Semantic Web -1 (2024) 1–15
DOI 10.3233/SW-243726
IOS Press
CORRECTED PROOF

The Community Solid Server: Supporting research & development in an evolving ecosystem

Joachim Van Herwegen [*] and Ruben Verborgh
Ghent University – imec – IDLab, Department of Electronics and Information Systems, Belgium
E-mails: joachim.vanherwegen@ugent.be, ruben.verborgh@ugent.be

Editor: Katje Hose, Databases and Artificial Intelligence research unit at TU Wie, Austria
Solicited reviews: Christoph Braun, Karlsruhe Institute of Technology (KIT), Germany; Bart Buelens, Flemish Institute for Technological Research, Belgium; Three anonymous reviewers

Abstract. The Solid project aims to empower people with control over their own data through the separation of data, identity, and applications. The goal is an environment with clear interoperability between all servers and clients that adhere to the specification. Solid is a standards-driven way to extend the Linked Data vision from public to private data, and everything in between. Multiple implementations of the Solid Protocol exist, but due to the evolving nature of the ecosystem, there is a strong need for an implementation that enables qualitative and quantitative research into new features and allows developers to quickly set up varying development environments. To meet these demands, we created the Community Solid Server, a modular server that can be configured to suit the needs of researchers and developers. In this article, we provide an overview of the server architecture and how it is positioned within the Solid ecosystem. The server supports many orthogonal feature combinations on axes such as authorization, authentication, and data storage. The Community Solid Server comes with several predefined configurations that allow researchers and developers to quickly set up servers with different content and backends, and can easily be modified to change many of its features. The server will help evolve the specification, and support further research into Solid and its possibilities.

Keywords: Solid, RDF, linked data, semantic web

1. Introduction

Data plays an important role in multiple aspects of our lives: from the data our government manages about us, to the items we store and share on the internet, to even everything about our online behavior that is being tracked online. Companies use that data to predict our future behavior and gain a competitive edge in their respective industries. This poses a considerable challenge for newcomers trying to establish themselves in a particular sector, as established companies have already amassed extensive data, creating a substantial barrier to outperforming them.

As companies wield considerable power over vast troves of data, individuals find themselves with minimal influence over the fate of their personal information. Due to companies' control over such large data piles, individuals

[*] Corresponding author. E-mail: joachim.vanherwegen@ugent.be.

have very little influence over what happens to their own data. Fortunately, recent legislative changes, such as the GDPR in the EU and the CCPA in California, signal a positive shift, indicating that companies will eventually need to acknowledge their inability to exert complete control over all data. Once people regain control over their own data, they can decide with whom they wish to share it, thereby motivating companies to offer services of a high enough quality to earn that privilege. This dynamic would foster a mutually beneficial relationship, as end-users would experience tangible benefits from data sharing, such as personalized services, improved privacy, and greater transparency, ultimately resulting in a more valuable and sustainable data exchange.

Historically, Semantic Web research has focused on the exchange of *Open* Linked Data, with little consideration for data where access might be restricted by policies, as is the case with personal data. These restrictions drastically change the infrastructure and processes, such as publication and query processing. Nonetheless, Semantic Web technologies can play a crucial role in placing data closer to people, because they can give universal and connected semantics to data that is managed in a highly decentralized way.

Solid [22] is an ecosystem that rises to the challenge of tackling the private–public data spectrum. It does so by going from a horizontal centralization, where all data for a single company is centralized, to a vertical centralization, where data of all companies for a single user is centralized instead. The main idea is that everyone using Solid has one or more *data pods* containing their personal data. By setting relevant access policies, users can specify who can read or write parts of their data. Solid clients that know how to interact with such a server can then be used to access that data.

In an open ecosystem such as Solid, any party can implement a server, as long as they abide by the Solid Protocol specification [7]. Similarly, for the same reason anyone can also make a client to communicate with such a server. As a consequence – and this is a core part of Solid – any client can interact with the data created by any other client, on any Solid server. Since the data is stored in a user's data pod and not in a specific client, clients should be seen as views and controls over that data. For example, one application could be used to set a person's date of birth, which could then be used in a completely different one as a birthday reminder.

There are many invested parties in the Solid ecosystem: companies addressing real-world use cases, researchers want to evolve the specifications to suit the necessary demands, developers want to create clients and servers to extend the reach of Solid, and new users just want to try it out. In this paper we introduce the *Community Solid Server (*CSS*)* as open-source software [29] as a tool to support research and development of current and future Solid specifications.

The primary problem the CSS addresses is the need for a highly flexible and user-friendly platform, particularly for new users and researchers. This is significant because flexibility and ease of use are crucial for the adoption and experimentation of Solid technologies, which drive the advancement of the ecosystem. While existing servers offer ease of use and some degree of customization, they typically do not provide the full modularity required for comprehensive customization or experimentation, limiting their utility for a wide range of use cases.

Our CSS distinguishes itself by offering a fully modular architecture that allows users to swap out components to enable new features or add experimental ones using dependency injection. This flexibility is achieved through JSON configurations, supported by comprehensive tutorials, documentation, and tooling that assist users in generating these configurations. While other servers offer valuable features, our server's design specifically caters to the needs of both new users and researchers, providing an unprecedented level of customization and making it a valuable tool for the Solid community.

In Section 2 we discuss related work. Section 3 covers specific use cases we want to solve with the server, which are generalized into requirements in Section 4. Section 5 and Section 6 explain how those requirements are fulfilled, the former through the software architecture and the latter through the configuration the server allows. Section 7 gives an overview of how the server is currently already being used, the impact it has, and how it solves the originally proposed use cases. Finally, we conclude in Section 8 where we also look towards the future of the server.

2. Preliminaries & related work

Before going into the specifics of our solution, we first give an overview of the relevant and related research.

2.1. Basic solid interaction

Before diving deeper into the specifications that define the client–server contract, we start with an overview of what happens during a request to such a server, in order to describe the high-level interaction.

2.1.1. Prerequisites

Combining Linked Data with authentication and authorization is an ongoing research topic with different potential solutions. One suggested solution is through the usage of WebIDs [20], which are HTTP URIs that uniquely identify a person or any other kind of agent.

In the example below, we assume the user already registered their WebID with a Solid Identity Provider (IDP), which is able to prove that they are the owner of that WebID. The server is free to implement its own identity verification mechanism (email/password combination, a specific kind of token…).

2.1.2. Performing an HTTP *request to a Solid server*

When a client wants to access data on a Solid pod on behalf of a user, the following steps are performed:

1. The client asks the user to authenticate with their IDP, and receives the necessary authentication data in turn.
2. The client uses that authentication data to generate HTTP headers which prove the identity of the user to the Solid pod containing the data.
3. An HTTP request for a read/write/update/delete operation on a specific resource is sent to the Solid pod.
4. The Solid pod contacts the IDP to determine the validity of the authentication headers.
5. If valid, the server uses those headers to determine the user's credentials (such as the WebID) and their client.
6. The server determines which permissions are available for the given credentials on the targeted resource, such as read, write, create, delete, etc.
7. The server determines if the requested operation can be performed with the available permissions.
8. If allowed, the server performs the operation and returns the result to the client.

2.2. Solid specification documents

The Solid ecosystem consists of a collection of specifications that clients and servers are required to adhere to. The interactions outlined above are captured in these specifications, of which the CSS is one implementation.

2.2.1. History and current status

The initial version of Solid was developed in tandem with prototype implementations such as the Node Solid Server (NSS) [2]. While the specification and other implementations were still in development, the definition of a "Solid server" was defined by the behavior of NSS. This behavior was first documented as notes and eventually as specifications in a W3C Community Group. At the time of writing, the transition to a W3C Working Group is being undertaken, which is able to create W3C Recommendations that serve as recognized standards.

However, the Solid specifications are still evolving, with both changes to existing documents and additional documents being suggested as part of a multiphase process. Hence, there is a need for an implementation of the specifications that can be used to implement and test changes to the specifications, and to explore and prototype desired future behavior of Solid implementations.

2.2.2. Solid protocol

The Solid Protocol specification [7] is the main entry point into the collection of specification documents that define Solid. As the main entry point, it defines which other specifications are required to be fulfilled for a server to be Solid-compliant, which we will cover in the following subsections. In particular, it contains an adaptation of the Linked Data Platform (LDP) specification [1]. On top of the existing HTTP methods (`GET` to read data, `POST` to create new resources, `PUT` to write data, `PATCH` to modify, and `DELETE` to remove), it defines specific semantics for patching RDF documents, and for interacting with containers of resources. Containers are resources that group other resources together by providing RDF descriptions with containment triples.

2.2.3. *Authentication*

The Solid-OIDC specification [8] is the authentication solution recommended by the Solid specification. It expands upon the OAuth 2.0 [15] and OpenID Connect Core 1.0 [19] standards and defines how clients can identify by requesting specific tokens from a server. It also defines how servers can provide these tokens and how they should verify their authenticity.

To conform to the Solid Protocol specification, a server of Solid pods needs to be able to accept requests containing these tokens and verify them. An OpenID Provider, on the other hand, is a server where clients can register to generate such tokens. To verify correctness of a token, the Solid server communicates with the OpenID Provider.

2.2.4. *Authorization*

A Solid server needs to be able to restrict access to private data. The specification defines two possible access control systems that can be used to do this: Web Access Control (WAC) [5] or Access Control Policy (ACP) [3]. At a high level these are quite similar systems: users can add system-specific resources to the server, indicating the credentials that are required to perform certain actions on their data. They can be used to, for example, provide public read access to certain resources so everyone can see them, or allow everyone to create new resources in a specific container as a way to allow people to leave comments or other communication, while still only allowing the data pod owner to edit the data. The protocols differ in how policies are inherited and how clients are identified.

2.2.5. *Notifications*

The Notification specification [6] clarifies how users can register to specific resources, after which the server will inform them of any changes. At the time of writing, the specification only clarifies how clients can register, and the data models used during the communication process, but not which kind of messages need to be sent out. It also specifies many different methods a server is allowed to use to send out those notifications, such as WebSockets or Webhooks.

2.3. *Existing implementations*

Several implementations of the specification exist, both on the server side and on the client side; we provide a non-exhaustive list of server implementations.

Non-commercial implementations include projects such as the open-source Node Solid Server (NSS) [2] and Solid Nextcloud plugin [18]. There are also commercial implementations such as the Enterprise Solid Server (ESS) [17] and TrinPod [14], both closed source at the time of writing.

These implementations each have their own priorities, none of which is the core of what we want our server to focus on. This is why we created the server discussed in this paper, of which we will discuss the priorities in the following sections.

NSS resulted from prototyping efforts during early phases of the Solid Protocol, and is used for development and testing. Its implementation is currently maintained by volunteers. Architected from a prototyping perspective back when the Solid Protocol was still forming, the NSS codebase is no longer well-suited to follow the current evolution of the specifications without a substantial rework and repurposing.

Solid Nextcloud adds support for the Solid protocol to Nextcloud, a collection of client and server software for using file hosting services. The main advantage here is that it builds on a known and stable storage method, adding extra functionality on top of it, which can be very useful for users already using Nextcloud as well.

The ESS focuses on security as scaling, positioning itself as a solution for companies that want to run a large Solid server that can guarantee safety.

TrinPod positions itself for Digital Twin use cases using the Solid specifications. It provides several utilities, independent of the Solid specification, focusing on supporting users working with such use cases, such as search functionality and advanced user interfaces.

3. Use cases

In this section, we give an overview of several use cases we wanted to support by creating a new server. These use cases were formulated based on discussions with community members and researchers who are familiar with the challenges and needs in developing and utilizing Solid servers. Each use case represents a practical scenario where the capabilities of the Community Solid Server (CSS) could address specific issues or provide enhancements.

Each of these builds upon the technology described in Section 2. In Section 7.2, we revisit these use cases and discuss how CSS addresses each scenario, demonstrating its utility and impact in advancing Solid server capabilities.

3.1. Benchmarking the impact of authorization to inform the specification

A protocol researcher aims to benchmark the differences between WAC and ACP for an HTTPS client. These are different authorization schemes a Solid server can have, and they want to find out how they impact a request. To this end, the researcher would need at least two Solid servers which are identical in every regard, except that one supports WAC, while the other supports ACP. Preferably they would also have a Solid server without authorization as a baseline. This would allow them to accurately measure the impact of either authorization scheme, which can be used to inform future specification changes.

3.2. Performing user experience research on the onboarding experience

A societal researcher wants to compare different welcome experiences to a Solid server, specifically the sign-up experience and the initial layout and contents of a pod. This way they can determine what might be needed to improve the experience with Solid for new users. For this, they want to have a server where they can easily replace the contents that get provided to new users, without having to write any code.

3.3. Supporting new operations

The behavior of `PATCH` is still under discussion within the Solid W3C Community Group. The first proposed `PATCH` format relies on SPARQL Update, which has the benefit of being an existing W3C standard, but lacks a semaphore mechanism. An alternative with N3 Patch defines a semaphore mechanism, but relies on the N3 language that is currently not a standard. Now a researcher wants to propose different `PATCH` formats along with an implementation, but without having to implement a full Solid server themselves.

3.4. Supporting the adaptation of research findings

Ongoing research looks at different aspects of the Solid Protocol and its implications on domains such as data management and security. Occasionally, findings from such research result in a necessity for changes or extensions to the specifications in order to alleviate the discovered concerns.

As a concrete example, recent research indicated a tension between the granularity of document organization and the granularity of the authorization system [10]. Their conclusion is that the same data needs to be exposed in different documents with different permissions. To investigate this, a server that allows for modifications in how to expose data is required.

4. Requirements

Out of the specific use cases in Section 3, we distilled several generic requirements that guide the design, architecture, and implementation of CSS. The main goal is to explicitly focus on the needs of two groups: *researchers* and *developers*.

4.1. Testable specification compliance

It stands to reason that the most important requirement for the server is that it is fully spec compliant. While that is an implementation objective, it is also necessary that we can verify and prove that this is the case. Ensuring compliance is critical for the use case of benchmarking authorization impacts, as it provides accurate and reliable results.

4.2. Evolve along with the specification

Solid is a combination of still evolving specification documents. It is imperative that the server can keep up with these changes; an outdated server could damage the ecosystem by sowing confusion about the correct behavior. Therefore, the server must be designed to be easily adapted with updates in the specifications.

Researchers and developers are at opposite ends here: researchers aim to inform the evolution of specifications, while developers prefer a more stable experience, yet want to be able to test their applications against the latest versions of the specifications. Both sides require a server that evolves to achieve their goals.

Since the server is a combination of several different specifications, the architecture needs to be designed in such a way that changes in one specification do not break a requirement of a different one: independence of all layers is important. This flexibility is necessary for the use cases of benchmarking authorization impacts and supporting new operations, ensuring the server remains relevant and accurate.

4.3. Support multiple server roles

Multiple servers are involved in a Solid interaction: the Solid server handles the core Solid protocol, and the OpenID Provider provides OIDC authentication. We need to provide a solution that covers all the necessary roles involved, thereby providing an out-of-the-box Solid experience.

Having an all-in-one system allows anyone to get started as easily as possible; modularity allows different kinds of instances from the same codebase. This modularity means different server roles can be configured and tested independently, allowing for comprehensive experimentation across the full range of Solid specifications. This requirement aligns with the use cases of performing user experience research and adapting research findings, providing a comprehensive testing environment.

4.4. Modularity and configurability

When conforming to the Solid specifications there is still room for variability, such as which authorization system to use, or even how the data is stored in the backend. Configuring changes like this in the server should be as easy as flicking a switch to go from one option to another.

For researchers, it is important to be able to compare different variations, so they can investigate the impact of certain changes. For example, having either WAC or ACP as an authorization system might have a major influence on the performance of the server, which might cause changes in the specification to bring those more in line with each other. To study such differences, they need to be able to set up server instances with different feature sets.

Developers want different server versions to make life easier for them when doing their work. Specifically, they need ways to simulate certain server situations to see how it reacts. E.g., force the authentication to extract specific credentials to simulate different users, disable authorization to focus on data management, cause the server to have faulty data for exception handling, etc. The use cases of benchmarking authorization impacts and supporting new operations benefit directly from this modularity, enabling varied configurations for thorough testing.

4.5. Allow extensions with new features

One part of doing research on Solid is designing new features based on emerging needs, with the ultimate aim of producing new specifications for uniform behavior. For example, currently the specification defines WAC and ACP, but one might want to investigate a new authorization scheme or way of enforcing policies [13].

Extensions could also replace existing parts of the server. Someone might want the data to be stored in a new type of backend for example, or provide a new implementation of a feature that is highly optimized for certain scenarios. The ability to extend and replace components is essential for all use cases, facilitating flexibility for new experiments and improvements.

4.6. Quick setup and teardown

A Solid server is not a simple piece of software. Generally there are additional steps that need to be taken before it can be started. These include configuration, starting external services, etc. Similarly, shutting down the server and resetting the system, so a clean restart is possible can also take multiple steps.

If we want the server to be used for rapid experimentation, it is important that there is as little overhead as possible. Researchers might want to quickly set up and switch between different kinds of servers to run their experimentation; for developers, this enables the server to be used within their test frameworks, automating the testing against a Solid server. For newcomers, it lowers the barrier of entry for getting started with Solid: the faster someone can go from reading about Solid to setting up a server, the better the introductory experience. This ease of setup and teardown supports all use cases, ensuring efficient experimentation and a smooth introduction to Solid.

4.7. Error handling and logging

Many steps happen during a Solid interaction, and when something goes wrong in a decentralized system, it is not always straightforward to determine which component is at fault. Therefore, it is necessary that the server has extensive error handling and logging. Researchers can use this to detect potential issues with specific interactions they might not have considered. Developers trying to debug or troubleshoot specific applications can receive better feedback this way. Extensive error handling and logging is crucial for all use cases, providing insights and feedback for debugging and improvements.

5. Architecture

Based on the requirements in the previous section, we now introduce the architecture of the Community Solid Server as an open-source implementation of the Solid specifications, tailored towards research and development.

For specification conformance, CSS needs to not only provide the HTTP request handling for data interactions, but also to authenticate clients, authorize requests, send out notifications, etc. It also acts as an Identity Provider and handles user account management in that context. All of these features are orthogonal to each other and depend on the different specifications described in Section 2.2. Figure 1 shows how these orthogonal features interact with each other through the steps of a request to a Solid server. Each of the displayed components (except for the Client initiating the request), represents a part of the CSS architecture.

5.1. Main components

The different roles the server supports are independent of each other in the architecture. They might use some of the same utility classes and store data in the same way, but besides that, changes for one major component will have no bearing on one of the other ones. This allows us to limit the impact of evolving specifications as mentioned in the requirements.

As CSS is a server that handles HTTP requests, each interaction is initiated by an incoming request, and terminated with an outgoing (data or error) response. After an initial routing step, the modus operandi is the same for every request: each component iteratively chips away at a small part of the request to facilitate the next step in the process, until all these small steps result into the output object that is then serialized towards the client.

The first step determines which major component should handle the request, based on the request type:

Static resources Requests for a fixed list of static resources, such as images and styling used on HTML pages, or having a static page to welcome the user.

OIDC interactions Interactions related to the OIDC specifications defined in Section 2.2.3. The server generates the necessary tokens, validates all the input data and exposes the necessary APIs.

Account interactions Registering an account and pod on the server is completely independent of the OIDC protocol. This is done through a set of JSON APIs, for which we also provide user-friendly HTML pages.

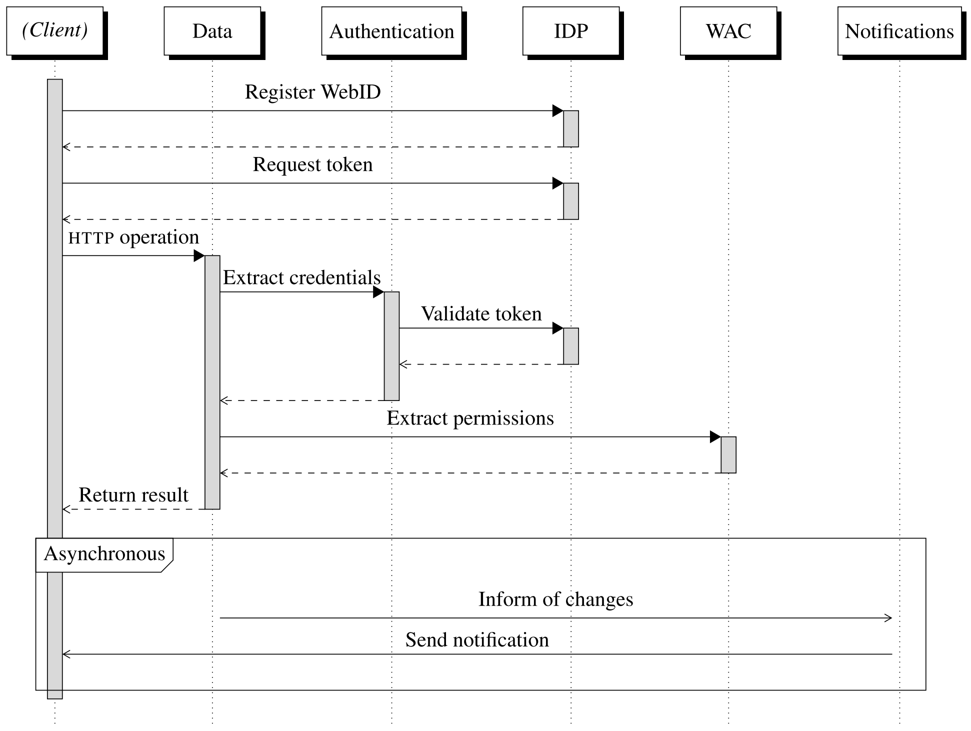

Fig. 1. Describes how the different components of the server interact with each other to resolve a request.

Notifications The notification specification defines several interactions that need to be supported through an RDF API, such as requesting all the available notification types and subscribing to a specific resource.

Solid Protocol operations Any HTTP operation that is not handled by one of the previous major components will be covered by the final block, which implements the Solid protocol. Since this is the most defining part of a Solid server and also the most extensive one, we cover it in detail below.

5.2. Handling a Solid protocol operation

When handling a request targeting a Solid resource, 4 core steps need to happen sequentially:

1. Parsing the request.
2. Extracting and validating the credentials of the client.
3. Verifying the authorization of the request.
4. Performing the operation described in the request.

The result of these steps will then be used to generate the HTTP response that is sent back to the client.

5.2.1. Request parsing

In this step, the raw input is normalized and validated, so later classes do not have to worry about edge cases arising from permitted differences in description. The full URL of the request is reconstructed, as this is the identifier of the resource that is being targeted. The `Accept` headers are parsed into a preferences object to be used for content negotiation. The headers related to conditional request, such as `If-Match`, are all combined into a single conditions object that can later be used for validating the request. All other relevant headers are combined into a single RDF metadata object.

Finally, the body is processed in case of `PUT`, `POST`, or `PATCH` requests. For `PUT` and `POST`, this involves verifying the relevant HTTP headers and wrapping the data stream to prevent unexpected asynchronous errors. In the case of `PATCH`, the stream is immediately fully parsed into a `Patch` object containing all requested changes, as this determines the exact kind of operations involved and hence the required permissions.

5.2.2. Authentication

While authentication involves many separate steps, CSS outsources most of this to an external library [4] of existing specifications that are not exclusive to Solid. The result of executing this step is an object containing all the identifying information of the client, including, most importantly, the WebID.

5.2.3. Authorization

The authorization step determines if the agent identified by the credentials in the incoming request has the necessary permissions to perform the operation expressed therein. During the authorization step, we determine which internal access modes are required to fulfill the operation. For example, a `GET` request requires Read, a `PUT` requires Write, and a `PUT` creating a new resource requires Write and Create. With `PATCH`, we inspect the parsed request body.

For each resource, CSS can determine which of those required access modes are actually available for the given agent. These can be specified using an access control system such as WAC or ACP. Each such system comes with its own implementations to parse the access controls and convert them to the CSS internal access modes. For example, WAC does not have a native Create mode; instead, this behavior is inherited through Append permissions on a resource's parent container.

Once both of sets have been determined, they are compared to each other. If any of the required access modes is not found in the set of available modes, access to the resource will be forbidden, and the request will be terminated with an appropriate `401` or `403` status code.

5.2.4. Operation

After authorization succeeds, the request has to be handled according to rules defined in the Solid Protocol specification. There is a specific class for each HTTP method, which calls the correct operation responsible for data management. These operations are captures in the `ResourceStore` interface, which contains key functions for performing CRUD (Create, Read, Update, Delete) operations on resources.

To support everything needed in the backend, we use multiple `ResourceStores`, each with their own specific function, which are then chained together following the Decorator pattern to represent a single store. The first store uses a read/write locking mechanism to make sure it is not possible to perform simultaneous operations that would result in data conflicts. The lock is only released when the other stores are finished.

`PATCH` is the more peculiar method of those defined in the Solid Protocol. Whereas other HTTP methods correspond to a single function, the behavior of `PATCH` is defined entirely by the request body. To prevent the need for backend-specific `PATCH` operations, the CSS architecture includes a `ResourceStore` that performs `PATCH` operations as a set of elementary resource manipulations, independent of how data is stored. Depending on the chosen `PATCH` body (currently N3 Patch and SPARQL Update are supported), a different algorithm is applied to an in-memory RDF document, which is only persisted in case of success.

Content negotiation The modular CSS architecture relies on content negotiation, not just for end-to-end reformatting for clients based on `Accept` headers, but also for internal use. For instance, during the aforementioned handling of `PATCH`, the CSS uses internal content negotiation to correspond an RDF serialization into triples and back. All of this happens in a single `ResourceStore` that checks the preferences of requests and converts the response data to match what is preferred.

To perform any kind of data conversion, we make use of a set of very narrowly focused conversion classes. All of these have a set of specific media types they can parse, and similarly a set it can convert to. For example, there is a converter that accepts various RDF serializations and outputs memory-native Quad objects, while another converter specifically converts Markdown to HTML. Exceptions and errors also internally pass through this system, such that detailed error reports can be relayed correctly to clients in different serialization formats.

A pathfinding algorithm chains multiple converters together as necessary to create a path starting from the resource media type as found in storage to the preferred type requested by the client, streaming data rather than materializing whenever possible. This allows for new conversion paths to be supported through a single new converter, rather than needing to implement all variations. For example, CSS requires no dedicated converter from Turtle to JSON-LD, because there are converters from Turtle to Quads, and from Quads to JSON-LD. Chaining those together produces the desired result.

Performing the requested operation Depending on the specific HTTP method and the type of the target resource, several checks and steps are selected. For example, `POST` only works when targeting a container. Similar to the idea behind the `ResourceStore` that handles `PATCH` operations, many of the steps here are independent of how the data is actually stored. The final `ResourceStore` implements this general behavior. Backend-specific implementations are hidden behind a more elementary `DataAccessor` interface that can be used to read and write data. There then is a `DataAccessor` implementation for storing data in memory, on a file system, etc.

6. Configuration

While we use the term "Solid server" throughout this paper, this can be a misnomer, as it might give the impression that a server is one opaque monolith. As indicated in Section 5, there are instead many components that play a role in a Solid operation. While these can all be realized on the same server, this is no such requirement; they could be split up over different servers with their own responsibility. Even then, there is still significant wiggle room as to how a server fulfills one or more of these roles.

One of the core parts of CSS is that these roles can be configured differently depending on the needs of the researcher or developer. For example, one might want to change the authorization system used by the server, or perhaps they want CSS to only support some of the necessary roles. In case someone already has their own Solid IDP setup, they might want to disable that part of CSS and have their server only handle the data operations.

All the CSS classes focus on solving a specific problem, and there are no classes that instantiate these and link them all together. Instead, we make use of Dependency Injection with the Components.js [23] framework. Components.js is non-invasive as all of its configuration happens outside the source code, in external configuration files that describe how classes are linked to each other and which parameters they require. These descriptions are RDF, usually JSON-LD, which means that configuration files are valid RDF serializations. It thereby provides the flexibility that is necessary to compose a server as the user wants, at the cost of increased complexity.

In Components.js, TypeScript class instantiations correspond to RDF class instantiations. Every subject is a class, with its type corresponding to a TypeScript implementation. Other predicates are used to define its constructor parameters and can, since it is RDF, link to other objects. To then change which components are used in a server instance, only the Components.js configuration has to be changed; the actual TypeScript code does not need to be touched.

To help users get started, the server comes with several pre-defined configurations, covering a range of possible feature combinations. These already cover several of the more expected server setups, with some variations in data storage methods or authorization systems for example. An overview of these can be seen in Table 1.

Table 1
An overview of some preset configurations provided with the Community Solid Server. A more detailed overview can be found at https://github.com/CommunitySolidServer/CommunitySolidServer/tree/main/config

Name	Data storage	Internal storage	Access root	Authorization	Accounts	HTTPS	Subdomains	Quota
default	Memory	Memory	✓	WAC	✓	X	X	X
memory-subdomains	Memory	Memory	X	WAC	✓	X	✓	X
file	File	File	X	WAC	✓	X	X	X
file-acp	File	File	X	ACP	✓	X	X	X
quota-file	File	File	X	WAC	✓	X	X	✓
file-root	File	File	✓	WAC	X	X	X	X
file-root-pod	File	File	Pod	WAC	X	X	X	X
https-file-cli	File	File	X	WAC	✓	✓	X	X
sparql-endpoint	SPARQL	Memory	X	WAC	✓	X	X	X
sparql-endpoint-root	SPARQL	Memory	✓	WAC	X	X	X	X
sparql-file-storage	SPARQL	File	X	WAC	✓	X	X	X
oidc	X	File	X	X	✓	X	X	X

The preset configurations are made by clustering related components together in partial configuration files, and importing them in the main entry point. Features can then easily be chosen by changing what is imported. For example, to swap between WAC and ACP as authorization system on the server, the WAC configuration import would have to be replaced by the ACP import. There are plenty of choices that can be made by using the imports like that, such as how data is stored, whether users can register, HTTP vs HTTPS for connections, etc.

We chose TypeScript because of its inherent advantages such as design-time safety and the resulting typing documentation for developers who aim to reuse CSS modules or integrate with them. Note that developers of external modules are not bound by our choice for TypeScript, as they can implement in plain JavaScript or any language that compiles to it. An extensive collection of both RDF and Solid-related libraries is available in TypeScript. Furthermore, Components.js leverages TypeScript type definitions to automatically provide and validate configuration options.

7. Sustainability, usage & impact

In this section we give an overview of the support we provide to users wanting to get started with the server. We also provide several examples of people making use of the server, showcasing how what we described in the previous sections is well-received. We encourage readers to try out the server for themselves. A good starting point is following the tutorial at https://github.com/CommunitySolidServer/tutorials/blob/main/getting-started.md, which covers how to set up a server, send HTTP requests to it, and how to change the server configuration.

All code of the Community Solid Server is open source under the MIT license. At the time of writing, the repository [29] has been starred 473 times, forked 117 times, and is used in 149 other repositories. 46 people have contributed to the repository.

7.1. Sustainability

In the context of this work, sustainability refers to the ability of the server to remain reliable, maintainable, and up-to-date with evolving standards over time. During the development of the server, we have always focused on making sure the code base remained of high quality. One aspect of this is requiring the unit tests to always have 100% code coverage on all code in the project. While this is not immediately an indication of everything working as intended, it does make sure that a developer checks that new classes output data as expected. Besides the unit tests there are also extensive integration tests, setting up complete instances of the server and verifying these instances conform to all the necessary specifications.

The Conformance Test Harness (CTH) [11] is a server-independent test framework for Solid servers. It runs a test suite to verify if a specific server fulfills the specification requirements. Besides our own internal tests, we also run the CTH as a form of external audit on the server functionality, to prove that the server fully conforms to the specifications.

This rigorous testing strategy contributes to the sustainability of the tool by ensuring that each new feature or change is thoroughly vetted for compliance and functionality. This helps maintain a high level of quality and reliability, ensuring that the server can evolve alongside the specifications without compromising on features.

This test harness is also an example of the community impact of creating this server. During the creation of this test harness, we provided feedback on certain tests being too strict or incorrect, while other tests showed us where the server implementation was wrong. This bidirectional approach caused both systems to improve.

7.2. Use case relevance

Having covered how the server works, we will now discuss how this solves the use cases described in Section 3.

7.2.1. Benchmarking the impact of authorization to inform the specification

A researcher aimed to compare WAC and ACP with a baseline. This means they need a server that loads the WAC component, one that loads the ACP component, and one that loads neither. As we have seen in Section 6, they can easily change which components are used by changing the imports in their configuration file.

Configuring HTTPS involves passing the certificate, which can either be set there in that configuration, or passed as command-line arguments. This allows the same configuration to be reused with different certificates.

In conclusion, they end up with three different configurations, which can all be used to independently set up the servers they need. Specifically the server without authorization can also be useful for developers not wanting to be bothered with restrictions during development.

7.2.2. Performing user experience research on the onboarding experience

The societal researcher wants to customize the welcome experience of a Solid server. Which HTML files to use, and what the template of a new pod is, is defined in the CSS configuration as part of a JSON-LD file. Components.js allows specific parts of a configuration to be replaced, so this can be used to replace the HTML parts wherever necessary. Similarly, the pod contents are determined based on templates, which can also easily be updated. Again developers can also make use of these ideas, as they can quickly set up pods with specific contents, without having to perform the initialization themselves.

7.2.3. Supporting new operations

A researcher wants to support a new PATCH format. How a PATCH is resolved, and which components are necessary, was covered in Section 5.2.4. Due to the nature of Components.js the researcher can develop the new components in a separate repository independent of the CSS core. Afterwards these can be linked together with the already existing configurations. This allows the new PATCH algorithm to be shared as a separate piece of code with the W3C group. Due to its independence it can easily be tested with different authorization frameworks, such as ACP and WAC, which is an additional bonus as PATCH has specific interpretations in authorization.

7.2.4. Supporting the adaptation of research findings

Recent research showed that exposing the same data through different resources would solve several problems [10]. Even adding support for something like that in CSS is possible by creating new components. As the proof is in the pudding, we implemented a new component that supports this idea [28]. This new component allows users to define so-called derived resources, which are generated by determining a set of resources and a query to perform upon them. As with other components, this can be combined with existing configurations to still provide the full flexibility that is possible with a CSS configuration.

7.3. Community impact

Since people started using the server, several of them have fed back input to help improve CSS. There have been several pull requests by community members to extend the server or fix specific issues. At the time of writing there have been 44 contributors to the repository.

Several external components have also been developed which either make use of the server, or create a new component that extends the server functionality, including:

– integration with calendar management systems [16]
– Internet of Things integrations [33]
– Data-Kitchen, a desktop app combining local files and Solid pods [35]
– Solid-Redis, a component for the server to use Redis as data storage [12]
– input validation using shape files [21]
– publishing event streams via Solid [34]

Another goal we wanted to achieve was to support client developers that need a server to test their client against during development. Below are some applications that use the server specifically for this purpose:

– viewing and manipulating personal data in your Solid pod [24]
– recipe manager to collect all your recipes [9]

The server has also been used for research & development by institutions, such as CERN, Geonovum, and Arbetsförmedlingen, the Swedish Public Employment Service.

7.4. Supporting people who use the server

Due to the server aiming to support many different scenarios, it can be overwhelming for new users to know how and where to start. To help users there, we have created extensive documentation, tutorials, and tools explaining and helping with different parts of the server.

The documentation [31] of the server is the best place to start as it links to all other resources available. The documentation itself covers several server-specific features and how to use them, such as automating pod creation on server startup for testing purposes. Besides the user documentation there is also an overview of core parts of the architecture. The components discussed in Section 5 are explored more deeply there. Finally, there are the full technical definitions of all the classes the package exports, which can be used by projects aiming to extend the server.

The server has extensive logging to help users keep track of the server internals. The logger makes use of log levels, so users can fine-tune the details of the output, going from just error levels that indicate an internal issue with the server, to debug levels where every resource lock is logged. It also includes additional details such as timestamps, the worker thread, and the component responsible

We created several extensive tutorials which guide the user along to solve specific problems. One tutorial [25] helps users who are completely new to Solid and shows how to interact with all the Solid core principles by setting up a CSS instance. It starts with showcasing the core Solid HTTP requests, after which it also introduces how authentication and authorization can be combined. For users who want to extend the server there is a specific tutorial [26] that covers all the possible options in which configurations can be modified.

An example repository [27] creates a Hello World component and extends CSS with it. Developers can copy that repository and replace it with their own code where necessary. It is fully documented on what the function is of every file there and includes examples on how to add your new component to an existing server configuration or how to set up automated testing for it.

Many people prefer using a GUI when interacting with a server for easier accessibility. Several Solid servers come with one built-in. The NSS for example, comes with the *mashlib databrowser*. CSS does not come with a built-in interface. Instead, thanks to the configuration solution, any Solid-compatible interface can be used, including the same mashlib interface used by NSS. We have provided a repository [32] where examples show how to configure such a server.

Finally, as mentioned in Section 6, the server offers a very large number of configurable parameters for every one of its features. A disadvantage of using this method to configure a server, is that this can be overwhelming for users not familiar with Components.js. To this end, we created a Web-based graphical interface [30] that can generate such configurations automatically, based on the selection of desired features.

8. Conclusions & future work

We set out to create a Solid server with a specific set of requirements, focusing on flexibility, extensibility, and support for specific kinds of Solid users. Due to the usage of dependency injection, it is possible to run many variations of the server with different features, and to create new components that can be added. By structuring the configuration and providing plenty of supporting tools and documentation, we have lowered the barrier of entry as much as possible, making the server accessible for people looking to experience Solid, without hindering users looking for more advanced features.

There are several situations in which the server is being used: people created different components to be added to a default installation, it is being used during the testing of client applications, and there are several running Solid server instances making use of this software. More and more people are finding their way to the repository and interacting with it, showing a growing demand for a server that fulfills these needs.

Creating a new server in the Solid ecosystem also helps in improving the Solid specifications. By providing an alternative implementation it can reveal hidden assumptions that are not specified, but are depended upon due to the existing implementation having this specific behavior.

Work on the server is not finished yet, there are still many open issues that need to be resolved, many of which are feature requests on how the server can be extended. The CSS aims to take a leading role in shaping future Solid specifications, by providing researchers and developers with a flexible environment for testing and experimentation.

Acknowledgements

The research in this paper was supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project VV023/10). The development of the Community Solid Server has been supported by Inrupt, Inc.

The authors would like to thank Tim Berners-Lee for his dedication to the Solid project, with one of his many contributions being design and architecture for the Node Solid Server, which crucially informed CSS. In addition, he has been a daily user and tester of our technology, and we are very grateful for his continued feedback and support.

Appendix. Definitions

The Solid project A Web decentralization project. It aims to fundamentally change the way web applications work today, resulting in true data ownership as well as improved privacy. The core idea is to decouple data from the applications that use it so that individuals have control over their own data.

Solid Shorthand for The Solid project.

Solid Ecosystem The Solid Ecosystem refers to the network of technologies, applications, servers, and protocols that support and implement the principles of Solid.

Solid Server A system that hosts resources and manages data storage. A Solid server implements the Solid specifications and protocols to allow users to store, retrieve, and manage their data.

Solid Client/Application An application or tool that interacts with data stored on a Solid Server.

Resource A resource in Solid refers to any data object stored within a Solid server. This can be any type of data, such as documents, images, or metadata. Resources are identified by URIs and can be accessed, modified, or deleted based on the access controls set by the data owner.

Container A type of resource used to group other resources together on a Solid server. Containers are analogous to directories or folders in a file system and can contain both data resources (such as documents, images, and other files) and other containers, allowing for hierarchical organization of data.

Pod While not defined explicitly in the Solid specification, this is a term commonly used for a specific namespace where a user stores all their data, and they have full control over.

WebID A unique identifier used in the Solid ecosystem to represent an individual. It is a URI that points to a profile document describing the person and their associated metadata. WebIDs are crucial for authentication and authorization within Solid, enabling users to control access to their data.

Identity Provider A service that creates, manages, and verifies the identity of users. It issues authentication tokens that are used by servers to grant or deny access based on the user's identity and the associated access policies.

References

[1] J. Arwe, A. Malhotra and S. Speicher, Linked Data Platform 1.0, W3C Recommendation, Vol. W3C, 2015. https://www.w3.org/TR/2015/REC-ldp-20150226/.

[2] T. Berners-Lee et al., Node Solid Server, 2023. https://github.com/nodeSolidServer/node-solid-server/.

[3] M. Bosquet, Access Control Policy, Editor's Draft, W3C Solid Community Group, 2022. https://solidproject.org/TR/2022/acp-20220518.

[4] M. Bosquet, Solid OIDC Access Token Verifier, 2023. https://github.com/CommunitySolidServer/access-token-verifier/.

[5] S. Capadisli, Web Access Control, Candidate Recommendation, W3C Solid Community Group, 2022. https://solidproject.org/TR/2022/wac-20220705.

[6] S. Capadisli, Solid Notifications Protocol, Editor's Draft, W3C Solid Community Group, 2022. https://solidproject.org/TR/2022/notifications-protocol-20221231.

[7] S. Capadisli, T. Berners-Lee, R. Verborgh and K. Kjernsmo, Solid Protocol, Editor's Draft, W3C Solid Community Group, 2022. https://solidproject.org/TR/2022/protocol-20221231.

[8] A. Coburn, elf Pavlik and D. Zagidulin, Solid-OIDC, Editor's Draft, W3C Solid Community Group, 2022. https://solidproject.org/TR/2022/oidc-20220328.

[9] N. De Martin, Umai, 2023. https://github.com/NoelDeMartin/umai.

[10] R. Dedecker, W. Slabbinck, J. Wright, P. Hochstenbach, P. Colpaert and R. Verborgh, What's in a Pod? – A knowledge graph interpretation for the Solid ecosystem, in: *Proceedings of the 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs*, M. Saleem and A.-C. Ngonga Ngomo, eds, CEUR Workshop Proceedings, Vol. 3279, 2022, pp. 81–96. ISSN 1613-0073. https://solidlabresearch.github.io/WhatsInAPod/.

[11] P. Edwards, Conformance Test Harness, 2023. https://github.com/solid-contrib/conformance-test-harness/.

[12] A. Faulkner, Redis database backend extension, 2023. https://github.com/comake/solid-redis/.

[13] M. Florea and B. Esteves, Is automated consent in solid GDPR-compliant? An approach for obtaining valid consent with the solid protocol, *Information* **14**(12) (2023). https://www.mdpi.com/2078-2489/14/12/631. doi:10.3390/info14120631.

[14] Graphmetrix, TrinPod, 2023. https://graphmetrix.com/trinpod.

[15] D. Hardt, The OAuth 2.0 Authorization Framework, RFC, 6749, RFC Editor, 2012. http://www.rfc-editor.org/rfc/rfc6749.txt.

[16] P. Heyvaert, Solid Calendar Store, 2023. https://github.com/KNowledgeOnWebScale/solid-calendar-store/.

[17] Inrupt, Enterprise Solid Server, 2023. https://www.inrupt.com/products/enterprise-solid-server/.

[18] PDS Interop, Solid Nextcloud, 2024. https://pdsinterop.org/solid-nextcloud/.

[19] N. Sakimura, J. Bradley, M.B. Jones, B. de Medeiros and C. Mortimore, OpenID Connect Core 1.0, Technical Report, OpenID Foundation, 2014. https://openid.net/specs/openid-connect-core-1_0.html.

[20] A. Sambra and S. Corlosquet, 2014, WebID 1.0, W3C Editor's Draft, W3C WebID Community Group. http://www.w3.org/2005/Incubator/webid/spec/drafts/ED-webid-20140305/identity.

[21] W. Slabbinck, Shape validation module, 2023. https://github.com/CommunitySolidServer/shape-validator-component/.

[22] Solid team, Solid Project, 2023. https://solidproject.org/.

[23] R. Taelman, J. Van Herwegen, M. Vander Sande and R. Verborgh, Components.js: Semantic dependency injection, *Semantic Web Journal* (2022). https://linkedsoftwaredependencies.github.io/Article-System-Components/.

[24] V. Tunru, Penny, 2023. https://gitlab.com/vincenttunru/penny/.

[25] J. Van Herwegen, Getting started with the Community Solid Server, 2023. https://github.com/CommunitySolidServer/tutorials/blob/main/getting-started.md.

[26] J. Van Herwegen, Customizing configurations, 2023. https://github.com/CommunitySolidServer/tutorials/blob/main/custom-configurations.md.

[27] J. Van Herwegen, Hello World component, 2023. https://github.com/CommunitySolidServer/hello-world-component/.

[28] J. Van Herwegen, Derived resources component for the Community Solid Server, 2024. https://github.com/SolidLabResearch/derived-resources-component/.

[29] J. Van Herwegen et al., Community Solid Server, 2023. https://github.com/CommunitySolidServer/CommunitySolidServer/.

[30] J. Van Herwegen and T. Dupont, Community Solid Server configuration generator, 2023. https://communitysolidserver.github.io/configuration-generator/.

[31] J. Van Herwegen, T. Dupont et al., Community Solid Server documentation, 2023. https://communitysolidserver.github.io/CommunitySolidServer/.

[32] J. Van Herwegen, R. Verborgh et al., Community Solid Server, 2023. https://github.com/CommunitySolidServer/Recipes.

[33] R. Verborgh, Philips Hue module, 2023. https://github.com/RubenVerborgh/solid-hue/.

[34] A. Vercruysse and W. Slabbinck, LDES Solid Server, 2023. https://github.com/TREEcg/LDES-Solid-Server/.

[35] J. Zucker, Data-Kitchen, 2023. https://github.com/solid/data-kitchen/.