MediCheck: A Symptom Based Disease Prediction Application

Software Design Specification

02.05.2025

Şükrü Can Mayda – 150120031
Turgut Köroğlu – 150121065
Nurbetül Çakır – 150121545

Prepared for
CSE3044 Software Engineering Term Project

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this Software Design Specification (DSD) is to outline the structural, behavioral, and architectural design of the MediCheck system—a web-based application that predicts diseases using machine learning. This document acts as a link between the requirements laid out in the Software Requirements Specification (RSD) and the actual implementation phase, providing a clear picture of how the system will be built to meet those needs. It aims to serve as a detailed design reference for developers, testers, and academic supervisors by detailing:

- Architectural structure and modular components
- Data flow and interaction patterns between components
- Design constraints, assumptions, and dependencies
- Component-level behaviors, interfaces, and responsibilities
- Dynamic behavior through sequence and interaction diagrams

This document ensures a shared understanding for everyone of how MediCheck will be developed and maintained, guiding consistent and traceable implementation throughout the project lifecycle.

1.2. Statement of scope

MediCheck is a web-based application designed to provide users with preliminary insights into potential medical conditions based on their self-reported symptoms. The system utilizes machine learning algorithms trained on publicly available datasets to generate intelligent and ranked condition predictions. It does not provide professional medical diagnosis but serves as an informative tool for self-awareness and early-stage guidance.

Major Inputs:

- User-submitted symptoms with an interactive web interface (free-text with selectable list).

Major Processing:

- Standardization of input symptoms (lowercase conversion, synonym mapping).
- Prediction of possible diseases using a pre-trained ML model.
- Ranking predictions by confidence score.
- Optional feedback collection from users.

Major Outputs:

- Ranked list of probable conditions with brief descriptions and confidence levels.
- Informational messages suggests users seek professional medical help if necessary.
- User feedback storage for potential model improvement.

Essential, Desirable and Future Requirements:

Requirement	Priority	Rationale
Symptom input module	Essential	Core input functionality; basis for entire workflow.
Disease prediction module	Essential	Fundamental feature for delivering system purpose.
Prediction result display	Essential	Enables communication of results to the user.
Responsive user interface (web, mobile)	Essential	Required for accessibility across platforms.
Feedback submission module	Desirable	Valuable for improving the ML model over time, but not essential.
User profile management (basic login/history)	Desirable	Useful for personal tracking but not required for initial release.
Admin model update panel	Future	Complex to implement and out of scope for initial version.

This scope is aligned with the **incremental development model**, enabling the delivery of core features in early iterations, while allowing flexibility to add desirable and future enhancements as time and resources permit.

1.3. Software context

MediCheck is developed as a part of an academic term project under the CSE3044 Software Engineering course. It aims to simulate a real-world healthcare support application by combining web development practices with machine learning capabilities.

This application addresses a growing need for AI-supported health information tools, especially for individuals who seek quick insights into their symptoms without visiting a healthcare facility. The project does not aim to replace professional medical care but supports user decision-making through intelligent symptom-based predictions. The software distinguishes itself from traditional rule-based systems (e.g., WebMD) by making a trainable ML model, which allows MediCheck to improve over time as new datasets are introduced. It uses open-source technologies and publicly available datasets to remain cost-effective and adaptable within an educational setting.

MediCheck is not intended for commercial deployment or clinical use. Instead, it serves as a proof-of-concept system to demonstrate how modern data-driven software can be designed, implemented, and evaluated for performance, usability, and maintainability within the constraints of an academic environment. The product is delivered as a independent web application with a modular backend–frontend architecture and no external third-party service dependencies beyond open-source libraries.

1.4. Major constraints

The design and implementation of the MediCheck system are subject to several academic, technical, and resource-related constraints, the given constraints on RSD mostly did not involve here:

- **Open-source**: Due to the non-commercial and educational nature of the project, all development tools, libraries, and datasets must be open-source and free to use. This affects choices such as using Flask, Scikit-learn, and public medical datasets (Kaggle).
- **No Cloud Services or Paid APIs**: The system must function without reliance on paid third-party APIs or commercial cloud services. All data processing, including model prediction, must occur on a local server for beginning.
- **Platform Independence**: MediCheck must run smoothly on all modern browsers and across operating systems (Windows, macOS, Linux) without needing custom installations. This limits the use of platform-specific features and requires responsive web design.
- **Limited Computational Resources**: Machine learning models must be lightweight and optimized for fast performance, as the system will be hosted on standard academic-grade infrastructure.
- **No Storage of Sensitive Medical Data**: The system must comply with basic privacy guidelines, meaning no personally identifiable health information or long-term medical records can be collected or stored. Only session-level data may be retained temporarily.
- **Development Timeline**: All planned features must be developed, tested, and delivered within a single academic semester. This restricts the project scope to essential and a few desirable features only.
- **User Assumptions**: It is assumed that users will access the system via modern browsers and have basic familiarity with symptom terminology. Misinterpretation or vague symptom entries may affect prediction quality.

1.5. Definitions

• **Symptom**: A physical or mental indication of a medical condition, as reported by the user.

• **Prediction**: A list of possible medical conditions generated by the ML model based on input symptoms.

• **User Session**: A temporary interaction between a user and the system, lasting until inactivity or logout.

• **Feedback**: User-provided evaluation of the accuracy or relevance of prediction results.

• **Model Update**: The process of retraining the machine learning model with new or extended datasets.

1.6. Acronyms and Abbreviations

Acronym/Abbreviation	Description
SRS	Software Requirements Specification
SDS/DSD	Software Design Specification / Document
ML	Machine Learning
UI	User Interface
UX	User Experience
CSV	Comma-Separated Values (data file format)
API	Application Programming Interface
HTTP/S	HyperText Transfer Protocol / Secure
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
JS	JavaScript

1.7. References

- IEEE Std 830-1998 – IEEE Recommended Practice for Software Requirements Specifications
- Scikit-learn Documentation – https://scikit-learn.org
- WebMD Symptom Checker – https://symptoms.webmd.com
- Mayo Clinic Symptom Index – https://www.mayoclinic.org/symptoms
- Kaggle Disease Symptom Dataset https://www.kaggle.com/datasets/dhivyeshrk/diseases-and-symptoms-dataset
- Lecture Notes https://mimoza.marmara.edu.tr/~mehmet.kaya/Courses/CSE3044/index.html
- *Software Engineering: A Practitioner's Approach* 8th Edition, Pressman & Maxim, 2014
- Software Engineering, Ian Sommerville, 8th edition, Addison-Wesley, 2007

2. Design Consideration

2.1. Design Assumptions and Dependencies

This section outlines the key assumptions and external factors that affects the design of the MediCheck system. The following are categorized under common design dependency themes:

- Related Software and Hardware
  - The system depends on Python-based libraries such as Scikit-learn (for ML modeling), Pandas and NumPy (for data preprocessing).
  - Backend development assumes the use of Flask, a lightweight Python web framework.
  - No dedicated hardware is required. Only necessary CPU-based computation on standard academic-grade machines and personal computers.

- Operating Systems
  - The system is designed to be platform-independent, working reliably on Linux, Windows and macOS.
  - Web access assumes server-side hosting using Python, JS and HTML.

- End-User Characteristics
  - The target users are non-technical individuals with basic web browsing skills.
  - Users are assumed to have ability to describe common symptoms using text and access to modern browsers like Chrome, Firefox, or Edge.
  - The system assumes basic health literacy, users can recognize and report observable symptoms which are optional.

- Possible and Probable Changes in Functionality
  - It is assumed that the system may require periodic model retraining as new datasets are introduced.
  - Future integration of multilingual support and admin dashboard for uploading new ML models

2.2. General Constraints

The following constraints affect the design and implementation of the MediCheck system. Environmental, technical, and educational considerations must be acknowledged to ensure a realistic and maintainable software design.

- Hardware or Software Environment

    - The system must operate on standard desktop or laptop hardware without requiring specialized devices.
    - The backend stack is restricted to Python and open-source libraries such as Flask, Scikit-learn, and Pandas.
    - No proprietary software is allowed due to licensing and academic constraints.
    - **Impact:** Limits the use of heavy or complex ML models and encourages lightweight, efficient design.

- End-User Environment

    - Users are assumed to access the system via modern web browsers (Chrome, Firefox, Edge).
    - The system must be fully functional on desktop devices.
    - **Impact:** Requires a responsive front-end UI and compatibility with HTML5/CSS3 standards.

- Availability or Volatility of Resources

    - The development team is limited to academic contributors with part-time availability.
    - No dedicated IT infrastructure is provided and there is no need.
    - **Impact:** All deployment and testing must occur on local or minimal institutional servers, impacting scalability and load testing.

- Standards Compliance

    - The frontend must comply with web development standards.
    - All backend communication must use secure HTTPS protocol.
    - **Impact**: Enforces semantic HTML, accessibility rules, and encryption practices.

- Interface/protocol Requirements

    - All client-server communication must occur via HTTP/HTTPS.
    - **Impact**: Simplifies API design but limits real-time interaction.

- Data Repository and Distribution Requirements

    - Initially, all training and inference data are stored in CSV files.
    - Feedback and user session data may later be used by SQLite or MongoDB.
    - **Impact**: Reduces infrastructure complexity but imposes data size and performance limitations.

- Security Requirements

    - No personally identifiable information (PII) is collected.
    - User inputs must be validated and sanitized to prevent injection problems.
    - Session data must expire after 15 minutes of inactivity.
    - **Impact**: Enhances privacy compliance but limits personalization features.

- Memory and Other Capacity Limitations

    - The system must function under limited memory and CPU usage.
    - **Impact**: Encourages efficient algorithms and avoidance of large ML models.

- Performance Requirements

    - Symptom input to prediction output must occur within 3–5 seconds under normal usage.
    - The system should support at least 10 concurrent users during demo periods in the very beginning.
    - **Impact**: Requires optimization of data flow and model inference time.

- Network Communications

    - As a web-based app, a stable internet connection is essential for both users and the server.
    - No offline or partially connected mode is supported.
    - **Impact**: Limits usability in low-connectivity environments.

- Verification and Validation Requirements (Testing)

    - Manual testing will be used to validate major modules (symptom input, prediction, result display).
    - Automated testing is optional due to the time and compatibility of teammates.
    - **Impact**: Increases manual testing effort and potential maintenance cost.

- Reliability

    - In the event of backend failure, the system shall return a message and not break the user session.
    - **Impact:** Reliability ensures consistent and accurate predictions that users can trust.

- Maintainability

    - Each module should be developed as an independent component to allow for modular updates.
    - **Impact:** Maintainability enables the MediCheck system to be easily updated with new symptoms or models.

2.3. System Environment

The MediCheck system is developed and deployed within a constrained academic environment using free and open-source technologies. The development and runtime environment are defined as follows:

- **Software Tools**
  - Python: For backend logic and machine learning.
  - JavaScript: For frontend logic.
  - HTML: For UI interface.
  - Flask: Python web framework used for routing, request handling and API.
  - Scikit-learn: Python library for classification model.
  - Pandas: Python library for data handling.
  - NumPy: Python library for numerical operations.
  - CSV: Used to load dataset into machine learning.
  - Git/GitHub: For code collaboration and issue tracking.
  - Operating Systems: Windows 10+, or Linux 22+ recommended.
  - Database: Used to make new analyses with new inputs.
  - Python IDE: Visual Studio Code use for code implementation.

- **Hardware Tools**
  - CPU: System needs a CPU to run the application.
  - Internet Connection: It needs a stable internet connection.
  - Web Server: Used for UI interface analysis.

2.4. Development Methods

The software design for MediCheck follows the **Incremental Development Model**, an approach where the system is developed and delivered in small, manageable parts, incrementals. Each increment focuses on a specific subset of functionality before expanding into auxiliary features such as feedback collection and user profiles. This model is formally defined in software engineering literature as a risk-managed, evolutionary strategy that improves flexibility and user feedback incorporation over time.

We also considered the following alternatives and rejected them for those reasons:

Waterfall Model: Rejected due to its rigid phase boundaries and lack of early testing opportunities, which are impractical in a semester-based project with evolving needs.

Agile Approach: While Agile offers strong team collaboration and iterative cycles, its formal ceremonies (daily stand-ups, sprints, retrospectives) were considered too complex to implement fully in a student team with varying availability.

Why Incremental Model was selected:

- Aligns well with semester constraints.
- Allows for early delivery of functional prototypes.
- Supports gradual integration of modules without affecting existing functionality.
- Encourages continuous feedback from academic advisors and peers.

3. Architectural and component-level design

The MediCheck application is structured using a layered modular architecture that separates system concerns across three main functional domains: user interaction, application logic, and prediction. This architectural design supports modularity, reusability, and independent component evolution.

3.1. System Structure

MediCheck's system structure is designed to provide a modular, maintainable, and scalable architecture that supports machine learning-based disease prediction based on user-entered symptoms. The architecture follows a layered design pattern and is composed of the following main layers:

- **Presentation Layer**

  The presentation layer is responsible for interacting with the end users and collecting their input through a user-friendly interface. In MediCheck, this includes:
  - HTML forms for symptom input (home.html).
  - Result display page (result.html).
  - JavaScript codes (script.js, auto_correct.js) that handles user interaction enhancements such as autocomplete.

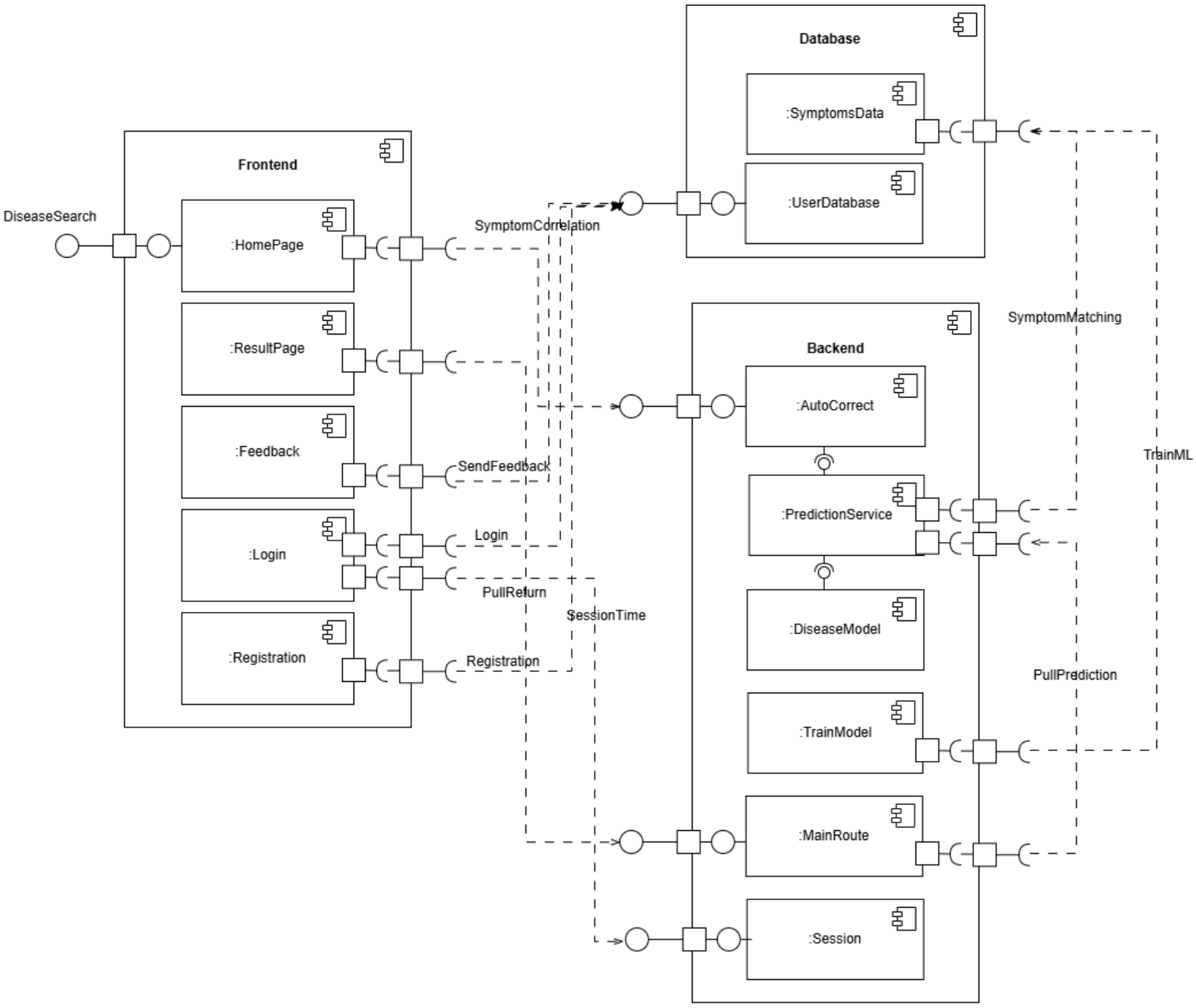
- **Application Layer**
  - The application layer manages the application's workflow, including handling user input, routing, and coordinating the logic between other layers. In MediCheck, this includes:
  - Flask routes in routes (routes/main_routes.py), which receive and respond.
  - Request handling and response rendering.
  - Error handling and redirection.
  - This layer defines how the system behaves when a user submits a form, navigates to a page.

- **Data Management Layer**

  This layer manages the data of the system. In MediCheck, this includes:

  - Loading and parsing symptom definitions from symptoms.
  - Storing and retrieving the trained ML model from disease model.
  - Processing and vectorizing user input symptoms for model inference.
  - This layer ensures that data is consistently structured and accessible to the application and ML layers, enabling accurate predictions and future extensibility.

3.1.1. Architecture diagram



3.2.1 Description for Component PredictionService

3.2.1.1 Processing narrative (PSPEC) for component PredictionService

PredictionService component is responsible for handling the interaction between the application logic and the machine learning model. It receives cleaned user input (symptoms), converts it into a vector format suitable for the model, invokes the trained model, and returns ranked disease predictions with confidence scores.

It ensures that model-related logic is isolated from the routing and presentation layers, enabling better maintainability and testability.

3.2.1.2 Component PredictionService interface description.

Input:

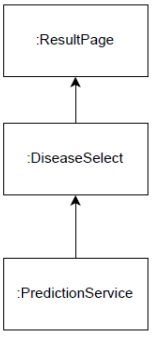
- symptom_list: A list of standardized symptoms received from the frontend and controller layer.

Output:

- Ranked list of dictionaries.

3.2.1.3 Component PredictionService processing detail

3.2.1.3.1 Design Class hierarchy for component PredictionService



3.2.1.3.2 Restrictions/limitations for component PredictionService

- Only supports models trained on symbipredict_2022.csv dataset.
- Cannot dynamically switch models without restarting backend.

3.2.1.3.3 Performance issues for component PredictionService

- Inference time must remain in 2 seconds.
- Cannot dynamically switch models without restarting backend.

3.2.1.3.4 Design constraints for component PredictionService

- It must work with .pkl format models trained using Scikit-learn.
- It must remain decoupled from UI logic.

3.2.1.3.5 Processing detail for each operation of component PredictionService

- predict_disease_from_symptoms(symptom_list)
  - Receives a list of symptoms from the user input.
  - Normalizes the symptoms.
  - Compares input symptoms with the model's feature list.
  - Generates a binary feature total symptom count, marking 1 where a symptom matches.
  - Converts the data to a DataFrame.
  - Loads and invokes the trained model (disease_model.pkl) to predict the associated disease.
  - Returns the predicted disease name as a string.

3.2.2 Description for Component TrainModel

3.2.2.1 Processing narrative (PSPEC) for component TrainModel

TrainModel component is responsible for training and saving the machine learning model used by MediCheck. It loads a structured symptom-disease dataset, preprocesses the data, trains a classification model, evaluates its performance, and serializes the model into a .pkl file for later use in prediction. This component is typically executed manually by a system maintainer or developer whenever the dataset is updated or improved.

3.2.2.2 Component TrainModel interface description.

Input:

- symbipredict_2022.csv: Dataset file

Output:

- disease_model.pkl: Trained model file
- Console output with performance metrics (accuracy, F1-score)

3.2.2.3 Component TrainModel processing detail

3.2.2.3.1 Design Class hierarchy for component TrainModel

- Implemented as a procedural script, not class based.
- All steps are organized in sequential function calls.

3.2.2.3.2 Restrictions/limitations for component TrainModel

- Dataset format must be strictly compatible with expected schema (binary symptom features + disease labels).
- Does not currently support hyperparameter tuning or cross-validation.

3.2.2.3.3 Performance issues for component TrainModel

- Training time depends on dataset size and model type.
- Current implementation trains in-memory; large datasets may lead to memory usage issues.

3.2.2.3.4 Design constraints for component TrainModel

- Uses only Scikit-learn for compatibility with prediction_service.py.
- Must output model in .pkl format for downstream use.

3.2.2.3.5 Processing detail for each operation of component TrainModel

- load_dataset(filepath)
  - Loads symptom-disease data from a CSV file using Pandas.
- preprocess_data(df)
  - Converts disease labels into categorical format, checks missing values.
- train_model(X, y)
  - Trains a RandomForestClassifier (or similar classifier).
  - Splits data into train test and prints evaluation metrics.
- save_model(model, filename)
  - Serializes the trained model using joblib or pickle to disease_model.pkl.

3.2.3 Description for Component MainRoute

3.2.3.1 Processing narrative (PSPEC) for component MainRoute

The MainRoute defines the routing logic of the MediCheck web application using the Flask framework. It manages HTTP requests from the frontend and coordinates interactions between the user interface, symptom processing, and prediction modules. This component handles:

- Rendering UI pages (home.html, result.html)
- Receiving user symptom input via POST requests
- Calling backend services such as PredictionService
- Managing optional feedback submission and error messaging

3.2.3.2 Component MainRoute interface description.

Input:

- GET /: Request to load the symptom input page.
- POST /predict: Receives symptoms from the user via a web form.
- POST /feedback: (optional) Receives user feedback about predictions.

Output:

- Rendered HTML pages (home.html, result.html)
- Redirects or confirmation messages upon successful form submission

3.2.3.3 Component MainRoute processing detail

3.2.3.3.1 Design Class hierarchy for component MainRoute

- This is a functional Flask blueprint and routing module, not class based.
- It defines route handlers using app.route decorators.

3.2.3.3.2 Restrictions/limitations for component MainRoute

- Routes are stateless; session-based tracking is minimal.
- No real-time interaction.

3.2.3.3.3 Performance issues for component MainRoute

- Prediction requests depend on backend model performance.
- Slow client-side input (e.g., large symptom lists) can delay POST submission.

3.2.3.3.4 Design constraints for component MainRoute

- Flask-specific routing conventions must be followed.
- Tightly coupled with HTML form field names (requires form validation alignment).

3.2.3.3.5 Processing detail for each operation of component MainRoute

- **GET/**

- Loads home.html page with symptom input field.

- **POST/predict**

- Retrieves symptoms from the form.
- Passes them to PredictionService.
- Receives and formats prediction results.
- Renders result.html with prediction output.

- **POST/feedback**

- Collects optional feedback (rating/comment).
- Stores via FeedbackManager.
- Returns a thank-you message or reloads result page.

3.2.4 Description for Component AutoCorrect

3.2.4.1 Processing narrative (PSPEC) for component AutoCorrect

AutoCorrect component enhances the user experience by providing real-time suggestions and basic typo correction during symptom entry. It uses JavaScript to compare user input against the list of valid symptoms from symptoms.json and suggests the closest matches. This functionality helps reduce errors caused by inconsistent or misspelled symptom names, increasing the likelihood of valid predictions from the ML model.

3.2.4.2 Component AutoCorrect interface description.

Input:

- User-typed text in the symptom input field (e.g., "headakhe").
- List of valid symptoms loaded from symptoms.json.

Output:

- Autocomplete dropdown of closest matching symptoms.
- Optionally replaces or highlights the corrected term.

3.2.4.3 Component AutoCorrect processing detail

3.2.4.3.1 Design Class hierarchy for component AutoCorrect

- Not a class-based component.

- Purely functional JavaScript code using event-driven architecture.

3.2.4.3.2 Restrictions/limitations for component AutoCorrect

- Basic string similarity matching only (e.g., Levenshtein distance or simple substring).

- No advanced NLP or spell-check algorithms are implemented.

3.2.4.3.3 Performance issues for component AutoCorrect

- Performance may degrade slightly if the symptom list becomes large.

- Executes entirely in the browser, so performance depends on client device.

3.2.4.3.4 Design constraints for component AutoCorrect

- Must work on all major browsers (Chrome, Firefox, Edge, Safari).

- Should not interfere with input unless a match is confidently suggested.

3.2.4.3.5 Processing detail for each operation of component AutoCorrect

- **loadSymptoms()**

- Uses fetchsymptoms.json to load the list of valid symptoms asynchronously.

- **onUserInput(event)**

- Captures current input value as the user types.
- Calls getClosestMatches(inputValue) to generate suggestions.

- **getClosestMatches(input)**

- Compares input against symptoms using basic string similarity.
- Returns top 3–5 suggestions to be displayed in a dropdown.

- **suggestionClickHandler()**

- If the user selects a suggestion, it replaces the input field's value.

3.2.5 Description for Component DiseaseModel

3.2.5.1 Processing narrative (PSPEC) for component DiseaseModel

DiseaseModel contains the serialized version of a trained machine learning model with a RandomForestClassifier used by the MediCheck system to make disease predictions based on user-submitted symptoms.

This component is not a code module but is loaded and used by the predictionService component at runtime. It allows the system to perform fast and consistent predictions without retraining the model each time. The model is trained offline via train_model.py and saved using pickle or joblib.

3.2.5.2 Component DiseaseModel interface description.

Input:

- Binary symptom vector ([0, 1, 0, 1, ...]) representing the presence of the symptoms.

Output:

- List of predicted diseases with associated confidence scores.

3.2.5.3 Component DiseaseModel processing detail

3.2.5.3.1 Design Class hierarchy for component DiseaseModel

- Not a class or module. It is a trained instance of a scikit-learn classifier stored as a binary file.

- Loaded using joblib.load() or pickle.load() into memory.

3.2.5.3.2 Restrictions/limitations for component DiseaseModel

- Must be retrained using the same symptom feature order and structure.
- Incompatible with models trained using different libraries (e.g., TensorFlow, PyTorch).
- Cannot be updated while the app is running unless hot-swapping logic is added.

3.2.5.3.3 Performance issues for component DiseaseModel

- Very fast inference (<1 second) when preloaded into memory.
- File size may grow with complex models or feature-rich datasets.

3.2.5.3.4 Design constraints for component DiseaseModel

- Model format must remain `.pkl` and follow scikit-learn conventions.
- Prediction output must be standardized.

3.2.5.3.5 Processing detail for each operation of component DiseaseModel

- **load_model(path)**

- Reads the .pkl file and loads the model object into RAM.
- Typically called once during Flask app startup.

- **predict(input_vector)**

- Accepts a binary vector as input.
- Returns ranked list of diseases and probabilities.

3.2.6 Description for Component SymptomsData

3.2.6.1 Processing narrative (PSPEC) for component SymptomsData

The symptomsData file is a static data source that stores a structured list of medical symptoms used throughout the MediCheck system. It supports:

- Autocomplete functionality on the frontend (via JavaScript)
- Validation and mapping of symptom entries on the backend
- Consistency in symptom naming between model training and inference

3.2.6.2 Component SymptomsData interface description.

Input:

- No direct inputs. The file is loaded by both frontend and backend components at runtime.

Output:

- A list or dictionary of symptoms in JSON format (e.g., ["itching", "headache", "vomiting", ...]).

3.2.6.3 Component SymptomsData processing detail

3.2.6.3.1 Design Class hierarchy for component SymptomsData

- Not an object-oriented component, JSON is data file.
- Loaded as a dictionary using json.load() in Python.

3.2.6.3.2 Restrictions/limitations for component SymptomsData

- Manually maintained; any type of missing entries can break model input consistency.
- No schema validation is performed at runtime.

3.2.6.3.3 Performance issues for component SymptomsData

- It is negligible, the file is small and loaded once in memory at runtime.

3.2.6.3.4 Design constraints for component SymptomsData

- Must match symptom names used in the model training dataset.
- Should remain lightweight for fast client-side loading.

3.2.6.3.5 Processing detail for each operation of component SymptomsData

- **Frontend Usage**

- Loaded via JavaScript (fetch('/static/symptoms.json')).
- Used to populate a dropdown or autocomplete field in homePage.

- **Backend Usage**

- Loaded during symptom vectorization
- Used to verify if user input exists in model's vocabulary

3.2.7 Description for Component HomePage

3.2.7.1 Processing narrative (PSPEC) for component HomePage

The homePage component is the main entry point of the MediCheck system where users input their symptoms. It includes a symptom input form that interacts with JavaScript to support autocomplete functionality and sends the entered symptoms to the backend for processing. This page is designed for ease of use and responsiveness on desktop devices.

3.2.7.2 Component HomePage interface description.

Input:

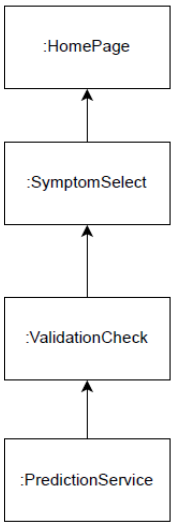
- List of available symptoms (symptoms.json)
- User input text.
- Optional UI hints or error messages from the backend

Output:

- Symptom input data submitted with POST/predict.

3.2.7.3 Component HomePage processing detail

3.2.7.3.1 Design Class hierarchy for component HomePage



3.2.7.3.2 Restrictions/limitations for component HomePage

- No real-time validation unless handled in JS.
- Dependent on correct loading of symptoms.json.

3.2.7.3.3 Performance issues for component HomePage

- Minor lag may occur if JS-based autocomplete is slow.

3.2.7.3.4 Design constraints for component HomePage

- Input field name must match backend field key.
- Page must adapt to various screen sizes using Bootstrap classes.

3.2.7.3.5 Processing detail for each operation of component HomePage

- Loads symptom list from symptoms.json via JS.
- Populates input dropdown or textbox with suggestions.
- On submit, sends POST request with user symptoms to /predict.

3.2.8 Description for Component ResultPage

3.2.8.1 Processing narrative (PSPEC) for component ResultPage

The ResultPage component displays the prediction results returned by the backend. It shows a ranked list of potential diseases with confidence scores and may include a feedback form allowing users to rate the result's relevance. This page focuses on clarity and user trust by presenting results in a clean, readable format.

3.2.8.2 Component ResultPage interface description.

Input:

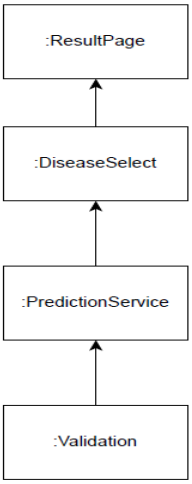
- Disease prediction results (from backend)
- Optional feedback form flags.

Output:

- Displays results on-screen
- Sends feedback via POST/feedback (optional)

3.2.8.3 Component ResultPage processing detail

3.2.8.3.1 Design Class hierarchy for component ResultPage



3.2.8.3.2 Restrictions/limitations for component ResultPage

- Assumes correct structure of prediction_result context variable.

3.2.8.3.3 Performance issues for component ResultPage

- Rendering a large number of predictions may break UI.

3.2.8.3.4 Design constraints for component ResultPage

- Feedback form must submit using POST with matching field names.
- Result rendering must remain responsive and minimalistic.

3.2.8.3.5 Processing detail for each operation of component ResultPage

- Receives prediction results from Flask backend.
- Displays them in a ranked list or table.
- Collects optional user feedback and posts it to /feedback.

3.2.9 Description for Component Feedback

3.2.9.1 Processing narrative (PSPEC) for component Feedback

The Feedback component is responsible for handling user feedback related to the prediction results. After receiving disease suggestions from the system, users may rate the accuracy or relevance of the prediction and optionally submit a comment. These feedback entries are stored for future evaluation or model improvement analysis. This component ensures that feedback data is collected in a structured and secure manner, separate from prediction logic, and optionally stored in a lightweight database or as a log file.

3.2.9.2 Component Feedback interface description.

Input:

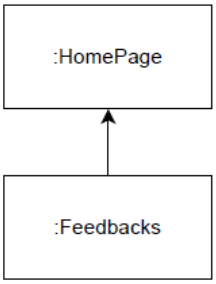
- user_id or session ID
- predicted_disease_list
- feedback_score
- user_comment

Output:

- Confirmation of successful feedback submission
- Optional log entry or database insert

3.2.9.3 Component Feedback processing detail

3.2.9.3.1 Design Class hierarchy for component Feedback



3.2.9.3.2 Restrictions/limitations for component Feedback

- No real-time model adaptation based on feedback
- Feedback is anonymous unless user login is enabled
- Only accepts feedback after a valid prediction session

3.2.9.3.3 Performance issues for component ResultPage

- Minimal performance impact due to small data size
- Slight latency if writing to remote or slow storage backends

3.2.9.3.4 Design constraints for component ResultPage

- Must not expose user input publicly or log sensitive data
- Data schema must be consistent for later analysis

3.2.9.3.5 Processing detail for each operation of component ResultPage

- **collect_feedback(request_data)**
  - Extracts prediction feedback and comment from POST request
  - Validates fields and formats data structure
- **save_feedback_to_file(data)**
  - Appends the feedback as a line in a JSON file
- **save_feedback_to_db(data)**
  - Inserts structured data into a MongoDB/SQLite collection if used
  - Ensures data integrity and indexing

3.2.10 Description for Component Registration

3.2.10.1 Processing narrative (PSPEC) for component Registration

The Registration component allows new users to create accounts within the MediCheck system. It captures personal credentials such as name, email, and password through a registration form and stores this information securely in the backend database. The component performs input validation, password strength checking, and uniqueness enforcement for usernames and email addresses. It ensures that account creation is safe, consistent, and compliant with basic authentication standards.

3.2.10.2 Component Registration interface description.

Input:

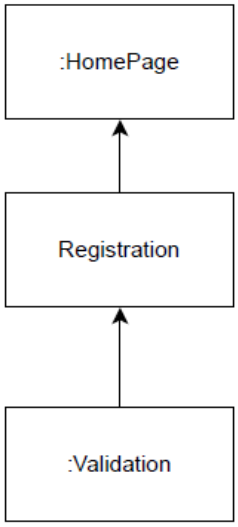
- username (string)
- email (string)
- password (string)

Output:

- Success message and redirection to login page
- Error messages for invalid or duplicate credentials

3.2.10.3 Component Registration processing detail

3.2.10.3.1 Design Class hierarchy for component Registration



3.2.10.3.2 Restrictions/limitations for component Registration

- Email and username must be unique in the system
- Password must meet defined complexity rules
- Registration is blocked if any required field is missing or malformed

3.2.10.3.3 Performance issues for component Registration

- Minimal performance overhead; only a single DB insert operation per request
- May slow down slightly if email uniqueness is checked inefficiently in large datasets

3.2.10.3.4 Design constraints for component Registration

- Passwords must be hashed (e.g., using bcrypt or hashlib)
- Inputs must be sanitized to prevent injection or XSS attacks
- Must follow form-validation feedback practices for UX clarity

3.2.10.3.5 Processing detail for each operation of component Registration

- **validate_input(form_data)**

- Checks for missing fields and validates email format and password strength

- **check_uniqueness(email, username)**

- Queries the database to ensure no duplicate entries exist

- **hash_password(password)**

- Hashes the password using a secure algorithm before storage

- **register_user(data)**

- Inserts the new user into the database with hashed credentials and timestamp

3.2.11 Description for Component Login

3.2.11.1 Processing narrative (PSPEC) for component Login

The Login component enables registered users to securely log in to the MediCheck system. It accepts credentials (email and password), verifies them against stored user records, and establishes a user session upon successful authentication. The component ensures secure handling of login data, implements password hashing comparison, and restricts access for incorrect or invalid credentials. Upon successful login, it redirects the user to the symptom input page or their dashboard.

3.2.11.2 Component Login interface description.

Input:

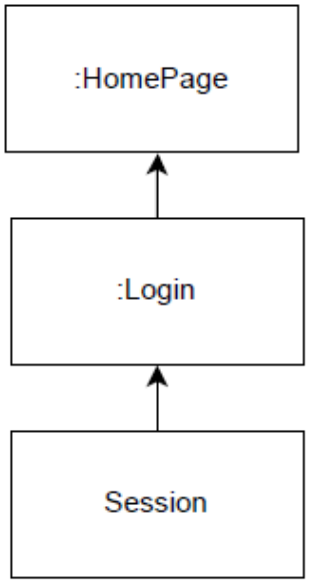
- email or username (string)
- password (string)

Output:

- On success: user session created, redirected to main page
- On failure: error message shown, no session created

3.2.11.3 Component Login processing detail

3.2.11.3.1 Design Class hierarchy for component Login



3.2.11.3.2 Restrictions/limitations for component Login

- Only users with correctly entered credentials can log in
- Password must be matched using hash comparison
- No account lockout or captcha support (due to academic scope)

3.2.11.3.3 Performance issues for component Login

- Minimal performance cost (single DB read + hash check)
- Slower if user table grows very large and lacks indexing

3.2.11.3.4 Design constraints for component Login

- Passwords must never be stored or compared in plaintext
- User sessions must be protected using secure cookies or tokens

3.2.11.3.5 Processing detail for each operation of component Login

- **validate_login_input(email, password)**

- Checks for empty fields and valid email format

- **retrieve_user_record(email)**

- Queries the database for a user with the given email

- **verify_password(input_pw, hashed_pw)**

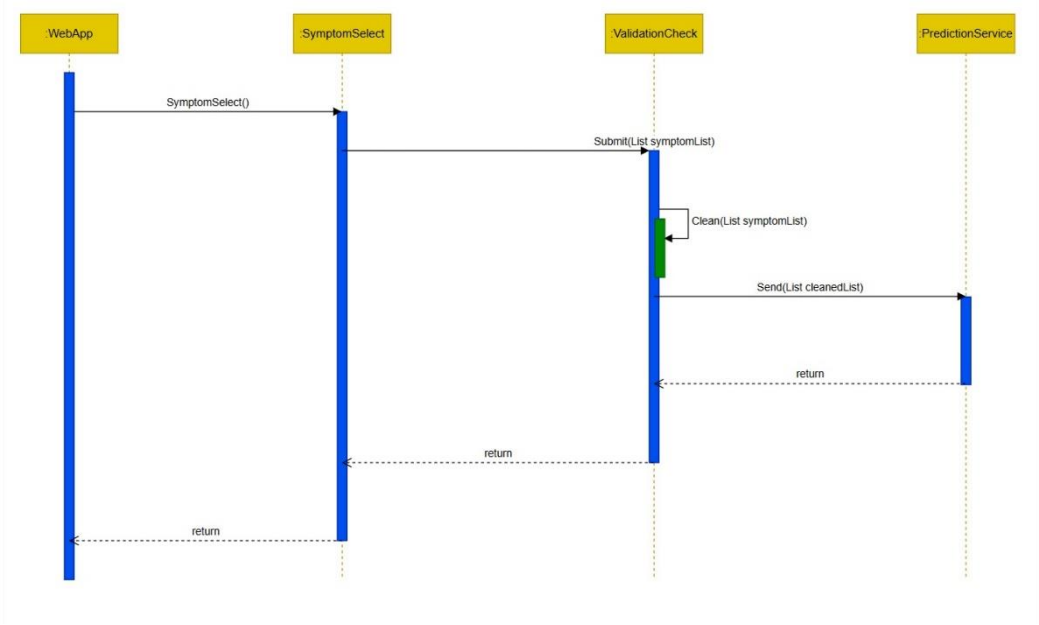
- Uses a hash comparison function (e.g., bcrypt.checkpw)

- **create_user_session(user_id)**

- Stores user info in the session object (Flask session or JWT)

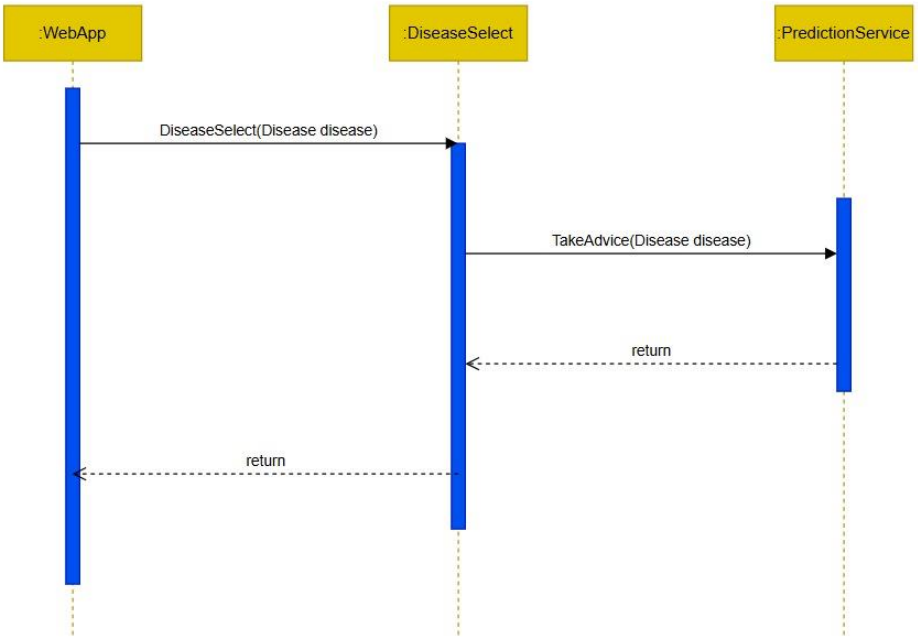
3.3.1 Dynamic Behavior for Component HomePage

3.3.1.1 Interaction Diagrams



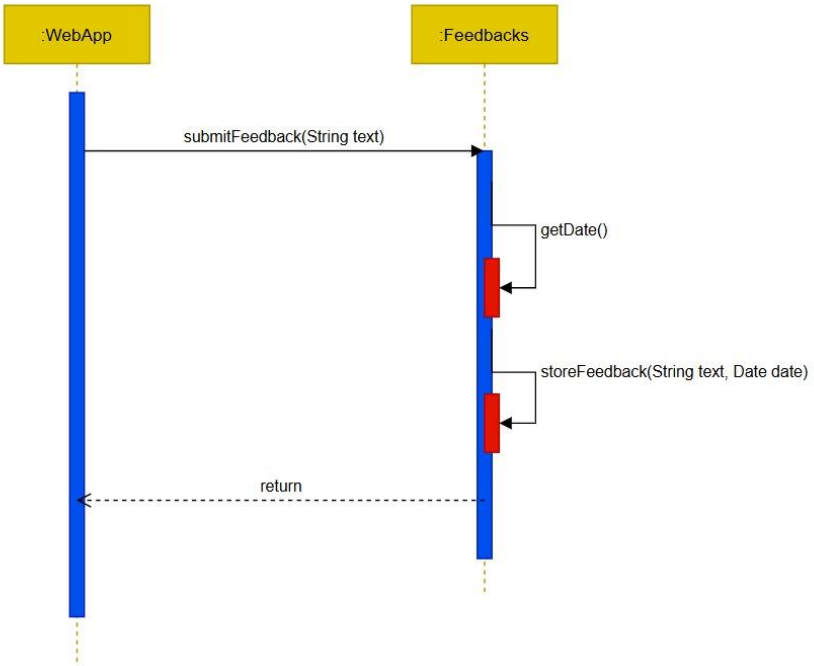
3.3.2 Dynamic Behavior for Component ResultPage

3.3.2.1 Interaction Diagrams



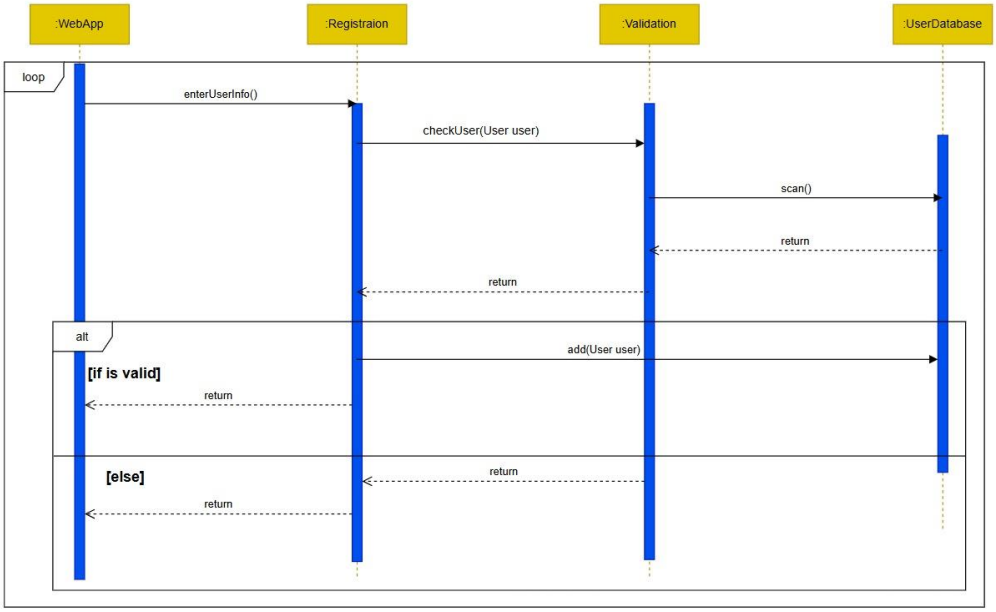
3.3.3 Dynamic Behavior for Component Feedback

3.3.3.1 Interaction Diagrams



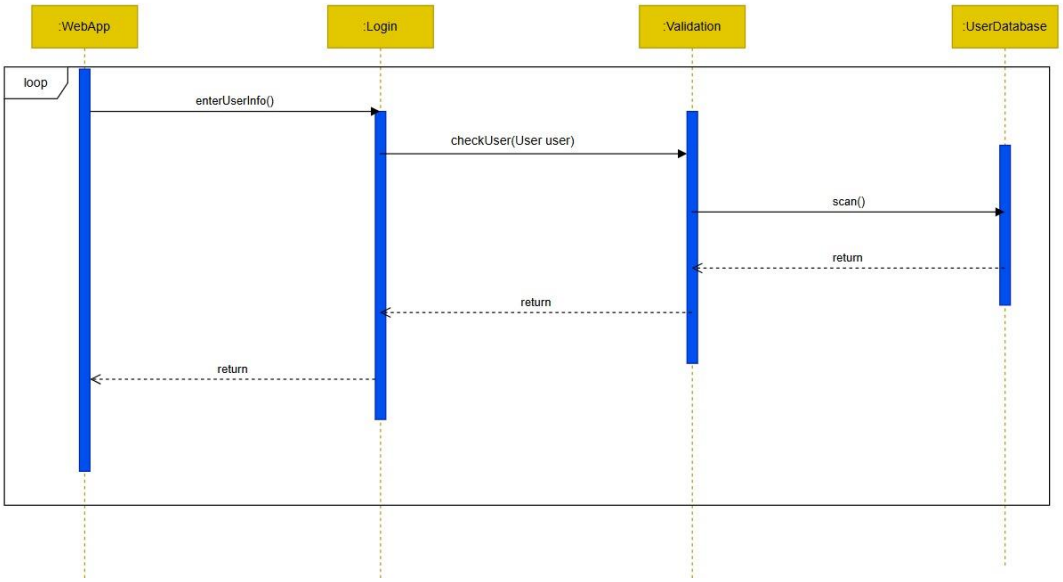
3.3.4 Dynamic Behavior for Component Registration

3.3.4.1 Interaction Diagrams



3.3.5 Dynamic Behavior for Component Login

3.3.5.1 Interaction Diagrams



4. Restrictions, Limitations, and Constraints

The following restrictions and limitations have a significant impact on the design and implementation of the MediCheck system:

- **Symptom Consistency**: Users must enter symptoms exactly as defined in the system. Typos or unrecognized entries may cause the model to fail or return inaccurate results.
- **Model Immutability at Runtime**: Once the ML model is loaded into memory (`disease_model.pkl`), it cannot be updated during active sessions. Any updates require a manual restart of the backend.
- **No Offline Mode**: MediCheck is a fully web-based system and requires a stable internet connection. There is no offline or partially connected functionality.
- **No Real Medical Diagnosis**: For ethical and legal reasons, the system only provides symptom-based predictions and explicitly does not replace professional diagnosis or treatment advice.
- **No Real-Time Learning**: Feedback submitted by users is collected but not applied in real-time to the prediction model. Model retraining must be done manually.
- **Limited Security Scope**: As an academic project, advanced security measures (e.g., multi-factor authentication) are not implemented. Basic hashing and session management are used.
- **Anonymous Feedback**: Feedback is optionally anonymous and may not be associated with user accounts unless login is enabled.
- **Frontend Limitations**: The system does not implement a Single Page Application (SPA); all pages reload on action. JavaScript is used for enhancing usability only.
- **Device Scope**: While the frontend is responsive, the application is not a native mobile app and does not utilize device-specific features such as push notifications.
- **Data Sensitivity Restrictions**: The system does not collect or store any personally identifiable health information (PIHI), limiting long-term personalization features.
- **Academic Time Constraint**: All development, testing, and deployment must be completed within the scope of one academic semester, limiting the number of features and depth of testing.

5. Conclusion

This Software Design Specification (SDS) document provides a comprehensive and structured overview of the MediCheck system's architecture, component design, and behavioral flow. MediCheck is a machine-learning-based web application intended to provide preliminary predictions of possible medical conditions based on user-entered symptoms. The system is built using modular components that allow for ease of maintenance and extension. Emphasis has been placed on usability, privacy, and responsiveness, ensuring that the application can function efficiently on standard academic infrastructure.

The use of the Incremental Development Model has enabled the team to prioritize essential features and deliver a minimum viable product early, while leaving room for desirable enhancements and future functionality. Though the system is not intended for clinical deployment, it serves as a valuable academic demonstration of applied software engineering practices in a healthcare-related use case. With proper feedback collection and future model updates, MediCheck can continue evolving into a more comprehensive and intelligent diagnostic assistant in educational or research settings.