# Representing the Virtual: Using AAS to Expose Digital Assets

Coen van Leeuwen[1], Cornelis Bouter[1], Rick Hindriks[1] and Robert Wilterdink[1]

[1]TNO ICT, Anna van Buerenplein 1, Postbus 96800 2509 JE Den Haag, Netherlands

**Abstract**

The Industry 4.0 Asset Administration Shell provides a standardized mechanism for collaboration between digital systems in the factory. Digital data within factories is typically stored in databases, we explore the requirements of providing an AAS as an interface to the data contained within the aforementioned databases. Based on these requirements, we describe and discuss a proof-of-concept implementation where an AAS is used to publish data stored in a relational database.

**Keywords**

Asset Administration Shell, Industry 4.0, Databases, SQL, Digital Twinning

## 1. Introduction

Representing virtual assets in a standardized way is the next step towards the Industry 4.0 digital factory. The Industry 4.0 Asset Administration Shell (AAS) is an industry standard for representing assets in factories, and allows the exposure of machine states in a standardized and organized manner. Its core use-case is the communication between assets within a factory, or to a management system, however the AAS also is very well suited for communication between parties. In this paper we will propose a different way of using the AAS, namely to represent digital or virtual assets instead of physical ones. The AAS specification explicitly gives a broad definition of an asset, including both physical and virtual assets [1]. However in practice the AAS has of yet always been applied for representing physical assets only. Using the AAS to represent virtual assets such as contacts, invoices, work orders or any other virtual entity, allows the industry to reason and communicate about these things. Moreover, such an approach allows the automation of processes using methods which are comparable to the existing methods for physical assets.

When the data is available as an AAS, that AAS can allow us to develop more meaningful mechanisms to automate systems within a factory. For instance, for a machine to start working we can use more standardized triggering mechanisms or signal an operator with a more informative message. As such, the AAS may serve as a driver for such processes in the factory as a

*Third International Workshop On Semantic Digital Twins (SeDiT 2022), co-located with the 19th European Semantic Web Conference (ESWC 2022), Hersonissos, Greece - 29 May 2022*

coen.vanleeuwen@tno.nl (C. van Leeuwen); cornelis.bouter@tno.nl (C. Bouter); rick.hindriks@tno.nl (R. Hindriks); robert.wilterdink@tno.nl (R. Wilterdink)

https://coenvl.nl/ (C. van Leeuwen)

0000-0003-4384-1443 (C. van Leeuwen); 0000-0002-5448-0543 (C. Bouter); 0000-0001-8798-7132 (R. Hindriks)



"motivation of work".

Representing virtual assets, however, is not a straightforward process. In the factory business process, it is day-to-day business that new virtual assets are added continuously. For instance, in a healthy production cycle, work orders are expected to be added, updated, and eventually removed. Although this may also happen for physical assets, for virtual ones this will happen on a much more frequent basis. Current AAS system infrastructures are insufficiently equipped to deal with the increased dynamicity of virtual AASs, yet alone for the expected increase in their sheer volumes.

## 2. Background

The Asset Administration Shell (AAS) is the proposed implementation of a Digital Twin by the German Plattform Industry 4.0. The normative documentation is available in three parts: 1) the data model [1], 2) the API definition [2], and 3) the communication language among AASs [3]. Additional non-normative material has been published on AAS composition [4, 5], structuring (sub)models [6], the various roles involved in AAS modelling [7] and examples [8].

Assigning semantics to AAS models has been recognised as a necessity to fully support interoperability [9, 10]. The *semanticID* has been defined to assign semantics to an AAS element, refering to an IRI for semantic web identification or IRDI eClass elements [1]. Two complementary RDF serialisations of the AAS have been realised to make "full use of the advantages of semantic technologies" [1]. The challenge nevertheless remains of providing models that refer to formal ontologies or international standards despite the recognition of the benefits semantics brings [11].

Some work exists on mapping established data specifications in AutomationML and OPC-UA to the AAS, since the AAS specification already to those languages [1]. For example, in [12] a mapping is constructed from the IEC 61131-3 PLC standard to a set of AAS submodels. The same authors also reflect on mapping between OPC-UA models and AAS submodels [13, 14]. [15] presents a translation from both MES data to AAS models and from ERP data to AAS models using an established AutomationML model for IEC 62264 [16]. In [17] AutomationML is also used as the intermediate data format. Transformation of one AAS model to another is covered in [18], who defined an AAS transformation language.

There is little research available on relational databases in an AAS context. The Data Administration Shell [19] adapts the AAS to a Digital Twin for datasets. It models only the metadata and preprocessing steps in the AAS models, but only contains a reference to the actual data set. The adequacy of the various types of SQL and NoSQL databases for the AAS is covered in [20]. Their literature study showed a lack of rigour in covering the employed data models. They also identified a gap in the implementation of database solutions.

## 3. Publishing databases using AAS

In order to expose virtual assets as an AAS, the AAS server instance needs to have access to the data underlying the virtual assets. In state-of-the-art factories several IT systems (e.g., ERP, PLM, MES) from different providers are responsible for managing and storing virtual assets.

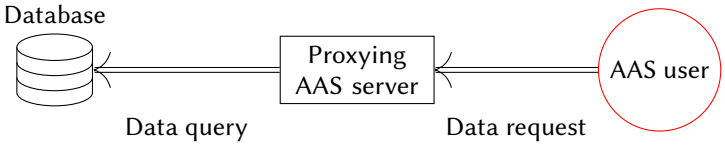


**Figure 1:** A schematic representation of a proxying AAS server, where the published data is fetched from the underlying database only on requests from the user.

Our intention is to enhance the inter-operability of these systems by creating a set of AASs that references data from all sources. The intention is not to replace the existing factory IT systems. Access methods to the underlying data can roughly be split in two: API access and database access. In this paper we explore database access, because in practice company IT administrators have unlimited database access. To transform data from an existing database into AAS form, the following aspects need to be taken into account:

### 3.1. Data authority

When designing factory IT systems, we typically strive to apply the principle of *separation of concerns*. By this principle, we need to decide which system components are responsible for which part(s) of the available data. When multiple components are responsible for the same subset of data, this requires them to coordinate, leading to additional complexity in those components. As such, the ideal case has only a single component responsible for each partition of the data; a single source of truth.

In the case of an AAS, we are often adding an additional component to an existing IT system. In order to facilitate collaboration between assets, existing data needed for this collaboration which exists within the IT system needs to be collected and published as one or more submodels on an AASs.

Given that the pre-existing IT system is and should remain the owner of the data, we need to carefully design the used data collection and publication mechanism(s). Notably, we need to ensure that we create as little additional copies of the data as possible, as this requires us to maintain and synchronize each copy of the data. Failing to do so would lead to inconsistencies between the system and AAS states.

A design pattern which prevents many of these problems is the *proxy* pattern, in which the AAS server retrieves the underlying data only and only when it needs to serve a request. A schematic example is shown in Figure 1. As a result, the underlying IT system remains the ultimate owner of the data, and data integrity and freshness are maintained.

### 3.2. Common data sources

The use of databases is already a common practice in any modern factory, but there are different implementations that will require different software to bind it to an AAS server. For most factories, it will be preferable to keep the database as it is, and transform the data using a "connector" component to connect the database to the AAS server instance. Different database implementations that are commonly used can be separated in relational databases versus NoSQL

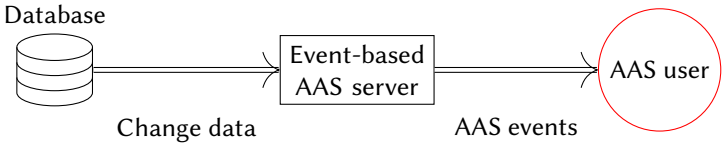


**Figure 2:** A schematic representation of the event data with database specific change data on the left side, and standardized AAS events on the right side.

databases. The difference between the two essentially means that in the first category data can be represented as tables with rows and columns, whereas in the second category data can be any format. Under the category of SQL databases there are for example: MySQL, PostgreSQL, MariaDB, Oracle Database or Microsoft SQL. Examples of NoSQL databases are: MongoDB, Cassandra, CouchDB or Neo4j. For a more in-depth comparison we refer the reader to [20].

### 3.3. Scaling

In the AAS servers that are currently operating, there are typically only a handful of AAS instances, and this number stays constant throughout the lifetime of the server. When representing virtual assets, the purpose of the assets is to trace their life-cycle: the creation of new ones, changing existing ones, and eventually removing assets that are no longer relevant.

For large factories, the expected number of assets that exists at any given time could be in the order to thousands to tens of thousands. This requires a scalable solution which can scale not only in storage size, but also in the processing capabilities of the compute nodes hosting the AAS assets. Iterating over all known instances in the database is probably not feasible while simultaneously keeping the AAS instances up-to-date with the underlying database. The AASs that are served need to be synchronized with the database at any point in time, where a delay greater than a couple of seconds is considered unacceptable. Instead, some kind of *push* mechanism is preferable, which brings us to the next requirement: reactivity.

### 3.4. Reactivity

The virtual AASs are expected to continuously change, both in number and in content. In order to keep the AASs synchronized with the underlying database, an event-based reactive system is essential; however, there is currently no standard for events. An event-driven mechanism makes sure that any changes in the underlying data source are actively pushed to the AAS server. Most databases support this type of events in the form of change data capture or change streams, hence this side of the data reactivity depends mostly on the backing data source, shown on the left side of Figure 2.

On the other side of the server, a set of change events that apply to the changes in the AASs needs to be published as well. This is the type of event on the right side of Figure 2 which is not yet supported in the AAS specification, but we suggest that such a standardization is added (see Section 5.1). With a standardized set of change events, the applications using the AAS can react to changes in order to start a new process, trigger a machine to perform an operation or to simply inform a user.

### 3.5. Read/write operations

Apart from getting data from a database in order to put data into a standardized form, the AAS/database integration can also be used to update the database. The AAS service specification allows for writing data to, for instance, change parameters. These updated parameters could be used to feed back into the database system. Although this is a feature which makes the integration stronger, we suggest *not* to implement this, but instead keep the AAS as a façade for the actual data. This not only limits the complexity of the implementation, but also makes sure the database will always remain the *single source of truth* in the case of any failures.

Database software is generally designed to ensure that all transactions maintain ACID properties: all operations must be atomic, consistent, isolated, and durable. Adding another interface to feed data into the database should not interfere with these statements, but the interaction with the database is already very reliable as it is, and it is not necessary to add another method. Most importantly, the way that people or machines interact with the database is already through the existing systems on the factory floor, adding another method to it would only make things unnecessarily complicated.

### 3.6. Finding assets

In the current AAS standard, there is no mechanism described filtering AASs on an existing AAS server based on some property; the only option is to list all available instances [2]. When dealing with a large number of assets, which is guaranteed to occur when using virtual assets, listing all available AASs is no longer a feasible option. Instead, a querying mechanism must be available to search for AASs, for example by means of searching for a part of the asset name, a keyword, a semantic id, or even a property.

## 4. Implementation example

As a proof-of-concept, we implemented an adapter for an existing AAS server implementation as shown in Figure 3, that takes data from a sample of a database as is used on a factory floor. We used a dataset containing work orders and the associated process steps of a week's worth of factory work. The database contained over $38.000$ work orders with a total of $131.000$ associated process steps.

### 4.1. Operation

Our adapter loads AASs with submodel templates, which contain qualifiers, as defined in [1], that indicate how to get data from a database as shown in Table 1. These qualifiers provide the necessary details to connect to the SQL database—a Microsoft SQL database in this example—and perform queries to feed the properties and the submodel elements of the template AAS. An alternative approach could be made by providing the connection detail in a separate file, but using qualifiers makes the AAS as self-contained as possible, especially when multiple database connections are required.

Using the template with these qualifiers, the adapter creates an AASProxy instance which represents the data in the form of an AAS, including all descriptors. The proxy is responsible

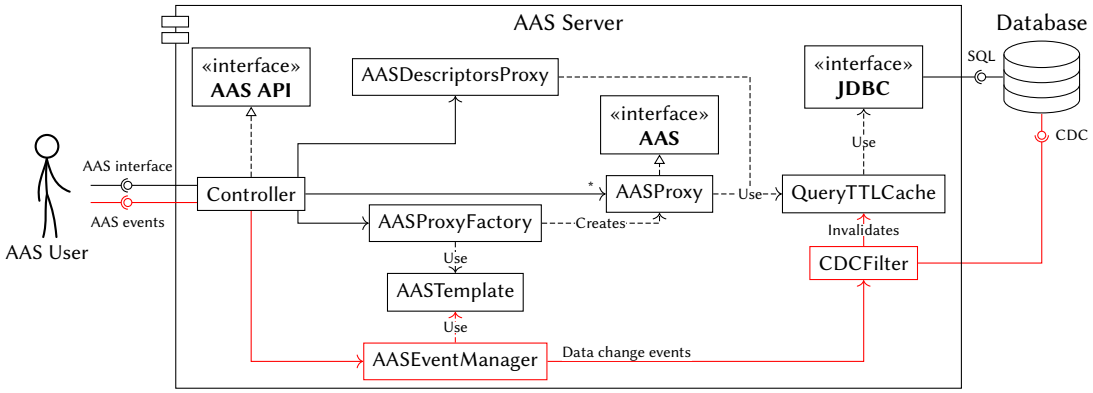


**Figure 3:** A graphical representation of the proposed AAS components to implement an AAS proxy for data in a database. The parts highlighted in red were not developed, but are needed in order to develop an event-based AAS interface.

**Table 1**
The different AAS Qualifiers used in the template to connect to the SQL backend.

| | |
|---|---|
| **SQL4AAS.Host** | The hostname or IP address of the SQL server. |
| **SQL4AAS.Port** | The port where the SQL database is listening. |
| **SQL4AAS.User** | The user name that is used for authorization. |
| **SQL4AAS.Password** | The password of the user to authorize with. |
| **SQL4AAS.Database** | The name of the database to use to on the server. |
| **SQL4AAS.ListQuery** | A SQL query that lists all different instances (AAS / submodel / submodel element) when executed. |
| **SQL4AAS.ContentQuery** | A SQL query that retrieves data for an instance that the list query returned. |
| **SQL4AAS.IDColumn** | The column which uniquely identifies the row; used to get a unique ID for the component. |
| **SQL4AAS.NameColumn** | The column which holds a human readable name for the row; when not used, the ID is used instead. |

for retrieving the data of the AAS from the database, based on the qualifiers. Whenever a user makes a query to the AAS server, the server's controller forwards this request to the current proxy, which translates this request into a set of SQL queries which are sent to the database. When the database responds to the aforementioned queries, the data is serialized into the AAS data format, and returned to the controller. Next, the controller responds to the user request, basing the response data on the internal data.

## 4.2. Limitations of implementation approach

By implementing a SQL controller into an existing platform, we managed to obtain a proof-of-concept, but there are some practical problems that make it infeasible for real-world application.

First and foremost, the large amount of changing data is a huge drain on the system resources. This is obviously very implementation specific, and could possibly be improved upon, but for

anything more than a few kilobytes per virtual AAS, but dealing with hundreds of thousands of AASs, the amount of memory needed to store the example data from our database quickly becomes in the order of gigabytes.

Even when that is handled properly, the amount of SQL queries that are required to retrieve the data also leads to a serious bottleneck. In our example every AAS, submodel and submodel element collection requires both a list query and a content query, which means that any asset quickly requires five to fifteen queries, meaning that we require half a million queries to retrieve the most recent state of all AASs. For any AAS structures that are more complex this number grows even faster. This problem may be mitigated by using lazy querying, by only listing the available objects, and only fetching their contents when a user retrieves the asset; but in this way the AAS server becomes just another database—which is not our goal—, except with output in standardized form. The above confirms our hypothesis that querying periodically is infeasible. As such, an event-driven or data-capture mechanism is required to connect with the data source.

On the side of the server implementation, the interface towards the AAS user is a complicating factor since there is currently no complete standard for disseminating information in an event-based manner. The current AAS specification describes a data model for events, but provides no mechanism for their real-time exchange. This means that with the current state of affairs, the user *must* periodically query each AAS in order to stay synchronized. For example, in order for a user to determine if an AAS has changed, and hence require action on the factory floor, the user must first list all the AASs that exist on the server. Then for all relevant assets, list all submodels, and for all relevant submodels list all submodel elements. This is a very tedious and time consuming process, and realistically impossible to use in a real-life scenario.

# 5. Discussion

## 5.1. Recommendations to improve the AAS standard

As we have seen, disregarding implementation specifics, there are issues with the AAS standard that make it infeasible to use as a *motivating of work* for other systems. In order to address this, we propose the following changes.

At the time of writing, the current AAS specification contains insufficient specification for Events, only describing their conceptual use and referring to underlying transport mechanisms to implement them. Therefore we propose to add to the standard a mechanism to publish events about changes in the assets. This mechanism should have at least the following properties:

- The mechanism should expose the events in a standardized format,
- the events inform users on about new, updated or removed objects,
- where objects can be any of the following: assets, submodels or submodel elements,
- a user can “subscribe” to all changes, but also to specific submodels or submodel elements.

Apart from the event mechanism, there should also still be a querying mechanism, that supports searching for AASs as well as for specific individual submodels. A simple “keyword”

querying mechanism would already be a very useful addition to the API. But a more elaborate querying mechanism can provide even more functionality by using the semantic information available in the AAS, allowing to search for submodels or keywords that match a certain semantic search criterion.

Moreover, clients may not always be interested in events from complete assets. Sometimes clients may even only be interested in a the value of single submodel element. To support such use-cases, the aas event API should allow subscribing to a individual submodels and submodel elements.

### 5.2. Event-based data interfaces

Above, we have described the requirements for event-based data exchange. Event-based data interfaces allow clients to selectively be notified of updates to data. Given our implementation scenario where we may have an AAS for each of the 38.000 work orders, an event-based system would allow us to only be notified which AASs exist at some point, and afterwards when the list of existing AASs changes. This requires less wasteful transfer of data which is unchanged, as well as require less queries to the database system underlying the AASs. On a more fine-grained perspective, an event-based interface allows clients to load a single AAS once, and then only be notified of updates of the subset of data that it is interested in, again preventing the wasteful exchange of unneeded data.

From a server perspective, an event-based AAS may also allow said server to provide virtual or placeholder assets and data. In a mechanism similar to lazy loading, only when a client issues a request for the placeholder data, the server will populate the actual AASs, submodels and submodel elements, and provide them to the user. Such a mechanism would alleviate the need for continuous inspection of the underlying database, as such inspection may be performed when the user makes a request.

### 5.3. Concluding remarks

In order to explore the "motivation of work" for the Asset Administration Shell, we have examined AAS-based interfaces to existing data sources. Based on our proof-of-concept implementation, it turns out that the currently available mechanisms in the AAS specification are lacking features which enable implementation in real-world scenarios. We have proposed extensions to said specification where we introduce the concept of events which allows for a more efficient and rapid interfaces to data and changes to that data.

## Acknowledgments

This paper is based on work funded from the European Union's Horizon 2020 research and innovation programme within the DIMOFAC and MAS4AI project under grant agreements No. 870092 and No. 957204.

## References


[1] Plattform Industrie 4.0, ZVEI, Details of the asset administration shell - part 1, 2019. Version 2.0.

[2] Plattform Industrie 4.0, ZVEI, Details of the asset administration shell - part 2, 2020. Version 1.0.

[3] Plattform Industrie 4.0, ZVEI, Details of the asset administration shell - part 3, [to be published].

[4] Plattform Industrie 4.0, ZVEI, Relationships between I4.0 components - Composite components and smart production, 2017.

[5] Plattform Industrie 4.0, AAS reference modelling, 2021.

[6] Plattform Industrie 4.0, ZVEI, Structure of the asset administration shell, 2016.

[7] Plattform Industrie 4.0, Functional view of the asset administration shell in an Industrie 4.0 system environment, 2021.

[8] Plattform Industrie 4.0, Verwaltungsschale in der Praxis, 2020.

[9] Plattform Industrie 4.0, Describing capabilities of Industrie 4.0 components, 2020.

[10] I. Grangel-González, P. Baptista, L. Halilaj, S. Lohmann, M.-E. Vidal, C. Mader, S. Auer, The Industry 4.0 standards landscape from a semantic integration perspective, in: International Conference on Emerging Technologies and Factory Automation, IEEE, 2017, pp. 1–8.

[11] S. Beden, Q. Cao, A. Beckmann, Semantic asset administration shells in industry 4.0: A survey, in: International Conference on Industrial Cyber-Physical Systems, 2021, pp. 31–38. doi:10.1109/ICPS49255.2021.9468266.

[12] S. Cavalieri, M. G. Salafia, Asset administration shell for PLC representation based on IEC 61131–3, IEEE Access 8 (2020) 142606–142621.

[13] S. Cavalieri, M. G. Salafia, Insights into mapping solutions based on OPC UA information model applied to the Industry 4.0 asset administration shell, Computers 9 (2020) 28.

[14] S. Cavalieri, S. Mulé, M. G. Salafia, OPC UA-based asset administration shell, in: Annual Conference of the IEEE Industrial Electronics Society, 2019, pp. 2982–2989.

[15] X. Ye, M. Yu, W. S. Song, S. H. Hong, An asset administration shell method for data exchange between manufacturing software applications, IEEE Access 9 (2021) 144171–144178. doi:10.1109/ACCESS.2021.3122175.

[16] B. Wally, L. Lang, R. Włodarski, R. Šindelár, C. Huemer, A. Mazak, M. Wimmer, Generating structured automationml models from IEC 62264 information, Proceedings of the AutomationML PlugFest (2019).

[17] A. Lüder, A.-K. Behnert, F. Rinker, S. Biffl, Generating Industry 4.0 asset administration shells with data from engineering data logistics, in: International Conference on Emerging Technologies and Factory Automation, IEEE, 2020, pp. 867–874. doi:10.1109/ETFA46521.2020.9212149.

[18] T. Miny, M. Thies, U. Epple, C. Diedrich, Model transformation for asset administration shells, in: Annual Conference of the IEEE Industrial Electronics Society, 2020, pp. 2207–2212. doi:10.1109/IECON43393.2020.9254649.

[19] A. Löcklin, H. Vietz, D. White, T. Ruppert, N. Jazdi, M. Weyrich, Data administration shell for data-science-driven development, in: Design Conference, 2021, pp. 115–120. doi:10.1016/j.procir.2021.05.019.

[20] V. F. de Oliveira, M. A. d. O. Pessoa, F. Junqueira, P. E. Miyagi, SQL and NoSQL databases in the context of Industry 4.0, Machines 10 (2022) 20. doi:10.20944/preprints202111.0019.v1.