# MyWatchCollectionHub Solution Design

## Overview

This document describes the implementation strategy for **MyWatchCollectionHub**, an application for cataloguing a personal watch collection. The goal is to automatically ingest images of watches, deduplicate them, identify each unique watch (brand, model, year, etc.), enrich the data with resale values and references, store the records in a database and expose them via a web UI. The project will be implemented in **Java** using **Spring Boot** for the backend and a modern JavaScript framework (e.g., React) for the frontend. Advanced tasks like image clustering and recognition will be delegated to Python microservices that employ deep‑learning libraries.

## High‑Level Implementation Plan

The application will be built iteratively. Each stage focuses on a key feature set to ensure incremental progress and maintainability. The table below summarises the major features and corresponding tasks.

| Feature | Tasks |
| --- | --- |
| **Image Ingestion & Deduplication** | \* Allow users to upload multiple images via the frontend.\* Generate image embeddings using a pre‑trained convolutional network (e.g., ResNet). Cluster embeddings with DBSCAN or hierarchical clustering to group images of the same watch. Each cluster becomes a unique watch record. |
| **Watch Detection & Classification** | \* Use an object‑detection model such as **YOLO v7/v8** to detect watches in images. YOLO is a single‑shot detector known for its speed and accuracy【31584787726438†L67-L79】.\* Crop detected watch regions and classify brand/model with a fine‑tuned classifier (e.g., ResNet, EfficientNet). |
| **OCR & Text Extraction** | \* Run OCR on cropped watch faces/case backs to extract text such as brand names and model numbers. Consider using open‑source OCR libraries like **Tesseract** or modern vision–language models (e.g., **TrOCR**, **EasyOCR**). Recent benchmarks show there are many OCR options (25 models tested in 2024) covering both local and cloud solutions【269821342371120†L88-L121】. |
| **Metadata Enrichment** | \* Query external sources to enrich watch records: brand histories, model specifications and estimated resale values. Use APIs (e.g., eBay search, Chrono24 pricing API) where available; otherwise, implement web scraping with care for rate limits and terms of service. |
| **Data Storage** | \* Design a relational schema using PostgreSQL. Core tables include **watch**, **image**, **price\_history** and **reference\_link**. Use JPA annotations in Java to map entities. |
| **Backend API** | \* Implement RESTful endpoints in Spring Boot for CRUD operations on watches, image uploads, search/filtering and data export. Use service classes to encapsulate business logic. |
| **Frontend UI** | \* Build a single‑page application in React. Provide pages for image upload, watch list overview, detailed watch view, search/filtering and editing. Connect to the backend via REST/JSON. |
| **Security & Auth** | \* Integrate basic authentication (JWT or OAuth2) to protect endpoints. Use Spring Security on the backend and a login flow on the frontend. |
| **Deployment & DevOps** | \* Containerise the backend using Docker. Provide Docker Compose for database and backend. Configure CI/CD pipelines (GitHub Actions) to run tests and build Docker images. |

## Detailed Solution Design

### 1. Image Ingestion & Deduplication

1. **User Upload**: The frontend allows batch upload of JPEG/PNG files. Files are sent to the backend as multipart form data.
2. **Storage**: Uploaded images are stored in a local file system or cloud storage (e.g., Amazon S3) and linked to a temporary “upload session”.
3. **Embedding Extraction**: A Python microservice exposes an endpoint that accepts image files and returns embeddings from a pre‑trained CNN (ResNet50) truncated before the classification layer. The Java backend calls this microservice after storing the image.
4. **Clustering**: With all embeddings for the upload session, perform clustering using DBSCAN (density‑based) to group similar images. Each cluster is treated as a candidate watch. If clustering yields noise points (images that don’t clearly belong to a cluster), they are reviewed manually.

### 2. Watch Detection & Classification

1. **Object Detection**: For each image, run a YOLOv7/v8 model to locate the watch bounding box. YOLO is a single‑shot detector known for its fast performance and accurate object detection【31584787726438†L67-L79】. The model can be obtained from open‑source model zoos and fine‑tuned on watch photos.
2. **Cropping**: Use the detected bounding boxes to crop watch regions. Pass the cropped images to the classifier and OCR components.
3. **Classification**: Fine‑tune a CNN (e.g., ResNet, EfficientNet) on a labelled dataset of watch images. Labelled data can be collected gradually by combining public datasets and manual annotation. The classifier outputs brand and model probabilities. If the confidence is low, rely on OCR and external metadata to determine the watch.

### 3. OCR & Text Extraction

1. **OCR Engine**: Use open‑source OCR libraries such as **Tesseract** for basic text extraction. For better accuracy on small watch dials, evaluate modern vision–language models like **TrOCR** or **EasyOCR**; a recent benchmark evaluated a broad range of OCR models, including TrOCR, Qwen2.5‑VL, EasyOCR and many others【269821342371120†L88-L121】.
2. **Post‑processing**: Clean the OCR results (remove special characters, normalise brand names). Use regular expressions or fuzzy matching to map model numbers to known patterns.
3. **Brand/Model Resolution**: Cross‑reference the extracted text with classification results. If there is a conflict, prioritise the more confident source or flag for manual review.

### 4. Metadata Enrichment

1. **Reference Sources**: Identify APIs or scraping targets: Chrono24 and WatchCharts for price data; eBay for recent sales; WatchBase or official brand pages for specifications and descriptions.
2. **API Clients**: Implement scheduled tasks in Java (Spring’s @Scheduled) to query these sources. For example, periodically update the estimated resale value using average sale prices over the last six months.
3. **Data Normalisation**: Normalise price currencies to a base currency (e.g., GBP). Convert dates and numeric fields to proper types. Store data in dedicated tables (e.g., price\_history).

### 5. Data Storage & Schema

The database schema will be defined using JPA entities:

* Watch (id, brand, model, year, resale\_value, description).
* Image (id, watch\_id, url, original\_filename, upload\_date).
* ReferenceLink (id, watch\_id, url, description).
* PriceHistory (id, watch\_id, price, currency, source, timestamp).
* Tag (id, watch\_id, name).

Relationships are mostly one‑to‑many (one watch has many images, price history records, links and tags). Use @ElementCollection for simple lists (as shown in the prototype) or separate entity classes for more complex relationships.

### 6. Backend API Design

* /api/watches – GET (list), POST (create new watch with metadata), GET /{id} (fetch details), PUT /{id} (update), DELETE /{id}.
* /api/watches/{id}/images – endpoints to upload and retrieve images.
* /api/search – search watches by brand, model, year or tags; support pagination and sorting.
* /api/export – export watch data as CSV/Excel.

Swagger/OpenAPI documentation will be generated using SpringDoc.

### 7. Frontend Design

The frontend will be a React application created with Vite or Create React App. Key pages/components include:

1. **Upload Page**: Allows drag‑and‑drop upload of multiple images. Shows progress and clustering results.
2. **Watch List**: Displays all watches in a grid/table with thumbnails, brand, model and value. Includes search and filter controls.
3. **Watch Detail**: Shows all images, metadata, price history chart and reference links for a single watch. Allows editing fields.
4. **Login / User Management**: Basic authentication and profile settings.

State management will use React Query or Redux Toolkit. UI components can be built with a design system like Material‑UI.

### 8. Security & Authentication

* Use Spring Security with JWT tokens for stateless authentication. Protect API endpoints and expose login/registration endpoints.
* Implement role‑based access control if multiple users are introduced.
* Ensure file uploads are scanned for malicious content and restrict file sizes.

### 9. Deployment & DevOps

* **Local Development**: Use Docker Compose to run the backend, database and Python microservice. Mount local volumes for persistence.
* **Continuous Integration**: Set up GitHub Actions to run unit tests on each commit and build Docker images on merges into main.
* **Hosting**: Deploy the backend to a cloud platform such as AWS ECS, Azure App Service or Heroku. Use an S3 bucket or Azure Blob Storage for image storage. The frontend can be hosted via a CDN (e.g., Vercel, Netlify).

### 10. Implementation Timeline

1. **Phase 1 – Documentation & Setup**: Finalise solution design, set up repositories, create base Spring Boot and React projects. (1 week)
2. **Phase 2 – Image Ingestion**: Implement image upload, storage and clustering service. Integrate with Python microservice for embeddings. (2 weeks)
3. **Phase 3 – Detection & Classification**: Fine‑tune and integrate YOLO and classification models. Build pipelines to detect and classify watches. (3 weeks)
4. **Phase 4 – OCR & Metadata**: Integrate OCR service, implement metadata enrichment, design price update tasks. (2 weeks)
5. **Phase 5 – API & DB**: Flesh out REST endpoints, implement search, refine database schema and write integration tests. (2 weeks)
6. **Phase 6 – Frontend**: Build UI screens, integrate with backend, implement authentication. (2 weeks)
7. **Phase 7 – Testing & Deployment**: Conduct end‑to‑end tests, add documentation, set up CI/CD and deploy. (1 week)

## Conclusion

The proposed design leverages state‑of‑the‑art computer vision techniques (YOLO for object detection【31584787726438†L67-L79】, CNNs for classification and modern OCR models【269821342371120†L88-L121】) in a modular architecture. A Java/Spring Boot backend manages data and business logic while Python microservices handle resource‑intensive AI tasks. A React frontend provides an intuitive user experience. The next step is to refine this design based on feedback and begin implementing Phase 1.