Section 1: Summary of implemented features.

The web application starts with by displaying a simple menu, which has seven primary functionalities:

1. View the Patient List: This feature presents an alphabetically sorted list of patients by their concatenated first and last names, which are displayed as clickable links. Unlike the original CSV format, where names may contain numbers, the web app ensures names are formatted without numerical characters. Upon clicking a name, detailed patient information is displayed, including full name, birthdate, age (if alive), death date (if applicable), gender, race, ethnicity, marital status(if available), address in concatenated form ("Home address" + "City" + "State" + "Zipcode"), maiden name (if available), Social Security Number (SSN), passport number, and driver's license number. Details are shown based on data availability; if a piece of information is not available, it is omitted.
2. Search: This functionality enables users to search for patients by entering keywords. The results, consisting of patient names containing the search term, are displayed in alphabetical order as clickable links. If no results are found, a "No results found" message is displayed.
3. View Patients Sorted By Age: This feature displays a list of living patients sorted in descending order by age. The names are presented as clickable links, which, when clicked, provide detailed patient information.
4. View Patients Living in the Same City: Users can select a city (which is sorted alphabetically) from a drop-down menu, which then displays a list of patients residing in that city. If multiple patients reside in the same city, the names of patients are presented in alphabetical order through which personalised data is accessible.
5. Save Data to JSON: This functionality allows users to convert and save patient data from CSV to JSON format. Users can access and download the formatted JSON file by clicking the provided link.
6. View Age Distribution Chart: This feature presents a pie chart displaying the age distribution of patients. The age categories on the x-axis are labelled in ascending order (e.g., 0-9, 10-19, etc.).
7. Add New Patient: This functionality allows users to add new patients to the dataset. The application generates a unique ID for each new patient. Certain fields are mandatory, while others are optional. The "Add patient" button is restricted unless mandatory fields are properly filled out. Once filled out, new patients added are seamlessly integrated into the existing functionalities, such as the patient list and search features.

Section 2: Description and evaluation of design and programming process.

I started the project by addressing the requirements methodically, which helped organize my workflow and understand subsequent stages of development. Initially, I structured the classes based on the first set of instructions. However, I made an early error by using generic classes for handling data frames. This was a mistake because it could lead to data inconsistency and type errors since different data types require different handling methods. Realizing this, I adjusted the approach to cater to the specific data types needed. At this point, I understood the importance of designing first, and then modifying my classes to fit that design. Therefore, in the DataFrame class, I utilized a HashMap to store column names and their corresponding data instead of an ArrayList to enable faster searching and data retrieval.

In the design of the web application, I was also careful with how each class and method is structured, particularly in the Model class. Inside this class, methods like removeNumbers and formatString are kept private for specific reasons. These methods are internal utilities used to clean up and format the data before it's either processed further or displayed. For example, removeNumbers strips away numeric characters from strings, which is essential for ensuring that patient names are presented correctly without any irrelevant digits. Similarly, formatString adjusts the text format for better readability. Making these methods private restricts their access to the Model class only, which is intentional because these operations are internal concerns of the data handling process.

In my design, the Model class not only facilitates the interaction within the application but also keeps the controller and view layers separate. This was done in order to align with the MVC architecture. Furthermore, the static feature of the Model ensures that only one instance of this class exists throughout the application's lifecycle, which shows the Singleton pattern. This approach is adopted to prevent multiple instances of the Model class from being created and used.

In the JSONWriter class, I used the ObjectMapper from the Jackson library to convert data into JSON format, The method writeDataFrameToJson takes a DataFrame and an OutputStream, converts the DataFrame into a list of maps which is then wrapped in a single map under the key "patients" to structure the JSON. Furthermore, the writeDataFrameToJson method in the JSONWriter class was made static to allow direct access to JSON writing functionality without the need to instantiate the JSONWriter class each time. This approach follows the principle of utility: since the method does not depend on instance variables and solely focuses on converting a given DataFrame into JSON format, making it static simplifies its use. It enables other parts of the application to convert data frames to JSON format efficiently and consistently, without managing the state or lifecycle of a JSONWriter object.

Finally, the ChartGenerator class generates pie charts based on input data. The generatePieChart method takes a map of data representing the distribution of values and creates a corresponding JFreeChart pie chart. It constructs a DefaultPieDataset from the input data, then uses JFreeChart's ChartFactory to create the pie chart with customizable settings such as title, labels, and appearance.

For my servlet, I focused on creating them tailored to specific functionalities that I mentioned earlier. Each servlet was developed to handle its task efficiently, directly interacting with the ModelFactory for data operations, which streamlined access to the data layer. This direct approach allowed for a clear separation between the servlets' responsibilities and the model without adopting an abstract class hierarchy to generalize shared behaviours as the repetitive code was minimal. For instance, the ViewPatientListServlet and SearchServlet were each designed to fetch and manipulate data in ways that, while unique, followed a consistent pattern of retrieving data from the model, setting it as request attributes, and forwarding to JSPs for rendering. Furthermore, the AddPatientServlet facilitated the addition of new patients by retrieving form data from the request and updating the model accordingly. Meanwhile, the GenerateAgeDistributionPieChartServlet class was created to generate a pie chart. Lastly, the SaveDataFrameToJsonServlet was designed to handle the conversion of DataFrame data to JSON format and enable users to download the data in a structured format.

For my first web app, I kept the design of the JSP pages really simple. Since I was new to web development, I focused on making sure everything worked properly rather than trying to make it look fancy. On pages like addPatient.jsp, I added basic JavaScript to check the forms, like making sure dates were correct, which helped prevent mistakes when entering patient info. This was pretty straightforward but really useful for making the app easier to use. In other parts, like showing the list of patients or their details, I used JSP tags to pull and display data from the server. The main page has a simple menu that links to all the different parts of the app. The search results page is designed to be as straightforward as possible, showing a list of patients based on what the user looked for. Through all this, I learned a lot about how to set up web pages and connect them to data, sticking to the basics to make sure the app did what it was supposed to do. This approach might not have resulted in the most visually stunning website, but it was a great way to learn and ensure that the app was functional and user-friendly.

Overall, I am pleased with how much I've learned and accomplished with this coursework, especially considering it was my first web app. I managed to meet nearly all the project requirements, except "Delete Patient" and "Edit Patient" features. This project helped me to understand Java programming and the application of object-oriented principles in a practical project. I also recognize there was an opportunity for better exception handling to streamline errors. Though initially, I considered using interfaces and abstract classes to refine my servlets' design, I concluded that for the scope and scale of my application, the benefit might not justify the complexity, given the minimal repetition in my code.