What's in a Pod?

A Knowledge Graph Interpretation For The Solid Ecosystem

Ruben Dedecker[1], Wout Slabbinck[1], Jesse Wright[2], Patrick Hochstenbach[1], Pieter Colpaert[1], Ruben Verborgh[1]

[1]*IDLab, Department of Electronics and Information Systems, Ghent University – imec*
[2]*Australian National University, College of Engineering & Computer Science*

Abstract

The Solid vision aims to make data independent of applications through technical specifications, which detail how to publish and consume permissioned data across multiple autonomous locations called "pods". The current document-centric interpretation of Solid, wherein a pod is a single hierarchy of Linked Data documents, cannot fully realize this independence. Applications are left to define their own APIs within the Solid Protocol, leading to fundamental interoperability problems and the need for associated workarounds. The long-term vision for Solid is confounded with the concrete HTTP interface to pods today, leading to a narrower solution space to address core issues. We examine the mismatch between the vision and its prevalent document-centric interpretation, and propose a reconciliatory graph-centric interpretation wherein a pod is a hybrid, contextualized knowledge graph. In this article, we contrast the existing and proposed interpretations in terms of how they support the Solid vision. We argue that the graph-centric interpretation can improve pod access through different Web APIs that act as views into the knowledge graph. We show how the latter interpretation provides improved opportunities for storage, publication, and querying of decentralized data in more flexible and sustainable ways. These insights are crucial to reduce the dependency of Solid apps on implicit API semantics and local assumptions about the shape and organization of data and the resulting performance. The suggested broader interpretation can guide Solid through its evolution into a heterogeneous yet interoperable ecosystem that better supports the diverging read/write data access patterns of different use cases.

1. The Solid Vision Of Data Interoperability And Control

Data privacy and control have lost ground on today's Web. User-generated data is stored in centralized data silos, in which people have neither the control nor the knowledge to manage how their data is being used [1]. As a response, the *Solid* project [2,3] was created with the aim of revitalizing the Web. Where the current system of centralized data silos create an ecosystem of limited integration, availability and innovation, Solid brings a course correction for the Web. Based on the *separation of data and applications*, the vision defines an ecosystem that facilitates the integration of data in different applications, while keeping people in direct control of their data.

To this end, Solid introduces the concept of a *pod* as an online data space for an individual to control and manage their data on the Web. Together, these pods form a decentralized Solid ecosystem, from which applications can directly integrate data from people's Solid pods, after receiving their permission. This contrasts with current Web applications, where this data first had to be collected in a centralized

location, after which the platform-specific API had to be integrated, where all the while user control is at the mercy of the platforms maintaining the data.

In order to separate data from applications, the semantic contents of the data must be captured such that they can be accurately interpreted and reused in different contexts. Semantics allow applications to interpret data without requiring specific knowledge encoded in the API over which the data is retrieved. A key driver is the use of the Resource Description Framework (RDF) [4], which provides an infrastructure for capturing this semantic information. This again contrasts with current Web APIs, where data is served in formats that require additional semantics to be described in the API's documentation. By shifting the focus from the API to the data, the Solid ecosystem aims to transition the Web from an ecosystem of API integration towards an ecosystem of data integration [5].

Unfortunately, we observe a significant gap between theory and practice: current Solid apps do not succeed in API-independent reuse of data across use cases. Rather than only relying on the data and its semantics, apps resort to implicit knowledge about how this data is structured across documents in a pod's Web API. Furthermore, different use cases impose conflicting requirements on that structure in order to satisfy their own constraints. As such, app developers struggle to make sustainable decisions on how to structure data for reuse, since their individual choices impact the interoperability of the entire ecosystem.

In this article, we identify the root cause of this interoperability problem as the mismatch between Solid's current single hierarchical API and the modeling requirements of real-world use cases. We describe the properties of this document-centric interpretation of a pod, and introduce a graph-centric interpretation that can bridge differences between use cases. We compare both interpretations, explain the consequences for concrete Solid implementations, and argue why we consider the graph-centric interpretation a more sustainable candidate to realize the Solid vision of data and application independence under control of the user.

2. Motivating Use Cases

In order to pinpoint the concrete differences between interpretations, we introduce two small use cases that we will carry throughout the paper.

2.1. Contacts Use Case

The *contacts* use case is a rather trivial example, but we introduce it to evidence that even very simple use cases can expose issues in the interpretation of a pod. The implication is thus that, if a certain interpretation cannot adequately handle the contacts use case, then it is likely to break more complex cases as well. The case is as follows:

- The data consists of a set of **contacts**, each of which have associated attributes such as name, address, email, phone number, date of birth.
- An **address book** app provides read and write access to each attribute of a contact, and allows creating new contacts.

- A **birthday app** shows daily reminders of contacts with upcoming birthdays, and allows editing birthdays and adding new ones.

2.2. Medical Use Case

The *medical* use case is conceptually simple, but it involves highly sensitive data. Its purpose is to demonstrate that issues identified in the *contacts* use case easily generalize to more core complex data and real-world problems. In this use case, the user is a patient storing the following data:

- A set of medical records reflecting **blood test results**, with records containing various vitamin levels as well as HIV status results.
- A set of **heart rate and blood pressure measurements**, captured by the user's wearable device.

3. Preliminary Definitions

Before describing the interpretations of a Solid pod, we start with a couple of definitions that we will use as building blocks throughout the article.

- We consider a ***protocol*** to be a generic set of rules for data transmission between systems.
  - The ***HyperText Transfer Protocol (HTTP)*** [6] structures the exchange of data between a server and a client as *resources* identified by a URI.
  - The ***Linked Data Platform (LDP)*** [7] constrains HTTP with interaction rules for recursive containers of RDF and non-RDF documents.
  - The ***Solid Protocol*** [8] constrains HTTP with *authentication* and *authorization*, and with interaction rules for recursive containers of RDF and non-RDF documents (inspired by LDP).
- A ***Web API*** is a specific structuring of resources on top of HTTP (or a specialization thereof, such as the Solid Protocol).
- ***Authentication*** means identifying the agent issuing a request to a Web API.
  - The ***WebID*** [9] is an HTTP URL that identifies an agent. When dereferenced, it leads to a ***profile document*** describing various agent details.
  - ***Solid-OIDC*** [9] establishes some authoritative identification of an agent by a specific WebID.
- ***Authorization*** means determining to what extent a server can respond to a certain Web API request from a specific agent.
  - ***Web Access Control (WAC)*** [10] is an Access Control List (ACL) mechanism that allows assigning inheritable permissions to documents and containers through so-called *ACL documents*.
  - ***Access Control Policies (ACP)*** [11] is a policy-based mechanism that allows assigning inheritable permissions to documents and containers through so-called *Access Control Resources (ACR)*.

Let us exemplify some of these definitions through our use cases:

- An HTTP interface at `https://sasha.pod/` implements the Solid Protocol when its containers and documents follow the interaction rules, and when it correctly authenticates users using their WebID and applies authorization to each resource.

- A Web API within `https://sasha.pod/` structures documents in containers.
  - Contacts are stored in `https://sasha.pod/people/` as individual documents:
    - `https://sasha.pod/people/sasha.ttl`
    - `https://sasha.pod/people/lucian.ttl`
  - Medical records are stored in `https://sasha.pod/private/acme-hospital/` by date, such as:
    - `https://sasha.pod/private/acme-hospital/2022/10/15/test-results.ttl`
- The WebID `https://sasha.pod/people/sasha#me` identifies a person named Sasha.
- The agent identified by `https://sasha.pod/people/sasha#me` is allowed to access all documents on `https://sasha.pod/`.

4. Document-Centric Interpretation Of A Pod

This section discusses the currently prevalent interpretation of a pod, which is *document*-centric.

4.1. Definition

As described in Section 3, the Solid Protocol models interactions with data as recursive containers with RDF and non-RDF documents. When a server offers this protocol, clients of this server can iteratively define a Web API by creating containers and documents.

The document-centric interpretation assumes that the structure and contents of the Web API, which a pod exposes through the Solid Protocol, *is* that pod in its entirety. Within this interpretation, the complete state of the pod is thus equivalent to the single Web API through which it is available; the *source of truth* is solely that specific Web API. That brings us to the following definition:

*In the document-centric interpretation, each **Solid pod** is a single specific hierarchical structure of containers and documents exposed through the Solid Protocol, where data and access control rules are stored in specific RDF and non-RDF documents within that hierarchy.*

4.2. Example

For example, a Solid pod would be fully defined by the following hierarchy and the contents of its documents:

- container `https://sasha.pod/`
  - RDF document `.acl` *(for access control)*
  - container `people/`
    - RDF document `.acl` *(for access control)*
    - RDF document `amal.ttl`
    - RDF document `lucian.ttl`
    - ...
  - container `private/`
    - RDF document `.acl` *(for access control)*

    - container `medical-records/`
      - non-RDF document `2022-09-15.pdf`
      - non-RDF document `2022-10-15.pdf`
      - …
  - …

In the above example, the access control document `https://sasha.pod/private/.acl` could contain WAC rules such that only the agent `https://sasha.pod/people/sasha#me` is allowed to access the container `https://sasha.pod/private/` and below.

4.3. Practical Usage

The above document-centric definition of a pod leaves several degrees of freedom as to how the pod is structured and how the resulting structure is interpreted. We now describe how today's Solid apps handle those degrees of freedom in practice.

4.3.1. Structure Of The Main Web API

Importantly, the current Solid technical reports [12] do not impose any specific container structure onto a pod beyond the presence of a root container `/`. Therefore, *the* Solid Web API does not exist; only the Solid Protocol to create an API for each pod. Some past suggestions are nonetheless present in certain server implementations as defaults (such as `/profile/`, `/inbox/`, and `/settings/` containers). Since these are not standardized across the ecosystem, their presence is not server-enforced, and as such cannot be relied upon.

As a consequence, client-side applications have to invent their own (sub-)API within the pod's URL space available through the Solid Protocol, by defining a certain container structure and data distribution across documents within this structure. We observe two kinds of behavior:

- Some apps use **hard-coded paths** to certain containers (e.g., `/contacts/`) or documents (e.g., `/profile/card`).
- Some apps use **link traversal**, which means they find the URLs of documents and containers by following links from the user's WebID profile document and/or via another index [13].

We also observe *hybrid behavior*, for instance where an initial path is obtained via traversal (e.g., `/private/medical/`), but deeper relative paths are hardcoded (e.g., `/private/medical/2022/10/`). In particular, link traversal is *bootstrapped* via hardcoded paths: if no link exists to the certain kind of data, then a specific document is created at a hardcoded path and then linked from a profile or index for future usage.

4.3.2. Aspects Of RDF Document Boundaries

From the way current apps organize data in RDF documents, we can observe the meaning they ascribe to such a document. Noting that Solid typically uses RDF 1.0 documents (so only triples, and not

quads as in RDF 1.1), the occurrence of an RDF triple in a document seems to carry various degree of meaning with regard to the following aspects:

- *(implicit)* **Context**: the occurrence of certain triples within the same document often implies that they are somehow interrelated, and that these triples somehow relate to the document. This topical relation is sometimes visible within certain triples, whose subject (e.g., `https://sasha.pod/people/sasha#me`) defines a URL fragment (e.g., `#me`) on the document identifier (e.g., `https://sasha.pod/people/sasha`).
- *(explicit)* **Policy**: both the WAC [10] and the ACP [11] specifications assign authorizations on a *document* level of granularity. Either the document can be accessed by a given agent in its entirety or not at all, thus resulting in all triples within a document sharing the same authorization rules.
- *(implicit)* **Provenance**: the document somehow captures the notion that its triples originate from a specific source or event, of which the document was a result.
- *(implicit)* **Trust**: the document defines a single boundary of trust for all of its triples. For example, a user's profile document is typically fully trusted by the user (because they are usually the only party with write access to it), whereas inbox documents created by third parties might contain triples that are not trusted.
- *(implicit)* **Performance**: the document groups together a number of triples because it improves the performance of certain use cases. For instance, triples that are often used together might be in the same document, and triples that are less needed might be in extension documents, in order to optimize the number of HTTP requests and the used bandwidth.

We remark that of these 5 aspects, only the *policy* on the document is modelled explicitly. The *context* is implicitly assumed because triples occurring in the same place were usually created by the same or related write operations, and because those triples are necessarily read together by apps. The *provenance* and *trust* are similarly derived from implicit assumptions about a shared origin, and the knowledge of specific policies and thus agents that could have written to the document. Notably, the fact that an identifier (e.g., `https://sasha.pod/people/`) is contained within a certain pod root container (e.g., `https://sasha.pod/`) does encode some explicit provenance about the document and its triples (e.g., "they were found in Sasha's pod"), but not necessarily about its creator or level of trust (e.g., multiple actors might have write access to the document). Finally, the *performance* is typically based on educated guesses, but seldom the result of actual performance measurements.

4.3.3. Alternative Web APIs To The Pod

Many applications encounter practical limitations when the data they require happens to be structured across multiple documents in the main API. In an attempt to address such cases, alternative Web APIs were proposed.

One proposal [2] suggests exposing a server-side SPARQL endpoint [14] over all RDF data in a pod, enabling fully server-side SPARQL query [15] processing. Another proposal [16] suggests to expose this data through a read-only Quad Pattern Fragments (QPF) [17] interface, to speed up the client-side processing of SPARQL queries over the entire pod. Whereas these alternative APIs can alleviate part of the

context and *performance* aspects of the main API, they come with challenges to implement *policy* and to adequately model *provenance* and *trust* in their responses.

Crucially, such alternative APIs are always derived from the main API, which is equivalent to the pod in the document-centric interpretation. The derived APIs thereby unavoidably inherit some of the explicit and implicit modeling aspects from the document-based main API. Concretely, the direct derivation from the main API manifests itself in the choice of the data model for the SPARQL and QPF interfaces. The RDF 1.1 quads they expose are constructed by loading the triples from each document, adding as a graph component the URL of that document. The following example quad reflects this:

- subject: `https://sasha.pod/people/sasha#me`
- predicate: `https://example.org/ontology#birthDate`
- object: `"1984-04-03"`
- graph: `https://sasha.pod/private/medical/2022/10/15.ttl`

Its components signify that there exists a triple with that specific subject, predicate, and object in the document with URL `https://sasha.pod/private/medical/2022/10/15.ttl`. In other words, the document-centric interpretation of a pod considers the birthdate statements' occurrence in this specific document on the pod to be an integral part of the statement itself.

4.4. Consequences Of The Single Hierarchy

In this section, we will examine and critique the consequences of the document-centric interpretation of a pod. Specifically, we study the limitations of the single hierarchy it causes (Subsection 4.1), and the effects of the implicit semantics in its structure (Subsubsection 4.3.1) and documents (Subsubsection 4.3.2). While some consequences could in theory be mitigated by alternative APIs (Subsubsection 4.3.3), the effectiveness thereof is hindered by the necessity of those alternatives to derive from the main API structure.

4.4.1. Single-App Modeling Mismatches

Each app needs to have a single consistent hierarchy to serialize its data, which does not reflect the complex nature of real-world organization. For instance, the address book app could organize people in categories such as `/contacts/work/` and `/contacts/sports/`, which leads to duplication when a person's colleague is also a member of their badminton team. A similar situation occurs when we need to decide whether to group health measurements by date (`/medical/2022/10/15.ttl`) or by topical evolution over time (`/medical/vitamine-d-levels.ttl`)

Hierarchical organizations are thus either constrained by their necessity to commit to a single representation of the real world, or in need of mechanisms to cope with the effects of data duplication or virtualization. An alternative API circumvents some of these limitations when it comes to reading, although the provenance and trust of the resulting responses are even less explicitly defined than in the main API. Writing is still fully coupled to the destination documents in the main API, since the quad components of each triple need to contain a specific document URL.

4.4.2. Cross-App Modeling Mismatches

The document-centric view of the Solid Protocol does not inherently provide interoperability, because apps are still responsible for determining the specific API that defines the document and container structure they will access. To make matters worse, different apps and use cases can have competing interests that lead them to prefer one API structure over another.

For example, interoperability requires the address book and birthday apps to use the same data, and hence to store it in the same place, such as the `/people/` container. However, their preferences regarding the organization of that container vary. The address book app, which lets the user edit contacts one by one, has a *context* and *performance* incentive to place all contact attributes in a single RDF document per contact, leading to an organization such as:

- `/people/work/dani.ttl`
- `/people/work/kiran.ttl`
- `/people/personal/kai.ttl`
- `/people/personal/luka.ttl`
- …

In contrast, the birthday app aims to quickly determine which celebrations are coming up, probably preferring a structure more like:

- `/people/work/birthdays.ttl`
- `/people/personal/birthdays.ttl`

Or possibly even:

- `/people/birthdays/january.ttl`
- `/people/birthdays/february.ttl`
- …

Similarly, whether heart rate and blood pressure measurements are organized by date or by evolution over time, depends on the specifics of a current use case.

4.4.3. Policy Modeling Mismatches

Context- and performance-based grouping are trade-offs that can be overcome with compromises, such as accepting that certain use cases will be slower than others. Unfortunately, the imposed grouping of multiple different aspects in the same document can also lead to more sensitive and insurmountable conflicts for the *policy*, *provenance*, and *trust* aspects.

Since the coupling of policies to document organization provides the only mechanism of control in the document-centric view, some use cases with conflicting requirements cannot effectively be realized today. For example, assume that the address book app indeed organizes contacts as one person per document:

- `/people/dani.ttl`

- `/people/kiran.ttl`
- …

In that case, the birthday app would be able to read (and needing to parse) people's personal details such as addresses and phone numbers, whereas the expectation is that it should only access names and birthdays.

Insurmountable conflicts become even more apparent with the medical use case. The results of a given blood test might be stored in a single document, and thus have a single policy boundary associated with it. If that test result contains both vitamin levels and an HIV status, then the document-based access control prevents users from only giving access to their vitamin levels.

The fact that conflicting requirements between aspects would necessitate the complexity of copies, flags a strong limitation of any single hierarchical API. One partial solution is to split these pieces of data into different documents, but this results in suboptimal boundaries for purposes of context, provenance, and trust. Users thus find themselves torn between giving apps too much access, or having to deal with overly granular control—in the extreme case causing situations that necessitate managing micro-documents with only one or a handful of triples. Whilst the degrees of freedom in the Solid Protocol allows for any such structures, the resulting API would be highly impractical for humans and machines alike. Another solution involves creating and maintaining a copy of the document with a subset of the data, which—in addition to the overhead of managing such copies—would also generate a different associated context, provenance, and trust—especially if writing to such derived documents is needed. Furthermore, all these aspects would necessarily be reflected in any derived APIs, which are tied to the main API's document-based structure and boundaries.

We conclude that document-centric pods inherently contain a *large amount of implicit semantics* in their API structure, hindering the realization of the data and application independence that is paramount to the Solid vision. Some semantics that are supposed to be entangled with the data are in practice assumed by the API, the construction of which happens in an uncoordinated way over time. The resulting spontaneous contracts are not made explicit by a single app, nor shared across multiple apps, meaning that interoperating apps have to be coded against each other rather than against the data, creating undesired inter-application coupling.

5. Graph-Centric Interpretation Of A Pod

In this section, we give the concept of a Solid pod a new interpretation, which is *graph-centric*.

5.1. Design Considerations

We start from the limitations of the document-centric pod interpretation, which essentially assumes the Web API exposed by a pod to be equivalent to the pod itself. On the one hand, we acknowledge the *universality* and *simplicity* of document-based APIs, and in particular of the Solid Protocol, which offers the building blocks to construct such APIs with the appropriate authentication and authorization. On the other hand, we showed concrete evidence in Subsection 4.4 that no single such hierarchy is able to

reconcile the conflicting constraints of different use cases, especially given that core aspects such as policies and provenance can only be applied at a document-level granularity.

While we recognize the importance of document-based Web APIs, we also observe that the simultaneous support for multiple use cases clearly requires multiple perspectives into the same data, each satisfying the constraints of particular cases. Even though the creation of multiple views has been attempted for Solid pods with SPARQL and QPF interfaces, their direct derivation from the main API still leads them to inherit that API's mismatched constraints on the modeling of policies, provenance, and trust.

In other words, aiming to derive richer views from the main pod API is akin to using a real-world object's *two-dimensional projection* to derive alternate two-dimensional projections of that same object. Because any two-dimensional projection is inherently designed to discard information of the original, the creation of complementary alternative projections actually requires the object's underlying *three-dimensional reality*. The two-dimensional projection was only ever meant as a helpful approximation of the three-dimensional object.

Translating this dimensionality metaphor to the world of pods, we conclude that today's single hierarchical API to a pod serves as a *proxy* for the underlying knowledge graph formed by the union of the pod's interlinked RDF documents—except that this union cannot adequately be reproduced because significant parts of its semantics are being discarded during its exposure in a specific API. Solid applications looking to leverage the potential of this larger Linked Data knowledge graph, will thus always be hindered by the limitations of one arbitrarily formed document API acting as its sole access gateway.

5.2. Definition

Given the above observations within the document-centric interpretation, we create a new interpretation that shifts the function of a pod's main API from being the pod itself to acting as *one of many* possible interfaces to an underlying knowledge graph, which *is* the pod. The *source of truth* is a knowledge graph consisting of documents as well as RDF statements, from which multiple Web APIs can be derived. Hence, no particular API is more prominent than any other. We define this as follows:

*In the graph-centric interpretation, each **Solid pod** is a hybrid, contextualized knowledge graph, wherein "hybrid" indicates first-class support for both documents and RDF statements, and "contextualized" the ability to associate each of its individual documents and statements with metadata such as policies, provenance, and trust.*

5.3. Example

For example, the pod `https://sasha.pod/` could be a hybrid knowledge graph consisting of:

- RDF triples expressing contact details of `https://sasha.pod/people/amal#me`
- RDF triples expressing contact details of `https://sasha.pod/people/lucian#me`
- a PDF document containing medical images dated 2022-09-15
- RDF triples representing a blood test result dated 2022-10-15
- …

Examples of associated metadata within this pod are:

- The RDF triples about Amal form a shared context with specific trust and provenance.
- A policy states that professional contact names can be publicly readable.
- A policy states that contacts' phone numbers are only visible to Sasha.
- The provenance of Lucian's phone number is a specific email.
- We trust the test result of 2022-10-15 is unmodified and accurate, because it is certified by a medical professional.

Below is one possible Web API on top of this pod that conforms to the Solid Protocol:

- container `https://sasha.pod/`
  - container `contacts/work/`
    - RDF document `.acl` *(for access control)*
    - RDF document `amal.ttl`
    - RDF document `lucian.ttl`
    - …
  - …

The same pod could simultaneously offer other Web APIs through the Solid Protocol:

- container `https://sasha.pod/`
  - container `people/birthdays/`
    - RDF document `.acr` *(for access control)*
    - RDF document `january.ttl`
    - RDF document `february.ttl`
    - …
  - …

Furthermore, the pod could have SPARQL and QPF interfaces on top of its knowledge graph, in addition to other Web APIs.

5.4. Aspects Of RDF Data In The Graph

Importantly, the "hybrid" and "contexualized" qualifiers are equally important as the "knowledge graph" term in the definition. This means that off-the-shelf triplestores or quadstores do not qualify as implementations of a pod. Whereas they could possibly be used as physical storage for such a pod (see Subsection 6.1), any implementation requires native support for documents and metadata.

The ability to generate multiple Web APIs that act as views on the data is required functionality for the pod. Below, we discuss how the aspects from Subsubsection 4.3.2 are modeled in the pod and the resulting APIs.

- **Context**: Each triple or document in the pod can be associated with multiple contexts. For instance, users could assign triples to specific resources in Web APIs ("this triple is exposed in the documents `/records/2022-15-10.ttl` and `/records/2022-15-10-summary.ttl`"), or smaller ad-hoc groupings

could be created ("these triples have a topical relationship"). An API can reflect this context through its resource structure, or by including explicit metadata in its response.

- **Policy**: Policies can be assigned to resources in one of the Web APIs and/or to selections of triples. In case policies are assigned to triples, their inclusion in a response can be conditionally determined by whether or not the requesting agent has access.
- **Provenance**: Provenance can be associated with individual triples or groups of triples, similar to how generic context is modeled.
- **Trust**: Trust can be expressed either as a function of provenance, or explicitly assigned to (groups of) triples, like context.
- **Performance**: The performance is no longer a function of the pod structure itself, but rather reflected in how well a specific API matches the access patterns of a given use case. In other words, performance concerns need to be addressed by defining relevant APIs on top of the underlying knowledge graph.

The implementation of those aspects differs from the document-centric interpretation in two crucial ways:

1. All 5 aspects are **modeled explicitly**: either as metadata in the knowledge graph that can then be reflected in an API (for *context*, *policy*, *provenance*, and *trust*), or in the definition of a specific API (for *performance*).
2. Each aspect can have a **different granularity**. For instance, a certain group of triples could share the same provenance, but have different policies applied to them.

5.5. View-Based Use Case Modeling

We will now revisit how the graph-centric interpretation addresses the mismatches of the document-centric interpretation (Subsection 4.4).

5.5.1. Single-App Modeling

Applications are no longer restricted to a single hierarchy for modeling information, because the view-based approach allows overlapping resources without requiring copies.

For example, if a person is both a colleague and a badminton player, their details can be available through both the documents `/contacts/work/amal.ttl` and `/contacts/sports/amal.ttl`, wherein the containers `/contacts/work/` and `/contacts/sports/` can have different policies associated with them. From the perspective of a consumer, neither document is more authoritative than the other, as they are generated from the same triples in the underlying knowledge graph. Similarly, medical records can be organized by both date (`/medical/2022/10/15.ttl`) and topical evolution (`/medical/heart-rate-2021-2022.ttl`), allowing them to have different provenance and trust.

5.5.2. Cross-App Modeling

Every app can choose the modeling that best fit its use case, as the same knowledge graph can be exposed through multiple Web APIs. This allows applications to find the APIs that best match their access patterns or, conversely, the pod to define APIs based on the needs of use cases.

For example, the address book app can be built using an API that organizes the attributes of a person in a single document, whereas the birthday app can group all birthdays together. That way, neither application downloads more data than necessary. Similarly, medical data can be grouped depending on the analyses that will be performed. Note that exact matches of access patterns to APIs will not always be possible nor desired; the idea is rather to offer a couple of core APIs that minimize overhead across a number of use cases [18].

5.5.3. Permission Modeling

Finally, the granularity of resources in a given API can be aligned with the granularity of policies, since this granularity does not have to be shared with the other aspects.

For example, the birthday app could have a read/write API to the contacts that does not include access to their email addresses or phone numbers. A derived medical report could only display vitamin levels but leave out highly sensitive information such as HIV status, while leaving intact the provenance and trust of the original context.

6. Comparative Analysis

In this section, we compare the impact of the document- and graph-centric pod interpretations in terms of storage, publication, and query processing. We highlight open research problems that need to be tackled for efficient graph-centric pod support.

6.1. Storage

In the document-centric interpretation, the underlying storage of a pod is typically managed as a document-based system, as evidenced by current Solid server implementations that leverage the file system or document-oriented databases. This preference is explained by the direct match of the interface to the pod; in some ways, the document-centric view on a pod can be seen as a permissioned file system for the Web. Of course, quadstore-based storage can also be applied, in which case the fourth element of each quad is usually necessary to indicate the document to which it belongs (and, hence, the associated metadata such as permissions). The benefit of the latter is that they might enable higher performance for derived APIs, although there is still the requirement of supporting non-RDF documents, such as images or movies.

The graph-centric interpretation imposes stronger requirements on the underlying storage. On the one hand, different metadata might have different granularities, so there is no longer a natural document grouping. On the other hand, different selections need to be created from the raw data, so their specific document organizations provide no universal benefit across APIs. Consequently, document-based storage systems are not a good technological fit. In the long term, native implementations of hybrid, contextualized will be a necessity. In the medium term, they can be emulated on top of quadstores

and RDF-star stores [19]. Whereas quadstores can easily associate metadata with RDF 1.0 triples, RDF-star stores could extend this functionality to RDF 1.1 when Solid pods start using graphs to model data.

6.2. Publication

The generation of API resources in the document-centric interpretation is relatively straightforward. Assuming the underlying storage system is indeed document-based, then each public-facing pod URL corresponds to an internal document identifier. Responding to an HTTP request involves looking up the internal document, verifying whether the document-based policies allow access, possibly performing a format translation, and sending it back to the client. Offering alternative APIs might involve more computations, depending on the underlying storage system.

The generation of API resources in the graph-centric interpretation involves a more complex process, which harbors a significant part of future work in this area. First, we need the notion of a *view definition*, which essentially specifies how public-facing pod URLs correspond to triples or documents from the underlying hybrid knowledge graph. Next, a *view application* process is necessary for materializing such view definitions to generate responses to concrete API requests. Furthermore, a *policy definition* is required to express the allowed operations per triple, and a *policy application* process to apply this policy to a specific request. To realize views, technologies such as RDF mappings [20], specifically from Web APIs to triples [21], can be extended to support the reverse direction. Existing work on policies [22, 23], as well as the WAC [10] and ACP [11] specifications, can be applied on the triple or shape level rather than to documents.

6.3. Querying

In the document-centric interpretation, the complexity of the required query processing is mainly impacted by whether or not the data resides in a single document, i.e., whether the concrete API of the pod happens to coincide with the access patterns of a specific use case. Any mismatch requires a multi-document selection, which might involve traversal-based query processing [24], possibly guided by the pod's known indexes or structure [25]. Furthermore, in order to achieve result set completeness, all relevant data and intermediary links need to be accessible by the client, which might not be the case if there is a mismatch between the granularity of the documents and the policies on the data.

Since the graph-centric interpretation allows for multiple APIs, client can select those APIs that best match the access patterns of their use case [18]. Further work on query engines for heterogeneous interfaces [26, 27] will help identifying the most relevant APIs and, over time, servers could analyze data usage to determine new API structures that might improve performance. This includes hierarchical Solid Protocol interfaces, where the partitioning in resources is largely driven by the server, or more expressive interfaces such as QPF and SPARQL, whose resources are more client-driven. We note that no particular interface is expected to become the best or ultimate choice. Since performance is essentially a function of the match between server-side API partitioning and client-side request patterns, a document-based API might outperform an expressive query-based API if cached documents happen to align with client-side requests. Furthermore, the needed data for any given use case could be located across a small or large number of different pods, and more expressive interfaces do not necessarily out-

perform less expressive interfaces for federation [18]. Hence, achieving high query performance is another reason why the multi-API aspect of the graph-centric interpretation is important.

7. Conclusion

Different interpretations of the Solid vision come with different abilities to support that vision. The dominant document-centric interpretation, specified by the Solid Protocol [8] as an LDP derivative with per-document access control, has the benefit of being conceptually simple and hence developer-friendly. However, we argue in this article that the simplicity of a hierarchical document collection stems from unsustainable shortcuts taken by applications, as they burrow their own ad-hoc APIs with implicit semantics into the container meta-model. These shortcuts hinder the Solid promise of data and application independence in the long term. While the data itself contains explicit semantics that can be interpreted by apps, the organization of that data is such that its storage and retrieval necessitates out-of-band agreements between different applications, resulting in significant app-to-app coupling and subsequent systematic breakage when the ad-hoc APIs inevitably evolve. Furthermore, we show that the abstraction easily breaks when apps pursue slightly conflicting goals.

These issues have largely gone unnoticed so far, because singular apps only encounter smaller problems, and different apps in a young ecosystem can still coordinate behind the scenes. However, an emergence of more Solid apps that need to interoperate has increased the frequency of such conflicts. Stopgaps that are being created to address them unfortunately only prolong unsustainable practices, as many issues are inherent to the implicit semantics within ad-hoc APIs. Proposals typically focus on making API semantics explicit [13, 28], yet they inherit the abstraction's mismatches when it comes to aspects such as trust and provenance. This is why we instead suggest pushing the vision forward by separating the semantics from the API altogether.

We observe that the document-centric interpretation seems to act as a proxy for a more fundamental concept, namely a hybrid, contextualized knowledge graph that attaches metadata to data as well as documents. Our proposed graph-centric interpretation put this concept at the heart of each pod, and considers the Solid Protocol's document interfaces to be specialized views of an underlying richer source of truth. This affords more flexibility—and hence independence—in how multiple apps can interact with the same data. Crucially, it also broadens the solution space for tackling interoperability challenges. We are no longer confined to working around problems inherent to the single hierarchy, but can create solutions at the knowledge graph level and mint associated API views tailored to the needs and constraints of specific applications and use cases. To continue an earlier comparison: instead of being restricted to a two-dimensional environment, we have the full three-dimensional space at our disposition for modeling solutions, which can then be projected onto as many two-dimensional surfaces as needed.

The graph-centric interpretation shifts the understanding from a data graph instantiated from a document organization, to a dynamically instantiated document interface over a data graph, paving the way for fundamentally graph-centric data management within Solid. The separation of pod and APIs allows apps to seamlessly interact via different interfaces to a pod, depending on what they are optimizing for. It also opens the door for authorization with different granularities beyond one specific document

structure, and even for asymmetric read/write interfaces. They can eliminate the need for elaborate mechanisms that explain where apps should write data [28]: graph-centric writing can happen by posting triples or documents directly to the graph, since it is the responsibility of *views* to expose all written information in the correct place within APIs.

The consequences of different interpretations of the Solid vision are not purely conceptual or theoretical: although implementers might have been unaware previously, even an implicit interpretation influences how the storage, publication, and querying functionality of pods is realized into concrete products. For instance, current implementations of alternative Web APIs to pods are affected by the document-centric interpretation, as a current Quad Pattern Fragments interface [16] uses for its quads' graph components the URI where the document API happens to expose them. Ironically, the server thereby unwittingly imposes some of the same constraints and mismatches on an interface that is supposed to mitigate them. This explains, for example, why implementers struggle to find the right permissioning for such derived APIs. The graph-centric interpretation helps by considering document URIs to be ephemeral artifacts of one particular API, which carry no further meaning in other views or apps.

We stress that the current technical reports, notably the Solid Protocol [8], support the document-centric as well as the graph-centric interpretation. This serves as a testament to both the orthogonality of the specifications and the compatibility of the proposed interpretation with the Solid vision. Specifically, the graph-centric interpretation embraces and leverages document-based access as a core building block for APIs. What changes is the role played by document hierarchies: rather than seeing one of them as a pod's absolute source of truth, we find more value in a native ability to construct many views of a much richer knowledge graph, for which Web API resources act as a vessel. Thereby, what is in a pod can be in the eye of the beholder: its knowledge graph facilitates a flexible matching of views to use cases.

8. Acknowledgements

Supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project V023/10).

References

[1] Berners-Lee, T.: Socially aware cloud storage. https://www.w3.org/DesignIssues/CloudStorage.html (2009).
[2] Sambra, A.V., Mansour, E., Hawke, S., Zereba, M., Greco, N., Ghanem, A., Zagidulin, D., Aboulnaga, A., Berners-Lee, T.: Solid: A Platform for Decentralized Social Applications Based on Linked Data. http://emansour.com/research/lusail/solid_protocols.pdf (2016).
[3] Verborgh, R.: Re-decentralizing the Web, for good this time. In: Seneviratne, O. and Hendler, J. (eds.) Linking the World's Information: A Collection of Essays on the Work of Sir Tim Berners-Lee. ACM (2022).
[4] Cyganiak, R., Wood, D., Lanthaler, M. eds: RDF 1.1: Concepts and Abstract Syntax. World Wide Web Consortium, https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (2014).
[5] Verborgh, R.: Reflections of knowledge. https://ruben.verborgh.org/blog/2021/12/23/reflections-of-knowledge/ (2021).
[6] Fielding, R.T., Nottingham, M., Reschke, J. eds: HTTP Semantics. Internet Engineering Task Force, https://www.rfc-editor.org/rfc/rfc9110 (2022).

[7] Speicher, S., Arwe, J., Malhotra, A. eds: Linked Data Platform 1.0. World Wide Web Consortium, https://www.w3.org/TR/ldp/ (2015).
[8] Capadisli, S., Berners-Lee, T., Verborgh, R., Kjernsmo, K. eds: Solid Protocol. W3C Solid Community Group, https://solidproject.org/TR/2021/protocol-20211217 (2021).
[9] Coburn, A., elf Pavlik, Zagidulin, D. eds: Solid-OIDC. W3C Solid Community Group, https://solidproject.org/TR/2022/oidc-20220328 (2022).
[10] Capadisli, S. ed: Web Access Control (WAC). W3C Solid Community Group, https://solid.github.io/web-access-control-spec/ (2022).
[11] Bosquet, M. ed: Access Control Policy (ACP). W3C Solid Community Group, https://solid.github.io/authorization-panel/acp-specification/ (2022).
[12] Solid Technical Reports. https://solidproject.org/TR/
[13] Turdean, T. ed: Type Indexes. W3C Solid Community Group, https://solid.github.io/type-indexes/ (2022).
[14] Feigenbaum, L., Williams, G.T., Clark, K.G., Torres, E.: SPARQL 1.1 Protocol. World Wide Web Consortium, https://www.w3.org/TR/sparql11-protocol/ (2013).
[15] Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. World Wide Web Consortium, http://www.w3.org/TR/sparql11-query/ (2013).
[16] Query (QPF). https://docs.inrupt.com/ess/latest/query/qpf-endpoint/ (2022).
[17] Taelman, R.: Quad Data Fragments. W3C Hydra Community Group, https://linkeddatafragments.org/specification/quad-pattern-fragments/ (2020).
[18] Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. Journal of Web Semantics. 37–38, 184–206 (2016). doi:10.1016/j.websem.2016.03.003
[19] Hartig, O., Champin, P.-A., Kellogg, G., Seaborne, A. eds: HTTP Semantics. W3C RDF-DEV Community Group, https://w3c.github.io/rdf-star/cg-spec/editors_draft.html (2022).
[20] Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In: Bizer, C., Heath, T., Auer, S., and Berners-Lee, T. (eds.) Proceedings of the 7th Workshop on Linked Data on the Web (2014).
[21] Van Assche, D., Haesendonck, G., De Mulder, G., Delva, T., Heyvaert, P., De Meester, B., Dimou, A.: Leveraging Web of Things W3C Recommendations for Knowledge Graphs Generation. In: Brambilla, M., Chbeir, R., Frasincar, F., and Manolescu, I. (eds.) Web Engineering. pp. 337–352. Springer (2021).
[22] Kirrane, S., Villata, S., d'Aquin, M.: Privacy, security and policies: A review of problems and solutions with Semantic Web technologies. Semantic Web. 9, 153–161 (2018).
[23] Debackere, L., Colpaert, P., Taelman, R., Verborgh, R.: A policy-oriented architecture for enforcing consent in Solid. In: WWW '22 Companion, Proceedings. Association for Computing Machinery (ACM) (2022). doi:10.1145/3487553.3524630
[24] Hartig, O.: An Overview on Execution Strategies for Linked Data Queries. Datenbank-Spektrum. 13, 89–99 (2013).
[25] Bogaerts, B., Ketsman, B., Zeboudj, Y., Aamer, H., Taelman, R., Verborgh, R.: Link Traversal with Distributed Subweb Specifications. In: Moschoyiannis, S., Peñaloza, R., Vanthienen, J., Soylu, A., and Roman, D. (eds.) Proceedings of the 5th International Joint Conference on Rules and Reasoning. pp. 62–79. Springer (2021). doi:10.1007/978-3-030-91167-6_5
[26] Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a Modular SPARQL Query Engine for the Web. In: Vrandečić, D., Bontcheva, K., Suárez-Figueroa, M.C., Presutti,

V., Celino, I., Sabou, M., Kaffee, L.-A., and Simperl, E. (eds.) Proceedings of the 17th International Semantic Web Conference. pp. 239–255. Springer (2018). doi:10.1007/978-3-030-00668-6_15
[27] Heling, L., Acosta, M.: Federated SPARQL Query Processing over Heterogeneous Linked Data Fragments. In: Proceedings of the ACM Web Conference 2022. pp. 1047–1057. Association for Computing Machinery (2022). doi:10.1145/3485447.3511947
[28] Bingham, J., Prud'hommeaux, E. eds: Shape Trees Specification. W3C Solid Community Group, https://shapetrees.org/TR/specification/ (2021).