Introduction 3](#_Toc509411583)

[Fetching data using GSON 3](#_Toc509411584)

[Using charts in JavaFX 5](#_Toc509411585)

[Using tables in JavaFX 7](#_Toc509411586)

[Additional functionality 9](#_Toc509411587)

[Conclusion 12](#_Toc509411588)

[References 13](#_Toc509411589)

# Figures

[Figure 1 Store the data source URL, read as JSON and enter to LinkedList 3](#_Toc509411562)

[Figure 2 Establish connection to data source 4](#_Toc509411563)

[Figure 3 Constructor for Cars data 4](#_Toc509411564)

[Figure 4 Line chart displaying vehicles sales per year 5](#_Toc509411565)

[Figure 5 Clear chart data before loading data 5](#_Toc509411566)

[Figure 6 Mapping the streamed data to the to the LineChartYear 6](#_Toc509411567)

[Figure 7 Pie Chart displaying the sales per region by vehicle 6](#_Toc509411568)

[Figure 8 Clear pie chart data before loading data 7](#_Toc509411569)

[Figure 9 Mapping data to pie chart and slice total 7](#_Toc509411570)

[Figure 10 TableView with multiple selected functions 8](#_Toc509411571)

[Figure 11 Column values and calculations on selected rows 9](#_Toc509411572)

[Figure 12 Output of current time and date to the label 9](#_Toc509411573)

[Figure 13 Displaying current time and date to a label 9](#_Toc509411574)

[Figure 14 Live time output updating every second 10](#_Toc509411575)

[Figure 15 Method used containing thread to calculate the live time data 10](#_Toc509411576)

[Figure 16 Information box opened when selecting Help>About 10](#_Toc509411577)

[Figure 17 ActionEvent for information dialog 11](#_Toc509411578)

[Figure 18 Progress indicator to display application is loading 11](#_Toc509411579)

[Figure 19 Non-functional print and styling options 12](#_Toc509411580)

# Abstract

This short report documents the development activity throughout the implementation of this business intelligence control panel, showing technologies that have been employed followed by the resulting functionality achieved by this. This coursework involves using NetBeans and Scene Builder to create a JavaFX application to hold the functionality of various charts, tables and more into this coursework. This group consists of Joshua Collins and Michael Grant who both have contributed 100% each for this coursework.

# Introduction

This development report will address problems that the team have been encountered along with solutions and reasons behind the chosen resolutions throughout this group coursework. In addition, this report will cite any useful sources to aid the solutions to some problems during the development of this control panel. This coursework involves developing a Business Intelligence Control Panel using JavaFX along with Windows Communication Foundation that displays corporate data in an appropriate graphical way

The aim of this business intelligence system is to dynamically inject data from a URL and construct the data using GSON and a constructor. From this, the data will be into various charts efficiently so that if data is updated, the charts will also update accordingly. Appearance of data is incredibly important in this business intelligence system. For example, in a commercial environment, business managers may be required to present this system to clients or shareholders to portray the sales grouped by various categories. Therefore, different methods of presenting the data in this system include; bar charts, line charts, area charts, pie charts and tables, all showing a different aggregation of data.

# Fetching data using GSON

Using GSON for this business intelligence system is incredibly important to interpret the data that is required. The data to be used is fetched from the URL address shown in figure 1 below. First, the GSON jar file must be added to the project in NetBeans, this is done by adding the required JAR file along with the Javadoc to ensure this is able to function properly. The GSON import reads the data from the website and reads is into JSON, which can then be imported into the project straight into a LinkedList named Cars. Figure 2 shows the DashService class which essentially establish the connection to the URL stated in figure 1.

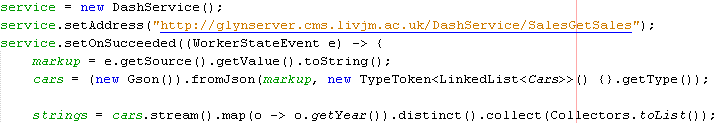


Figure 1 Store the data source URL, read as JSON and enter to LinkedList

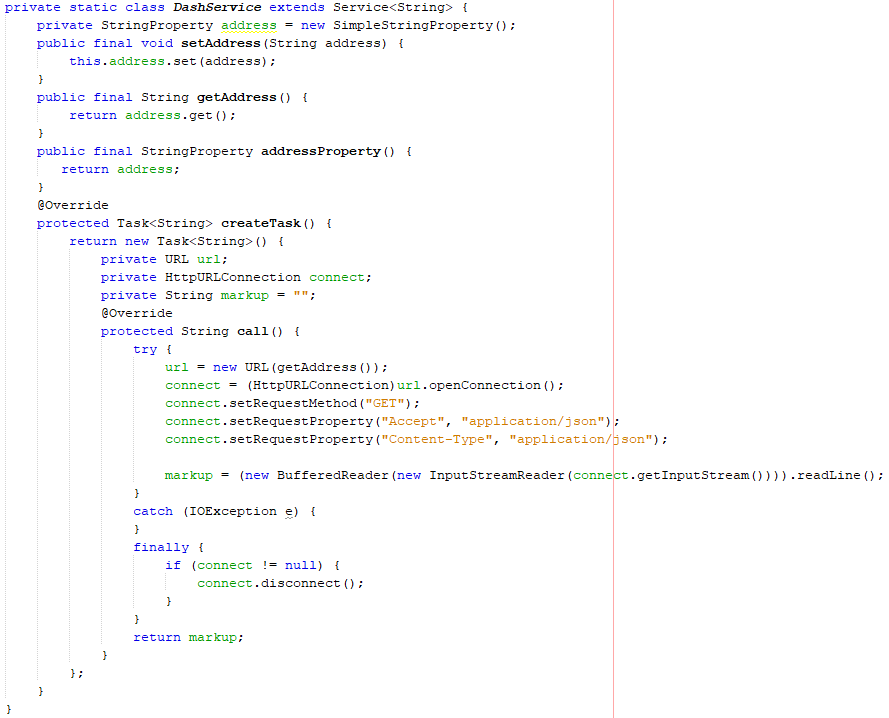


Figure 2 Establish connection to data source

When the data has been read by the input stream from the URL using GSON and JSON, a constructor must be put in place to ensure the data is stored under the correct variables in the right order. In addition, the variables used in the constructor must match the related variables as portrayed in the Dash Service web page found at: <http://glynserver.cms.livjm.ac.uk/DashService/help/operations/SalesGetSales> . Additionally, the format of the string that can be used to print the data from the source includes “%s%s%s%s%s” to declare there are five columns per line when reading the data that will also match with the XML response. This can be seen in figure 3.

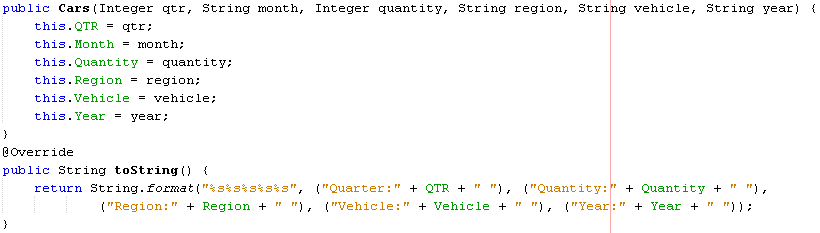


Figure 3 Constructor for Cars data

# Using charts in JavaFX

Using charts in to present commercial data in a business environment is an excellent way to portray the progress of data within an organisation, to then product analysis, predictions and more. When using scene builder in NetBeans, it is possible to make use of line, bar, area, pie, bubble, scatter, stackedBar and stackedArea charts. In the case of this control panel, the team have made use of line, bar, area and pie charts to display the data from the required source dynamically using multiple series.

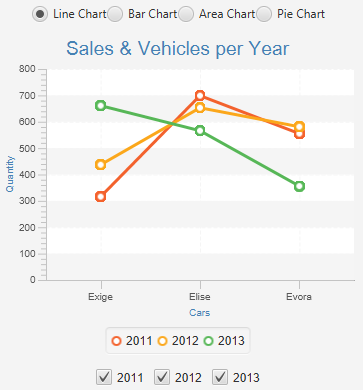


Figure 4 Line chart displaying vehicles sales per year

When adding the data into whichever desired chart for this project, it is important to ensure all previous data is cleared from the chart. Doing this ensures data accuracy and reducing the likelihood of duplicate data. This must be declared for each chart used in the application; therefore, for this system all 12 charts in this system use an identical line of code in figure 5.



Figure 5 Clear chart data before loading data

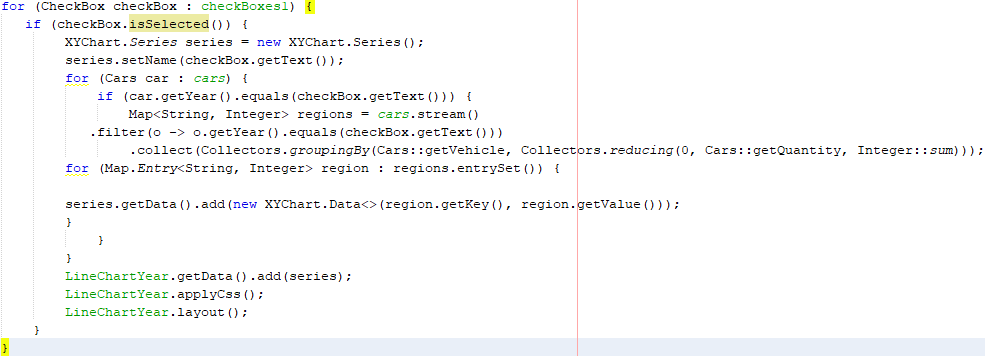
After clearing the data from the chart, the required information for each chart is ready to be loaded. Each functionality for the bar charts, line charts and area charts are wrapped in a for loop to ensure the data is mapped correctly when toggling the filters. Filters vary on the charts between region, year and vehicle to ensure all separate charts display different data, yet still beneficial to the client. In Figure 6 below, the data is filtered by the year and mapped to the check box to ensure when a box is selected or de-selected, the data alters with this. A collector is used which groups the filtered data by vehicle, whilst retrieving the total sales by calling “Collectors.reducing(0, Cars::getQuantity, Integer::sum)))”. From this, the data is then added into the XYChart followed by applying the css and layout. 

Figure 6 Mapping the streamed data to the to the LineChartYear

In addition, implementing a pie chart was slightly different in comparison to producing the line charts, bar charts and area charts. Pie charts are beneficial to produce when displaying to data to clients as there is showing the data in a different style, even though the difference between data values may not be as clear. Figure 7 shows a pie chart that has been implemented into this business intelligence system. You should notice below the check boxes of this figure, that when selecting a slice of the pie chart the value is shown. In this example, the Asia value is 817, America 46 and Europe 551.

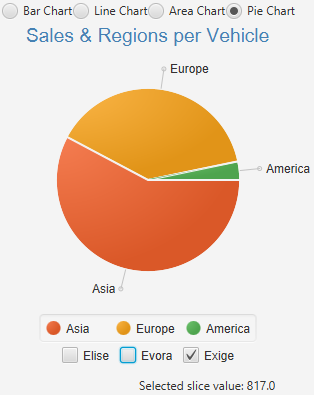


Figure 7 Pie Chart displaying the sales per region by vehicle

Similar to the figure 5 above, the data of the pie chart should be cleared before adding any new data into the chart. This again is to ensure no old data is previously stored, reducing duplicate and dirty data to be stored in the chart.



Figure 8 Clear pie chart data before loading data

The streamed data can now be inserted into the pie chart mapping necessary series to areas of the constructor presented in figure 3. In the example below, figure 9 shows the data being filtered to ensure that the data presented is mapped accordingly to any changes in selecting the check boxes for this chart. A collection is used from this to calculate the sum of the sales and group the values by region to display as the series. A key difference between implementing the pie chart and other charts in this way is there is no “for (Cars car: cars)” around the data being filtered and collected. If this were to be in place, then the pie chart would display a series for every piece of individual data, although this is not the case for other charts.

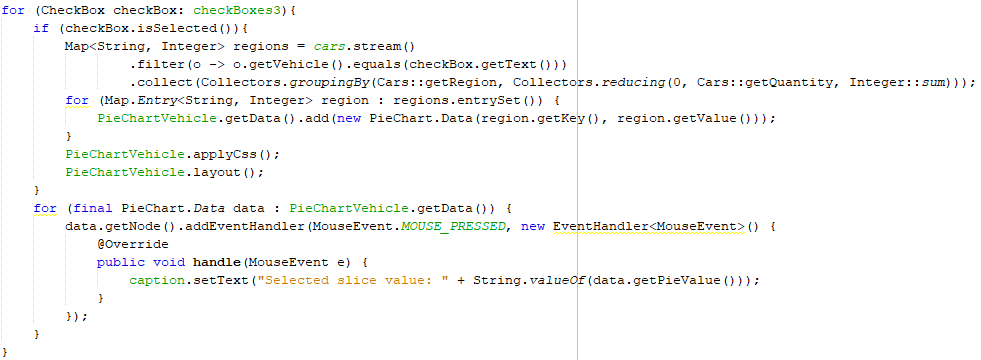


Figure 9 Mapping data to pie chart and slice total

# Using tables in JavaFX

Using TableViews in JavaFX are an in important way to present data for almost any purpose, with many different functions available to put in place. TableViews allow you to enter data dynamically or fixed depending on the purpose, each table column automatically uses a sort function to adjust the data if required. Additional functions that may be used in the tables that have not been implemented into this system include: editing, adding and deleting rows whilst the application is live. Figure 10 displays the table used in this business intelligence system that portrays the quarter, month, year, sales, region and vehicle. In addition, the bottom of figure 10 shows the total and average of multiple rows when selected.

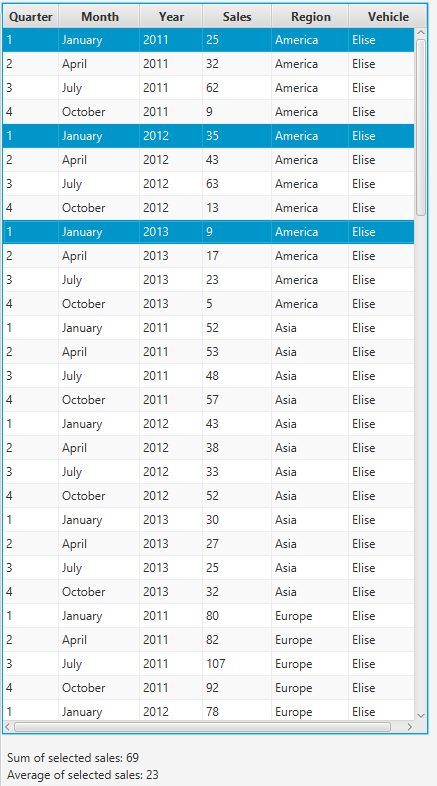


Figure 10 TableView with multiple selected functions

When creating the table in Scene Builder, the columns must also be added into the table and all given an appropriate fx:id. Figure 11 below shows that each individual column has been named accordingly such as regionColumn and vehicleColumn etc. In addition, although the charts for this business intelligence system are dynamically injected with data. The table brought more difficulties in this aspect to also dynamically inject the data. Therefore, the solution around this was to hard code the data into the table. This reduces efficiency of the code and data but would certainly be an area for improvement in terms of future development. In contrast, if the data were to be modified on the data source URL, the charts would automatically update but the table would require the data to manually be inserted.

As previously mentioned, the table in figure 10 allows the user to select multiple rows by a line of code in figure 11, “tableView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);”. From this, additional functionality to include a listener when items are selected can be added. The changed method in figure 11 shows the operations of calculating the sum and average of sales depending on selected rows. Once the calculation has been complete, the according labels are then set with text to display the result.



Figure 11 Column values and calculations on selected rows

# Additional functionality

Furthermore, multiple simple functions in a system such as this business intelligence control panel can make the system more complex. For example, when running this application the data is fetched from the URL stated above. Therefore, it may be beneficial to inform the client of when the data was last updated in this application as the data could be updated to live business results at any time. Figure 13 below shows the simple function to do this, also figure 12 showing the visual representation of this.



Figure 12 Output of current time and date to the label



Figure 13 Displaying current time and date to a label

Using functions to calculate the current time is useful for different purposes. Another functionality included into this business intelligence system is the current time that updates to display the live time. Figure 14 shows the graphical output of this changing accordingly. Figure 15 illustrates the method used to carry out this function. Using a thread that sleeps for 1 second, at the end of the thread the new time is fetched and then set to the specific label required. Therefore, having the thread to sleep for 1 second updates the live time every second with no delay. However, there may be a possibility of the time skipping a second depending on the performance of the machine being used.

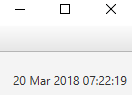
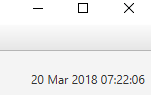


Figure 14 Live time output updating every second

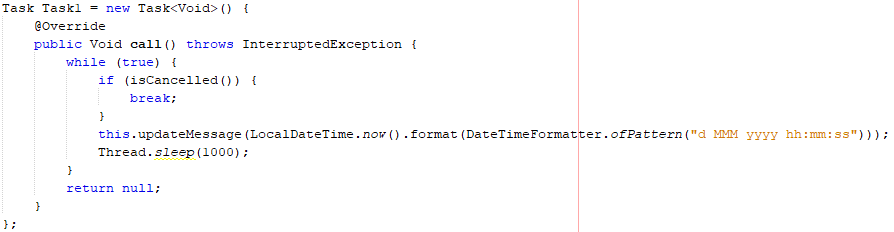


Figure 15 Method used containing thread to calculate the live time data

Almost every application developed to a professional standard today will contain a menu bar at the top of the application. Popular menu headings include; File, Edit and Help. Therefore, it would be desirable to implement a similar approach into this business intelligence system. Menu options implemented and are functional in this application include File>Exit and Help>About. There are other functions implemented into this application but are not functional that include File>Print, Edit>Add Styling, and Edit>Clear Styling. Figure 16 below shows the result when the user selected Help>About in this system. Simply displaying information of the team members who have worked on this system.

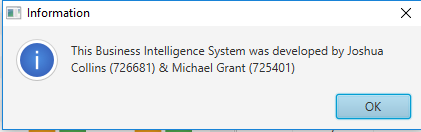
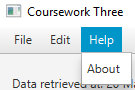


Figure 16 Information box opened when selecting Help>About

The code the implement the information box to display was very straight forward. This included using an ActionEvent for whenever the About option was selected, if this were to be selected then the information box details are declared in figure 17 below.

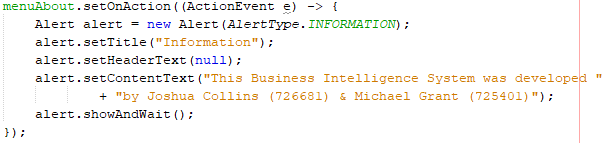


Figure 17 ActionEvent for information dialog

Furthermore, depending on the size of the data to be input into an application, this could be timely on the client side to wait for the results. The client should be able to see that something is working in the application when the data is not loaded, to re-assure them the application has not crashed. Figure 18 shows a loading icon being displayed when the application first runs in this system. This progress indictor will spin around to show the user that the data is loading. Once the data is loaded, the progress indictor will disappear.

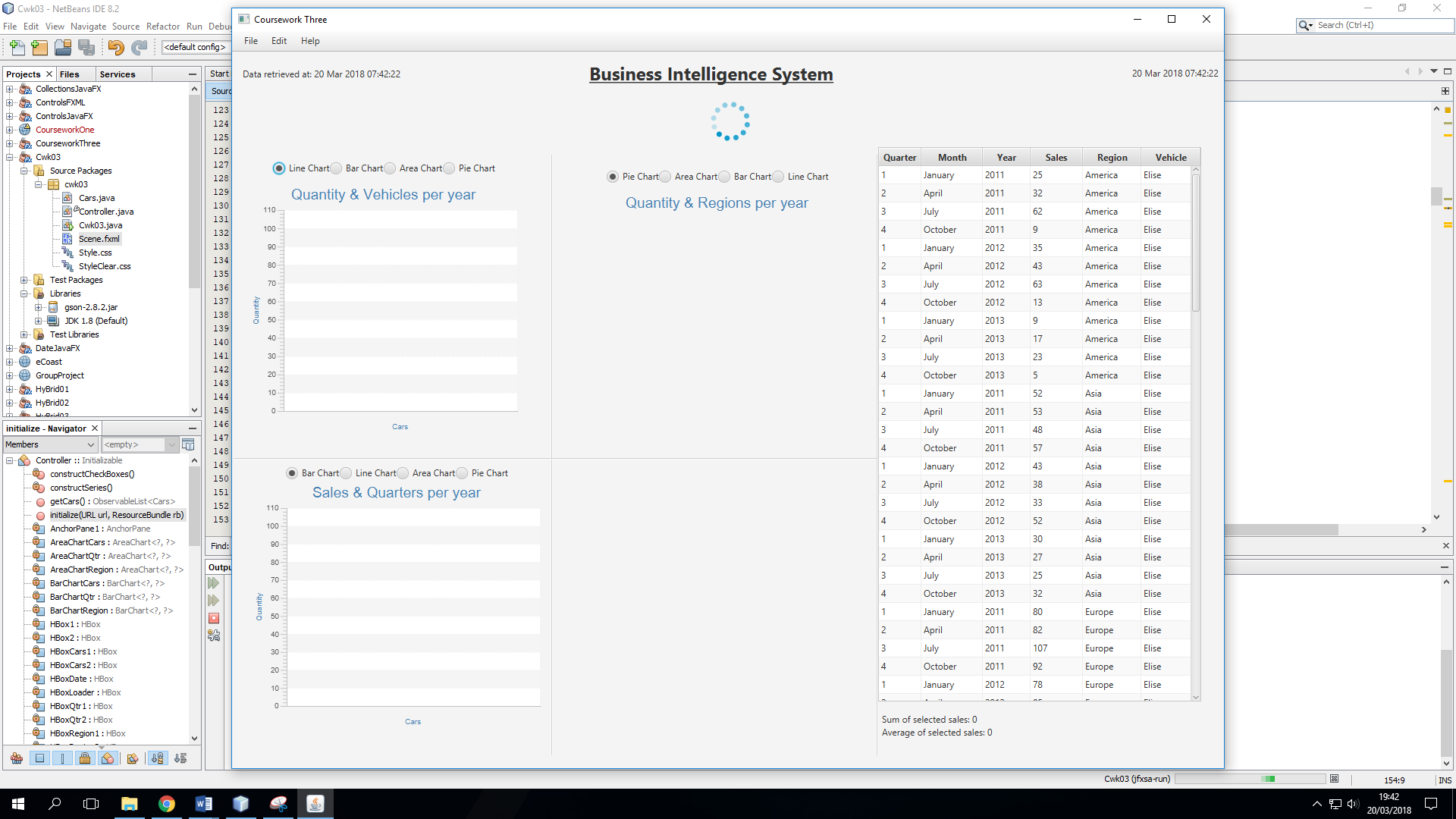


Figure 18 Progress indicator to display application is loading

# Conclusion

To conclude on this business intelligence Java FX application created using NetBeans and Scene Builder. The sole sim of this system was to display commercial data using a myriad of reporting tools. This system must use one or more query parameters which users can specify along with the retrieved data. This system meets these aims with a total of 12 charts able to choose from for the user that dynamically injects the data from the URL data source provided for the benefit of the charts being able to be updated automatically according to the data. In addition the user is able to select filters on each of the 3 chart topics to display specific data when necessary. In addition, the table is used in the control panel to display each specific data row to the user, which when selecting one or more rows will calculate the total and average sales for whichever combination of data the client desires.

Additionally, the system includes additional functions for the clients pleasure displaying the time and date of when the data was retrieved, the current date and time, values of selected pie chart slices, information dialog and exit option. There are also additional functionally that were attempted to implement into this system that were not able to execute properly. These non-functional operations include allowing the user to print the business intelligence system, allowing the user to add or clear a style sheet used in the system.

Possible future development improvements for this business intelligence system identified by the team include; dynamically injecting the data into the table likewise with the charts to ensure the data consistency and accuracy rather than manually entering the data. Other future improvements include resolving the problems of the non-functional operations with the print option and allowing the user to change the styling of the system. These operations are shown in figure 19.

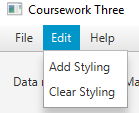
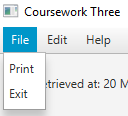


Figure 19 Non-functional print and styling options

# References

Hughes, G. (2018). *GlynH's HyBrid03*. Retrieved March 2018, from Canvas: https://canvas.ljmu.ac.uk/courses/12276/pages/glynhs-hybrid03?module\_item\_id=314329

Hughes, G. (2018). *Misc JavaFx*. Retrieved March 2018, from Canvas: https://canvas.ljmu.ac.uk/courses/12276/pages/misc-javafx?module\_item\_id=315100

Hughes, G. (2018). *SalesGetSales*. Retrieved March 2018, from DashService: http://glynserver.cms.livjm.ac.uk/DashService/help/operations/SalesGetSales

o7planning. (2018). *JavaFX PieChart Tutorial*. Retrieved March 2018, from o7planning: https://o7planning.org/en/11105/javafx-piechart-tutorial

Oracle. (2014). *Using JavaFX Charts*. Retrieved March 2018, from Oracle Docs: https://docs.oracle.com/javafx/2/charts/pie-chart.htm

Oracle. (2015). *Packages*. Retrieved March 2018, from Oracle Docs: https://docs.oracle.com/javase/8/javafx/api/

Oracle. (2017). *Reduction*. Retrieved March 2018, from The Java Tutorials: https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html

Wright, J. (2017, March 27). *Adding and populating a TableView object using SceneBuilder*. Retrieved March 2018, from YouTube: https://www.youtube.com/watch?v=uh5R7D\_vFto&t=851s&list=LLEN7P2MUaBD23LbxO\_tKDPg&index=2