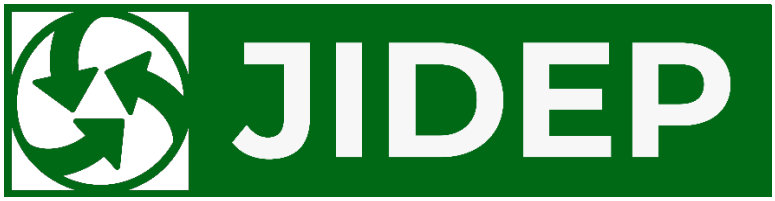
PROJECT DELIVERABLE REPORT

Grant Agreement Number: 101058732



Joint Industrial Data Exchange Platform

Type: Deliverable Report

Deliverable Title: Report on distributed data contributions

Issuing partner	Technovative Solutions (TVS)
Participating partners	THE CHANCELLOR MASTERS AND SCHOLARS OF THE UNIVERSITY OF CAMBRIDGE (UCAM)
Document name and revision	D2.7 Report on distributed data contributions(Final)
Author(s)	Miah Raihan Mahmud Arman (TVS) Rasel Ahmed (TVS) Dr Feroz Farazi (UCAM)
Reviewer(s)	Tanvir Islam (TVS)
Deliverable due date	31-05-2024
Actual submission date	29-05-2024

Project Coordinator	Vorarlberg University of Applied Sciences
Tel	+43 (0) 5572 792 7128
E-mail	florian.maurer@fhv.at
Project website address	www.jidep.eu

Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission services)	
CO	Confidential, only for members of the consortium (including the Commission services)	
SEN	Sensitive, limited under the conditions of the Grant Agreement	

Executive Summary

This report aims to demonstrate the distributed data contributions of the JIDEP project. In this document, the selection and implementation methodology is described in detail. In this report, our choice of GunJS for implementing the distributed storage system of the JIDEP architecture is described and justified. Finally, based on a preliminary analysis of JIDEP business processes and data flows, we devise the appropriate architecture design process and create an architectural plan that comprises a high-level architecture of our distributed storage system and its main components:

- Distributed Storage Peers
- Distributed Storage Server

Apart from these two, we have also used the Ontology service from D2.4 to configure our storage system for use-case-specific Data Sharing Unit (DSU) types. The DSU type for the automotive, wind turbine and printed circuit board (PCB) use cases has been designed.

The Distributed Storage Peers receive requests from the users to store and manage their data securely and distributedly. In addition, the Distributed Storage Server ensures the backup and restoration of those data: the peers and the server sync in real-time to ensure that each has the latest data version.

We augmented, modified, refined, and refactored our services to provide more features and improve performance and security, we also support use-cases specific DSU types and also developed a system for access control so that the data can be securely shared among trusted entities according to the confidentiality terms and usage limit specified by the architecture.

Table of Contents

List of Figures

List of Tables

Disclaimer

JIDEP has received funding from the European Commission's under the Grant Agreement no.101058732. The content of this document does not represent the opinion of the European Commission, and the European Commission is not responsible for any use that might be made of such content.

Acronyms

API	Application Programming Interface
DSUs	Data Sharing Units
EoL	End-of-Life
OBDI	Ontology Based Data Integration
RDF	Resource Description Framework
SDK	Software Development Kit
SPARQL	SPARQL Protocol and RDF Query Language
TVS	Technovative Solutions
UCAM	University of Cambridge

1. Introduction

In an era marked by rapid technological advancements and increasing interconnectivity, industries worldwide are faced with the imperative to harness the power of data for innovation, collaboration, and sustainable growth. The Joint Industrial Data Exchange Platform (JIDEP) emerges as a pivotal solution, offering a comprehensive framework to revolutionize how industrial data is exchanged, leveraged, and applied across diverse sectors. This project elucidates the strategic objectives that underpin the evolution of JIDEP, aiming to foster enhanced data exchange through ontologies, establish a collaborative ecosystem across the value chain, develop tools for industry sustainability and circularity, reinforce the autonomy and resilience of European industries, and validate the platform's efficacy through real-world use-cases and demonstrations. As industries navigate a dynamic landscape, this report helps underscore JIDEP's role as a catalyst for transformative change, poised to drive innovation, efficiency, and competitiveness while promoting standard data practices across industries.

In the ever-evolving landscape of knowledge management and data utilisation, Task T2.7 takes centre stage as a pivotal endeavour to create a seamless and efficient medium for storing and propagating data. The task revolves around developing specific Data Sharing Unit (DSU) types, each meticulously tailored to cater to distinct domains, enabling the seamless sharing and manipulation of domain-specific knowledge.

We are at the forefront of innovation, engrossed in developing cutting-edge tools poised to reshape the landscape of data management. Among these tools, the JIDEP Ontology API stands as a testament to our commitment to precision and efficiency. Through this API, we enable seamless integration between domain-specific ontologies and the JIDEP Platform, along with the chance to connect with the broader data ecosystem, fostering a harmonious relationship between structured knowledge and data utilization. Additionally, our endeavours extend to data storage, where we harness the power of distributed storage systems. This strategic approach facilitates secure and efficient storage, ensuring the accessibility and availability of data across diverse domains and applications. This visionary approach enhances data resilience and lays the foundation for collaborative data sharing and analysis. With these tools as our driving force, we are positioning ourselves at the forefront of a new era of data-driven innovation. As we continue to develop and refine these technologies, we embark on a journey that promises to unlock new possibilities, revolutionize data management, and pave the way for a more connected and insightful future.

2. Distributed Storage

2.1 Background

Data is a collection of factual and raw information, ranging from textual records and numerical values to multimedia content. This information is systematically organised, stored, and processed to extract insights, trends, and knowledge. The essence of data lies in its ability to serve as a foundation for analysis, decision-making, and generating meaningful conclusions [1]. Usually, any software is comprised mainly of data and programs. The program uses the data to extract desired information. The data and the program must be stored somewhere and interact with the data whenever authorised users perform any action. The systemic way in which they are stored and how they interact can be broadly categorised into the following three categories:

a) Centralised System
b) Decentralised System
c) Distributed System

These three types of systems have different architectures, characteristics, advantages, and limitations.

Centralised Systems: Centralized systems employ a client/server architecture wherein a central server connects with one or more client nodes. This prevalent system structure is widely adopted in various organisations, where clients send requests to a company's server and subsequently receive responses [2]. For example, Wikipedia is a centralised system.
Centralised systems have the following characteristics [2]:

- Presence of a global clock: All client nodes sync up with the clock of the central server.
- One single central unit: The central server serves or coordinates all the other nodes in the system.
- Dependent failure of components: If the central server fails, the entire system fails.
- Scaling: Only vertical scaling on the central server is possible. Horizontal scaling would contradict this system's single central unit characteristic.

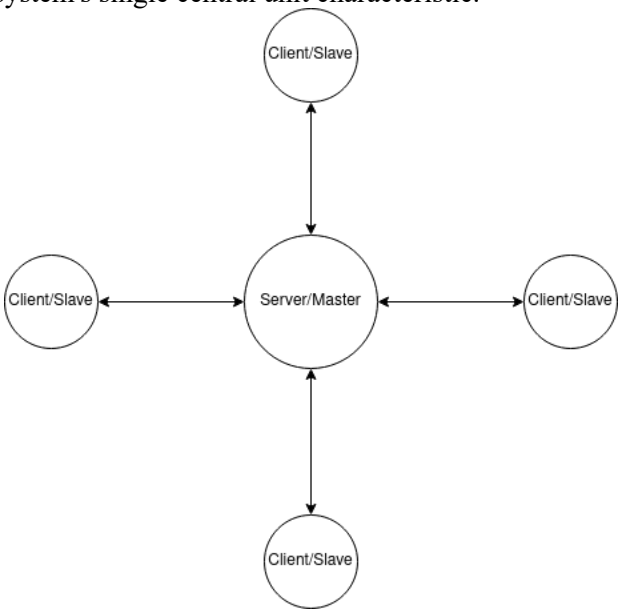


Figure 1: Centralized System

Centralised Systems have the following advantages [3]:

- Easy to physically secure: The server and client nodes can be located securely and serviced quickly.
- Easy to implement and manage: The system has a simple and clear structure and logic.
- High performance and reliability: The system can handle high traffic and provide consistent service quality.

Centralised Systems have the following limitations [3]:

- Can't scale up vertically after a specific limit: After a limit, increasing the hardware and software capabilities of the server node will not improve the performance significantly.
- Bottlenecks can appear when the traffic spikes: The server can only have a finite number of open ports to listen to connections from client nodes. When high traffic occurs, the server can suffer from denial or distributed attacks.
- Single point of failure: The system is vulnerable to attacks or disasters that target the central server. The entire system will stop working if the server is compromised or damaged.

Decentralised Systems: Decentralized systems have multiple central nodes, each overseeing and coordinating client nodes. In this structure, no overarching central node governs or communicates

with all other nodes across the system. Each central node is an independent entity with distinct policies, rules, and protocols [4]. For example, BitTorrent is a decentralised system.

Decentralised Systems have the following characteristics [4]:

- Decentralised storage systems fragment data into multiple segments known as "blocks."
- It distributes these blocks among diverse systems and nodes, achieving data dispersion across the network, a concept known as "sharding."
- Multiple central nodes serve or coordinate their respective client nodes. Each central node has its own domain of authority and responsibility.
- The failure of one central node does not affect the other central nodes or their client nodes. The system can continue to function with partial failures.
- Both vertical scaling and horizontal scaling are possible. Vertical scaling can be done on each central node individually. Horizontal scaling can be done by adding more central nodes or client nodes to the system.

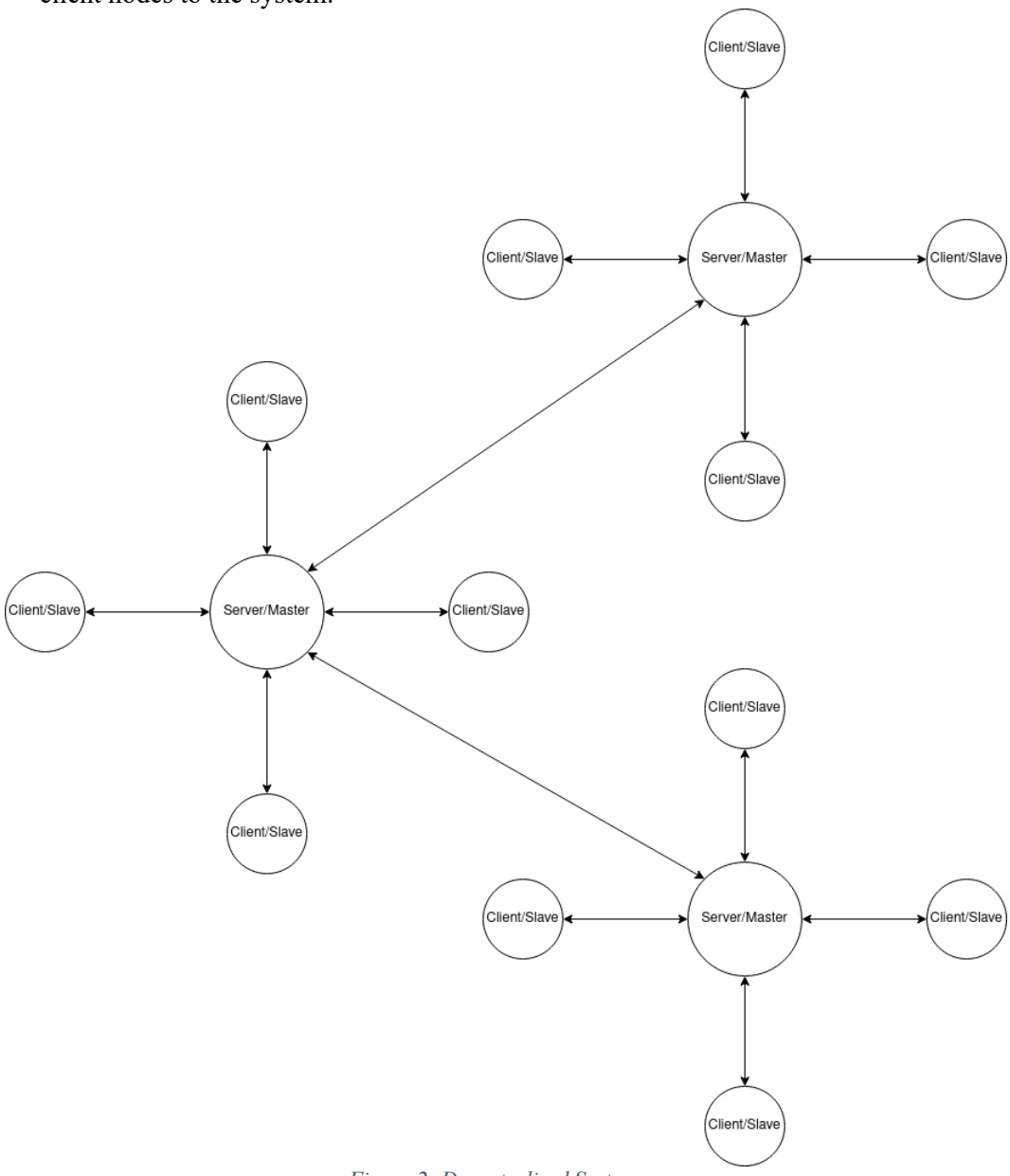


Figure 2: Decentralized System

Decentralised Systems have the following advantages [5]:

- Fault tolerance and resilience: The system can tolerate and recover from failures of some components without affecting the whole system. The system can also adapt to changing conditions and demands by adjusting its policies, rules, and protocols.
- Scalability and flexibility: The system can scale up or down easily by adding or removing components. The system can also accommodate different users' different needs and preferences by allowing them to choose their preferred central node or protocol.
- Privacy and security: The system can protect the privacy and security of users by avoiding centralised data storage or processing. Users can control their own data and transactions without relying on a third party or intermediary.

Decentralised Systems have the following limitations [2]:

- Complexity and overhead: The system has a complex and unclear structure and logic that requires more coordination and communication among components. The system also incurs more overhead in terms of resources, time, and bandwidth.
- Inconsistency and unreliability: The system may suffer from inconsistency and unreliability due to the lack of a global clock, a global consensus mechanism, or a global authority. Different components may have different versions of data, different states of transactions, or different outcomes of operations.
- Conflicts and disputes: The system may encounter conflicts and disputes among components due to divergent interests, goals, or policies. There may be no clear or fair way to resolve these conflicts or disputes without compromising each component's autonomy or sovereignty.

Distributed Systems: A distributed system comprises autonomous components situated on separate machines, communicating messages to attain shared objectives collaboratively [6]. There is no distinction between server nodes and client nodes in a distributed system. Each node acts as both a server and a client, depending on the situation. For example, Google is a distributed system.
Distributed systems have the following characteristics [7]:

- Heterogeneity in distributed systems enables operation across diverse hardware and software using middleware abstraction. Middleware interprets calls, facilitating distributed processing on varied nodes.
- The openness of a distributed system pertains to the challenge of enhancing or extending it. This quality enables the system to be reused for diverse tasks or data processing.
- Concurrency denotes a system's capacity to manage shared resources' access and utilisation, which is crucial to prevent data corruption or loss. The absence of proper measures can lead to errors as multiple nodes make disparate changes to the same resource, propagating inaccuracies across processes.
- Scalability is a crucial attribute influencing the efficiency of a distributed system, indicating its ability to adjust to varying sizes. This flexibility is vital due to computers' dynamic nature, as devices can join or depart from the system unpredictably due to factors like power downs or network instability.
- Component failures are common in distributed systems with diverse hardware. Fault tolerance, achieved through recovery and redundancy, enables the system to manage such failures—recovery ensures controlled responses, while redundancy deploys backups for critical processes in case of system breakdowns.
- Transparency in distributed systems ensures users perceive a unified entity rather than individual cooperating parts.

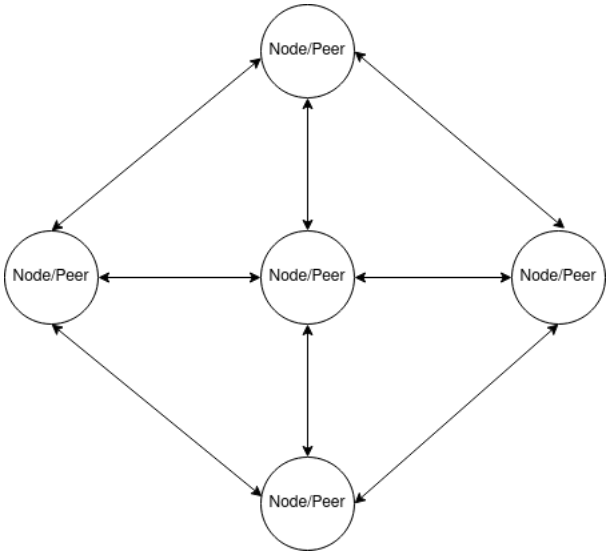


Figure 3: Distributed System

Distributed systems have the following advantages [8]:

- Fault tolerance and resilience: The system can tolerate and recover from failures of some components without affecting the whole system. The system can also adapt to changing conditions and demands by adjusting its algorithms, protocols, or strategies.
- Scalability and performance: The system can easily scale up or down by adding or removing components. It can also improve its performance by distributing the workload, data, or computation among components.
- Availability and accessibility: The system can provide high availability and accessibility to users by replicating or caching data, services, or resources among components. Users can access the system from anywhere and anytime without depending on a single point of access.

Distributed systems have the following limitations [8]:

- Complexity and overhead: The system has a complex and unclear structure and logic that requires more coordination and communication among components. The system also incurs more overhead in terms of resources, time, and bandwidth.
- Inconsistency and unreliability: The system may suffer from inconsistency and unreliability due to the lack of a global clock, a global consensus mechanism, or a global authority. Different components may have different versions of data, different states of transactions, or different outcomes of operations.
- Security and privacy: Due to the network's openness and heterogeneity, the system may face security and privacy challenges. It may also be vulnerable to attacks or breaches that target the network, the data, or the components.

2.2 Methodology

2.2.1 Criteria

To find a desired data-storage solution for our project, we had to first define a set of criteria depending upon which we compared and selected the data-storage solution for our project. The table below summarises the criteria we used to evaluate the different data-storage solutions:

Table 1: Criteria for Distributed Database Selection

Category	Aspect	Question
Correctness	Conflict handling	How are conflicts handled? Does it require the client to write bespoke conflict resolution code?
	Robustness	How "bullet proof" is it? How easy is it to get it into a broken state where different clients see inconsistent data despite syncing?
	Consistency verification	Is there consistency verification built-in, to detect if you're in a broken state?
	Intent preservation	How well does sync preserve intent? In what cases would a user's work be "lost" unexpectedly?
Cost	Performance	What is the performance impact of using this solution? How fast is it to read and write data? How much bandwidth does it use for syncing?
	Complexity	How complex is it to set up and use this solution? How much code do you have to write or maintain? How easy is it to debug or troubleshoot problems?
	External dependency	How much does it rely on external services or resources? Do you have to pay for them or manage them yourself? How reliable are they? What happens if they go down or change their APIs?
Flexibility	Data model	How flexible is the data model? Can you store any kind of data or do you have to follow a schema or a format? Can you query or manipulate the data in different ways?
	Sync protocol	How flexible is the sync protocol? Can you customize how and when data is synced? Can you sync with different peers or servers? Can you handle different network conditions or offline scenarios?
	Extensibility	How extensible is the solution? Can you add new features or functionality to it? Can you integrate it with other libraries or frameworks?

2.2.2 Data Type

While searching for these criteria in different distributed databases, we identified a particular data type called CRDT, which stands for Conflict-free Replicated Data Type. It relies on a data structure that can be replicated across multiple computers in a network. CRDTs allow each replica to be updated independently and concurrently without requiring coordination or communication between the replicas. CRDTs also ensure that the replicas can always be merged into a consistent state without any conflicts or data loss. CRDTs are helpful for systems that need to store and sync data across different devices, such as mobile apps, distributed databases, and collaboration software.

2.3 Distributed Storage Selection

In our study, we found several libraries, compared them and drew conclusions. Some libraries were of interest for our purpose, which are listed as follows:

a) Gun.js
b) RemoteStorage.js
c) RxDB + PouchDB
d) Hypermerge + Automerge
e) YJS
f) orbitDB

2.3.1 Different Distributed Storage Comparison

In our quest for the optimal Distributed Storage solution, we diligently evaluated several prominent contenders in the field: Gun.js, RemoteStorage.js, RxDB & PouchDB, as well as Hypermerge & Automerge. Among these options, our discerning analysis led us to choose GunJS as the most fitting solution for our project. We narrowed down our list of distributed data-storage solutions to the following:

a) Gun.js
b) RemoteStorage.js
c) RxDB + PouchDB
d) Hypermerge + automerge

The subsequent table offers a concise synthesis of key attributes and distinctions across these four libraries, each catering to the construction of decentralised applications with data synchronisation capabilities:

Table 2: Comparison Table of Different Distributed Data Storages

Library	Description	Pros	Cons
Hypermerge	A Node.js library that combines Automerge, a CRDT, with hypercore, a distributed append-only log. It allows apps to have data sets that are conflict-free, offline-first, and serverless.	Flexible, fast, reliable, and supports multiple data models and sync protocols.	Deprecated, slow to synchronize, incompatible with new Automerge file format.
RxDB+PouchDB	A combination of RxDB, a reactive and offline-first database, and PouchDB, a JavaScript database that syncs with CouchDB. It allows apps to store data locally in the browser or Node.js and sync it with a remote CouchDB server.	Scalable, easy to use, supports complex queries and analytics, compatible with different	Lacks the support for automatic conflict resolution.

		environments and adapters.	
Gun.js	An open-source and peer-to-peer graph database that can store any kind of data and sync it in real-time across any device. It does not require any servers or cloud services to operate and handles conflicts and offline scenarios automatically.	Flexible, scalable, fast, secure, supports multiple data models and APIs. An active community of users.	Their codebase is hard to comprehend. It does not support the storage of arrays.
remotestorage.js	A JavaScript library for storing user data locally in the browser as well as connecting to remoteStorage servers and syncing data across devices and applications. It is also capable of connecting and syncing data with a person's Dropbox or Google Drive account (optional).	Simple, stable, versatile, powerful, supports different data models and interfaces.	Slow sync speed, no end-to-end encryption mechanism-built in.

2.3.2 Analysis and Selection of Distributed Storage

The versatility and production readiness of GunJS, along with the other pros, impressed us the most. So we decided to go with GunJS. The difference between GunJS and other decentralized databases is that GunJS is a graph database that can store any kind of data and sync it in real-time across any device, without requiring any servers or cloud services. GunJS uses a peer-to-peer protocol that automatically handles conflicts, errors, and data consistency. GunJS also supports offline scenarios and local-first storage, as well as end-to-end encryption for data security. GunJS is designed to build decentralised applications that are fast, secure, and scalable.

2.3.3 Details of the Selected Distributed Storage

The key features of GunJS are briefly described below. Here are some of the main features of GunJS that make it a powerful and flexible database for building decentralised applications:

a) **Graph data**: GunJS stores data as a graph of nodes and edges representing any data structure, such as key/value, tables, documents, videos, and more. Graph data allows users to model complex relationships and queries without worrying about schemas or formats.
b) **Freedom**: GunJS is a database for freedom fighters. GunJS gives users the most potent digital weapons on the internet: encryption for privacy and independence through decentralization. There are now enough phones on the planet to power all of Facebook by the people, for the people, and of the people. Users own their data and will not need to share it through any central entity.
c) **Realtime synchronisation**: GunJS syncs data in real-time across any device without requiring servers or cloud services. GunJS uses a peer-to-peer protocol that automatically handles conflicts, errors, and data consistency. GunJS also supports offline scenarios and local-first storage, so users can access and update their data even when disconnected from the network.
d) **End-to-end encryption**: GunJS provides end-to-end encryption for user data using a library called SEA (Security, Encryption, Authorization). SEA lets users create secure user accounts with normal logins and even peer-to-peer password resets using 3FA (three-factor authentication). SEA also encrypts user data with public-key cryptography, so only they and the people they trust can access it.

2.4 Augmentation and Maintenance of Distributed Storage

2.4.1 Stability

The initial version of our distributed storage system faced frequent downtimes when multiple users tried to use the system at once. We addressed the root causes of this instability and overcame those by gradually implementing better practices.

2.4.2 Code Quality

Better code is easy to maintain and improve. So, we prioritised code quality from the beginning. We followed the DRY (Do not Repeat Yourself) philosophy to clean and organise our code into reusable functions.

2.4.3 Performance

As the project advanced, we approached the stage of real user interaction. Recognising the criticality of performance, we proactively fine-tuned and adjusted our distributed storage system. We aimed to ensure users enjoy a seamless and satisfactory experience while utilising our platform.

2.4.4 Troubleshooting

Even the most improved and stable systems and our system face downtime. In such instances, we meticulously examine the different aspects of our system to diagnose the issue and take the necessary steps to get the system up and running.

2.5 The Architecture of Distributed Storage

After the decision to adopt GunJS as our Distributed Storage solution, the architectural framework was carefully crafted around this chosen library. Intending to grant the users complete control over their data, a fundamental requirement was ensuring secure storage on their devices. By harnessing the capabilities of a graph database, our overall database structure emerged as a composite of all data dispersed across peers.

In this design, each peer was a node, retaining a portion of the comprehensive database proportional to its accessed data. This ingenious arrangement accelerated user data access, aligning with their specific needs. Additionally, a distinct inclusion was made by incorporating a server as one of the peers. This strategic addition supported data persistence, mainly when all other peers might be offline.

A deeper exploration of this architecture will be offered in the forthcoming sub-section, where we will delve into its intricate details and mechanisms.

2.5.1 Architecture Components

The different components of our architecture play different vital roles in creating a reliable, safe and scalable system for the sharing and storage of DSUs. The main components of our distributed data contribution system are:

a) **Distributed Storage Server:** As the sole server-peer, this component's primary role revolves around securely archiving our complete database within Amazon's S3 storage infrastructure. This particular component is synchronized seamlessly with the live data from the Distributed Storage Peers, guaranteeing persistent and real-time data preservation. Notably, this element exclusively encompasses the data backup and restoration logic and needs no user engagement.

Data updates are seamlessly propagated through the intrinsic synchronisation mechanism inherent in GunJS.

b) **Distributed Storage Peers:** This group of components receives user requests to access and manage their data via the Distributed Storage API. Our report, Deliverable D3.5 within the JIDEP project, delves into a detailed exploration of this API and the accompanying SDK documentation. These components are designed to operate independently, maintaining functionality even when other peers are not online. This capability is achieved by integrating data and the associated data access logic. In this setup, the Distributed Storage Server also serves as a peer with a specific role. The data access logic is meticulously designed to ensure that only authenticated and authorised users are granted access to the data.

c) **Backup Cloud Storage:** This cloud-based object storage service delivers industry-leading scalability, data integrity, security, and speed. It empowers organisations across diverse sectors and scales to securely store and safeguard extensive data volumes, catering to various applications, including data lakes, websites, mobile apps, backup and recovery, archives, and large-scale data analytics. Amazon S3 additionally provides administrative tools that empower users to streamline expenses, structure data, and establish tailored access parameters. We utilised this service to create backups and reinstate data when information might be compromised or lost.

d) **JIDEP Ontology API:** This service, developed as a component of the JIDEP project, delivers ontology-based data essential for generating, confirming, and enhancing DSUs (Data Storage Units). It supplies diverse metadata concerning the structure of DSUs tailored to specific use cases, along with accessible information for user interfaces. This service facilitates interoperability and standardisation of the information within our system and extends this consistency to our partners' systems.

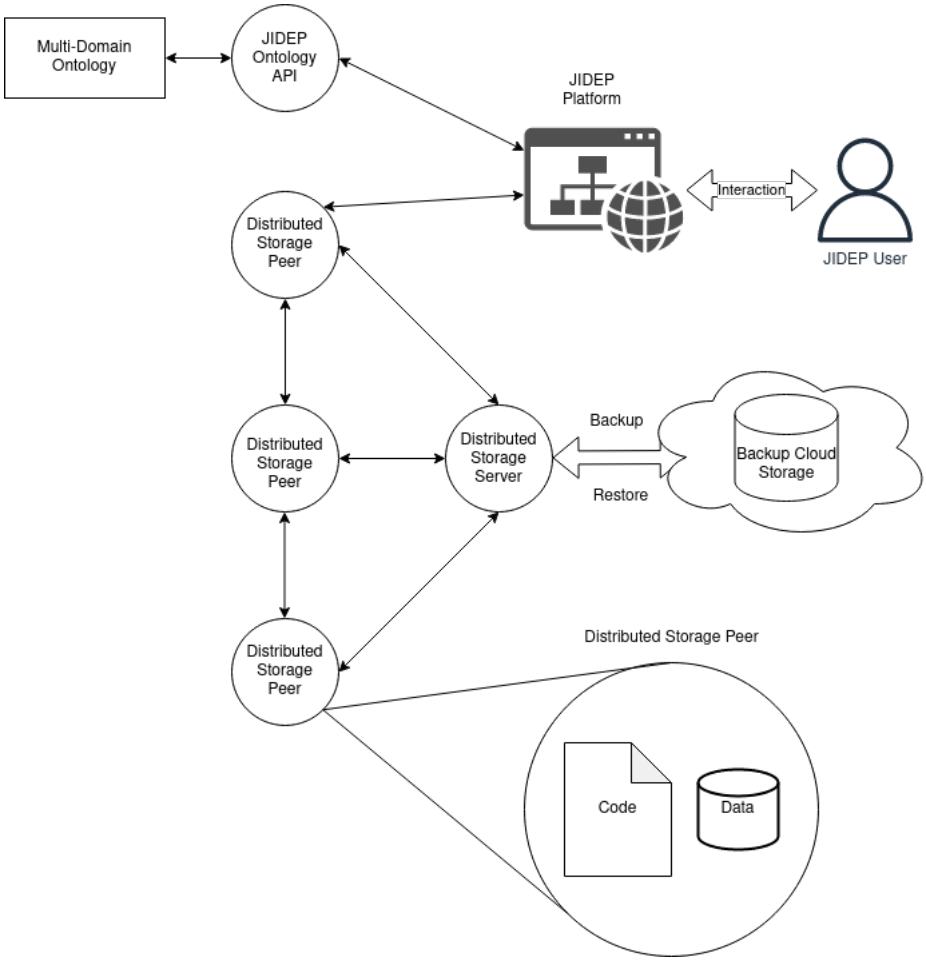


Figure 4: System Architecture of JIDEP

2.5.2 Distributed Storage Methods

We used many methods of Distributed Storage in our **Distributed Storage Peers and Distributed Storage Server** components. Here is the list with a brief summary of those methods:

a) **Put(data, callback)**: This method is used to save data to the gun database. It takes a data object as the first argument and an optional callback function as the second argument. The callback function receives an acknowledgement object as a parameter, which contains information about the success or failure of the operation. The gun.put method returns the gun instance itself, so it can be chained with other methods.
b) **Get(key)**: This method is used to retrieve data from the gun database. It takes a key string as the argument and returns a gun reference that points to the data associated with that key. The gun reference can be used to access or modify the data using other methods, such as gun.on, gun.once, gun.map etc.
c) **Opt (options)**: This method is used to configure the gun instance with various options. It takes an options object as the argument and applies it to the gun instance. Some of the options that can be passed are peers, localStorage, radisk, store, and super. The gun.opt method returns the gun instance itself, so it can be chained with other methods.
d) **User Create(alias, pass, callback, opt)**: This method is used to create a new user account in the gun database. It takes an alias string, a password string, an optional callback function, and an optional options object as arguments. The callback function receives an acknowledgment

object as a parameter, which contains information about the success or failure of the operation. The options object can be used to specify additional parameters, such as change, pin, or prove. The gun.user.create method returns the user instance itself so that it can be chained with other methods.

e) **User Auth(alias, pass, callback, opt)**: This method is used to authenticate an existing user account in the gun database. It takes an alias string, a password string, an optional callback function, and an optional options object as arguments. The callback function receives an acknowledgement object as a parameter, which contains information about the success or failure of the operation. The options object can be used to specify additional parameters, such as change, pin, or prove. The gun.user.auth method returns the user instance itself, so it can be chained with other methods.

f) **User Leave()**: This method logs out from the current user account in the gun database. It takes no arguments and returns nothing. It clears any cached data and removes any listeners associated with the user account.

g) **User Delete(pass, callback):** This method deletes the current user account in the gun database. It takes a password string and an optional call-back function as arguments. The callback function receives an acknowledgement object as a parameter containing information about the operation's success or failure. The gun.user.delete method returns nothing.

h) **Load(cb, opt):** This method loads all data from a gun reference into memory. It takes a callback function and an optional options object as arguments. The callback function receives each node of data as a parameter and can perform any operations on it. The options object can be used to specify additional parameters, such as wait or off. The gun.load method returns nothing.

2.5.3 Security features of our architecture

Our architecture ensures security so only the users own their data and can share it with people they trust. Some of the security features of our architecture are:

a) **End-to-end encryption:** GunJS uses a library called SEA (Security, Encryption, Authorization) for end-to-end data encryption. SEA lets users create secure user accounts with normal logins and even peer-to-peer password resets using 3FA (three-factor authentication). SEA also encrypts data with public-key cryptography, so only the user and the people they trust can access it.

b) **Decentralisation:** GunJS does not require servers or cloud services to operate, so users do not have to rely on third parties to store or sync their data. Users own and control their data and can choose who to share it.

c) **Conflict resolution:** GunJS uses a peer-to-peer protocol that automatically handles conflicts, errors, and data consistency. It uses a CRDT algorithm that ensures the data is always conflict-free and consistent across all devices, even when they are offline or disconnected. Users do not have to worry about losing or corrupting their data due to network failures or malicious attacks.

These are some of the security features of GunJS that position it as a suitable database for advocates of freedom.

2.6 Integration of Domain-Specific & Application-Specific Ontologies

Overview

Ontologies explicitly define concepts (terms) and represent domain entities and relations, thereby reflecting a specific conceptualisation of reality [9]. The storage and dissemination of specialised knowledge have found an innovative approach in the form of a domain-specific ontology. This ontology encompasses a comprehensive array of concepts within a specific field and their interconnections and distinctive attributes. As a result, it provides a cutting-edge platform for preserving and sharing expertise precisely tailored to that particular domain's needs [10]. Application-specific ontologies refer to ontologies designed and developed for a particular application domain, addressing the unique requirements and characteristics. These ontologies are tailored to capture the specific concepts, entities, and relationships relevant to the targeted application, allowing for more efficient and accurate knowledge representation and retrieval within that domain [11]. Integrating ontology into T2.7 to define Data Sharing Units (DSUs) as JSON schema involves creating domain-specific ontologies connected to the JIDEP platform via Ontology API that represent the concepts, entities, and relationships relevant to the specific use cases and application domains. These ontologies served as the foundation for defining the DSU Types, encapsulating data, and coding for ontology manipulation. A domain-specific ontology facilitates efficient data sharing and manipulation by providing a standardised and shared understanding of domain knowledge. Advantages include improved data interoperability, enhanced information retrieval, and better decision-making capabilities. The ontology ensures consistency, reduces ambiguity, and enables seamless integration with various applications and data sources. Moreover, it promotes reusability and scalability, allowing for easier adaptation to evolving domain requirements, ultimately leading to more effective knowledge management and problem-solving within the specific domain and its application.

The categories in Table 1 cover a broad range of attributes associated with entities or objects. Identifiers provide unique labels for referencing, while physical, chemical, and biological properties offer insights into their intrinsic characteristics. Circular economy attributes focus on sustainable resource usage, and environmental performance properties gauge an entity's environmental impact, facilitating informed decision-making and assessment of ecological implications.

Table 3: Categories of a Material Passport

Category	Summary	Status
Identifiers	The identifiers section in the provided sample encompasses essential details for the entity's identification. It includes a brand name, manufacturer's details (name, registration country, registration number), product name, trade name, and the entity type (product). This comprehensive information contributes to a clear and accurate identification of the entity, enhancing its traceability and recognition within relevant contexts.	Defined
Physical Properties	These properties collectively provide insights into the entity's physical characteristics, including size, thermal behavior, and weight. Detailing these aspects facilitates a comprehensive understanding of the entity's physical	Defined

	attributes and behaviour.	
Composition Properties	This information offers insights into the underlying constituent materials of the entity, aiding in understanding its makeup.	Defined
Circular Economy	The circular economy attributes provided in the sample offer insights into sustainable resource management. These include end-of-life (EoL) use potentials, recycling practices, and indicators of circularity. The recycling details encompass materials and processes, while the circularity indicator and related values quantify the extent of resource recycling and waste reduction, facilitating a clear assessment of the entity's contributions to circular economic practices.	Defined
Environmental Performance	The environmental performance details in the provided sample encompass various aspects. This includes measurements such as carbon footprint, impact on human health, aquatic acidification, and water footprint, each quantified with values (though units are not provided). Additionally, the functional unit and impact assessment methodology play a role in gauging the entity's environmental impact. This comprehensive data enables an evaluation of the entity's ecological effects and resource usage, enhancing understanding and decision-making regarding its environmental performance.	Defined

Utilizing JSON Schema as a DSU type brings structured clarity to Data Sharing Units (DSUs). By defining the schema, we precisely outline the expected format, data types, and validation rules for each DSU. This JSON Schema ensures that the data encapsulated within DSUs adheres to predetermined standards, reducing errors and enhancing data reliability. JSON Schema acts as a blueprint, allowing seamless integration and interaction between different DSU instances, applications, and APIs. Its flexibility also permits easy adaptation to changing requirements, making it a robust foundation for creating DSUs that efficiently store, exchange, and manipulate domain-specific information while maintaining data integrity.

Here's a tabular representation of use case-specific DSU types for the Automotive, PCB, and Wind Turbine domains:

Table 4: Domain-specific DSUs

Use Case	**DSU Type**	**Description**
Automotive	Automotive DSU	This DSU facilitates efficient storage, sharing, and manipulation of automotive-related information, enhancing interoperability and decision-making within the automotive sector.
PCB	PCB DSU	This DSU streamlines the storage and exchange of PCB-related data, facilitating seamless collaboration and efficient management within the electronics industry.
Wind Turbine	Wind Turbine DSU	This DSU optimizes the storage and sharing of wind turbine data, fostering effective collaboration and informed decision-making within the renewable energy domain.

Here is an example of an Automotive DSU type for Automotive use case in JSON Schema format,

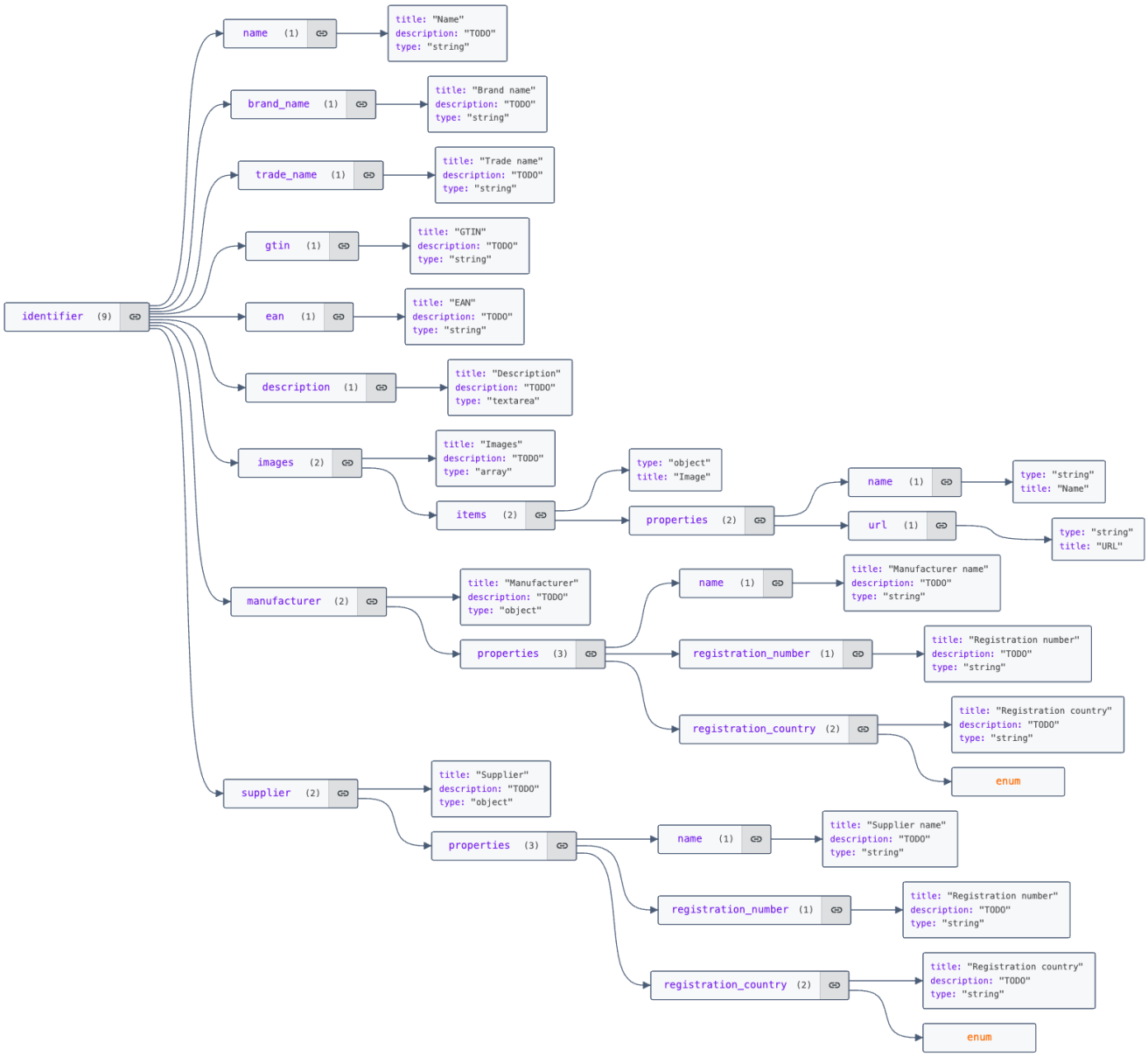


Figure 5: identifier section

2.6.1 Multi-Domain Ontology Toolkit

The Multi-Domain Ontology Toolkit we've developed is a versatile solution to streamline ontology manipulation and JSON schema generation processes. It is a powerful toolkit for querying domain and application-specific ontologies while creating JSON-based schemas that adhere to the defined ontology structures.

JSON Schema Generation:

The Multi-Domain Ontology Toolkit dynamically generates JSON schemas that comply with the input requirements of the ontology-based user interface generation engine, enabling the creation of material/product passports. It queries properties, subclasses, instances, and units associated with the product class in the JIDEP ontology suite uploaded to the JIDEP Knowledge Graph (KG). The toolkit has the potential to enable circularity by supporting industrial users in providing data, among others, about recyclable composite material-made products of the automotive and wind turbine industries. UCAM has designed an algorithm that underlies this toolkit. The algorithm takes an ontological class as the input and composes a SPARQL query to retrieve all object and data properties directly associated with the input class. Retrieved properties, along with data type or range information, are included in a JSON object. It traverses through the object properties by retrieving the classes defined in the range to identify their properties until the termination condition is evaluated as accurate. These properties are appended to the JSON object under corresponding classes. If the property retrieving query returns an empty set of properties, it composes and performs a subclass retrieving query. If the result of the subclass query is an empty set, it conducts an instance retrieval query. If the outcome of the instance query does not contain anything, it composes and performs a query to retrieve the unit and has-value property. It terminates if the unit and value query result is empty, and no other classes are available for further querying for properties, subclasses, instances or units. This toolkit promotes consistency and alignment and expedites the development of schema and ontology-driven semantic-enabled applications and APIs.

2.6.2 Ontology APIs:

The Ontology API is a part of the Joint Industrial Data Exchange Pipeline (JIDEP) Platform, which is responsible for generating schemas (DSUs) based on the ontology developed by UCAM. These schemas (DSUs) represent the structure and properties required for creating Material Passports. The API exposes endpoint(s) to access schema information for different aspects of the Material Passport. These schemas will be utilised to create user interfaces in the JIDEP Platform to capture relevant data during the Material Passport creation process. A detailed overview of Ontology API is as follows:

Ontology Development: UCAM has developed an ontology, a formal representation of knowledge in a specific domain related to materials, products and their properties. The ontology defines concepts, relationships, and properties associated with Material Passports.

Ontology API Endpoints: The API exposes endpoint(s) to retrieve schema information for different categories of properties of the Material Passport creation process. The specific endpoint(s) similar to /schema/step/{name}, where the name represents a step of the Material Passport.

Table 5: API endpoints of Jidep Ontology

Method	Endpoint	Description	Returns
GET	/schema/category/identifier	Get schema for the "Identifiers" category.	Schema as JSON format from a file or Ontology.

GET	/schema/category/physical_properties	Get schema for the "Physical Properties" category.	Schema as JSON format from a file or Ontology.
GET	/schema/category/composition_properties	Get schema for the "Chemical Properties" category.	Schema as JSON format from a file or Ontology.
GET	/schema/category/circular_economy	Get schema for the "Circular Economy" category.	Schema as JSON format from a file or Ontology.
GET	/schema/category/environmental_performance	Get schema for the "Environmental Performance" category.	Schema as JSON format from a file or Ontology.

Properties Validation: When a request is made to the Ontology API's endpoint with a specific name, the API validates whether the provided name corresponds to a valid step in the Ontology. If not, it may raise an appropriate error response (e.g., 404 - Not Found).

Ontology Schema Generation: Upon receiving a valid request, the API accesses the Ontology developed by UCAM to extract the relevant schema information corresponding to the specified step name. This process involves querying the Ontology and retrieving the necessary structure and property details.

Schema Representation: The extracted schema information is then formatted into a suitable representation in JSON format, which is appropriate for conveying the schema structure and properties.

Response: The API sends the generated schema as part of the response to the client, allowing the client application (such as the JIDEP Platform) to access the schema information required for creating Material Passports.

User Interfaces Creation: The JIDEP Platform dynamically generates user interfaces using the received schema information. These forms are designed to capture specific data related to the Material Passport, following the structure and properties defined in the schema. Users interact with the forms to provide necessary information for each step.

Material Passport Creation: The data collected through the generated forms is used to create comprehensive and standardized Material Passports containing all the relevant information about the materials and their properties.

3. Adaptation in Use Cases

The Ontology API and Multi-Domain Ontology Toolkit (developed by UCAM) combine standardised data representation and schema generation for various industries. This overview highlights their application, as shown in Figure 9, in JIDEP's three use cases: the Automotive Industry, the Wind Turbine Sector, and the PCB Industry. In the JIDEP project, we leverage an ontology schema to design user-friendly interfaces for our tools, ensuring consistency and adaptability. This schema significantly enhances the functionality of the tools, including the Environment Analytic Tool for analysing environmental data, the Material Passport for tracking material lifecycles, the Circularity Calculator for assessing sustainability, and the Collaborative Space for facilitating stakeholder communication and data sharing. By adopting this approach, we improve user interactions and make our tools more efficient and effective, enhancing the overall end-user experience.

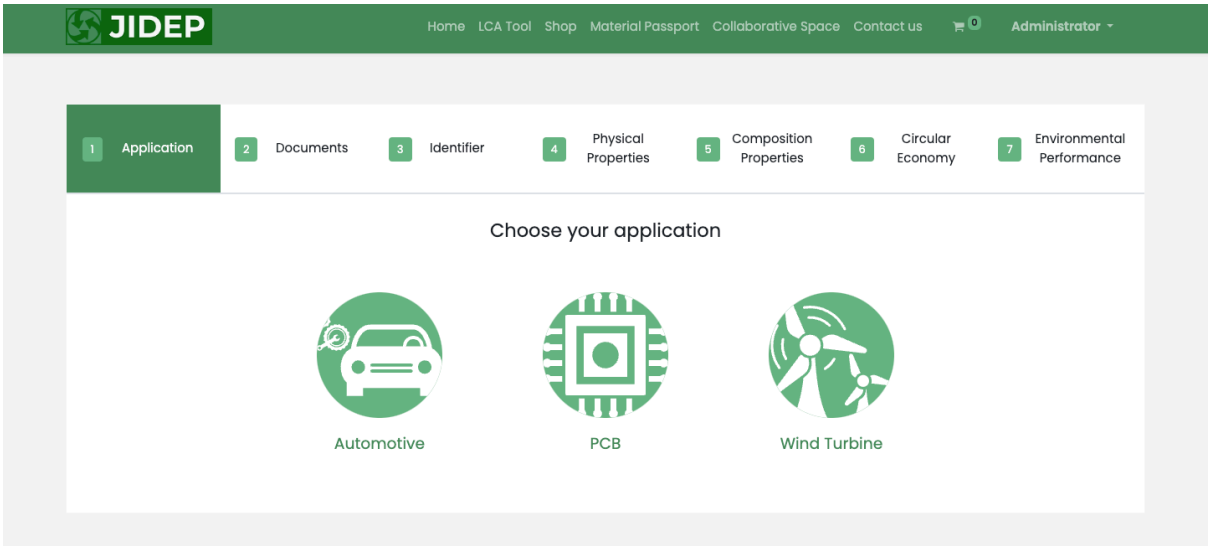


Figure 6: Use Case Selection

The screenshots provided in Figure 7 and Figure 8 illustrate the automatically generated user interfaces for the JIDEP tools from the ontology schema. Deliverable D4.1 provides further details about these tools.

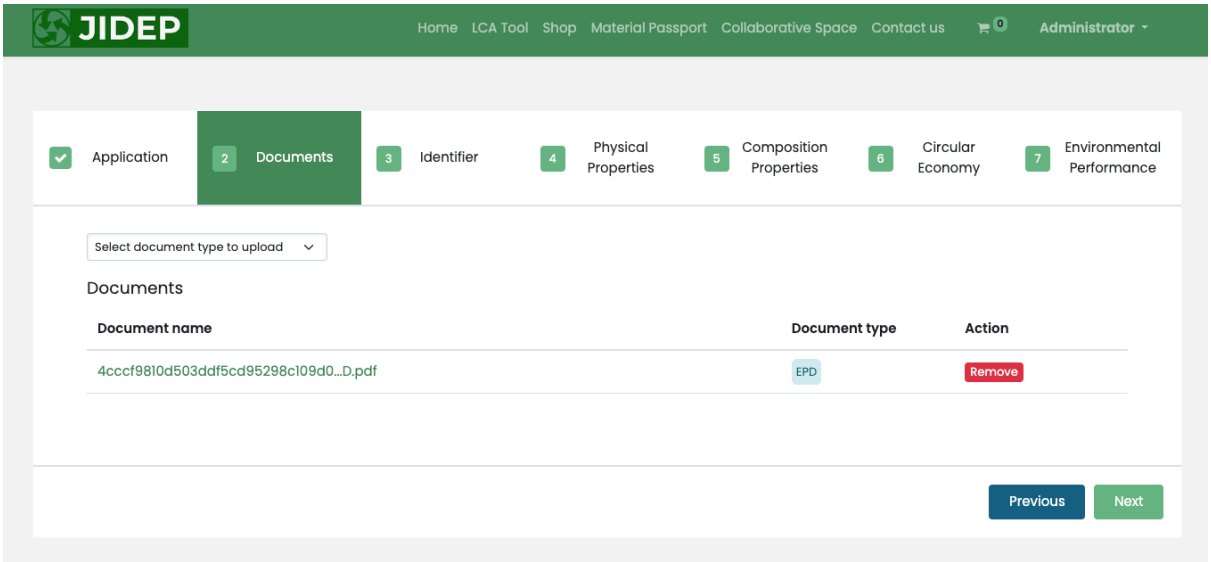


Figure 7: Documents section for Material Passport

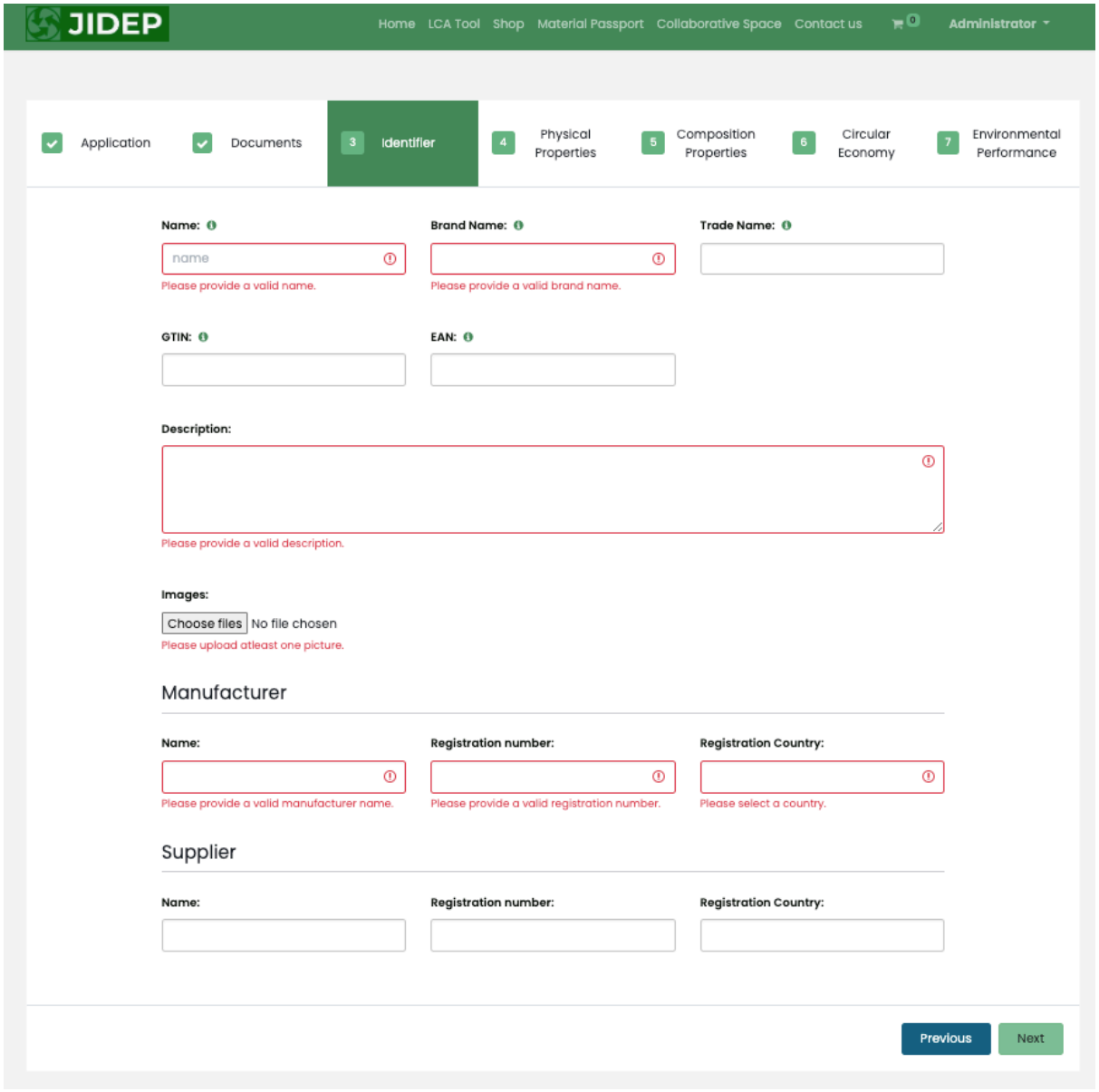


Figure 8: Material Passport Identifier Section

4. Conclusions

The successful integration of the Ontology API, ontology and Multi-Domain Ontology Toolkit developed by UCAM within the Automotive, Wind Turbine, and PCB industries signifies a remarkable achievement in advancing standardised data representation, streamlined information exchange, and informed decision-making across these sectors. Each industry has meticulously harnessed the Ontology APIs and Multi-Domain Ontology Toolkit's potential to fulfil specific objectives and tackle domain-centric challenges. The beta version of D2.7 was dedicated to refining the automotive use case and crafting a specialized data-sharing unit (DSU) tailored to the intricacies of this domain. This iteration showcased our unwavering commitment to precision and innovation. Excitingly, this final version of D2.7 encompasses not only the Automotive use case but also further polishes the Wind Turbine and PCB use cases. This expansion is a testament to our dedication to holistic solutions that cater comprehensively to diverse domains.

This report has effectively demonstrated the meticulous criteria and methodology employed in selecting the distributed data contribution technology for JIDEP. Furthermore, it delved into the intricacies of designing and implementing our distributed storage tools, a pivotal aspect in achieving the targeted objectives of JIDEP's use-case-specific applications. The development of the Distributed Storage Peers and Server is a noteworthy milestone. The report elucidated that while the Distributed Storage Peers effectively managed user data securely and distributed, the Distributed Storage Server ensured data backup and restoration, with real-time synchronisation ensuring data consistency. With an enhanced and comprehensive experience, we uphold our commitment to excellence and adaptability, reflecting our dedication to pioneering advancements within the JIDEP project.

References [11]

[1] J. Vaughan, “What is Data,” 2019. [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/data. [Accessed 2023].

[2] P. Hooda, “Comparison – Centralized, Decentralized and Distributed Systems,” Geeks for Geeks, 24 12 2018. [Online]. Available: https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems/. [Accessed 22 08 2023].

[3] B. Gaille, “15 Centralized Database Advantages and Disadvantages,” BrandonGaille.com, 16 07 2018. [Online]. Available: https://brandongaille.com/15-centralized-database-advantages-and-disadvantages/. [Accessed 18 07 2023].

[4] K. Taylor, “Everything you need to know about Decentralized Storage Systems,” HitechNectar, [Online]. Available: https://www.hitechnectar.com/blogs/everything-you-need-to-know-about-decentralized-storage-systems/. [Accessed 01 08 2023].

[5] J. Howell, “What is Decentralized Storage and How does it Work?,” 101 Blockchains, 05 05 2023. [Online]. Available: https://101blockchains.com/decentralized-storage/. [Accessed 01 08 2023].

[6] “Distributed Systems: An Introduction to Distributed Computing,” Confluent, 2014. [Online]. Available: https://www.confluent.io/learn/distributed-systems/. [Accessed 07 06 2023].

[7] A. Francis, “Characteristics of a Distributed System,” MBA Knowledge Base, 13 08 2015. [Online]. Available: https://www.mbaknol.com/information-systems-management/characteristics-of-a-distributed-system/. [Accessed 25 07 2023].

[8] “Advantages and Disadvantages of Distributed System,” Java T Point, 2020. [Online]. Available: https://www.javatpoint.com/advantages-and-disadvantages-of-distributed-system. [Accessed 27 07 2023].

[9] P. &. M. R. &. M. P. Faber, “Linking a Domain-Specific Ontology to a General Ontology,” in *Proceedings of the 24th International Florida Artificial Intelligence Research Society, FLAIRS - 24*, 2011.

[10] N. &. M. D. Noy, “Ontology Development 101: A Guide to Creating Your First Ontology,” *Knowledge Systems Laboratory,* vol. 32, 2001.

[11] T. D. &. S. E. D. Noia, The Semantic Web - ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers, Crete, Greece: Springer, 2016.

