Architectural Considerations for a Data Access Marketplace

Uwe Hohenstein, Sonja Zillner and Andreas Biesdorf
Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, D-81730 Munich, Germany

Keywords: Marketplace, Data Access, Access Control, Filtering, API Management.

Abstract: Data and data access are increasingly becoming a good to sell. This paper suggests a marketplace for data access applications where producers can offer data (access) and algorithms, while consumers can subscribe to both and use them. In particular, fine-grained controlled data access can be sold to several people with different Service Level Agreements (SLAs) and prices. A general architecture is proposed which is based upon the API management tool WSO2 to ease implementation and reduce effort. Indeed, API Management provides many features that are useful in this context, but also unveil some challenges. A deeper discussion explains the technical challenges and alternatives to combine API Management with user-specific filtering and control of SLAs.

1 INTRODUCTION

Data and related data-processing algorithms are becoming more and more a product or service to be sold. At the same time and with the advent of integrated data analytics, data and algorithms may form an ecosystem if a protected and controlled usage and exchange is possible.

However, public research on algorithms is typically carried out outside of a data provider's premises in – partially profit-oriented – research organizations. They are not allowed to directly access the data due to a lack of interest to share data for free. But without access to the data, promising algorithms and applications cannot be tested and their results cannot be disseminated. Vice versa, the largest benefit of research activities could be realized if the algorithms or their results are made available to data owners. Indeed, since algorithmic research is expensive and the result represents intellectual property of the research institution, sharing the algorithms and/or their results for free is not an intention.

Giving monetary incentives could help out of the dilemma. Access to data can be given to algorithm developers for a certain fee. In case of regulations or data protection laws, data can be made anonymous before granting access. The other way round, algorithms and/or the data they produce can be sold to professionals. However, an open and secure marketplace platform for trading, sharing, and exchanging data, data processing algorithms as well as data analytics results is a prerequisite. On the one hand, a marketplace makes it easier for providers to publish data and algorithms, disseminate the offerings, and finally attract potential consumers. On the other hand, it helps data consumers to discover and request access to data published in the marketplace.

One important requirement towards such a marketplace is security with a particular focus on authentication: Access should be granted for particular portions of data to known and privileged consumers only. Moreover, a controllable consumer-specific filtering of data needs to be ensured, i.e., different consumers should obtain access to different portions of the data.

Partial solutions have been discussed in the domain of databases to address the problem of consumer-specific access to data sources: There are several approaches for column-level access control (i.e., to omit or mask out specific columns) and row-level access control (i.e., filter out rows) discussed in research (cf. Section 6). However, advanced concepts are required to be also able to limit data quality, the result size, or to throttle too many accesses depending on a selected Service Level Agreement (SLA).

While various approaches have been developed to restrict visibility of data by filtering, little attention has been paid to monetization of data. Marketplaces can help but appropriate solutions are missing to integrate advanced database filtering,

Hohenstein, U., Zillner, S. and Biesdorf, A.
Architectural Considerations for a Data Access Marketplace.
DOI: 10.5220/0006855303220333
In *Proceedings of the 7th International Conference on Data Science, Technology and Applications (DATA 2018)*, pages 322-333
ISBN: 978-989-758-318-6

especially due to quite a dynamic set of users that requests a high degree of automation.

In this paper, we present a marketplace architecture that provides high flexibility by combining API Management and flexible filtering. The use of API Management as off-the-shelve is reasonable because it already implements a lot of functionality for marketplaces and thus let the architecture focus on core functional aspects. In fact, many features such as OAuth-based authentication, scalability for a high amount of users, auditing, monitoring, and billing are available or can be integrated at least.

The proposed marketplace allows data providers to generate, advertise, and sell access to APIs (especially for data access), while consumers are enabled to purchase access to APIs at the marketplace. There are dedicated provider and consumer web interfaces. Figure 2 and 3 show an example that we developed and tested in a funded project named "Klinische Datenintelligenz" (Sonntag et al., 2015).

In the following, we propose in Section 2 a customizable Data Delivery Service, i.e., a generic REST service for database access to be offered as an API in a marketplace. This service provides consumer-specific query results.

API Management in general and the tool WSO2 in particular are introduced in Section 3. We explain why API Management reduces effort to implement a marketplace by already offering fundamental concepts.

While API-Management usage appears to be highly appealing, there are still a couple of challenges to tackle. Section 4 details the problem space and discusses major challenges of combining API Management and user-specific filtering, e.g., how to maintain a highly automated environment, especially for an unknown and potentially large consumer community.

Solutions for tackling those problems, alternatives, and an architectural proposal are discussed in Section 5.

Section 6 presents some related work in order to underline the novelty of our approach. Finally, conclusions are drawn in Section 7.

2 DATA DELIVERY SERVICE

Before diving into the details of the marketplace architecture, we first propose a Data Delivery Service (DDS) for relational database systems. The DDS offers a generic REST service that allows for executing arbitrary SQL queries passed as a string. The DDS is REST-based, but in fact, HTML/JavaScript based user interfaces can be put on top of the REST API. A DDS request has the following form:

```
POST Host/Type/Database?Options
```

The SQL query to be executed is placed in the request body. The result of such a POST request can be obtained in XML or JSON, controllable by the "Accept" header of the request. The *Type* of the database, e.g., PostgreSQL, and a concrete *Database* are part of the URL. For instance, the query

```
SELECT encounter_num, patient_num,
       concept_cd, provider_id,
       start_date,quantity_num,
       units_cd, observation_blob
FROM  i2b2myproject.Observation_fact
WHERE observation_blob <> ''
```

to a medical i2b2 database returns a JSON result like the following:

```
{
  "columns": [  // columns of the result
    "encounter_num",
    "patient_num",
    "concept_cd",
    "provider_id",
    "start_date",
    "quantity_num",
    "units_cd"
  ],
  "types": [ // data types for those columns
     "int4",  //  (in the same order)
     "int4",
     "varchar",
     "varchar",
     "timestamp",
     "numeric",
     "varchar"
   ],
   "elapsedMs": 36, // server-side execution
                    // time in ms
   "size": 2,  // number of records in result
   "content": [ // records
     {
       "no": 0, // 1st record
       "values": [
           "1791",
           "1",
           ... ]
     },
     {
       "no": 1, // 2nd record
       "values": [ ... ]
     },
     ... // further records
   ]
}
```

The JSON result possesses a generic structure and includes a meta-data description (if requested by the query parameter `metainfo=include` in *`Options`*) which describes the column names and data types. Hence, the result becomes interpretable and machine readable.

The DDS provides further features such as pagination with query parameters `top=` and `limit=`, streaming, and data compression.

Similar approaches for other data sources are also possible, e.g., ontologies like DBpedia (http://wiki.dbpedia.org/) with other APIs like SparQL or general computing services.

3 API MANAGEMENT

According to Wikipedia (https://en.wikipedia.org /wiki/API_management), API Management is "the process of creating and publishing Web APIs, enforcing their usage policies, controlling access, nurturing the subscriber community, collecting and analyzing usage statistics, and reporting on performance." The technical platform is composed of tools, services and infrastructure developed to support two types of users: producers and consumers.

Figure 1 illustrates the basic principle of recent API Management (API-M) tools such as WSO2. At the left side, providers can provision existing backend APIs to API-M at any time. The API is defined by its URI and might be of REST or SOAP style. Indeed, the DDS of Section 2 is one potential service to be offered. All offerings obtain a new URI by the API-M, which is mapped to the original backend service URI. Thereby, each provider obtains a base URI like `https://server:8243/t/a_provider` which is extended by a service-specific part like `/dds` for the published API-M service. API-M acts as a proxy and protects the backend service.

A user can browse through all the API offerings of all the providers (cf. left side of Figure 2). If interested, a user can sign-up to the API-M with a name and password. Having then logged in, a user can subscribe to a service of his interest, thus becoming a consumer. For using a subscribed API, a consumer has to request an OAuth security token for access issued by an Identity Provider.

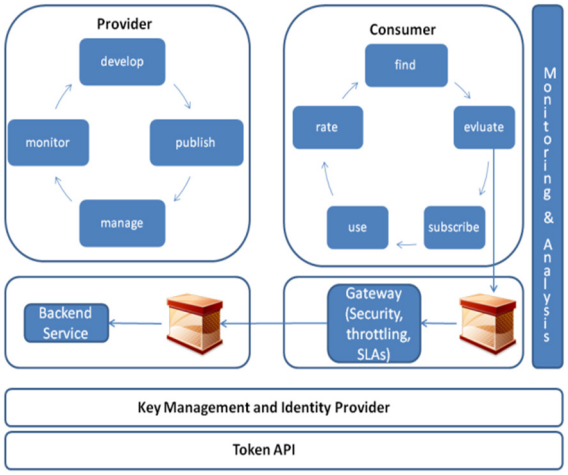

Figure 1: API Management.

API-M mainly handles the producers and consumers and their interplay. API-M is a good basis for a marketplace for data-access applications, especially a DDS. One important feature is the addition of an OAuth-based access control to even unprotected services.

The approach we present relies on the WSO2 API Management: WSO2 is an open source API Management, which is also offered as a payable Cloud offering (the latter having special features already built-in). Figure 2 and 3 present some screenshots for a marketplace that we have set up for a funded project (Sonntag et al., 2015) in the medical area. Figure 2 illustrates the consumer's web interface `https://server:8243/store` with all the subscribed services (at the bottom), the possibility to sign-up to the API-M as a new user, to log in (both not visible in Figure 2), and to subscribe to a service as a consumer. Figure 3 illustrates the provider's web interface `https://server:8243 /publisher`. The interface shows all the offered APIs with icons. Clicking on an icon, details about the mapping to the backend service occur and the lifecycle management can be entered, e.g., stopping a service at all or for particular customers only. Using the functionality at the left side, new APIs can be published by "Add".

As already mentioned, a consumer has to request an OAuth security token for invoking an API. Such a token can be requested by pushing the "Regenerate" button (cf., "Access Token" box in Figure 2). The token must be passed with every REST request in the "Authorization" header. API-M then authenticates the security token and allows invocation of the API. The token expires after a configurable time, but can be renewed at any time.

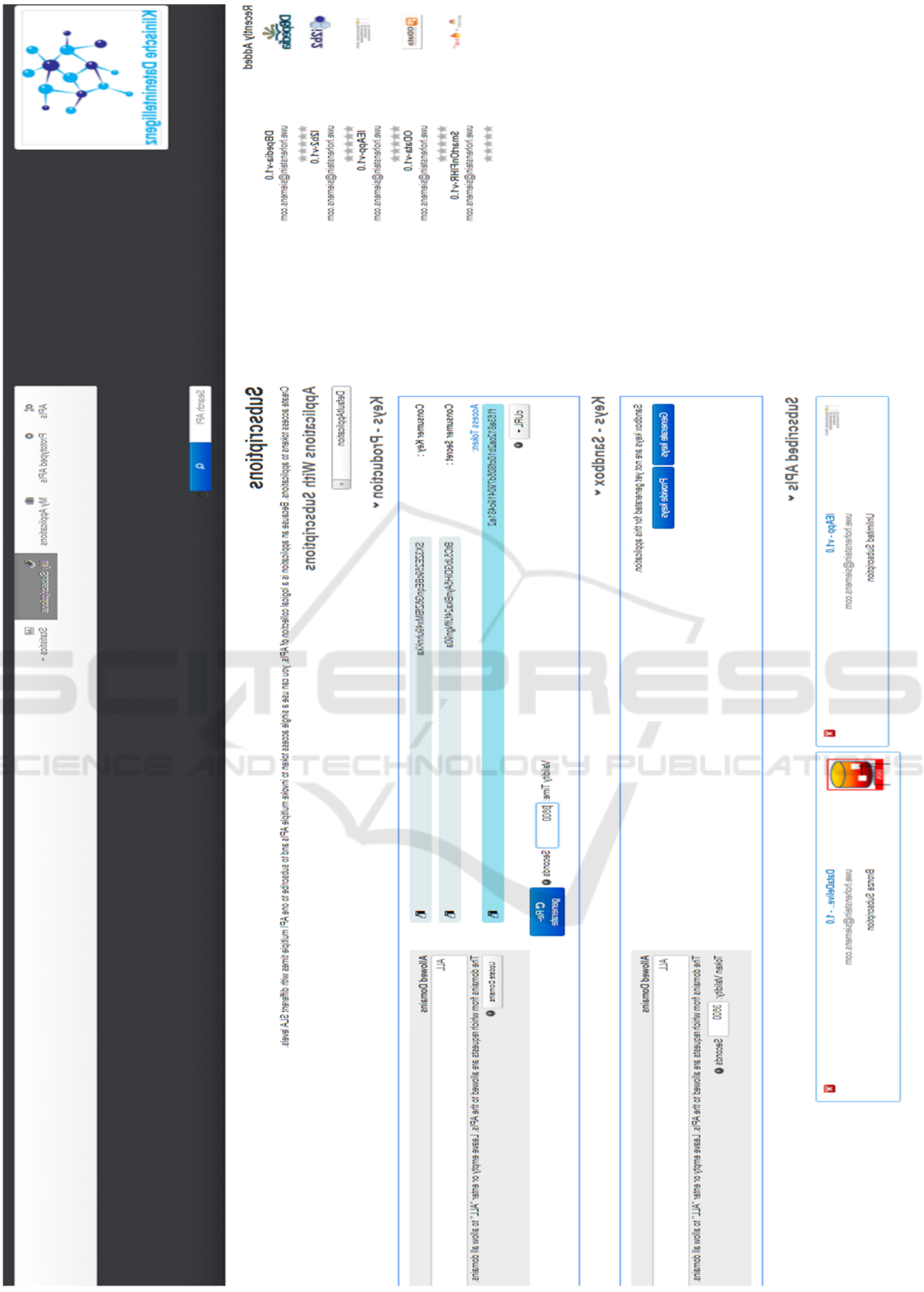

Figure 2: Consumer's web interface.

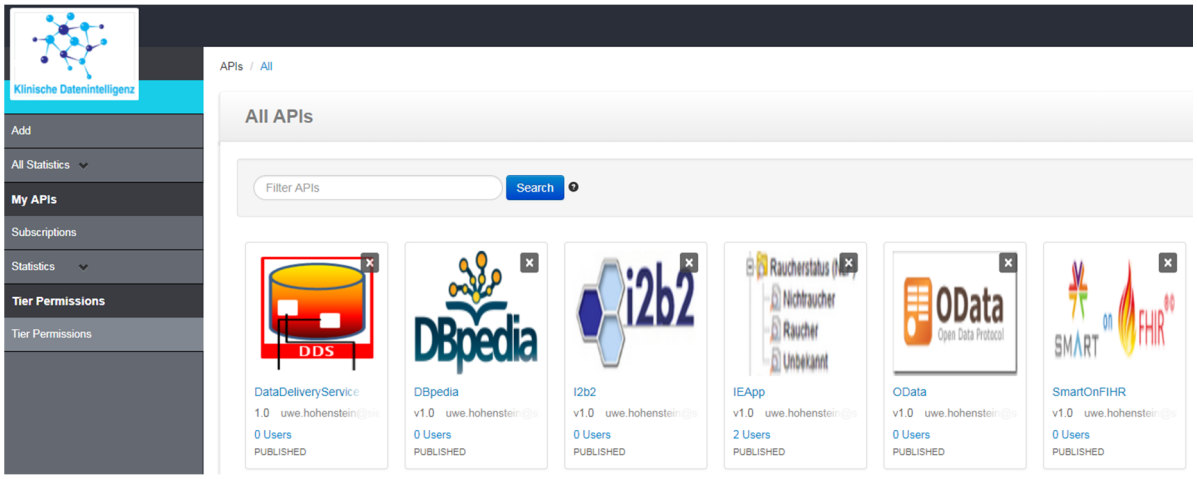

Figure 3: Provider's web interface.

We have chosen WSO2 as it provides several useful features:

- Several dashboards are available in addition to the providers' and consumers' views. For instance, an administration view allows providers to manage consumers. Moreover, the platform administrator can handle tenants/providers.
- The API provider (owner) of a service is able to approve every user sign-up and/or subscription in an administration view; corresponding pre-defined business processes can be used, and new ones can be specified. These are triggered if activated. A self-approval without provider interaction can also be configured.
- An application's publication can specify information about usage conditions and pricing: To this end, APIs can be offered in several tiers such as Gold, Silver, or Bronze with certain SLAs and prices associated. By subscribing, a consumer accepts the usage conditions (e.g., about payment). Moreover, a throttling of access according to tiers can be configured.
- A corresponding billing component can be integrated. Billing is based upon consumption and can use a monitoring component that tracks all consumers' activities.
- There are powerful concepts to map a frontend URL as published in the API-M to the backend URL of the service, for instance, to change the URL by switching query parameters and path elements.
- Similarly, the authentication of the WSO2 service at the backend system can be configured.
- A scope concept enables a consumer to further restrict accesses.
- A versioning of APIs is supported.

4 PROBLEM SPACE

Using API management (API-M) such as WSO2 for implementing a marketplace is helpful and useful because a lot of marketplace functionality like producer and consumer web interfaces is available. But API-M also introduces some challenges if we want to offer a Data Delivery Service (DDS) as a service.

Suppose a provider has exposed a DDS for his data to the API Management as a REST service. Any consumer interested in using the DDS can subscribe to the DDS service in the API-M. After a successful approval by the provider, a consumer can ask the API Management for issuing an OAuth security token; the OAuth token is consumer and DDS-specific. The security token has a specific expiration time and must be passed to the API-M as an integral part of any invocation. API Management checks the validity of the security token and – if valid – invokes the Data Delivery Service.

Figure 1 shows that neither the DDS nor the database is directly accessed by consumers: API Management acts as a proxy in front and protects the DDS. API-M receives requests and forces consumers to authenticate with a valid OAuth token in order to let API-M forward the call to the DDS.

When a consumer subscribes to an API, a tier (which has been described by the provider as part of his offering) can be selected: In fact, the tier can be

used to establish some predefined SLAs (especially throttling) and to determine pricing.

This is all functionality already offered by API Management. However, one important point is missing: We have to take care of consumer-specific filtering in the DDS: Consumers should see specific data instead of the whole data set. Examples in case of relational database systems are row/column level filtering, reducing the size of a result set, or other SLA attributes that affect the quality of returned data. To achieve filtering, three major problems have to be tackled.

(1) Information about the consumer is required for filtering. Any kind of filtering should take place depending on the consumer who is invoking the request. The problem is how to get the consumer's identity from the API-M, especially as the consumer uses a cryptical OAuth token for authentication at the API-M. Indeed, the DDS at the backend must know the consumer. Moreover, the selected tier can also be used to control filtering (row/column level, limit on result sets, quality, de-personalization, aggregation, read/write permissions etc.).

(2) Next, suppose the consumer is known by the DDS: How to use the consumer information to reduce results in a flexible manner? How to perform filtering without large manual, administrative effort? The software of the DDS as a backend service should not be modified for every new consumer.

(3) Finally, API-M and DDS have to co-operate. In particular, there is an unknown consumer base, with potentially many unexpected consumers. Each provider wants to give consumer-specific access to the database, but does not know about potential consumers of the marketplace. Moreover, each subscribed user requires certain actions to be performed in the database such as creating some database access roles.

5 OVERALL APPROACH

The following discussion is based upon the WSO2 API Management, but other API-Ms have similar concepts.

5.1 Passing Authorization Tokens

As already mentioned, every invocation of an API via the API-M requires a security token. Concerning Point (1) in Subsection 4, there are two major problems to solve: How to pass the security token to the DDS and how to interpret the quite cryptic token like `6db65cdf3231be61d9152485eef4633b` in the DDS.

We investigated that WSO2 API-M can be configured to pass the security token (coming from the consumer) to the DDS backend service. There are mainly two steps to perform within the provisioning of an API. First, a WSO2 predefined mediation configuration "preserve_accept_header" has to be applied to In/Out Flow requests. Using this option, the request is forwarded from WSO2 to DDS with an `x-jwt-assertion` header that contains a so-called Java Web Token (JWT). At a first glance, the JWT with its 1445 bytes looks even more cryptic, but has the advantage that it can be parsed in a second step with a JWTParser like http://jwt.io. Even more important, the parsed JWT yields the information the DDS requires, for example:

```
{
  "iss": "wso2.org/products/am",
  "exp": 1467298671690,
  "http://wso2.org/claims
                /applicationid":
"1",
  "http://wso2.org/claims
     /applicationtier": "Unlimited",
  "http://wso2.org/claims
          /apicontext": "/dds/v0.1",
  "http://wso2.org/claims/version":
          "v0.1",
  "http://wso2.org/claims/tier":
          "Silver",
  "http://wso2.org/claims/keytype":
          "PRODUCTION",
  "http://wso2.org/claims/usertype":
          "APPLICATION",
  "http://wso2.org/claims/enduser":
          "a_user@a_company.com",
  "http://wso2.org/claims
            /enduserTenantId": "-1234"
}
```

Figure 4: Decrypted token.

That is, the DDS is finally able to obtain and interpret the JWT and to extract information about `tier` or `enduser` from the token.

5.2 Options for Implementing Filtering

Having extracted and parsed the OAuth token, the consumer information becomes available to the DDS. Concerning problem (2) and (3) in Subsection 4, there are two major alternatives.

A) The extracted consumer can be used to connect to the database and to perform the filtering in the database by using the database features that

are discussed in Section 6. This is an easy option, but has the drawback that for every new consumer (which is just known after subscription), a corresponding database user has to be created for database logon, and further GRANTs and database-specific actions are required. This must be integrated into the subscription approval business process. Hence, there is some administrative effort to keep API-M and database users in sync. As a consequence, self-approval of consumers is complicated. Moreover, a user and password management is required to secure the database. The security token might be a good candidate for the password. However, the token has a short expiration time in the area of a few minutes; the user password that the consumer has chosen during sign-up for the API-M is invisible. Another disadvantage is the huge amount of database accounts. This leads to many parallel open database connections – one for each consumer – since connection pools usually do not work with a user-spanning pooling over database accounts. This has a major impact on the performance and raises high resource consumption in the database.

B) As an alternative, it is possible to use one single connect string to connect to the database. This requires less administration effort in the database and also solves the connection pooling issue. Unfortunately, database filtering features can no longer be used since the real consumer is now unknown in the database due to the shared database account. Hence, filtering must be performed in the DDS with some implementation effort as a consequence. As a gain, this offers most flexibility for filtering, especially since further information such as the chosen tier can be taken into account. There are two major options:

B.1) The Data Delivery Service can perform filtering by query rewriting and sending consumer-specific, modified queries to the database. There are many approaches in the literature, for example (Barker, 2008), (Rizvi et al., 2004) and (Wang et al., 2007). That is, the filtering logic becomes part of the DDS, and the DDS has to be aware of the corresponding policies. The DDS must know which user is allowed to see what columns and rows. And query rewriting at runtime is a must. Moreover, some technical issues like SQL injection have to be handled carefully.

B.2) Alternatively, the DDS can submit the original query and perform the filtering on the received result sets. Again, the DDS has to know the filtering policies. Performing complex filtering results is challenging leading to some complex analysis, since the filter conditions must be interpreted; the columns are not directly obvious and must be derived from the submitted SQL query. Moreover, there are performance issues since such a client-side vertical filtering transports larger query result sets from the database to perform filtering.

5.3 Database Connect

We suggest a more generic approach combining the alternatives to achieve best benefit. In a nutshell, we proceed as follows:

a) In order to solve the issue with several database accounts, the DDS connects to the database with a *shared* account in the sense of alternative B), without giving any further privileges to this account except for the permission to connect.
b) However, the user (consumer) is passed as a hook-on to the connection. This allows us to set up specific user access privileges in the database.
c) The consumer information is used to control the result for a specific consumer by one *single* database view for all the users. This eases the administrative effort. Sections 5.4 will present the details.
d) Further filtering can occur in the DDS to achieve powerfulness, e.g., considering the tier and/or scope.

Indeed, we found for a) and b) some quite hidden, product-specific mechanisms that are available in some database systems and enable passing consumer information to the database. For example, in PostgreSQL it is possible to create an account to connect to the database. The account has no further access to tables and views beyond the allowance to login:

```
CREATE ROLE loginOnly NOINHERIT
      LOGIN PASSWORD 'Pw4LoginOnly';
```

The DDS uses this `loginOnly` account. Another account `a_user`, created without the `LOGIN PASSWORD` option, is unable to connect:

```
CREATE ROLE a_user;
```

However, if `a_user` is added to the `loginOnly` group by `GRANT a_user TO loginOnly`, DDS can login with `loginOnly` and issue the statement `SET ROLE a_user`. This lets the user privileges for `a_user` become effective for every successive query.

Teradata has a so-called Query Band mechanism that behaves similarly from a functional point of view. SQL Server offers an `EXECUTE AS` statement to switch the user after having connected.

Letting the DDS authenticate in the database with a (shared) predefined account avoids a user management and corresponding administrative effort as well as negative performance impact.

5.4 Column-and Row-Level Filtering

The goal to achieve is a flexible filtering approach with little administrative effort. We present a hand-made solution to become product-independent. However, note that the approach also allows for integrating with database features or the research approaches discussed in Section 6.

Our approach consists of *one* common view for each base table handling the filtering in a consumer-specific manner. Accessing the common view, consumer information is implicitly used to restrict results. The view performs column- and row-level filtering based upon consumer and tier in a *generic* manner.

The filtering principle uses additional tables similar to (Barker, 2008). A configuration table `Privileges` (cf. Table 1) controls the visibility for users. For example, if `Tab`*`X`*`Column1` contains a value `false` for a particular consumer, then `Column1` of table `Tab`*`X`* should not be visible for that user.

Consequently, the configuration of user privileges – which columns and which rows in tables should be visible – is done external to the DDS.

Table 1: Privileges.

Privileges	*Consumer*	*TabXColumn1*	*TabXColumn2*	…
	User1	true	false	
	User2	false	true	
	User3	true	true	

This table is used in a generic view `Tab`*`X`*`_Filter` to be created for each table `Tab`*`X`*:

```
CREATE VIEW TabX_Filter
SELECT CASE WHEN p.TabXColumn1
            THEN t.Column1
            ELSE NULL END AS Column1,
       CASE WHEN p.TabXColumn2
            THEN t.Column2
            ELSE NULL END as Column2,
       ...
FROM TabX t
LEFT OUTER JOIN Privileges p
        ON p.Consumer = current_user
AND  <Condition>
```

Figure 5: Filtering view.

Certainly, each user is withdrawn access to the base tables `Tab`*`X`*; only access to the `Tab`*`X`*`_Filter` views is granted.

`CASE` expressions nullify or mask out columns for dedicated users according to what is defined in table `Privileges`. Hence, the behaviour is similar to the nullification semantics of (LeFevre et al., 2004).

The views need the current user (as being hooked to the connect). Database products usually provide corresponding functions, e.g., there is the `current_user` function in PostgreSQL. Oracle offers a so-called system context that is accessible in a similar manner.

Row-level filtering, i.e., `<Condition>` in the view definition of Figure 5, is simple if the consumer is part of table Tab*X*, e.g., in a column `User`. This is in fact how row level security works in commercial database products. Then, the `<Condition>` is quite generic: "`User= current_user`" by applying the `current_user` function in PostgreSQL.

However, it seems to be rather unrealistic that the subscribed consumer already occurs in the column data. It would certainly be more flexible if a subquery could determine the visible records for a consumer. As an important requirement to be taken into account, the approach must avoid a re-compilation of the DDS for any new customer. Moreover, the communication between frontend API-M and DDS or database should at least be reduced because business processes for service subscription have to be implemented, the possibilities of which are limited.

One approach is to implement the functionality in the database by computing the keys of visible records for a table `Tab`*`X`* by a table-valued function `RowLevelFilter4Tab`*`X`*`(user VARCHAR)`:

```
CASE User
  WHEN "user1"
   THEN SELECT t.key FROM TabX t ...
  WHEN "user2"
   THEN SELECT t.key FROM TabX t ...
```

Hence, row-level filtering can rely on any condition, on any columns or data. Even more, the view `Tab`*`X`*`_Filter` (cf. Figure 5) remains stable by replacing `<Condition>` with:

```
LEFT OUTER JOIN
 RowLevelFilter4TabX(current_user) c
  ON c.key = t.key
```

This principle offers complete freedom for a consumer-specific row-level filtering. However, there is a small disadvantage: For each new consumer, the function must be extended in order to add a consumer-specific subquery in the `CASE` clause. Fortunately, this can be done in the database at runtime without any impact on the DDS and its implementation, and without downtime.

An alternative is to keep the condition in the `Privileges` table in a textual form and to rely on dynamic SQL to compose an overall query.

5.5 Tiers

As already mentioned, the provider of the DDS is able to offer a service in different tiers. There are predefined tiers such as Gold, Silver, and Bronze, but new tiers like Platinum can be defined, too. With each tier certain SLAs and the price scheme can be specified as part of the offering in order to become visible for potential consumers.

A consumer can subscribe to an API for a specific tier, thus accepting the associated prices and SLAs. S/he can also subscribe to several tiers. The generation of a security token is then done for each particular tier. During invocation, the tier is part of the security token (cf. Figure 4). Hence, the DDS can use it, e.g., by adding `TOP(n)` by query rewriting in order to limit the result size according to the tier.

Furthermore, throttling, i.e., allowing only a limited number of accesses per time unit, can be configured for each tier in WSO2 without any explicit implementation.

Using the tier to control filtering is also possible. A possible solution is to concatenate user and tier (both can be extracted from the token) to a single name with some separation symbol in between. This name is then passed to the database instead of the consumer as before. That is, a role for this name has to be created in the database, and both parts of the name have to be extracted from the role in the view.

5.6 Administration

Some administrative effort is required for the presented approach. At first, a common database connect user is required for the DDS, e.g., in PostgreSQL:

```
CREATE ROLE loginOnly NOINHERIT ...;
```

Furthermore, the `Privileges` table and the `TabX_Filter` views must be created. These activities occur only once.

Moreover, several statements have to be executed for every new consumer `<user>`:

- `CREATE ROLE <user>;`
- `GRANT <user> TO loginOnly;`
- `GRANT SELECT ON TabX_Filter TO <user>;`

Next, a record in the `Privileges` table specifies column and/or row access. New consumers require additional records in the table, otherwise default settings apply to them. And finally, the function `RowLevelFilter4TabX` has to be adjusted.

In total, the consumer-specific administrative operations are minimal.

In principle, the above consumer-specific activities have to be established as part of the subscription workflow of the API-M. DDS can offer a service to execute those tasks in the database. This service can then be used by the workflow process.

In case a specific subscription workflow cannot be defined in the API-M tool, we can let DDS keep a table `AllConsumers` of consumers who have already accessed the DDS successfully. If a new consumer signs in, i.e., not occurring in the `AllConsumers`, the consumer will be added and the setup is performed. This principle can also be applied in general to avoid a communication between API-M and database.

5.7 Authentication

So far, the API Management provides a REST API for the DDS and takes care of authentication by OAuth tokens. The REST API can be invoked by any type of client – an application or a graphical user interface – implemented in any language. Every invocation requires a security token that can be obtained by the consumer web interface of the API-M (cf. Figure 2). The security token can be used until it expires.

Requesting the token by the consumer's web interface is a manual interactive action. Fortunately, further support is available by WSO2: Applications can acquire the token programmatically by invoking another REST API of WSO2 and passing user and password as provided during sign-up.

Graphical user interfaces (GUIs) in HTML and JavaScript benefit from an advanced OAuth support. A GUI can be registered for the DDS service in such a way that whenever the REST API is invoked from the GUI, WSO2 is implicitly contacted. A login form pops up asking the consumer to authenticate with user and password. Furthermore, the user has to

confirm that the GUI is allowed to act on behalf of the DDS. Hence, there is a tight integration of authentication.

5.8 Implementation of DDS

The implementation of the DDS is done in a generic manner by an abstract class that concentrates the common functionality, while having several DBMS-specific concrete classes to focus on database concepts such as passing and using the customer information.

6 RELATED WORK

Various tools and approaches have been developed in the database area to restrict visibility of data by filtering whereas only little attention has been paid to selling and restricting data access at marketplaces.

proDataMarket is one rare architectural approach for a marketplace (Roman et al. 2017). The approach focuses on monetizing real estate and related geospatial linked and contextual data. The proposed architecture offers a provider and consumer view similar to ours, however, does not rely on API Management and does not tackle the major problems of user-specific filtering and handling SLAs.

In contrast, a lot of work is available on access control, i.e., to limit activities of legitimate users. The need for flexible access control policies has been already recognized for decades. (Bertino, Jajodia, and Samarati, 1999) presented a well-defined authorization model that permits a range of discretionary access policies for relational databases.

This and other research influenced database vendors as they also recognized basic standard mechanisms (mainly views, stored procedures, and application-based security) as inappropriate (Rjaibi, 2012). Standard mechanisms have several drawbacks such as the need for a further view for each additional policy or user. Nowadays, relational database systems have introduced advanced features. The concepts allow for mainly row-level and column-level access control and are quite similar although being named differently in the products. For example, (Rjaibi, 2012) describes the IBM DB2 features for row permissions and column masks definitions. The latter allows for a customizable obfuscation of data by patterns for XXX-ing parts of data, e.g., the last part of phone numbers or accounts etc. (Oracle, 2017) introduced the Virtual Private Database technology, while (SQL Server, 2016) has concepts named Dynamic Data Masking, Column Level Permissions, or Row Level Security. The policies usually rely on functions to be defined for each table. Moreover, they rely on the fact that users obtain individual connect accounts. In contrast to our work, no further SLAs can be integrated, thus achieving less flexibility.

(Pereira, Regateiro and Aguiar, 2014) distinguished three general architectural solutions:

a) Centralized architectures such as using views and parameterized views (Roichman and Gudes, 2007), query rewriting techniques (Barker, 2008) (Rizvi et al., 2004) (Wang et al., 2007), and extensions to SQL (Chlipala and Impredicative, 2010) (Chaudhuri, Dutta, and Sudarshan, 2007);
b) distributed architectures (Caires et al., 2011);
c) and mixed architectures (Corcoran, Swamy, and Hicks, 2009) (XACML, 2012).

The proposal of (Pereira, Regateiro and Aguiar, 2014) belongs to category (c) and extends role-based access control to supervise direct and indirect access modes. An indirect mode means that SQL queries are executed, the result set is modified (for instance in JDBC, Hibernate, or LINQ), and changes then committed to the database.

The work of (Rizvi et al., 2004) is somehow special. Their "Truman" mode behaves similar to row/column level security with a query rewriting technique. However, they stress on the disadvantage of such an approach: possible misinterpretations of query answers might arise as a consequence of suppressed records due to row-level filtering. A "Non-Truman" mode tackles this point: Based upon authorization views for filtering, a user query is said to be valid if the query can be answered by using the authorization views only. If a query passes this validation test, the query is executed against the table without any modification. Otherwise, the query is rejected.

(LeFevre et al., 2004) discussed a technique to control the disclosing data process and thereby focus on Hippocratic databases. LeFevre proposes a high-level specification language for representing policy requirements. A policy is based upon the premise that the subject has control over who is allowed to see its protected data and for what purpose. Thus, operations are associated with a purpose and a recipient. Policies can be defined in P3P and EPAL and are translated into SQL by a query rewriting technique. Each purpose-recipient pair is represented by a view which replaces prohibited cells values at the table level with null, and removes protected rows from the query result according to the purpose-recipient constraints.

(Barker, 2008) proposed a formally well-defined, dynamic fine-grained meta-level access control. Barker makes use of a high-level specification language, a tuple calculus, for representing policies. This specification is translated into SQL by means of query rewriting to let the policy become effective. By categorizing users to categories according to a trust level, job level etc., the potential problem of view proliferation is much more manageable than for a user-based view (as for example in (Rizvi et al., 2004)). Hierarchical and negative authorizations are also handled by allowing for policy overriding and withdrawals.

None of these proposals handle the marketplace aspects and the resulting challenges that we address. For example, they demand each user to individually connect to the database. i.e., a shared connect is not possible. Moreover, the approaches are not able to handle further SLAs beyond row/column level security.

Several approaches exist that generate application code to assure access control. All these approaches operate at compile time and thus are not applicable to our work.

For example, the Ur/Web tool developed by (Chlipala and Impredicative, 2010) enables programmers to write statically checkable access control policies. A new 'known' predicate is proposed for SQL that returns what secrets are already known by the user. Based upon a set of policies, programs are inferred in such a way that query results respect the policies. Validation of policies occurs at compile time, thus requiring programmers to know database schemas and security policies while writing source code.

A similar approach (Abramov et al., 2012) presented another validation process that takes place at compile time. A complete framework allows security aspects to be defined early in the software development process. Based upon a model, access control policies are derived and applied.

The approach of (Zarnett, Tripunitara, and Lam, 2010) can be applied to control the access to methods of remote objects via Java RMI. Remote objects and methods can be enriched with Java annotations that specify access control policies. Accordingly, RMI Proxy Objects are generated in such a way that policies are satisfied. Annotations are also used by (Fischer et al., 2009) to assign a more fine-grained access control to methods.

The SELINKS programming language (Corcoran, Swamy, and Hicks, 2009) focuses on building secure 3-tier web applications. Programmers write programs in a LINQ-like language called LINKS whereupon a compiler creates the byte-code for each tier according to the security policies. The policy functions are called by applications to mediate the access to data that is labelled as sensitive. The generation process guarantees that sensitive data is never accessed directly by bypassing policy enforcement. Policy functions run in a remote server and check at run-time what type of actions users are granted to perform.

(Komlenovic, Tripunitara and Zitouni, 2011) presented a distributed enforcement of role-based access control policies. Other work by (Jayaraman et al., 2013) discussed a new technique and a tool to find errors in role-based access policies.

A survey about further research can be found in (Fuchs, Pernul, and Sandhu, 2011).

7 CONCLUSIONS

This paper presents an approach to offer data, more precisely data access, at a marketplace. The approach gives providers an opportunity to offer data access as a service at various levels; consumers can subscribe to data access services and use them. As a specific requirement, the approach supports delivering data in a consumer-specific manner. That is, different users obtain different vertical and horizontal parts of the data depending on some configuration. To this end, a Data Delivery Service (DDS) is proposed.

We suggest a flexible architecture that relies on API Management, particularly the WSO2 eco-system, and integrates a row-/column-level access control for relational database systems. Any data access is protected by an OAuth security concept.

The use of API Management eases implementing a marketplace but also brings up some major challenges. In particular, there is a strong need for achieving a flexible configuration and avoiding manual administrative effort due to the unknown and possibly numerous users. The paper discusses these challenges and their solutions in detail.

This work has been conducted in a funded project in the medical domain (Sonntag et al., 2015). To illustrate effectiveness of the overall approach, we set up a marketplace including the DDS and other value-added services in the way described in this paper. The DDS offers access to a medical i2b2 database (https://www.i2b2.org/) in this particular context. We were able to successfully control accesses of medical professionals within a clinic. Moreover, several value-added services have been

developed on top of the DDS and integrated into the marketplace, too. Of particular interest is the combination of data access with making data anonymous.

Our future work will consider commonly used database interfaces such as OData (http://www.odata.org) and other high-level REST APIs as data providers. We also plan to evaluate in depth whether the approach is appropriate for advanced restrictions such as satisfying regulatory compliance, governmental or dictated by another body. Moreover, the performance impact of filtering will be investigated in detail.

REFERENCES

Abramov, J., Anson, O., Dahan, M. et al., 2012. A methodology for integrating access control policies within database development. *Computers & Security, 2012, Vol. 31, No 3, pp. 299-314.*

Balazinska, M., Howe, B., and Suciu, D., 2017. Data markets in the cloud: An opportunity for the fatabase community. *Proc. of VLDB Endowment 2011, Vol. 4, No 12, pp. 1482-1485*.

Barker, S., 2008. Dynamic meta-level access control in SQL. In: Data and Applications Security XXII. *Springer Berlin Heidelberg, 2008, pp. 1-16*.

Bertino, E., Jajodia, S., and Samarati, P., 1999. A flexible authorization mechanism for relational data management systems. In *ACM Transactions on Information Systems, Vol. 17, No. 2, April 1999, pp. 101-140.*

Caires, L., Pérez, J., Seco, J. et al., 2011. Type-based access control in data-centric systems. In: 20th European Conference on Programming Languages and Systems: Part of the joint European conferences on theory and practice of software. *Springer Berlin Heidelberg, 2011. pp. 136-155.*

Chaudhuri, S., Dutta, T. and Sudarshan, S., 2007. Fine grained authorization through predicated grants. In: *23rd Int. Conference on Data Engineering (ICDE), IEEE 2007, pp. 1174-1183*.

Chlipala, A. and Impredicative, L., 2010. Static checking of dynamically-varying security policies in database-backed applications. In*: The USENIX Conference on Operating Systems Design and Implementation, 2010, pp. 105-118.*

Corcoran, B., Swamy, N., and Hicks, M., 2009. Cross-tier, label-based security enforcement for web applications. In*: Proc. of the 2009 ACM SIGMOD Int. Conference on Management of Data. ACM, 2009, pp. 269-282*.

Fischer, J., Mario, D., Majumdar, R. et al., 2009. Fine-grained access control with object-sensitive roles. In: *European Conf. on Object-Oriented Programming (ECOOP 2009). Springer Berlin Heidelberg, 2009, pp. 173-194.*

Fuchs, L., Pernul, G., and Sandhu, R., 2011. Roles in information security – a survey and classification of the research area. *Computers & Security, 2011, Vol. 30, No 8, pp. 748-769.*

Jayaraman, K., Tripunitara, M., Ganesh, V. et al., 2013. Mohawk: abstraction-refinement and bound-estimation for verifying access control policies*. ACM Transactions on Information and System Security (TISSEC), 2013, Vol. 15, No 4, Article No 18.*

Komlenovic, M., Tripunitara, M., and Zitouni, T., 2011. An empirical assessment of approaches to distributed enforcement in role-based access control (RBAC). *CODASPY 2011, pp. 121-132*.

LeFevre, K., Agrawal, R., Ercegovac, V. et al., 2004. Limiting disclosure in hippocratic databases. In: *Proc. 13th VLDB, pp. 108-119.*

Oracle, 2017. Using Oracle Virtual Private Database to Control Data Access. [Online]. Available: https:// docs.oracle.com/cd /B28359_01/network.111/b28531/ vpd.htm#DBSEG007.

Pereira, O., Regateiro, D., and Aguiar, R., 2014. Distributed and Typed Role-based Access Control Mechanisms Driven by CRUD Expressions*. Int. Journal of Computer Science: Theory and Application, Vol. 2, No 1, October 2014, pp 1-11.*

Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P., 2004. Extending query rewriting techniques for fine-grained access control. In*: ACM SIGMOD Conference 2004, pp. 551–562*

Rjaibi, W., 2012. Data security best practices: a practical guide to implementing row and column access control. [Online]. Available: https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Wc9a068d7f6a6_4434_aece_0d297ea80ab1/page/A%20practical%20guide%20to%20implementing%20row%20and%20column%20access%20control.

Roichman, A. and Gudes, E., 2007. Fine-grained access control to web databases. In*: Proceedings of the 12th ACM symposium on access control models and technologies. ACM, 2007. pp. 31-40.*

Roman, D., Paniagua, J., Tarasova T. et al., 2017. proDataMarket: a data marketplace for monetizing linked data. Demo paper at *16th Int. Semantic Web Conference (ISWC'17), Vienna 2017.*

Sonntag, D., Tresp, V., Zillner, S., Cavallaro, A. et al., 2015. The clinical data intelligence project. *Informatik-Spektrum Journal 2015, pp. 1–11.*

Wang, Q, Yu, T., Li, N. et al., 2007. On the correctness criteria of fine-grained access control in relational databases. In*: Proc. of the 33rd Int. conference on Very Large Data Bases, 2007, pp. 555-566.*

Zarnett, J., Tripunitara, M., and Lam, P., 2010. Role-based access control (RBAC) in Java via proxy objects using annotations. In: *Proc. of the 15th ACM symposium on access control models and technologies. ACM, 2010, pp. 79-88.*

XACML, 2012. XACML – eXtensible Access Control Markup Language. [Online]. Available: http://www.oasisopen.org/committees/tchome.php?wgabbrev=xacml.