future internet

MDPI

Article

Entering the Metaverse from the JVM: The State of the Art, Challenges, and Research Areas of JVM-Based Web 3.0 Tools and Libraries

Vlad Bucur [1] and Liviu-Cristian Miclea [2,*]

[1] LSGE (London Stock Exchange Group) Romania, Bulevardul Iuliu Maniu 6G, 061344 București, Romania; vbucur1@gmail.com
[2] Department of Automation, Technical University Cluj-Napoca, 400114 Cluj-Napoca, Romania
* Correspondence: liviu.miclea@aut.utcluj.ro; Tel.: +40-740-135-928

Abstract: Web 3.0 is the basis on which the proposed metaverse, a seamless virtual world enabled by computers and interconnected devices, hopes to interact with its users, but beyond the high-level project overview of what Web 3.0 applications try to achieve, the implementation is still down to low-level coding details. This article aims to analyze the low-level implementations of key components of Web 3.0 using a variety of frameworks and tools as well as several JVM-based languages. This paper breaks down the low-level implementation of smart contracts and semantic web principles using three frameworks, Corda and Ethereum for smart contracts and Jeda for semantic web, using both Scala and Java as implementing languages all while highlighting differences and similarities between the frameworks used.

Keywords: metaverse; Web 3.0; IoT; blockchain; semantic web

check for updates

Citation: Bucur, V.; Miclea, L.-C. Entering the Metaverse from the JVM: The State of the Art, Challenges, and Research Areas of JVM-Based Web 3.0 Tools and Libraries. *Future Internet* **2023**, *15*, 305. https://doi.org/10.3390/fi15090305

Academic Editor: Michael Sheng

Received: 1 July 2023
Revised: 15 August 2023
Accepted: 5 September 2023
Published: 7 September 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Java is one of the most popular programming languages in the world as of today. According to an ample study conducted by HackerRank in 2020, Java ranks third in terms of most sought-after programming languages by hiring managers [1], and second in the best-known languages ranking for 2020 [2]. Additionally, Java has ranked as one of the most popular languages in the TIOBE Index since it has begun keeping a record of programming languages by their popularity in 2001 [3]. Given the popularity of Java, it is only natural to inquire about the tools available to developers when working with Web 3.0 and metaverse-specific applications.

The importance of the metaverse, which incorporates many Web 3.0 elements, extends beyond just the marketing appeal of the brand, closely tied to social media giant Facebook. The metaverse is poised to become one of the key pillars of human–machine interaction. What makes the metaverse interesting from a human–machine interaction standpoint is its abundant use of virtual and augmented reality technology. The basis of the metaverse concept rests on the creation of a persistent virtual world in which users could become "lost" and live separate, virtual lives. Even more interesting, though, is the likelihood of overlap between the virtual world and the real world, in the form of augmented reality. In comparison to virtual reality, augmented reality projects virtual images onto the real world, therefore making it a very good fit for bridging the divide between the fully virtual metaverse and the real world, especially in terms of business and revenue opportunities.

Given how great an impact the metaverse is likely to have on the virtual landscape, from things as "simple" as websites to complex applications and videogames, it is important to understand exactly how the metaverse operates on a low level, from a development perspective. This article explores a number of key frameworks and libraries used for developing two of the core components of Web 3.0, which rests at the core of the metaverse. The

two core components selected for analysis are blockchain applications, with an emphasis on smart contracts, and semantic web applications. The aim of this article is to provide detailed descriptions and analysis of these frameworks while comparing and contrasting the available solutions to each other, where applicable.

This article is broken down into six sections, including the introductory section, a section giving an overview of Web 3.0 and its core components, as well as its connection to the metaverse, and a brief overview of JVM tools. The third section is an ample analysis of blockchain technology and smart contracts, including a comparison between two of the most popular blockchain frameworks available in JVM-type languages, Corda and Ethereum. The fourth section focuses on implementing semantic web concepts using Apache Jena, an open-source semantic web framework written in Java, while the final section discusses research challenges in Web 3.0 in relation to the metaverse, Web 2.0, and IoT before, finally, leading into a series of conclusions.

2. Definitions and Methodology

The concept of Web 3.0 started emerging much earlier than the coining of the term "Metaverse" in 2021 by Meta, the parent company of Facebook. Web 3.0, known initially as the Sematic Web, was an initiative to make Web 2.0 machine-interpretable through heavy use of metadata and actually emerged sometime in the early 2000s. Various authors, namely O. Lassila et al. and J. Hendler in their papers [4,5], peg the "birth" of Web 3.0 at the beginning of the 21st century. The term was popularized in an article written by J. Markoff in The New York Times in 2006 which explored Web 3.0′s viability as a business model [6].

In the context of the metaverse, Web 3.0 acts as the "vehicle" which enables users to enter the virtual worlds of the metaverse. Concepts that have now been associated with Web 3.0, such as blockchain, AI, augmented reality (AR), virtual reality (VR), and decentralization are also considered intrinsically linked to the metaverse. Blockchain is a well-known concept that has witnessed a rise in notoriety due to its pivotal role in enabling cryptocurrency and NFT transactions. In the metaverse, cryptocurrencies are to be used as the de facto currency for any sort of transactions, therefore making blockchain a cornerstone of metaverse development. The concept of using cryptocurrencies is linked to another key factor of Web 3.0, decentralization. In the metaverse this leads to the elimination of middlemen like banks or financial institutions, and the decentralization of monetary policy. However, it is important to note that blockchain also has other uses in the metaverse such as recording options in virtual elections or preserving the privacy of data.

The role of AI and Machine Learning is not as clear in the metaverse as that of blockchain. Since the metaverse is in the very early stages of development, and is currently void, the role of AI is not very clear. In Web 3.0, AI and ML are linked to the semantic web. Artificial intelligence develops a better understanding of the context in which Web 3.0 functions by using the semantic web's metadata. This is true of an AI's basic functioning principles in the metaverse as well, but what exactly the purpose of AI will be for the end user is unclear. However, that does not stop one from speculating on a number of AI applications in the metaverse: to help users with searches, act as guides, automate day-to-day tasks, and more.

From a software development and computer programming perspective, all of Web 3.0′s facets can provide several challenges of varying degrees of difficulty. A simple use case, such as securing a cryptocurrency wallet, can quickly become complex when factoring in Web 3.0′s, and the metaverse's, decentralized nature. Wallets would be available on multiple devices, with different operating systems, could be accessed by different applications, both for viewing purposes and for purchasing, and would need tracking through blockchain-ledgers, involving smart contracts to some degree. A software developer would need to take into account all of these factors, as well as concerns related to architecture, design and clean code when implementing a Web 3.0 application.

Java is one of the most popular programming languages in use today, and it behooves us to take a closer look at specifically which tools and libraries from the vast Java ecosystem

are most popular, and useful, in developing Web 3.0 applications inside the metaverse. The robustness of Java as a software development ecosystem, not just as a programming language, makes it an interesting candidate for an in-depth Web 3.0 tooling review. The Java Virtual Machine (JVM) is one of the most performance-driven runtime environments currently available and has expanded its list of supported languages greatly. While Java is still, by and large, the most popular programming language that uses the JVM, others have emerged recently. The most notable of these is Kotlin. Kotlin strays from the strongly typed, verbose nature of Java and moves towards a freer flowing programming semantic while still making use of the power of the JVM.

Finally, JVM-based languages are not even required to solely use the out-of-the-box implementation of the Java Virtual Machine as a runtime environment anymore. A popular alternative to the JVM is GraalVM which allows developers to run native images on HotSpot JVM, with Graal just-in-time complier or even as an ahead-of-time compiled native executable [7]. Even more interesting is the prospect of running not just Java code in GraalVM's runtime environment, but Python or JavaScript code as well.

Having presented a brief overview of the building blocks on which the rest of this paper rests, the following two sections will now focus on the specifics of implementing JVM-based solutions for two of the cornerstones of Web 3.0 and metaverse development: blockchain and semantic web.

The methodology used during the sections pertaining to blockchain and semantic web was as hands-on as possible. Applications were implemented using all three of the libraries or frameworks described in each section. Coding was completed in Java, where applicable, using guides provided by the framework developers and/or coding samples exposed through public, open-source repositories. The proprietary code on which the in-depth analyses and descriptions are based is not presented in this paper, the authors having opted to present a generic, high-level UML diagram for each type of implementation. The types of implementations analyzed can be broadly categorized as smart contracts and semantic web applications.

The scope of the paper is, therefore, to compare Java-based libraries and frameworks from both a development and research perspective. The aim is to provide a breakdown of each of the main features of a library or framework and present the conclusions in an easily digestible way for both types of readers: academics and software development professionals.

3. Blockchain

Blockchain is a type of database that is managed by a network of computers, rather than a central authority, making it virtually impossible to alter or tamper with the records. The blockchain ledger consists of blocks containing a set of transactions. Once a new transaction is made, it is verified by a network of computers (nodes) that use complex algorithms to confirm its validity. Once the transaction is verified, it is added to the latest block, and the block is added to the existing chain of blocks, forming a permanent and immutable record. Blockchain security is maintained through cryptography and each transaction is secured by using a hash function. Figure 1 is part of the official development documentation of BitCoin, freely available for download on GitHub.com, and illustrates the basic implementation of transactions using witnesses in a blockchain context.

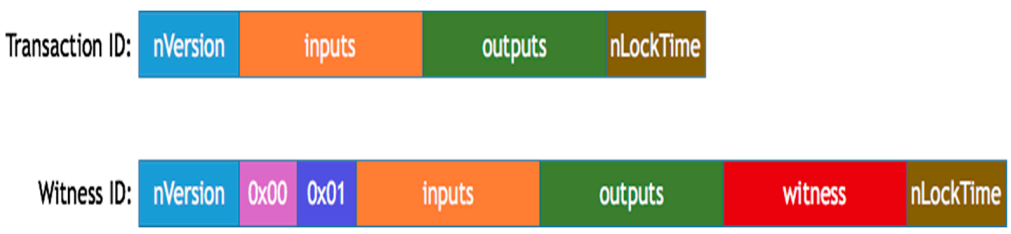

Figure 1. Transaction and witness hashing [8].

One of the key uses of blockchain is smart contracts. Smart contracts are self-executing computer programs that automatically enforce the rules and conditions of a contract. They make use of blockchain technology, because of its decentralization, immutability, and transparency in recording transactions. Once a smart contract is deployed on a blockchain network, it cannot be altered or tampered with, ensuring that all parties involved in the contract have access to the same information and that the terms of the contract are enforceable. Smart contracts can also be programmed to trigger automatic actions when certain conditions are met, such as releasing funds to a supplier once a shipment has been delivered.

In the context of the metaverse, smart contracts can be used to automate and enforce the rules of various transactions that take place within the virtual world. For example, smart contracts can be used to facilitate the buying and selling of virtual assets, such as in-game items or virtual real estate. They can also be used to create decentralized autonomous organizations (DAOs) within the metaverse, where members can vote on important decisions using a transparent and secure system.

Smart contracts can also be used to create unique non-fungible tokens (NFTs) that represent virtual assets within the metaverse. These NFTs can be bought, sold, and traded just like physical assets, and their ownership and transaction history are recorded on the blockchain. This creates a new economy within the metaverse, where users can create, own, and trade valuable virtual assets.

Presently, there are three well-known smart contract platforms employing JVM languages. These libraries are commonly used alongside other types of programming languages when implementing smart contracts:

- Corda is an open-source blockchain platform that supports the use of Java and Kotlin for developing smart contracts. Corda smart contracts are called "CorDapps" and can be written in Java or Kotlin using the Corda framework. CorDapps are designed to be highly modular, allowing developers to easily plug in their own custom code.
- Ethereum is a popular blockchain platform that supports the use of Solidity, a programming language that is similar to JavaScript. However, it is also possible to use other languages, such as Java, with tools such as EthereumJ or the Web3j library.
- Hyperledger Fabric is a permissioned blockchain platform that supports the use of multiple programming languages, including Java, JavaScript, and Go. Smart contracts on Hyperledger Fabric are called "chaincode" and can be written in any of these supported languages.

These smart contract platforms, along with four others, were comparatively analyzed by M. Suvitha and R. Subha, in an article published in 2021, and while the authors did not come to a clear conclusion as to which platform is the most fully featured overall, they did highlight each platform's unique features and capabilities. The analysis was not limited to only the programming languages supported, but also covered consensus mechanisms, scalability, interoperability, privacy, and security [9].

The two most JVM-friendly platforms, Corda and Ethereum, have their own strengths. In the case of Corda, its strengths lie in the ability to support complex, enterprise-level use cases that require privacy, scalability, and interoperability. Its focus on interoperability also makes it a good choice for businesses looking to integrate blockchain technology into their existing systems and workflows [9].

Ethereum's strengths, on the other hand, lie in its ability to support a wide range of use cases, its popularity and tooling. By comparison with Corda, its focus on programmability and decentralized application development makes it an attractive choice for businesses and developers looking to build decentralized applications and services, with a particular focus on NFTs [9]. In fact, one of the major differences between Corda and Ethereum, especially from a metaverse perspective, is Ethereum's NFT functionality, which enables it to create games and digital assets that are unique and verifiably scarce [9].

This section of the paper will analyze smart contract implementations in Corda and Ethereum to highlight the ways in which these platforms can be used in the metaverse.

3.1. Corda

The first application analyzed in this article uses Corda to implement a smart contract for financial transactions, as financial transactions are one of Corda's strengths. Figure 2 is a UML-like diagram of the most important classes involved in the creation of a Corda smart contract. Elements of Figure 2 will be used throughout this subsection to analyze the implementation of a Corda blockchain application.

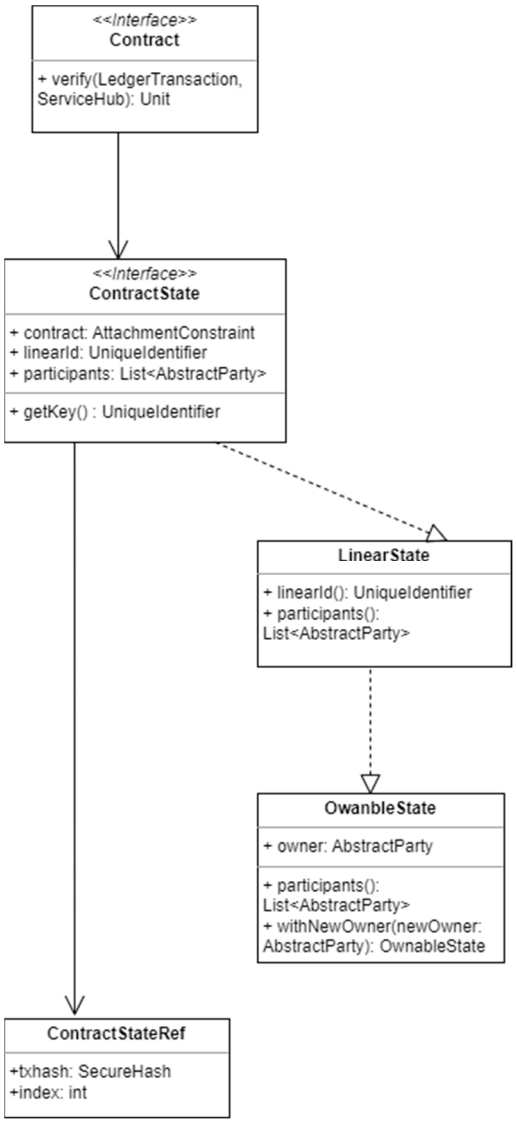

Figure 2. Basic classes and interfaces involved in a Corda smart contract implementation.

Contract is the key interface of any transaction involving Corda smart contracts. It is the interface implemented by all contracts, and while it does not have any direct subclasses, it provides a key method to all of the classes that implement it: the verify method. This is the main method of the Contract interface, and it contains the contract verification logic that is executed when a transaction is validated by a node. The verify method takes two parameters: a LedgerTransaction object, which represents the transaction being verified, and a ServiceHub object, which provides access to various services provided by the node, such as the vault and the network map.

An application handling financial transactions would use the verify method to check things such as the following: single transactionality per verification, uniqueness of payer and payee, checking the participants' keys, single states for financial transaction, and more.

The class composition of this sample financial transaction would also implement ContractState, the interface implemented by all state objects that are used as part of a contract. In ContractState, contract is a reference to the contract that governs this state, while linearId is an instance of UniqueIdentifier which is the unique identifier of a particular instance of a state. Participants, on the other hand, is a method that returns a List<AbstractParty>, where AbstractParty is an abstract class that represents a party on the Corda network. The list contains all parties that have some sort of right or interest in the state, such as the current owner or the regulator overseeing the contract.

There are multiple types of states agreements can have, all derived from ContractState, including OwnableState and LinearState, all of which work within the framework of a Corda smart contract to achieve different results when managing a transaction.

OwnableState is an interface that extends ContractState and adds the functionality of keeping track of the current owner of the state. This means that an OwnableState object contains an owner field, which is an instance of a Party object representing the current owner of the state. The purpose of OwnableState is to provide a simple way to track ownership of assets on the Corda network. By implementing the OwnableState interface, developers can easily create assets that can be owned and transferred between parties on the network. When the ownership of an asset changes, the withNewOwner method is called to create a new instance of the state with the new owner information.

OwnableState is often used in conjunction with other Corda features, such as the TokenSDK, to create digital assets that can be traded on the network.

LinearState is another type of ContractState that represents a state which evolves over time and can be uniquely identified using a linear ID. The linear ID is a unique identifier that remains constant throughout the life of the state, even if the contents of the state change. This allows transactions to refer to the state unambiguously and ensures that the state can only be consumed once. LinearState is used in scenarios where a state is expected to represent an ongoing agreement between parties that evolves over time. Examples include loans, insurance policies, and supply chain contracts. By using a linear ID to uniquely identify a state, parties can track the evolution of the agreement over time and ensure that all parties agree on the current state of the agreement.

These states are tied together by ContractStateRef, an object that references a specific instance of a ContractState, at a particular point in time. It consists of a hash of the transaction that created the state, as well as the index of the state within the transaction's outputs. ContractStateRef is an immutable data class that provides methods for serializing and deserializing reference as well as for creating new instances with different transaction hashes or output indexes.

When creating transactions, it is common to reference output states from previous transactions as inputs to new transactions. In such cases, a ContractStateRef is used to uniquely identify the output being referenced. This allows the transaction's contract code to verify that the referenced output has not been consumed or modified since it was created.

By combining all of these key classes together, Corda allows developers to secure, create, and verify smart contracts, keep a ledger of transactions, and ensure the uniqueness of each transaction, therefore ensuring that contracts in a virtual world, the metaverse, mimic their real-world counterparts as closely as possible.

3.2. Ethereum

Another major use of blockchain is NFTs. Non-Fungible Tokens (NFTs) combine both blockchain and smart contracts technology to create digital assets that work like physical assets. Using blockchain technology, NFTs are able to guarantee the authenticity and uniqueness of each digital asset. This makes them valuable for collectors and investors. Blockchain helps keep NFTs unique by using three key elements: a cryptographic hash function that creates a unique token which is then stored in blockchain as the NFTs unique identifier, smart contracts which can define the rules of creation and ownership of the NFT

and, finally, a consensus mechanism employed by the blockchain network where all nodes must agree on the creation and transfer of NFTs.

One of the most popular choices for coding NFTs is the Ethereum platform, and its Java toolkit called Web3j. Web3j is a Java library that provides integration with Ethereum blockchain. With Web3j, developers can interact with smart contracts and the Ethereum network, as well as manage accounts and transactions. Web3j supports the latest Ethereum specifications and provides features such as contract development and deployment, event filtering, and integration with various platforms.

Web3J provides a Java API to interact with the Ethereum network and the smart contracts deployed on it. It supports generating Java classes from Solidity smart contracts through the Solidity compiler, which makes it easier to interact with smart contracts using Java code. The generated Java classes correspond to the Solidity contract and include all its public functions, events, and variables. These classes can be used to create, deploy, and interact with the smart contract. This differs significantly from the Corda approach, where classes can be written natively in Kotlin, a JVM-based language.

Ethereum is a public network; as such, in order to keep transactions transparent and safe, it has several standards for tokens in place, most notably ERC-20 and ERC-721. The Corda network uses its own token standard by comparison called the Corda Token SDK. ERC-20 is a fungible token standard, meaning that each token is identical and interchangeable with other tokens of the same type whereas ERC-721 is a non-fungible token standard. This means that each token is unique and cannot be exchanged or replaced with another token. The NFT application analyzed throughout the rest of this section uses the ERC-721 token standard, as that is the standard that NFTs must adhere to while using the Ethereum network for blockchain transactions.

The class structure of a Web3j NFT also makes use of contracts, much like a financial transaction created in Corda, as can be seen in Figure 3, but it is slightly less complex. This is due to the Web3j class itself. The Web3j class is the main entry point for interacting with the Ethereum network. It provides methods for connecting to the network, sending transactions, and working with smart contracts.

The Web3j class could even be considered a god-class since it provides multiple functionalities all at once. Not only does it provide users with the ability to connect to the Ethereum network to check the validity of a transaction, but it can also be passed as a parameter during the instantiation of an NFT. In that scenario, the Web3j class acts as a means to connect to Ethereum network and implements the required checks for the NFT to be ERC-721 compliant.

Regardless of the approach in which the Web3j class is fed to an NFT, a token must make use of the Web3j class in order to be considered non-fungible, since the Web3j class is the enforcer of the ERC-721 standard. The ERC721 class, which is part of the Web3j library, acts as a wrapper on the Solidity contract implementing the ERC-721 standard. ERC721 provides an abstraction layer that allows Java code to interact with the Solidity contract in a more user-friendly and type-safe way, while still ensuring that the ERC-721 standard is being met. In order for any token to be ERC-721 compliant, and thus become an NFT, it needs to implement a number of methods. The ERC721 class defines methods that correspond to the ERC-721 standard, such as transferFrom(), balanceOf(), ownerOf(), and so on and therefore enforces non-fungibility in Ethereum.

For developers seeking a way to decouple their NFT from the Web3j class, a more manual approach is available requiring extension of the Contract class. In this case, the token needs to implement all of the methods that make it ERC-721-compliant manually, and it needs to return the result of RemoteCalls made to the Ethereum network to validate its authenticity. RemoteCalls are used in minting the token or during the creation of an NFT that extends Contract. A TransactionManager is used, which includes an instance of the Credentials class that represents the Ethereum account used to sign transactions.

Finally, in order to interact with Solidity, the programming language used by the Ethereum network, Web3j uses a Function class. The Function class is used to represent a

Solidity function in Java code. It also generates the encoded data required to call a Solidity function from Java.

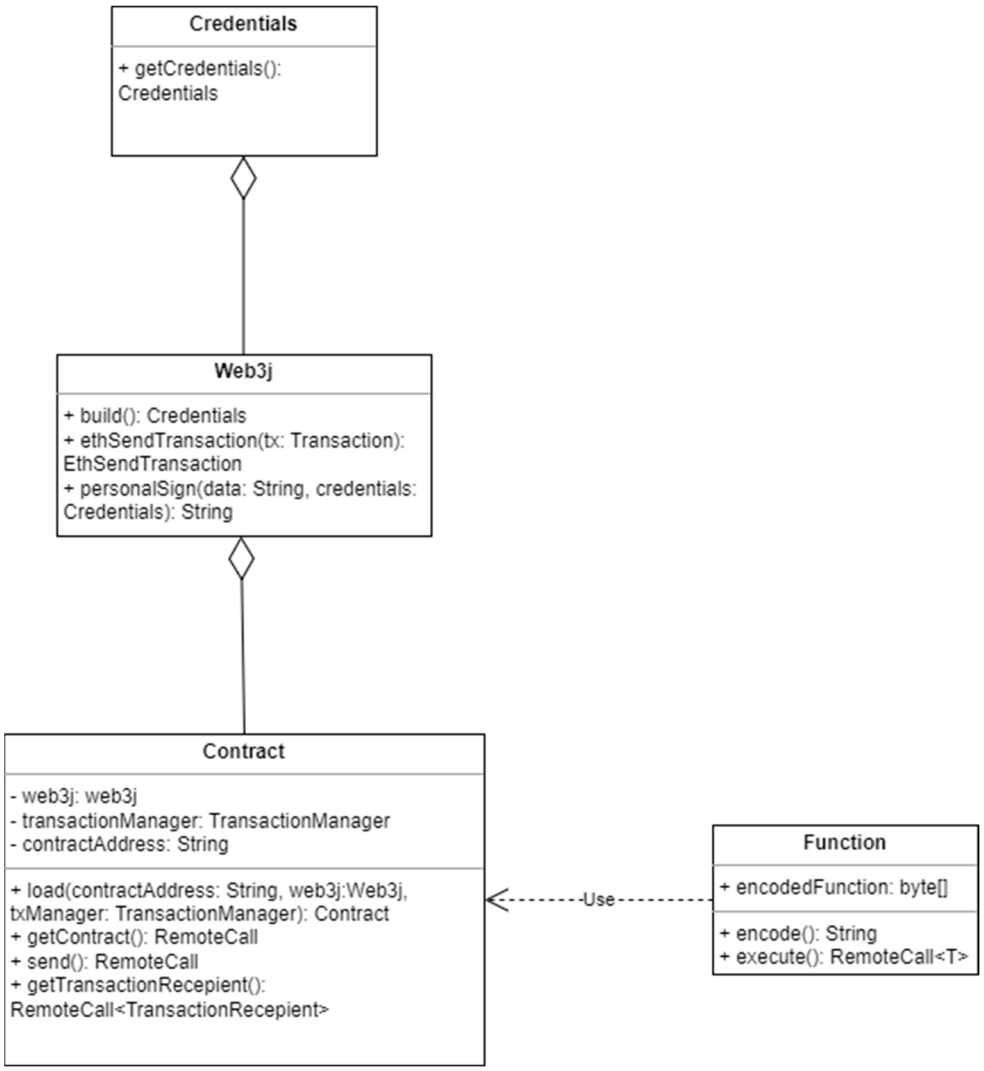

Figure 3. Basic classes involved in creating an NFT with Web3j.

The Function class has several members, including the function name, the input parameters, the output parameters, and the function's signature. It defines methods to generate the encoded function data and to execute the function call. Once a Function object is defined, it can be used to generate the encoded function data by calling the encode method. It can then send this encoded function data to the Ethereum network using the TransactionManager. Finally, it can execute the function by calling the execute method on the Function object, passing in the appropriate TransactionManager object and any other required parameters.

In terms of smart contracts, a comparison between Corda and Ethereum shows that each of the frameworks has different advantages, disadvantages and uses, as illustrated in Table 1.

In summary, Ethereum smart contracts run on the Ethereum virtual machine (EVM), which is a decentralized, distributed computer that executes smart contracts. These smart contracts are written in Solidity, a programming language specifically designed for Ethereum. Smart contracts in Ethereum are executed on a public, permissionless blockchain network, and their state is replicated across all nodes of the network. Corda smart contracts, on the other hand, run on a node-specific JVM. Corda smart contracts are written in Java or Kotlin, and their execution is limited to the nodes on which they are

deployed. Smart contracts in Corda are executed on a private, permissioned network, and their state is shared only with those nodes that are party to the contract.

Table 1. Corda and Ethereum feature overview.

	Corda	**Ethereum**
Network Type	Permissioned	Permissionless
Consensus Mechanism	Pluggable (e.g., BFT, Raft, Notary Services)	Proof of Work (PoW), transitioning to PoS
Language Support	Java, Kotlin	Solidity (Ethereum-specific language)
Privacy	Native support for privacy and confidentiality	Public by default, limited privacy features
Transaction Model	Unspent Transaction Output (UTXO) model	Account-based model (balances and transactions)
Scalability	Focus on scalability and performance	Scalability challenges (e.g., network congestion)
Interoperability	Native support for interoperability	Interoperability through standards and protocols
Smart Contract Features	Flexible contract design and governance	Rich ecosystem of smart contract functionalities
Integration	Suitable for enterprise integration	Suitable for public and open applications
Development Tools	Corda Development Kit (CDK), Corda Node API	Truffle Suite, Remix IDE, Web3.js, libraries
Language Libraries	N/A	Web3j (Java library for Ethereum interaction), ethereumj (Java implementation of Ethereum client), web3j (Java bindings for Ethereum JSON-RPC API)

It is important to note that this table provides a general comparison of the features and capabilities of a Corda and Ethereum snapshot in time. These frameworks and their interconnected networks are in a constant state of flux and are evolving at a rapid pace due to their novelty. As such, the choice between Corda and Ethereum for implementing smart contracts and blockchain functionalities depends on the specific requirements of the use case, the desired network characteristics, and the development preferences of the project.

4. Semantic Web

The semantic web is a vision for the future of the web in which information is organized in a way that is more meaningful and machine-readable. It involves the use of metadata, ontologies, and linked data to enable computers to understand the content and context of web resources. Web 3.0 seeks to provide users with more control over their data and online interactions, as well as enabling seamless and secure transactions between parties.

One of the key challenges facing the development of Web 3.0 is the need for intelligent and efficient handling of the vast amounts of data that will be generated by decentralized applications. This is where the semantic web comes in, as it provides a framework for organizing and processing data in a way that is more efficient and meaningful.

The semantic web is implemented programmatically using a set of standards and technologies that enable machines to understand and process the meaning of data on the web. Some of the key technologies and standards used in the semantic web include the following:

- RDF (Resource Description Framework) is a data model that provides a standard way to describe resources on the web. It defines a set of concepts and relationships for representing metadata about resources.
- OWL (Web Ontology Language) is a formal language for representing and sharing ontologies on the web. An ontology is a set of concepts and relationships that define a domain of knowledge.
- SPARQL (SPARQL Protocol and RDF Query Language) is a query language for RDF data. It allows users to retrieve and manipulate data stored in RDF format.
- Linked Data are a set of best practices for publishing and interlinking data on the web. It enables data to be shared and reused across different applications and domains.

While there are several technologies that enable Java applications to implement the semantic web, including Sesame and OWL API, arguably the most popular and well supported is Jena. An Apache project, Apache Jena is an open-source Java framework for building semantic web and linked data applications. It provides a set of tools for RDF and SPARQL standards, as well as for building and managing ontologies [10].

Apache Jena exposes several APIs and tools for working with RDF data. The core concept of the RDF model is represented by the Model interface. A model is a collection of RDF statements, where each statement consists of a subject, a predicate, and an object. The model can be seen as a graph structure, where nodes represent resources or literals, and edges represent relationships between them. Furthermore, Apache Jena provides parsers and writers for reading and writing RDF data in various formats such as RDF/XML, Turtle, N-Triples, and JSON-LD. These input/output operations allow developers to load RDF data from files or streams into a Jena model or write RDF data from a model to a file or stream.

To create an empty RDF model, the ModelFactory.createDefaultModel() is used. Namespaces are then defined for resources and properties using their URIs. The resources and properties are created using the model.createResource() and model.createProperty() methods, respectively. Statements are added to the model using the model.add() method, specifying the subject, predicate, and object of each statement. Finally, statements in the model are iterated over using model.listStatements() and can then be printed to the console or other outputs.

This high-level breakdown of classes and methods exemplifies the creation of an RDF model using Apache Jena. However, more complex operations can be performed on an RDF model, such as querying, reasoning, and writing to different RDF formats using other APIs provided by the framework.

A SPARQL query string can be added to an RDF that would, for example, select all the persons and their names from the RDF model. A Query object is created from the query string using QueryFactory.create() which is then executed by using QueryExecution-Factory.create() and by providing the query and the RDF model. The QueryExecution is customarily wrapped in a try-with-resources block to ensure proper resource cleanup. The ResultSet returned by qexec.execSelect() can be iterated over in order to extract the relevant data using sol.get(). This is just an example of a basic query; however, the SPARQL query strings can be modified to perform different types of queries, such as filtering, joining, or aggregating the data in the RDF model.

Another important aspect of the semantic web is ontologies, and Apache Jena provides support for working with ontologies and performing reasoning on them as well. It includes an OntModel interface for working with OWL ontologies, as well as an InfModel interface for creating models with reasoning capabilities based on rule engines or more advanced reasoning systems. A basic definition of an OntModel can be seen in Figure 4.

The OntModel represents an ontology model, which is an extension of the Model interface. It provides additional features for working with ontologies, such as reasoning, consistency checking, and ontology-specific operations. OntClasses represent an implementation of an ontology; they can be used to define the structure and hierarchy of concepts. The provided methods work with class-related information such as subclasses, super classes, and class instances. The OntProperty interface represents a property in an ontology and allows implementing classes to work with domain, range, and sub-properties. Finally, the Individual interface represents an individual or an instance of a class in an ontology and it provides methods to work with individual-related information, such as the class to which it belongs, and the properties associated with the individual.

By using these interfaces and their implementations, developers create ontologies, which are formal representations of knowledge that define the concepts, relationships, and constraints within a specific domain. In the case of the semantic web, ontologies play a crucial role in facilitating interoperability and semantic understanding of data.

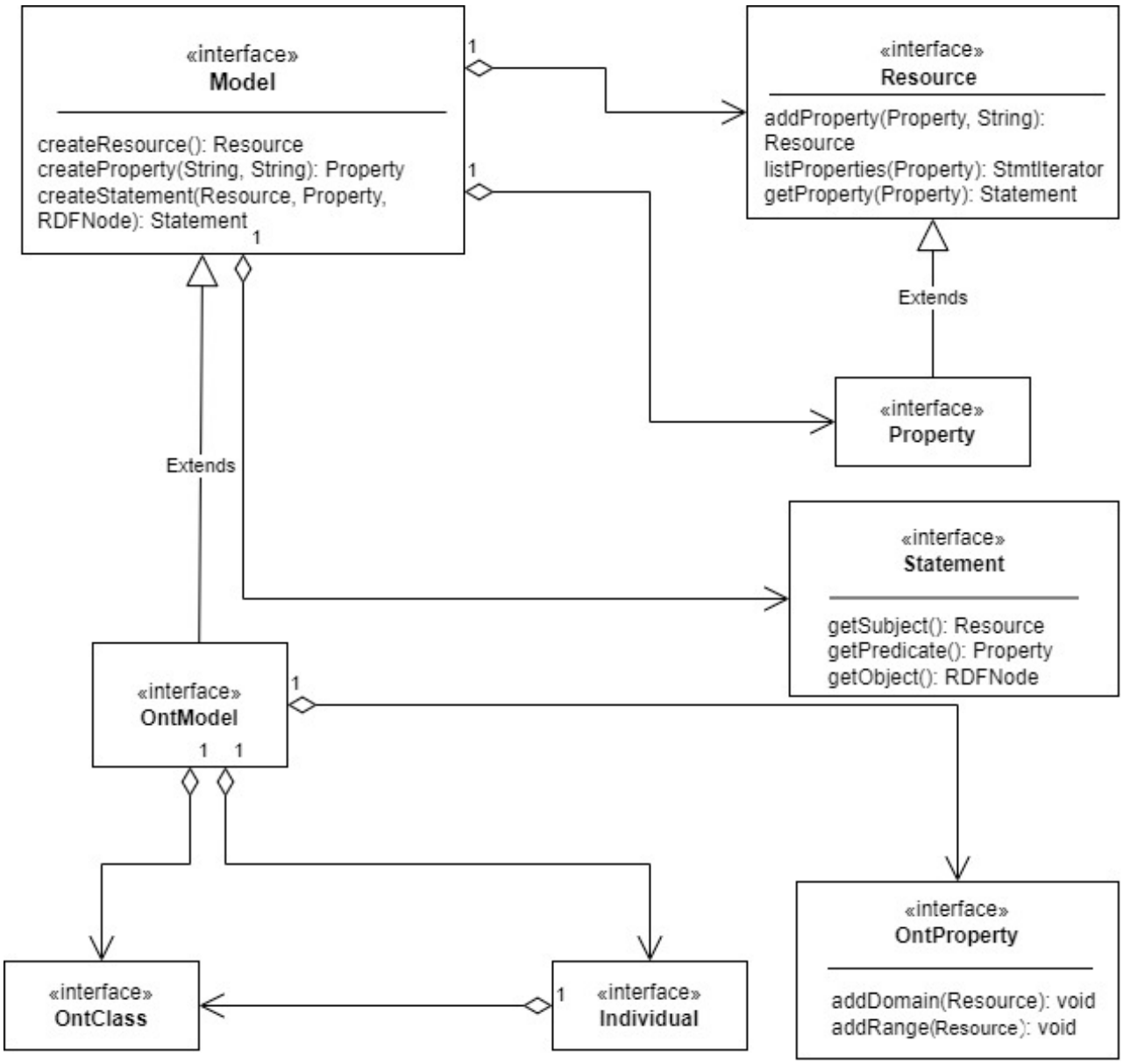

Figure 4. Basic definition of an ontology in Apache Jena.

Ontologies start with a conceptual modeling phase, where the domain experts define the key concepts, properties, and relationships within the domain. This involves capturing the relevant entities, their attributes, and the associations between them. Once the conceptual model is defined, it is translated into a formal representation using ontology languages such as RDF, OWL, or RDFS (RDF Schema). These languages provide a standardized syntax and semantics for expressing ontological concepts.

Classes and ontologies have a tight relationship, as ontologies define classes to represent concepts or entities within the domain. As classes have attributes and relationships with other classes, the ontologies can be hierarchical and/or associative. Furthermore, classes have properties which define their attributes or characteristics. These properties can represent simple data such as name, age, or complex relationships or data such as being part-of, having-color, etc. Properties can also have their own attributes, such as range, the class to which the property applies, and domain, the class that the property belongs to. Together these concepts are known as class and property definitions, in the context of ontology.

Another important feature of ontology is interfaces and reasoning. By explicitly defining the relationships and constraints in the ontology, inference engines can deduce new knowledge or make logical conclusions based on the existing information. This allows for automated reasoning and the ability to answer complex queries and derive new insights.

Finally, ontologies evolve; they are not static. They are updated over time to reflect changes in the domain or to accommodate new requirements. Ontology evolution involves adding new concepts, modifying relationships, or refining the constraints to capture the evolving knowledge accurately.

RDF Models and ontologies are closely related as RDF provides the data model and representation format for expressing information in the semantic web, while ontologies define the vocabulary, concepts, and relationships within a domain. RDF serves as the underlying data model for representing information in the semantic web. It is a graph-based model that represents data as triples, consisting of subject-predicate-object statements. Ontologies leverage the expressive power of RDF to define classes, properties, relationships, and constraints within a domain.

RDF data can be linked to ontologies by using ontology terms (classes and properties) to describe the resources and relationships in the data. Ontologies provide a vocabulary and a set of predefined terms that can be used to annotate and categorize the RDF data. This linking enables semantic interoperability, where data from different sources can be understood and integrated based on their shared ontological concepts.

Linked data are the final piece of the puzzle in the semantic web and they serve as a series of guidelines and principles that help associate and/or link RTFs and ontologies together. Linked data were defined by Breners-Lee in a presentation at the 17th international conference on World Wide Web and initially included four principles [11]. These principles have since been expanded upon and include the following:

- The usage of unique identifiers or URIs: Each resource in the linked data should have a unique identifier in the form of a URI. URIs are used to reference resources and provide a global naming scheme for data.
- Exposing data via HTTP: Linked data should be made accessible via standard HTTP protocols. This enables data retrieval using HTTP requests, such as GET, PUT, PATCH, etc. This model of exposing data is very popular with many web-based APIs in production applications today.
- Providing resource descriptions: Resources should be described using standardized formats such as RDF as it provides a flexible and extensible way of representing structured data.
- Linking related data: Linked data should include links to other related resources. These links are expressed as RDF triples, connecting resources together and forming a graph of interconnected data.
- Using RDF for structured data: Linked data relies on RDF as the primary data model for expressing information. RDF represents data as subject-predicate-object triples, where each triple represents a statement or fact about a resource.

Linked data are, therefore, more of a concept or a set of guidelines, whereas RDFs and ontologies are the technologies, or implementations, used to achieve the goals of linked data. In this sense, linked data are similar to REST, which is not an enforced standard of API communication, but rather a set of guidelines that many software products adhere to strictly.

The purpose of linked data, as envisioned by Breners-Lee, was to help end users navigating the semantic web as they would the regular web [11]; however, it is important to note that the semantic web and the regular web are not entirely similar. One of the key differences between the semantic web and the regular web is that the former is created, primarily, to service AIs and machines. This means that there are key differences in the data representation inside the semantic web and the regular web. For example, RDFs are the building blocks of the semantic web whereas in the regular web, resources are linked together by HTML, which is designed for human consumption.

When implementing semantic web-based applications, Apache Jena does not appear to have a rival in the Java ecosystem. It is the most well-supported framework and the most complete; therefore, it is easier to highlight the features and limitations of Jena, as seen in Table 2, rather than comparing it directly with other frameworks.

Table 2. Jena features and limitations overview.

Features	Description
RDF and OWL Support	Comprehensive support for RDF and OWL ontologies, including parsing, querying, and manipulation.
RDF Model	Provides a flexible data model for representing RDF triples and graphs. Supports various serialization formats.
SPARQL Query	Powerful query language and engine for querying RDF data using the SPARQL query language.
Ontology Support	Supports working with ontologies, including creating, manipulating, and reasoning with OWL ontologies.
Reasoning	Offers support for ontology reasoning, including inferencing and consistency checking.
Rule-Based Reasoning	Provides support for rule-based reasoning with rule engines like the Jena rules engine.
Inferencing	Supports semantic inferencing using rule engines and custom inference rules.
RDF Serialization	Supports various RDF serialization formats, such as RDF/XML, Turtle, N-Triples, and JSON-LD.
Integration	Integrates well with other Java frameworks and libraries.
SPARQL Update	Supports SPARQL Update, allowing updates to RDF data using SPARQL queries.
RDF Storage	Supports storing and retrieving RDF data in various storage systems, including in-memory and databases.

Apache Jena is a powerful Java framework for working with RDF data and ontologies. It provides extensive support for RDF and OWL, including parsing, querying, reasoning, and serialization. Jena could almost be considered the only choice for Java developers; however, it does have some quirks that are worth noting, such as limited support for RDF*, SHACL validation, and some advanced features of OWL.

5. Research Challenges

Table 3 attempts to summarize the current state of the four elements that make up this paper, the current version of the Web (Web 2.0), Blockchain and the Semantic Web (Web 3.0) and the metaverse (enabled by Web 3.0). Each area of research touched upon by this paper was graded from Not Important (NI), to Important (I) and Very Important (VI). The table offers an overview of what we, the authors of this article, believe are the key challenges to implementing metaverse-specific applications from a research perspective. The question of programming language was thoroughly addressed throughout Sections 3 and 4. The rest of these sections will address the other issues of importance such as performance, architecture, quality of data, and security.

Table 3. Research challenges in key areas related to the metaverse.

Area of Research	Web 2.0	Blockchain	Semantic Web	Metaverse
Programming language	NI	I	I	I
Performance	I	VI	I	VI
Architecture	I	I	VI	I
Data quality	I	VI	Vi	I
Security	VI	VI	VI	VI

Perhaps one of the most important aspects of any network of computers, of any kind, is the performance of said network. However, while some comparisons exist between Corda and Ethereum, from a feature perspective or a generic approach to smart contracts and blockchain, performance comparisons are scant. One of the more complete performance evaluations comes from A. A. Monrat et al. [12]. The article compares privately provisioned blockchain platforms to one another. The platforms are deployed in Azure cloud and their performances are compared using various benchmarking tools such as Caliper, Blockbench, or Corda's own enterprise test suite [12]. In terms of performance, Ethereum takes a significant lead in terms of throughput, but does worse in terms of latency [12]. While this article does serve to highlight some of the key performance differences in blockchain processing by several providers, an updated benchmark would be required to come to a more concrete conclusion about performance. Additionally, it is important to note that, in the benchmarks run by A.A. Monrat et al., all blockchains were run on private networks, and this in itself is one of the key differences between Corda and Ethereum.

As Valenta, M and Philipp G. S. highlight, one of the key differences between Ethereum and Corda is the platform itself [13]. Ethereum is a generic blockchain platform, which is publicly available, whereas Corda is a specialized distributed ledger platform aimed specifically at the financial industry [13]. While the article does not specifically compare the performance of different blockchains to one another, rather highlighting their functionalities, similarities, and features [13], it cannot be stressed enough that Corda is run on private networks. While the performance of Ethereum and Corda networks can vary depending on the specific use case and network configuration, there are key differences in performance that can be derived from their approaches to blockchain.

Ethereum is a public blockchain that uses a proof-of-work consensus algorithm, which can result in slower transaction times and higher transaction fees compared to private blockchains. Ethereum's network can also become congested during periods of high usage, which can further impact performance.

Corda, on the other hand, is a permissioned blockchain that uses a unique consensus algorithm called "notary services" to achieve finality of transactions. This allows Corda to achieve high throughput and low latency, making it well suited for enterprise use cases.

In general, Corda may offer better performance and scalability for enterprise use cases, while Ethereum may be more suitable for public and decentralized applications. However, the specific performance of each network will depend on factors such as network configuration, number of nodes, and the complexity of the smart contracts or transactions being executed.

Without a thorough benchmark, it is hard to tell which network offers the better performance. What that means for blockchain developers is that the choice of which network to use does not heavily rely on performance, but rather on types of features offered, implementation of smart contracts, programming language used, currency or other elements that supersede performance concerns.

In terms of architecture, the biggest challenges stem from semantic web applications, perhaps because there are far fewer applications that implement the semantic web by comparison to blockchain. However, a model architecture for the semantic web was proposed by B. Abrahams and Wei Dai [14]. The proposed solution uses annotation software to generate RDF markups describing the content of the website [14]. The extracted RDF content is stored in a database and it forms part of the semantic middleware application [14]. User requests are passed to web agents which formulate a query plan [14]. RDF annotations start with namespace declarations, which act like a prefix to associate individual resources with a schema, doubling not only as a representation of data but also acting as a markup language for the ontology [14]. A reasoner is used to read the ontology model and derive knowledge about the domain [14]. Resources are queried using a multiagent architecture, with Jena as middleware [14]. This architecture model, which predates linked data, helps illustrate a view of the web where the application owner services, primarily, the machine-learning algorithms and less so the end user.

Even after the creation of linked data, however, there are still numerous architectural challenges to implementing the semantic web, as described by J.L. Martinez-Rodriguez et al. Key. Amongst these challenges is the quality of data fed into the semantic web. Not only is trustworthiness an issue, but so are formatting problems and querying issues [15]. Then, there is the issue of the data complexity itself. Even using some of the proposed guidelines for selecting vocabulary or ontology for correctly describing the data is a very long and time-consuming task requiring supervision by a human expert [15]. The purpose of the semantic web is, primarily, to obtain the meaning contained by data in such a way that computers understand it, but web information does not have a heterogenous structure and making data associations with a pre-set vocabulary is not easy [15]. Another issue highlighted in the article is the reasoning or querying of data. The sheer amount of data is huge, according to J.L. Martinez-Rodriguez et al. Linked open data had created more than 50 billion RDFs as of 2015 [15]. Furthermore, some of those statements were incorrect or noisy, for example, when preamble information was omitted or when an instance was declared with disjointed classes [15].

As touched upon briefly in the fourth section of this article, there are also data quality concerns about semantic web applications. Trust is one of the key aspects of semantic web architecture because it helps take decisions over certain tasks [15]. A measure of trust degree is required, therefore, when taking data into account, such as provenance, reputation, or information quality [15]. But these features of trustworthiness must first be specified and secondly their definition must be ubiquitous. Herein lies part of the problem as highlighted by J.L. Martinez-Rodriguez et al. since quality is subjective and it depends on human-driven criteria to judge whether the information is suitable or not [15]. All of these annotations must also be added to the data, and adding them manually would be impossible, so other machine learning tools are leveraged in order to simplify the process [15]. However, it is important to note that these annotations vary from implementation to implementation. While an attempt was made to define eighteen quality dimensions and metrics, only four were commonly used in the more than 1 billion triples analyzed by the authors [15].

The article by J.L. Martinez-Rodriguez et al. does a very good job in highlighting many of the technical challenges of working with large sets of unregulated data, but there is one important aspect that is not addressed: security and privacy in the semantic web. Security issues have become increasingly prevalent in software development, with a cumulative total of nearly 230,000 vulnerabilities known between 1988 and 2022, according to IBM's IBM Security X-Force Threat Intelligence Index 2023 report [16]. Phishing is still the most prevalent technique in gaining unauthorized access to systems, and data theft is one of the most likely outcomes of a breach, accounting for 17% of total cybersecurity incidents in Europe according to the same report [16]. This represents a particular problem for semantic web applications as the interfacing capabilities of ontologies and RDFs can lead to privacy-leakage from one application to another, therefore exposing far more data in the case of a breach, than the regular web.

Furthermore, there is the issue of compliance with regulations, most notably GDPR (General Data Protection Regulation), or, in the case of countries outside of Europe, HIPAA (Health Insurance Portability and Accountability Act). Finally, it is important to highlight that semantic web architecture must account for data privacy through more than just regular web means, as it has to minimize the amount of personal data exposed on several levels as well as ensure encryption of various layers and between various connections so that one successful attack does not expose the entire system and all of its dependents.

6. Conclusions

Java continues to offer a robust tool set and plentiful resources as well as frameworks for working within a Web 3.0 and metaverse context. Furthermore, the expansion of Java beyond a simple programming language into an entire ecosystem of JVM-based languages, some of which offer better features than plain Java, are more robust or more secure, giving developers and tech companies room to pick and choose their technology stack.

It is also important to note that in the context of an entirely new approach to web development and web architecture, JVM-based languages have a considerable edge in terms of users and overall experience when compared to other programming languages that have been released more recently. The popularity of Java and its continued use has given rise to a large community of JVM users which is a key resource in software development, albeit one that is very often ignored. Leveraging the tools, community, and experience accumulated in using JVM-based programming languages can pave the path to a more feature-rich and profitable Web 3.0 and usher in a new era of human–computer interaction in the coming years.

Author Contributions: Writing—original draft, V.B.; supervision, L.-C.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the InoHubDoc project through European Social Fund financing agreement no. POCU/993/6/13/153437. The APC was funded by the Technical University of Cluj-Napoca via contract number 25426 signed on 07.08.2023.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: This paper was financially supported by the Project "Network of excellence in applied research and innovation for doctoral and postdoctoral programs"/InoHubDoc, project co-funded by the European Social Fund financing agreement no. POCU/993/6/13/153437.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. HackerRank. HackerRank Developer Skills Report—Skills. 2020. Available online: https://info.hackerrank.com/rs/487-WAY-049/images/HackerRank-2020-Developer-Skills-Report.pdf (accessed on 27 May 2023).
2. HackerRank. HackerRank Developer Skills Report—Tools. 2020. Available online: https://info.hackerrank.com/rs/487-WAY-049/images/HackerRank-2020-Developer-Skills-Report.pdf (accessed on 27 May 2023).
3. TIOBE Index. TIOBE Index—May 2023. Available online: https://www.tiobe.com/tiobe-index/ (accessed on 27 May 2023).
4. Lassila, O.; Hendler, J. Embracing Web 3.0. *IEEE Internet Comput.* **2007**, *11*, 90–93. [CrossRef]
5. Hendler, J. Web 3.0 Emerging. *Computer* **2009**, *42*, 111–113. [CrossRef]
6. Markoff, J. *Entrepreneurs See a Web Guided by Commonsense*; The New York Times, Business: New York, NY, USA, 2006.
7. GraalVM. Get Started with GraalVM. Available online: https://www.graalvm.org/latest/docs/getting-started/ (accessed on 30 April 2023).
8. Bitcoin. BIPS Wiki Implementation. Available online: https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki#hashes (accessed on 1 June 2023).
9. Suvitha, M.; Subha, R. A Survey on Smart Contract Platforms and Features. In Proceedings of the 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, 19–20 March 2021; pp. 1536–1539. [CrossRef]
10. Apache Jena. Getting Started with Apache Jena. Available online: https://jena.apache.org/getting_started/index.html (accessed on 1 April 2023).
11. Bizer, C.; Heath, T.; Idehen, K.; Berners-Lee, T. Linked data on the web (LDOW2008). In Proceedings of the 2008 17th International Conference on World Wide Web, Beijing, China, 21–25 April 2008; pp. 1265–1266.
12. Monrat, A.A.; Schelén, O.; Andersson, K. Performance Evaluation of Permissioned Blockchain Platforms. In Proceedings of the 2020 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE), Gold Coast, Australia, 16–18 December 2020; pp. 1–8. [CrossRef]
13. Valenta, M.; Philipp, G.S. *Comparison of Ethereum, Hyperledger Fabric and Corda*; Frankfurt School Blockchain Center: Hessen, Germany, 2017; Volume 8, pp. 1–8.
14. Abrahams, B.; Dai, W. Architecture for automated annotation and ontology based querying of semantic Web resources. In Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI'05), Compiegne, France, 19–22 September 2005; pp. 413–417. [CrossRef]

15. Martinez-Rodriguez, J.L.; Lopez-Arevalo, I.; Rios-Alvarado, A.B. A Classification of Challenges in the Semantic Web Based on the General Architecture. In Proceedings of the 2015 26th International Workshop on Database and Expert Systems Applications (DEXA), Valencia, Spain, 1–4 September 2015; pp. 197–201. [CrossRef]
16. IBM Corporation. *IBM Security X-Force Threat Intelligence Index 2023*; IBM: Armonk, NY, USA, 2013.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.