BRACT'S
VISHWAKARMA INSTITUTE OF TECHNOLOGY, KONDHWA, PUNE - 431001

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

Software Requirements Specification

Version 1.0

Digital Product Definition Model Implementation

Submitted by

Sr. No.	Name	Roll No.	PRN No.	Email
1.	Tejas Shastri	391057	22210110	tejas.22210110@viit.ac.in
2.	Krishnapriya Bandewar	391002	22211528	krishnapriya.22211528@viit.ac.in
3.	Sumeer Mehta	391059	22211576	sumeer.22211576@viit.ac.in

Table of Contents

1.0. Introduction

1.1 Purpose

The purpose of this Software Requirements Specification (SRS) document is to define the requirements for the Digital Product Definition (DPD) system. The system aims to manage model versioning and approvals through a role-based workflow involving Parent, Producer, and Subproducer roles. It will facilitate secure uploading, version control, and hierarchical approval of digital product models.

1.2 Scope of Project

This system is designed to be a centralized platform for managing digital product definitions in organizations where structured collaboration between hierarchical roles is essential. The platform will support:

- User authentication and role-based access.
- Version control for model data.
- Approval workflows among Subproducer, Producer, and Parent roles.
- Storage and retrieval of versioned models.
- A front-end interface for each user role to interact with the system.

The project will be developed using ReactJS with TailwindCSS and DaisyUI for the frontend, SpringBoot for the backend, PostgreSQL as the database, and JWT for authentication and data management.

1.3 Glossary

- SRS: System Requirements Specification
- UI: User Interface
- Admin: Administrator or Teacher managing the system
- PostgreSQL: Standard, tabular dataset for robust scaling
- ReactJS: Flexible framework for frontend development
- SpringBoot: Flexible backend Java environment
- Tailwind CSS: Utility-first CSS framework
- JWT: JSON Web Token Authentication

- **DPD**: Digital Product Definition

- **Modeldata**: The core entity representing a digital product model.
- **Modeldataversion**: Versioned instance of Modeldata.
- **Parent**: The top-level user who approves final versions.
- **Producer**: The intermediary user who reviews Subproducer submissions and forwards for Parent approval.
- **Subproducer**: The base-level user responsible for submitting new versions.
- **Appuser**: System user with associated role and authentication credentials.

1.4 References

- IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications
- Project documentation for Spring Boot and React
- Official documentation for PostgreSQL, Redux, and related libraries used in the tech stack

1.5 Overview of Document

This document contains a detailed overview of the DPD system, its functionalities, interfaces, constraints, and requirements. It begins with the overall product description, followed by specific functional and non-functional requirements, system features, constraints, assumptions, and associated diagrams.

2.0. Overall Description

2.1 Product Perspective

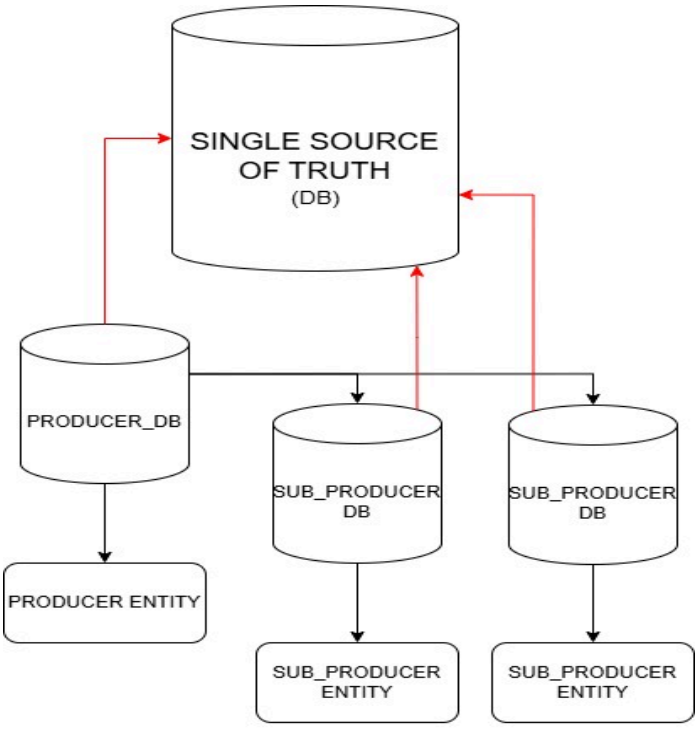
The DPD system is a standalone web application built with Spring Boot for backend services and React for the frontend interface. It integrates a relational database (e.g., PostgreSQL) to store user data, model versions, and access roles. The architecture emphasizes modularity, with clearly defined layers for data persistence, business logic, and API exposure, allowing for scalable development and easier integration of new features. Frontend and backend components communicate via RESTful APIs, ensuring decoupling and facilitating testing and

deployment automation. The application is designed with multi-tenancy in mind to support future expansions such as organization-level segregation, fine-grained access control, and audit trails. Moreover, the modular approach allows seamless integration of advanced functionalities like JWT-based stateless authentication, role-based access control (RBAC), and support for 3D CAD file handling via pluggable service components or microservices. This future-proof design ensures that the system can evolve alongside increasing complexity and business requirements.

2.2 Product Functions

The core functionalities of the **Digital Product Definition System** include:

1. **User registration and authentication**
2. **Role-based dashboards (Parent, Producer, Subproducer)**
3. **Upload and management of image-based model data**
4. **Version control for model updates**
5. **Approval workflows for promoting versions**
6. **Viewing historical versions and rollback functionality**
7. **Admin oversight and audit capabilities (optional future feature)**



DPD TENET - SINGLE SOURCE OF TRUTH

2.3 User Classes and Characteristics

The primary user classes for the **Digital Product Definition** include:

- **Parent**: Has full visibility over all Producers and Subproducers; approves final model data for use.
- **Producer**: Manages Subproducers, reviews their model data submissions, and submits to Parent for approval.
- **Subproducer**: Works on model data, submits new versions for review.

2.4 Operating Environment

- **Backend**: Spring Boot, Java 17+
- **Frontend**: React, Redux, JavaScript/TypeScript
- **Database**: PostgreSQL or MySQL
- **Deployment**: Cloud-native (e.g., AWS, Azure) or on-premise server
- **Browser Support**: Latest versions of Chrome, Firefox, Edge

2.5 Design and Implementation Constraints

- The current design of the DPD system imposes several practical constraints primarily driven by limited development resources and a focus on rapid prototyping. One of the key limitations is the use of standard image files (e.g., PNG, JPG) in place of more sophisticated 3D CAD models. This decision reflects both a constraint in available technical skills as well as the need to avoid the overhead of managing complex 3D rendering and storage pipelines during the initial development phase. While this approach suffices for basic digital product visualization, it inherently limits support for advanced engineering workflows that require true geometric data.
- The authentication mechanism currently in place is a basic email-password matching system. This was intentionally selected to keep the user onboarding and login process straightforward during the MVP stage. However, this method lacks the robustness and security features of modern authentication systems, such as JWT (JSON Web Token) or OAuth2. These more secure mechanisms are planned for future integration but are deferred until after the initial system is stable and functionally complete.
- Another design constraint involves how user role enforcement and data visibility are implemented. Rather than using declarative security mechanisms such as Spring Security annotations or method-level access controls, role enforcement is handled manually within the service layer. While this allows flexibility during rapid development, it increases the likelihood of bugs or inconsistencies and is not optimal for scalability or security.

2.6 Assumptions and Dependencies

The DPD system operates under a set of foundational assumptions and dependencies that inform its design choices and operational reliability. It is assumed that users will only upload supported image formats (such as PNG and JPG) when creating or updating models. The system does not currently implement format validation at a deep level, and improper files may lead to upload failures or incorrect metadata processing.

It is further assumed that users will be assigned appropriate roles—such as Parent, Producer, or Subproducer—at the time of registration. Role-based access and data visibility depend on the integrity of this assignment. Currently, there is no mechanism for dynamically changing roles after account creation, and doing so manually through database operations could lead to inconsistent application behavior.

A reliable, persistent file storage solution is assumed to be available and correctly configured for handling file uploads associated with model versions. The backend expects an accessible file system path where versioned files are stored, and any disruption or misconfiguration in this component would directly affect the system's core functionality.

Finally, Redux is employed on the frontend to manage user session state in lieu of JWT-based session handling. This temporary mechanism assumes that the browser retains state across session refreshes or until the user logs out manually. The absence of stateless session validation means the current system is not yet secure against common web threats, and therefore, it should not be used in production environments until JWT or OAuth is implemented.

3. Specific Requirements

3.1 Functional Requirements

The functional requirements specify the expected behavior of the system and its various components. For the **Digital Product Definition**, they are as follows:

3.1.1 User Authentication and Profile Management

The system shall provide a secure login and registration mechanism. During registration, users must provide a unique username, email address, password, and a selected role (Parent, Producer, or Subproducer). Passwords are stored in plaintext temporarily but will be hashed in future iterations. Upon login, the system authenticates the user by matching the email and password combination. Once authenticated, user session state is stored using Redux. Future versions of the application will introduce JWT-based stateless session management.

3.1.2 Role-Based Dashboard and Access Control

The system must present different dashboards and accessible features based on the user's role.

- **Parent Dashboard:** Displays all registered Producers and their corresponding Subproducers, along with model history and status.

- **Producer Dashboard:** Allows creation and upload of model versions, assignment of tasks to Subproducers, and viewing model lineage.

- **Subproducer Dashboard:** Displays tasks assigned by their Producer, allows for model review or comment submission, and enables version uploads.
  All data presented should be filtered server-side according to user roles to prevent unauthorized access.

3.1.3 Model Management and Version Control

Producers and Subproducers must be able to upload model files (image-based representations). Each model shall be versioned chronologically, and metadata such as uploader, timestamp, and comments must be recorded. Parent users should be able to browse the version history of any model, while Producers can only view and manage their own models.

3.1.4 Communication and Task Delegation

Producers must have the ability to assign tasks to their Subproducers. This includes selecting specific models or components for revision, attaching instructions, and setting deadlines. A basic notification system shall exist to inform users of newly assigned tasks.

3.1.5 User and Data Administration (Future Scope)

Future versions of the system shall introduce administrative features such as user role reassignment, account suspension, audit logs for all model uploads and changes, and integration with external storage services or version control systems.

3.2 External Interface Requirements

The external interface requirements define how the system interacts with other systems, devices, and users.

3.2.1 User Interface

The frontend is built using React and TailwindCSS to ensure responsive, modular, and accessible design. Three dashboards are implemented, each tailored to the respective role. Navigation must be intuitive, with minimal cognitive overhead. Modals, cards, and lists will be used for organizing model data and task flows.

3.2.2 Hardware Interfaces

No specific hardware dependencies exist beyond those required for typical web application usage (mouse, keyboard, monitor). The system is designed to be used on desktop or laptop environments, with mobile compatibility considered a secondary goal.

3.2.3 Software Interfaces

The backend is built using Spring Boot and connects to a PostgreSQL database for persistent data storage. File uploads are stored locally or via a mounted volume. API endpoints follow RESTful principles, and all data exchange between frontend and backend occurs via JSON. Redux is used to manage state on the client side. Authentication currently relies on simple REST calls; JWT-based security interfaces will be added later.

3.2.4 Communication Interfaces

All client-server communication is conducted over HTTP/HTTPS. There is no inter-process communication with external systems in the current version. File uploads use multipart/form-data encoding, and all other requests use JSON. Notifications are implemented using simple state polling; future versions may leverage WebSockets or server-sent events.

3.3 Non-Functional Requirements

3.3.1 Performance Requirements

The system should support up to 100 concurrent users without significant performance degradation. Basic caching strategies will be implemented to avoid repeated database calls on the frontend. File upload and rendering operations must complete within 5 seconds under normal load.

3.3.2 Security Requirements

Although advanced security features are deferred, basic protections must still be in place. User passwords are not to be exposed in API responses. Unauthorized users must not be able to access protected endpoints. Input sanitization is required to avoid injection attacks. Once JWT is integrated, role-based route protection and token expiration will be enforced.

3.3.3 Usability Requirements

The system must offer a clean, intuitive UI with clear differentiation between roles. Navigation menus, dashboard components, and modal windows should follow consistent design patterns. Accessibility compliance (e.g., keyboard navigation, color contrast) is expected in later releases.

3.3.4 Reliability Requirements

The system should operate with at least 99% uptime during working hours (9am–6pm). All critical operations such as user registration, login, and file upload

must be fail-safe, meaning the user should receive appropriate error messages on failure without data loss or crashes.

3.3.5 Maintainability Requirements

Codebase must be modular and follow clear separation of concerns (MVC architecture). Service, Controller, and Repository layers in the backend must be logically separated. React components must be organized by feature, with reusable UI primitives. Comments and documentation should be included for all major modules.

3.3.6 Portability Requirements

The application should be deployable on any standard Linux or Windows environment with Java 17+, Node.js, and PostgreSQL installed. The system should also run reliably in containerized environments (e.g., Docker), with a Dockerfile and docker-compose configuration provided for reproducibility.

4. System Features

This section provides a detailed description of all the major features of the system and their requirements.

4. System Features

4.1 Feature: User Registration and Authentication

Description:
This feature allows new users to create accounts and existing users to authenticate using their credentials. The system supports role-based user creation at the point of registration.

Primary Actors: All Users (Parent, Producer, Subproducer)

Key Operations:

- User registration with email, username, password, and role.
- Email and password-based login.
- Basic error handling for invalid credentials or duplicate accounts.

- User state is stored using Redux on the frontend.
- Future support for JWT-based authentication and token-based session management.

4.2 Feature: Role-Based Dashboard Navigation

Description:
Once authenticated, users are redirected to dashboards customized for their role. The data presented and the operations allowed vary based on user type.

Primary Actors: All Users

Key Operations:

- Parent Dashboard: Displays a hierarchical view of all Producers and their Subproducers, including access to all uploaded models.
- Producer Dashboard: Allows management of their own model uploads and Subproducers, and the assignment of model tasks.
- Subproducer Dashboard: Displays tasks assigned by their Producer and allows them to upload revised model versions or submit feedback.

4.3 Feature: Model Upload and Version Control

Description:
Enables Producers and Subproducers to upload image-based models, which are versioned and associated with specific tasks or stages in the product lifecycle.

Primary Actors: Producers, Subproducers

Key Operations:

- Upload image-based model representations (PNG, JPG).
- Maintain version history with timestamp and uploader info.
- View all versions of a given model (sorted chronologically).
- Allow comments or feedback to be attached to model versions.
-

4.4 Feature: Task Assignment and Communication

Description:
Producers must be able to delegate tasks to their Subproducers, specifying which model requires work and what needs to be done.

Primary Actors: Producers

Key Operations:

- Select model(s) and assign to Subproducers.
- Attach task instructions or notes.
- Subproducers can view assigned tasks in their dashboard.
- Optional comments or feedback can be sent back to the Producer.

4.5 Feature: Organizational Hierarchy and Role Enforcement

Description:
The system maintains a strict role-based hierarchy: Parents oversee Producers; Producers oversee Subproducers. Each user only sees data they are authorized to access.

Primary Actors: All Users

Key Operations:

- View restrictions and data filters based on user role.
- Producers see only their Subproducers and own models.
- Subproducers cannot access data from other teams.
- Parent users can view everything in a read-only capacity.
-

4.6 Feature: Model Browsing and History View

Description:
Allows users (primarily Parents and Producers) to browse through historical versions of models. Each version contains metadata and potentially attached feedback.

Primary Actors: Parents, Producers

Key Operations:

- Visual representation of model version lineage.
- Metadata display: uploader, date, comments.
- Open/download image models for inspection.

4.7 Feature: User Session Management (Redux)

Description:
Temporarily, user session and authentication state is stored using Redux on the frontend to control access and role-specific rendering.

Primary Actors: All Users

Key Operations:

- Redux state is initialized at login and cleared on logout.
- Role and user ID are stored for conditional rendering.
- This will be replaced by JWT for stateless session control in the future.

6. Assumptions and Dependencies

This section identifies any factors that are assumed to be true or any conditions that are necessary for the system to function as expected. It also outlines any external elements that the system depends on for its successful operation.

6.1 Assumptions

Assumptions are critical conditions or circumstances that are believed to be true during the system design and development. These assumptions may affect the scope, design, and performance of the system.

1. **Users will upload only supported image formats**
   The system assumes that all image files uploaded by users will adhere to the supported formats (e.g., PNG, JPG). The backend is designed to accept only these formats, and other file types may be rejected with appropriate error messages.

2. **All users are pre-assigned appropriate roles during registration**
   It is assumed that the roles for all users (Parent, Producer, Subproducer) are correctly

assigned at the time of registration. The system will rely on this role assignment to determine the access level and dashboard features available to each user.

3. **Users have access to a stable internet connection**
   It is assumed that users will have access to a stable internet connection, as the DPD system is web-based. Internet connectivity is crucial for accessing and interacting with the system's features such as model upload, version viewing, and dashboard navigation.

4. **Future JWT-based authentication integration will be straightforward**
   It is assumed that integrating JWT for authentication in the future will be a straightforward process and will not require significant changes to the system's existing user authentication model. This will involve transitioning from a basic email-password-based authentication to token-based management.

5. **The database will be properly maintained and backed up**
   The system assumes that the relational database (e.g., PostgreSQL) used to store user data, models, and versions will be properly maintained, backed up, and scaled as needed to support system growth.

6. **Users will have basic technical understanding of the platform**
   The system assumes that users (especially Producers and Subproducers) will have a basic understanding of how to upload images, navigate through model versions, and complete tasks. Training or onboarding processes will not be implemented in this initial version.

7. **No significant changes to the relational database schema**
   It is assumed that the schema design for user roles, models, and versions will remain relatively stable throughout the system's initial lifecycle. Any changes to the database schema (e.g., introducing new relationships) will be handled with backward compatibility in mind.

8. **The frontend is responsive and can function across different devices**
   The system assumes that the frontend built with React will be responsive and work seamlessly across various devices (desktop, tablet, mobile), allowing users to interact with the dashboard and upload models regardless of their device type.

6.2 Dependencies

Dependencies refer to external systems, services, or software components that the DPD system relies on to function correctly. These dependencies may influence the system's performance, security, and availability.

1. **Database Management System (PostgreSQL)**
   The system is dependent on a relational database (e.g., PostgreSQL) to store user data, model versions, and associated metadata. It assumes that the database will be reliable, scalable, and backed up regularly to ensure data integrity.

2. **Backend Framework (Spring Boot)**
   The DPD system relies on Spring Boot for backend services. The functionality of the

entire system depends on the performance, stability, and scalability of the Spring Boot framework. The backend also uses Spring Data JPA for database interaction and Hibernate for object-relational mapping (ORM).

3. **Frontend Framework (React.js)**
   The frontend of the system is built using React.js, which is responsible for rendering the user interface and handling client-side interactions. The system assumes that React will perform efficiently across all supported browsers and devices. The frontend communicates with the backend via REST APIs.

4. **File Storage System**
   The system relies on a file storage system (e.g., local disk storage or cloud storage) to store model images and version files. This storage system is crucial for file retrieval, versioning, and management. In the future, the system may require integration with cloud storage services (e.g., AWS S3, Google Cloud Storage) for scalability and performance.

5. **Authentication Libraries**
   For handling user authentication, the system uses basic email and password matching. It assumes that libraries such as Spring Security or similar frameworks will be available to manage authentication and authorization. Future upgrades may require support for more sophisticated token-based systems (e.g., JWT, OAuth).

6. **Redux for State Management**
   The frontend relies on Redux to manage user session state and role-based rendering. This dependency ensures that the UI adapts based on the logged-in user's role (Parent, Producer, or Subproducer). Any changes in the state management libraries may require updates to the frontend code.

7. **Browser and Device Compatibility**
   The system assumes compatibility with modern web browsers (e.g., Google Chrome, Firefox, Safari) and that the users' devices meet the necessary requirements for displaying the user interface. The system may not perform optimally on older browsers or devices with limited resources.

8. **External API Integrations (Future Considerations)**
   In future releases, the system may integrate with external services or APIs for features like real-time notifications, cloud storage for model files, or additional user authentication methods. These external services may affect system performance and availability if not managed correctly.

9. **Third-Party Libraries**
   The frontend uses third-party React libraries and components for UI elements (e.g., for file uploads, buttons, or forms). These libraries depend on active maintenance, security patches, and compatibility with React.

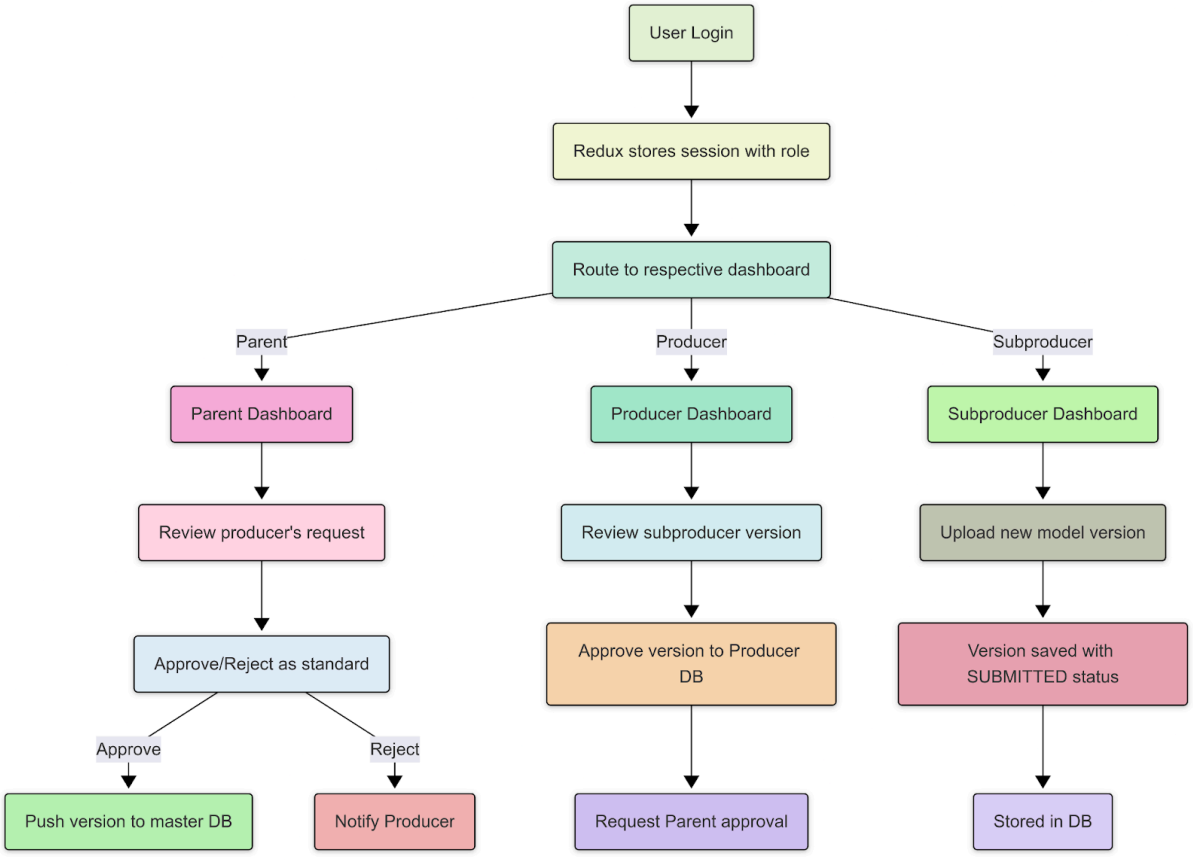
10. **Security and Compliance Standards**
    The system may rely on compliance with certain industry security standards (e.g., GDPR, CCPA) for managing user data and file storage. The system will need to be adaptable to meet these standards, especially when scaling.

6.3 Diagrams

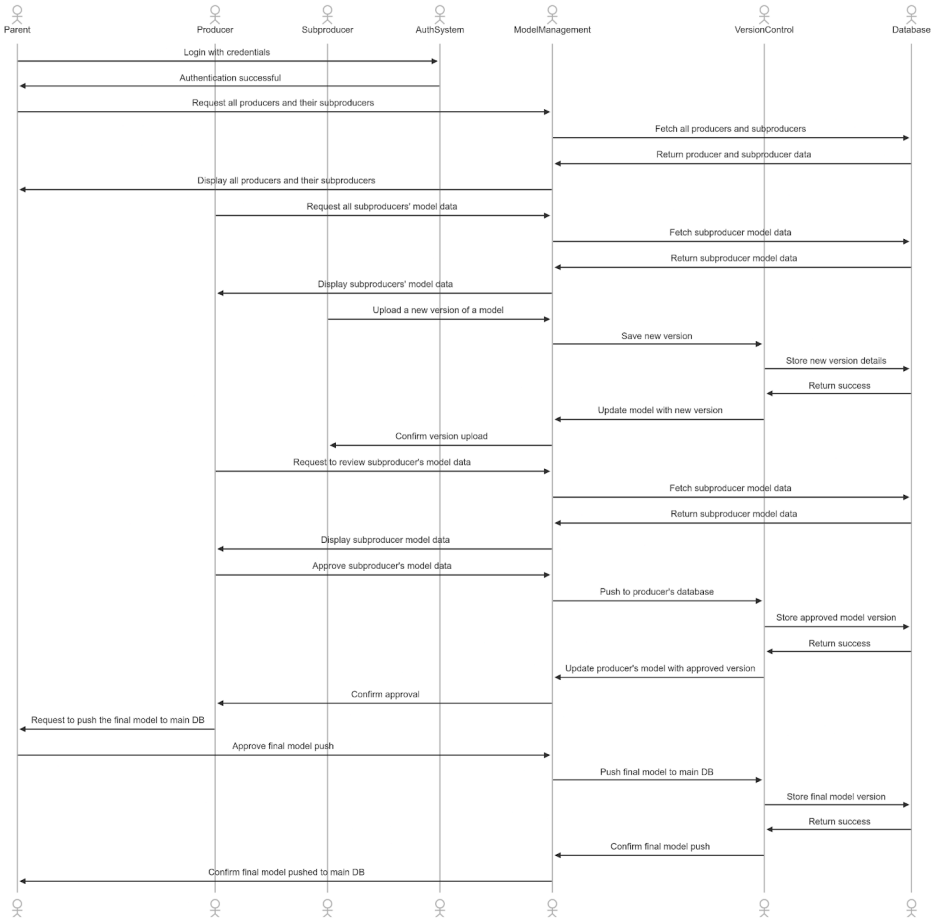
Diagrams are an essential part of understanding the system. For the **Digital Product Definition** , several types of diagrams will be included to cover different aspects of the system's functionality, architecture, data flow, and interactions. Each diagram provides a unique perspective on the system, ensuring a comprehensive and detailed understanding of how the project is organized and how it operates.

6.3.1 Flow Diagram

The **Data Flow Diagram (DFD)** provides a visual representation of how data moves through the **Digital Product Definition** . It captures the flow of data from **Producers to Subproducers to the Parent company**, as well as how **results and feedback** are stored in the database. This diagram helps to understand the system's data processing and interaction between various components

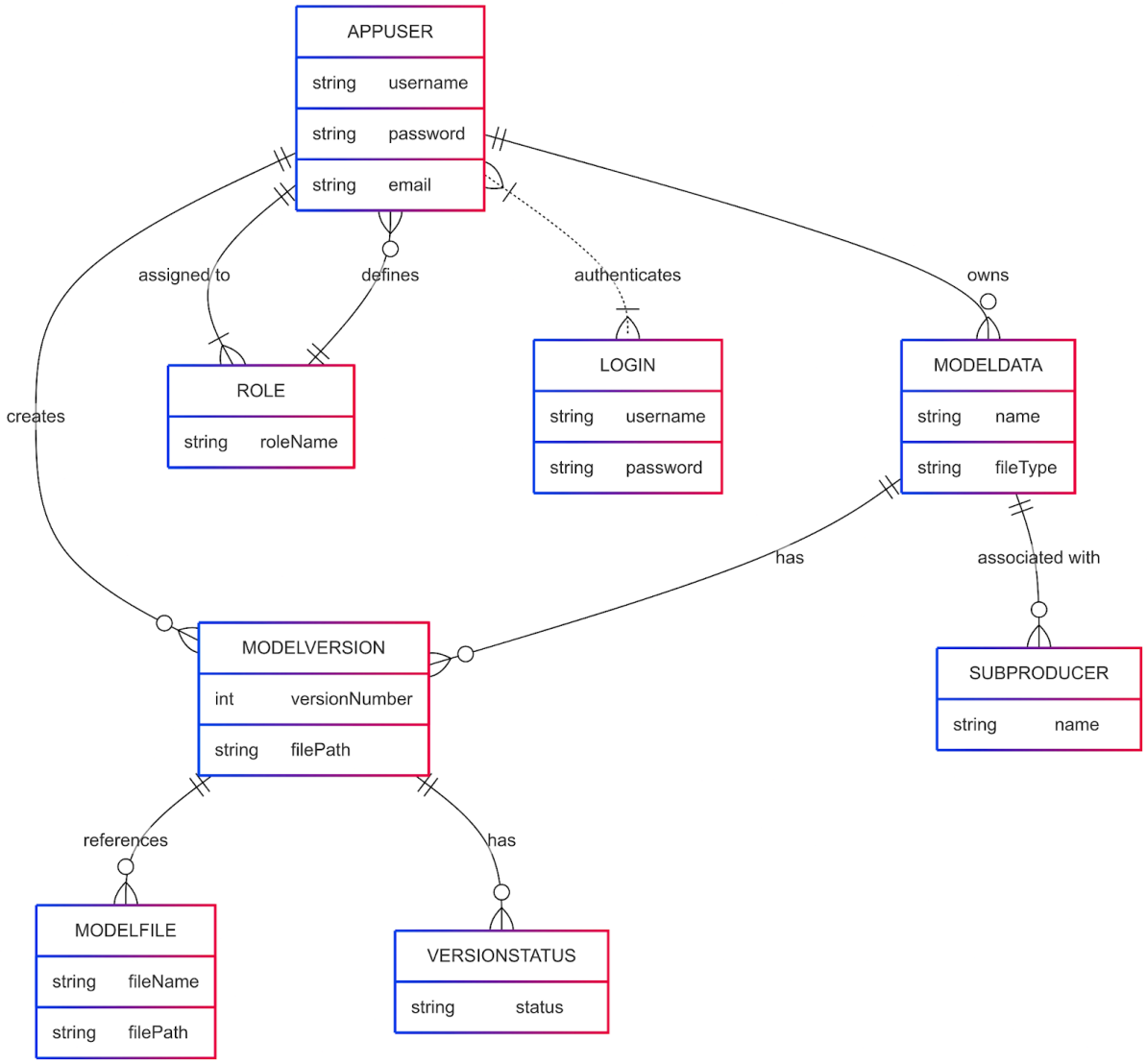


6.3.2 Sequence Diagram

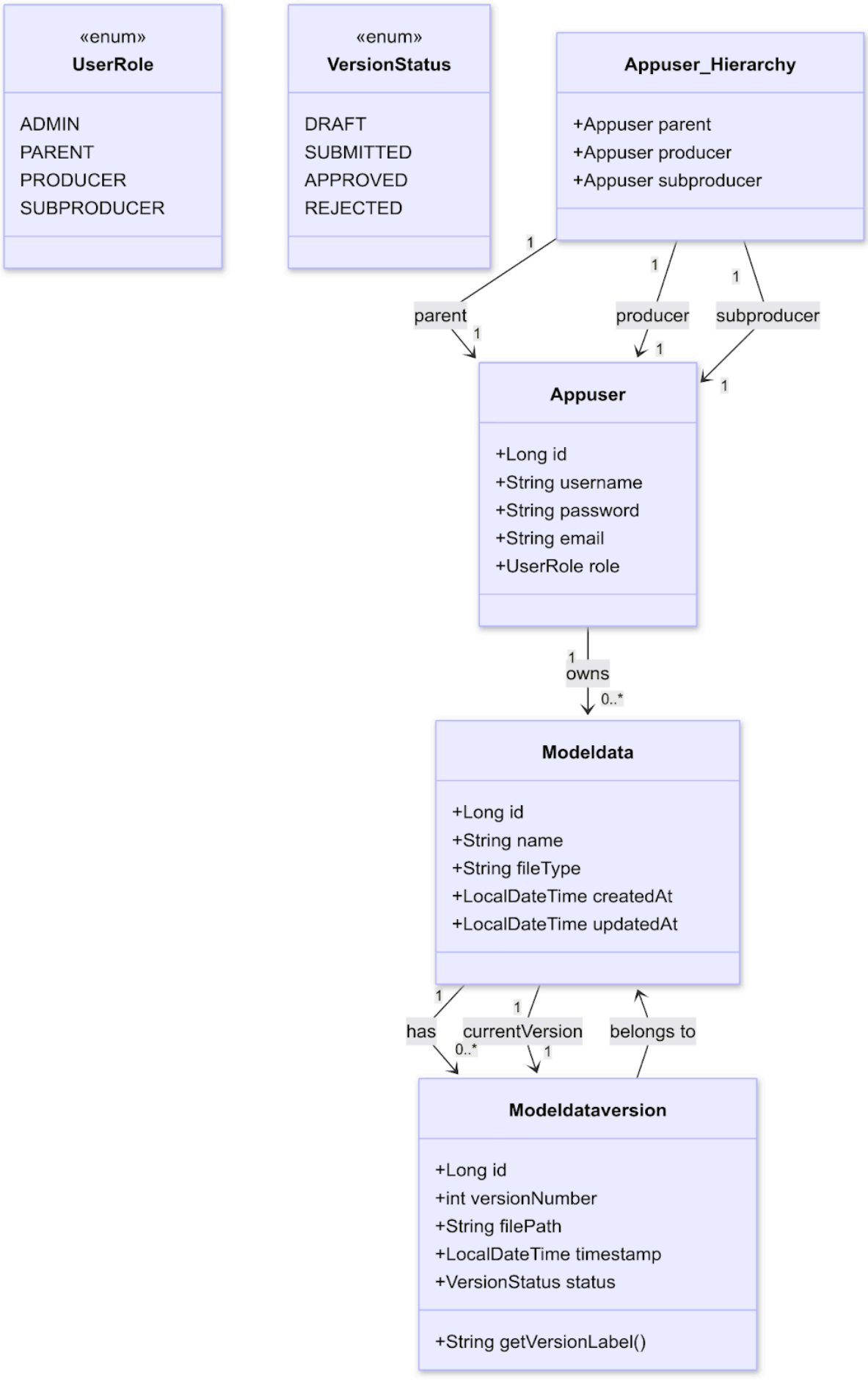


6.3.3 ER Diagram

The Entity-Relationship Diagram (ERD) illustrates the main data entities involved in the DPD system, such as User,Parent,Subproducer, Producer and Database, along with their attributes and relationships. It provides a clear understanding of how data is structured and connected, helping in designing the database for storing user details, feedback, and administrative data.



6.3.3 Class Diagram


«enum»
UserRole
ADMIN
PARENT
PRODUCER
SUBPRODUCER
«enum»
VersionStatus
DRAFT
SUBMITTED
APPROVED
REJECTED
Appuser_Hierarchy
+Appuser parent
+Appuser producer
+Appuser subproducer
1
1
1
parent
producer
subproducer
1
1
1
Appuser
+Long id
+String username
+String password
+String email
+UserRole role
1
owns
0..*
Modeldata
+Long id
+String name
+String fileType
+LocalDateTime createdAt
+LocalDateTime updatedAt
1
1
has
currentVersion
belongs to
0..*
1
Modeldataversion
+Long id
+int versionNumber
+String filePath
+LocalDateTime timestamp
+VersionStatus status
+String getVersionLabel()

7.0 Conclusion

The Digital Product Definition (DPD) system represents a robust, modular solution for managing product data, model versions, and user access roles. By leveraging Spring Boot and React, the system ensures scalability, flexibility, and a modern user experience. The separation of concerns between the backend and frontend allows for efficient development, with room for future integration of advanced features such as JWT-based authentication and 3D model handling.

The system's user role management ensures that different stakeholders (such as Parents, Producers, and Subproducers) can interact with the system according to their privileges, making the process of managing models and versions straightforward. By enabling role-based access control at the service level, the system ensures that sensitive data is only accessible to those with the proper permissions.

From a technical standpoint, the use of a relational database (e.g., PostgreSQL) ensures data integrity and consistency, while the application’s modular structure makes it easy to extend and scale as the project evolves. Despite current limitations—such as the absence of JWT and the use of images instead of 3D models—the system is designed with future-proofing in mind, allowing for these improvements in subsequent versions.

In terms of user experience, the frontend React application provides a dynamic interface, allowing users to interact with the system in an intuitive and streamlined manner. The system supports multiple dashboards for different roles, enhancing user engagement and simplifying workflow management.

Overall, the DPD system aims to streamline the product data management process, facilitating collaboration across different roles while ensuring a seamless and secure user experience.