Data Portability Using WebComposition/Data Grid Service

Olexiy Chudnovskyy, Stefan Wild, Hendrik Gebhardt and Martin Gaedke
Faculty of Computer Science
Chemnitz University of Technology
Chemnitz, Germany
olexiy.chudnovskyy@informatik.tu-chemnitz.de, stefan.wild@informatik.tu-chemnitz.de,
hendrik.gebhardt@informatik.tu-chemnitz.de, martin.gaedke@informatik.tu-chemnitz.de

***Abstract* - Web 2.0 has become the ubiquitous platform for publishing, sharing and linking of content. While users are empowered to create and manage their data, the latter is still scattered and controlled by distributed and heterogeneous Web applications. The data is usually stored in internal silos and is only partially exposed through platform APIs. As a result, the reuse possibilities, fine-grained access control and maintenance of distributed data pieces become a time-consuming and costly activity, if feasible at all. In this paper, we introduce our approach to develop Web applications based on the principles of data portability, decentralization and user-defined access control. Our main contributions are (1) a novel Web service-based storage solution for the read-write Web, combined with (2) a security framework for WebID authentication and authorization based on WebAccessControl lists (WAC), and (3) a corresponding systematic approach for Web applications development based on loosely-coupled and user-controlled storage solutions.**

Keywords - WebComposition; Data Engineering; REST; WebID; Web 2.0

I. INTRODUCTION

In the age of Web 2.0, it is the users, who produce a huge amount of data by contributing to blogs, wikis, discussion boards, feedback channels or social networks [1]. The numerous platforms on the Web facilitate this activity by providing sophisticated publishing tools, data sharing functionalities and environments for collaborative work. While users enjoy the simplicity and comfort given by these applications, they usually face the problem that "their" created data belongs to the service provider and is basically out of their control [2]. The data can be accessed, edited or linked only in the ways that were originally foreseen by platform developers. For example, user profile stored in Facebook cannot be synchronized with profiles in other social networks. An uploaded picture cannot be linked with others using new relationship types like "same event" or "same place". The accessibility and portability of data depends on APIs, usage terms and conditions of different platforms. In summary, users are not only hindered in their sharing, linking and publishing possibilities – they do not really control their data anymore. Application developers, on their side, are hindered in consuming the published content, either because existing platforms expose it in a restricted way or do not include enough metadata required for the particular domain.

There is a clear need for storage solutions, frameworks and tools, which would support both: users and developers in their corresponding activities. In this paper, we present our approach to decouple Web applications from storage solutions and analyze the resulting consequences regarding data access, security and application development process. In particular, the contributions of the paper are the following:

1. A novel Web-service-based storage solution enabling data publishing and linking based on the principles of Linked Data. The solution acts as a portable Web component, which provides repository functionality and also enables aggregation of access to distributed heterogeneous data sources.
2. A security framework for user-defined access control based on the social graph defined by the Friend-of-a-Friend ontology (FOAF). We apply the WebID concept [3] and WebAccessControl protocol [4] to design a reusable module for client authentication and authorization.
3. A systematic approach to develop storage-decoupled Web applications. We adapt modeling techniques and tools used to design application data domain and provide guidance in adoption of the presented architecture.

The rest of the paper is organized as follows: in Section II, we discuss the concept of application-independent storage solution and introduce our implementation based on WebComposition/Data Grid Service (DGS). In Section III, we describe the utilized authentication and authorization approach based on WebID and WebAccessControl protocol. Section IV discusses how traditional development process should be adapted to take the new architecture into account. In Section V, we illustrate our approach by implementing a simple photo management application. Finally, in Section VI, we summarize the paper and give an outlook into our further work.

II. WEBCOMPOSITION/DATA GRID SERVICE AND READ-WRITE WEB

WebComposition/Data Grid Service [5] is the core element of the fourth generation of the WebComposition approach [6]. It acts as an extensible storage and gateway

solution, which focuses on data integration and uniform access to distributed data sources [7][1]. Data Grid Service has several interfaces and applies the Linked Data principles to identify, describe and link internal resources.

Following, we describe the data model, interface and functional capabilities of Data Grid Service. Furthermore, we give an insight into its internal architecture and show extension possibilities.

A. *Data model*

The data space managed by WebComposition/Data Grid Service consists of a set of typed lists. Lists can have different nature and provide different operations on items inside. For example, the core modules of Data Grid Service implement operations on XML resources, which can be retrieved, updated, removed or linked with others. Extension modules implement handling of binary collections or structured access to external data sources, like relational databases, user tweets, blog entries, documents etc. In all cases, Data Grid Service provides a common view on distributed data spaces and exposes them to clients as lists of entries (Figure 1).

Beside typed lists, the so called virtual resources can be defined within Data Grid Service. While they do not offer any storage or gateway functionality, they are used to provide additional access and manipulation methods on the top of the existing lists. An example of such a virtual resource is the one enabling further representations of existing resources. With the help of transformation stylesheets like XSLT, the default XML representation of resources can be extended with RSS, Atom, JSON and other formats.

Collections and items within Data Grid Service can be annotated to provide additional information about their origin, author, creation time etc. Annotations also give information about resource behavior and defined access control. Furthermore, a repository-wide metadata is available, where specification of the lists and general service description are stored.

B. *Interface*

WebComposition/Data Grid Service follows the REST architectural style to identify and manipulate internal resources. All resources within Data Grid Service are identified using URIs. Some pre-defined URI-patterns are used to access metadata (`{resource URI}/meta`) or access control lists (`{service URI}/meta/acl`) etc. The standard HTTP methods GET, POST, PUT and DELETE are used to read, create, update and delete resources. Depending on the configuration, some of the resources may require authorization before executing the operations.

REST/HTTP interface provides several advantages for multi-tier architectures, where data storage is decoupled from services and third-party applications. First, it is simple, complies with well-proven Web standards and can be seamlessly integrated into the Web landscape [8]. Second, REST/HTTP enables loose coupling between services and clients. A REST-based storage solution can evolve independently and extend its functionality without breaking the contract. And finally, based on the HTTP protocol, third-party providers are empowered to provide additional services on the top of user-controlled storage solutions, e.g., caching, security etc.

Though REST/HTTP is the main and most suitable interface for decoupled data storages, also SOAP and XML/RPC endpoints are foreseen to support business

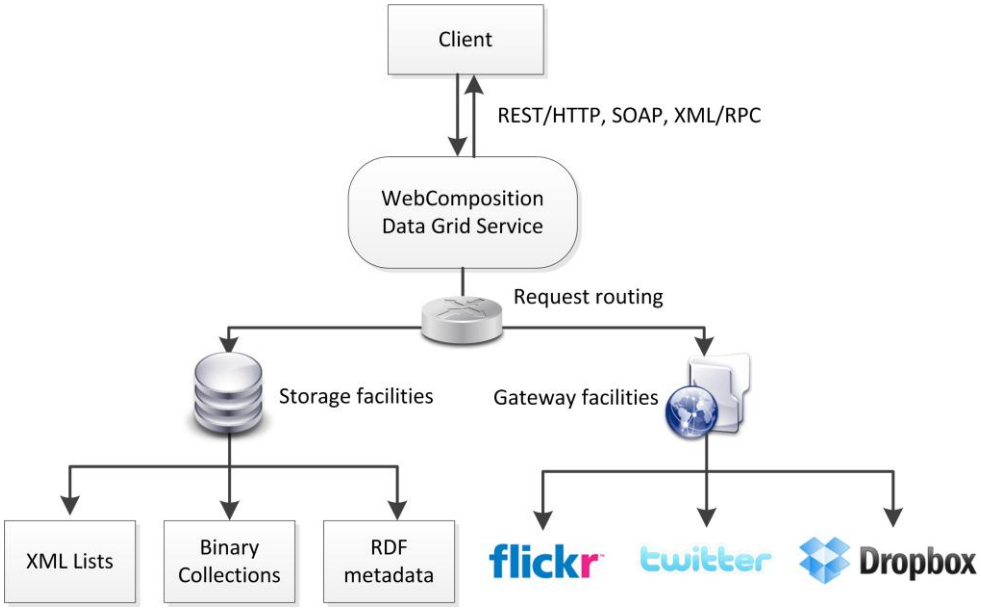

Figure 1 WebComposition/Data Grid Service

scenarios and proprietary clients.

C. Core functionality

Data Grid Service provides several core modules, which enable management of both structured and unstructured content. In particular, users have the possibility to upload their profile information, multimedia content and define fine-grained access control on the resources. Applications can utilize the published content and create new collections within Data Grid Service to store their internal data. In both cases, user is the only owner of the data and can extend, update or revoke access at any time.

Following example creates a new XML list within Data Grid Service:

```
<collection xmlns="http://www.w3.org/2007/app"
xmlns:atom="http://www.w3.org/2005/Atom"
xmlns:dgs="http://www.webcomposition.net/2008/02/dgs/">
<atom:title>profiles</atom:title>

<dgs:dataspaceengines>

  <dgs:dataspaceengine
dgs:type="http://.../CacheDataSpaceEngine" />

  <dgs:dataspaceengine
dgs:type="http://.../XmlDataSpaceEngine"
xmlns:dx="http://.../XmlDataSpaceEngine/" >
    <dx:primarykey>
          childnode1/childnode2/id
    </dx:primarykey>
  </dgs:dataspaceengine>

</dgs:dataspaceengines>
</collection>
```

Listing 1. Definition of new list

The request contains title and type of the list as well as list-specific metadata. The type of the list defines the module(s) (or so called Data Space Engines), which would be responsible for the HTTP requests on corresponding URI namespace. The metadata may describe the behavior of the module more precisely, e.g., security policies or configuration settings. After creation it is accessible under `{list URI}/meta` URI and can be retrieved in Resource Description Framework (RDF) format.

A single request can be processed by several modules, called within a pipeline, where output of one module is passed to another one as input. This enables pre- and post-processing of incoming data - for example, for caching or transformation purposes.

Following, we present the core modules of Data Grid Service, their capabilities and usage examples.

1) Data Space Engine for XML lists

XML is a well-understood and easy to use format, which is commonly used in distributed Web applications. Data Grid Service uses XML as the main data representation format, mainly because of many existing tools and standards to validate, to transform and to navigate through XML-based documents. The module "Data Space Engine for XML lists" provides a broad range of functionality to deal with XML resources. Though basic Create/Read/Update/Delete (CRUD) functionality is supported for all kinds of XML resources, the main purpose of this Data Space Engine is to manage so called XML lists, i.e., XML resources, which have a flat tree structure and contain a list of semantically and structurally related XML items. The list model can be applied for many kinds of Web applications, e.g., blogs, content-management-systems, e-commerce applications etc.

By restricting the view from general XML resources to XML lists, the Data Space Engine can provide additional functionality specific to lists. In particular, the module provides operations to identify and retrieve single items, append new or delete existing ones. Furthermore, XML Schema and XSLT stylesheets can be applied to perform data validation or create alternative representation formats of list items.

XML lists are useful to represent user- or application-produced content and make it available to others. For example, a simple address book service can model user contacts as XML items and publish them as a collection in the user's storage solution:

```
$curl https://dgs.example.org/contacts

<contacts>
 <contact id="1001">
    <firstname>John</firstname>
    <secondname>Smith</secondname>
    <address>
        <street>2nd Avenue</street>
        <number>54</number>
        <zip>11124</zip>
        <city>New York</city>
    </address>
</contact>
 <contact id="1002">
   …
 <contact id="1003">
   …
</contacts>
```

Listing 2. Example of XML list

Following the RESTful architecture style the contact items can be retrieved, created, updated or deleted using the corresponding HTTP methods. However, as soon as user enables write access to further service providers, she might want to restrict the structure of the list items or check them for valid content. For this purpose, an XML schema is defined for the list, causing validation of document after all incoming write requests.

The XML representation of list items empowers application developers to use complex data structures while describing their data. An XML item is a tree, whose depth can vary depending on the application needs. In order to update nested items, like the contact address in the example above (so called partial update operation), the module provides the concept of URI templates, which enables dynamic assignment of URIs to item pieces. URI templates are configuration settings for the module and can be defined at run-time by adding dedicated metadata to the XML list:

```
POST /contacts/meta HTTP/1.1
Host: user1.datagridservice.example.org
Content-Type: text/n3
…

@prefix meta:
<http://www.webcomposition.net/2008/02/dgs/meta/>.
<http://dgs.example.org/contacts>
meta:urlTemplate
[
meta:url "contacts/{value}/address";
meta:xPath "/contacts[@id='{value}']/address"
].
```

Listing 3. Definition of URI template

An URI template consists of 2 patterns - the one to be applied on URIs of incoming requests and the one to be applied on XML list to select the desired subnodes. As a result, the nested XML nodes get associated with the URI `https://dgs.example.org/contacts/{contact-id}/address` and can be manipulated the same way as the parent ones.

Furthermore, arbitrary views on the XML lists can be defined in the same way. The expressiveness of view definitions, however, is limited to the expressiveness of XPath query language. As an example, one could define a view on all persons living in a particular city and retrieve them using a dedicated URI pattern.

Many data-driven applications rely on entity-relationship models while designing and managing their data domain. To model the "JOIN" operations on resources, i.e., to retrieve all the related items for some given one, XML Data Space Engine introduces the concept of relationships. Relationships define the connection between two items in terms of some given ontology. The relationships are described using RDF and belong to the list metadata. The definitions can be consumed by service clients in order to discover and apply additional retrieval functions. A relationship is configured through 3 obligatory and 3 optional attributes:

- *Parent*: A URI of the list to act as a primary list, e.g., `http://dgs.example.org/contacts`
- *Child*: A URI of the list to act as a subordinate list, e.g., `http://dgs.example.org/pictures`
- *Predicate*: A URI of RDF predicate to act as a foreign key, defining a connection between primary and secondary list items, e.g., `http://xmlns.com/foaf/0.1/img`.

A relationship configured using the above attributes enables processing of the following URI pattern:

```
http://{service_host}/{parent_list_name}/
{parent_item_id}/{child_list_name}
```

As a result, only those items from child list are retrieved, which are linked to the parent list item using the relationship-specific predicate. For example, a GET request on `http://dgs.example.org/contacts/1001/pictures` would return picture descriptions associated with the contact 1001.

A POST HTTP request on the same URI is used to add new items to the child list linking them simultaneously with the specified parent list item.

To create an inverse link from child item to the parent one, each time a direct connection is established, the optional *Inverse Predicate* attribute is used, containing a URI of RDF predicate for the inverse relationship. A corresponding RDF statement is then automatically added to the child list metadata, acting as a foreign key to the parent list item. The same URI patterns can be applied to retrieve, create or delete parent items linked to some given child item.

If many relationships between the same parent and child list should be modeled (1:n, n:m), optional *Parent* and *Child Aliases* can be defined to match the incoming request with corresponding relationship definition. Listing 4 gives an insight into the list metadata and shows the internal relationship representation.

In summary, Data Space Engine for XML lists enables users and third-party applications to store and manage their data using simple and flexible data model. It follows principles of RESTful architecture style and can be applied to implement a broad range of data-centric Web applications.

2) Data Space Engine for binary resources

Current Web 2.0 applications require from storage solutions efficient support of both structured data and arbitrary binary content. Some of the data is supposed to be public, while access to other has to be limited. This fact elicits corresponding requirements on resource publishing and access control functions.

In case of public resources, users should be assisted by annotating and linking resources among each other's. For example, metadata available in the uploaded media files has to be exposed in machine-processable format, so that Web crawlers and service clients can utilize it to implement more intelligent search and discovery functions. Providing links to related content is also essential to enable third-party applications to explore the user space.

```
$curl https://dgs.example.org/contacts/meta

<rdf:Description rdf:about="http://dgs.example.org/contacts/">

<dc:creator rdf:resource="http://dgs.example.org/profiles/27" />

<dc:title rdf:resource="Contacts" />

...
</rdf:Description>

...
<rdf:Description
   rdf:about="http://dgs.example.org/contacts/meta/relationships/68">
   <dm:source rdf:resource="http://dgs.example.org/contacts" />
   <dm:target rdf:resource="http://dgs.example.org/pictures" />
   <dm:predicate rdf:resource="http://xmlns.com/foaf/0.1/img" />
   <dm:inverse-predicate rdf:resource="http://purl.org/dc/elements/1.1/creator" />
</rdf:Description>
```

Listing 4. Example of the list metadata

The Data Space Engine for binary resources implements basic CRUD functionality for the arbitrary content. Binary resources are grouped within collections. The GET request on the collection returns an Atom feed with basic descriptions of collection items. To create and update new resources, corresponding HTTP requests with MIME type specification are used. The annotations and metadata of the items is accessible using the `{resource URI}/meta` URI pattern. Third-party applications can consume this metadata according to their needs or update it using corresponding HTTP methods.

We have implemented automatic metadata extraction for some common MIME types (based on pre-defined mapping between file attributes and RDF properties). For example, just after uploading an MP3 file to Data Grid Service, the artist and album information are immediately published using RDF and terms coming from Music Ontology [9]:

```
<rdf:Description
rdf:about="http://dgs.example.org/music/31"
xmlns:ns0="http://purl.org/ontology/mo/">
    <dc:date>2005</dc:date>
    <dc:description>
    Amazon.com Song ID: 20206547
    </dc:description>
    <dc:title>Von Hier An Blind</dc:title>
    <ns0:album>Von Hier An Blind</ns0:album>
    <ns1:genre>Pop</ns1:genre>
    <ns2:singer>Wir Sind Helden</ns2:singer>
<rdf:Description>
```

Listing 5. Example of metadata extraction

Similarly, Data Space Engine analyzes PDF, JPEG and MPEG file encodings in order to extract the metadata and expose it using the RDF and common vocabularies.

3) Data Space Engine for XSLT transformations

Data Space Engine for XSLT transformations enables definition of further representations of XML resources. For example, contact details from the example above can be exposed as JSON list, CSV table, Atom Feed etc. For this purpose, new resources are added to Data Grid Service and configured to be processed by the XSLT module. The configuration contains the specification of the XML resource to be transformed, the MIME type of the resulting resource and the XSLT stylesheet with the transformation algorithm (Listing 6).

The resource configuration and the stylesheet are considered as resource metadata and can be updated later to adapt the module behavior.

4) Data Space Engines for external services

The design of the architecture foresees extensibility possibilities by implementing the pre-defined module interface. As such, further data spaces, e.g., user blogs, tweets, activity streams or multimedia content scattered across different platforms can be embedded into Data Grid Service. Third-party applications may access this content according to policies defined by the user. We implemented gateways to Twitter, Dropbox and Flickr as examples to let users and applications discover the data in one place and consume it using one single unified REST/HTTP interface.

5) SPARQL Endpoint

All of the resources within Data Grid Service can be annotated using RDF metadata. To let service clients find resources they are interested in, we implemented a dedicated SPARQL endpoint, accessible at the dedicated `{service URI}/meta/sparql` URI. By sending compliant queries, clients can search for resources within Data Grid Service's data spaces.

III. SECURITY FRAMEWORK FOR PORTABLE DATA

While consolidating the user data in one place, storage solution has to guarantee its safety, security and privacy. As such, only authorized clients are allowed to read and modify the data. The choice of the authorization mechanism is crucial in order to address different types of clients and the peculiarities of the Web domain. Our goal is to enable easy-to-understand but still fine-grained access control, where rules are expressive enough to take users' social graph and relationships in account.

```
<collection xmlns="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/Atom"
xmlns:dgs="http://www.webcomposition.net/2008/02/dgs/">
<atom:title>contacts-atom</atom:title>
  <dgs:dataspaceengines>
       <dgs:dataspaceengine dgs:type="http://.../XsltDataSpaceEngine"
                        xmlns:dsexslt="http://.../XsltDataSpaceEngine/" >
         <dsexslt:source>contacts</dsexslt:soruce>
         <dsexslt:mimetype>application/atom+xml</dsexslt:mimetype>
         <dsexslt:stylesheet>
           <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                <xsl:output method="xml" indent="yes"/>
                <xsl:template match="contacts">
                     <feed xmlns="http://www.w3.org/2005/Atom">
                        ...
                       <xsl:apply-templates select="contact"/>
                     </feed>
                </xsl:template>
              ...
           </xsl:stylesheet>
         </dsexslt:stylesheet>
       </dgs:dataspaceengine>
  </dgs:dataspaceengines>
</collection>
```

Listing 6. Data Space Engine for XSLT resources

The emerging WebID standard is a promising technology, which enables single sign-on, flexible identity management and complex authorization rules. Every agent (person, organization or application) is identified using a URI, which refers to the user's corresponding public profile.

To prove that the WebID belongs to the user, he creates a self-signed certificate, which points to his WebID. The X.509 v3 certificate standard allows the binding of additional identities to the subject of the certificate through a subject alternative names extension [10]. Furthermore, the user extends the profile document with certificate's public key. When the user agent tries to access a protected resource, the server asks for the client certificate, before connection is established. The possession of the certificate is verified during the SSL/TLS handshake and is the first step in the authentication process. As the second step, server compares the certificates' public key with the one stored in the WebID profile. The comparison is usually performed with a dedicated SPARQL query on user profile URI. The successful authentication process proves that the FOAF resource belongs to the user and provides additional information to the server about the user profile, social relationships, etc.

After the user has been authenticated, the storage solution determines if he has enough rights to access the requested resource. We utilize WebAccessControl mechanism, which complements WebID with access control lists based on Semantic Web technologies [11]. WebAccessControl is inspired by authorization mechanisms implemented in many file systems. It allows placing a set of permissions on a specific resource for a user or a group, identified by an URI:

```
@prefix acl: <http://www.w3.org/ns/auth/acl>.

[acl:accessTo <http://example.org/img.png>;
acl:mode acl:Read, acl:agentClass foaf:Agent].

[acl:accessTo <http://example.org/img.png>;
acl:mode acl:Read, acl:Write; acl:agent
<http://example.org/foaf#joe>]
```

Listing 7. Example of WebAccessControl list

The access control lists are RDF resources with precisely defined semantics. The example above is an N3 serialization of the access control list and protects the resource with URI `http://example.org/img.png`. The first rule makes the resource readable for every user agent with a valid WebID, while the second one grants write permissions to the identity `http://example.org/foaf#joe`.

Currently, WebAccessControl foresees four different access modes. In addition to the mentioned *Read* and *Write* modes, one is able to grant *Append* and *Control* permissions. *Append* is a restricted *Write* permission, where one is only allowed to attach new data to the resource (e.g., in case of log files). If the agent should be capable of modifying the access control list, the mode *Control* has to be set.

The WAC list is stored centralized within Data Grid Service. The owner of the storage solution has write permissions to the access control list and can define the access rules for other agents.

The presented authentication and authorization mechanism is implemented as independent and reusable module. It is invoked before the request reaches the Data Space Engines-pipeline and checks if user has a valid WebID and was assigned required permissions to access the resource. The check of the permissions is done by mapping the HTTP methods GET, POST, PUT and DELETE to its

equivalents in the WebAccessControl ontology. We mapped HTTP GET to *acl:Read*, HTTP PUT as well as HTTP DELETE to *acl:Write* and HTTP POST to *acl:Append*.

IV. Development of Web Applications using WebComposition/Data Grid Service

We envision that users will be the only owners of their data independently from its usage by third-party Web applications. Applications have to provide an added value on the top of the data and not limiting its reuse, sharing and linking possibilities. To deal with the fact, that the storage is decoupled from the application and is shared on the Web between many applications, the classical development processes, models and supporting tools for data-driven Web applications should be adapted (Figure 2).

To illustrate our approach, we consider a simple Web application for management of photo albums. The platform should enable users to manage their pictures, tag them, and assign them to photo albums. Users should be able to browse albums of others and search for pictures using different criteria. Though there are plenty of platforms on the Web providing similar functionality, all of them require users to put the data inside one single platform.

Following, we show how to engineer Web applications, which do not host the user data in internal data silos, but utilize user-controlled storage solutions.

The development of the Web application starts with a requirements engineering step. We analyze user needs and capture their requirements regarding functional and non-functional aspects of the Web application. For our example scenario, we refine and write down the functionality described above. Apart other possible non-functional requirements, we focus on the fact that the data should be stored decentralized in user-controlled storage solutions.

After the requirements are captured, we analyze the structure of business domain in order to produce the conceptual model of the application. The result of the analysis is usually an Entity-Relationship (ER) model, which captures different types of objects from the business world, their attributes and relationships. To meet the peculiarities of storage-decoupled Web applications, we extend the model and distinguish between local and global entities, which should indicate that entity belongs either to user or to the application data space. For example, pictures and albums are entities, which belong to user and should be maintained within his data space, while platform-wide categories and tags can be managed centralized in the application data space.

Entity-Relationship model is an important artifact, which is used, among others, for database schema specification and automated code generation. In our approach, we apply distributed and Web-based storage solutions instead of monolithic databases, so that the data can be shared between different application and services. To enable independent (and possibly serendipitous) data consumption, not only structure, but also semantics of the data should be unified and captured within dedicated models. To meet this need, we extend the Entity-Relationship model with semantic-specific aspects. In particular, we capture the semantics of entities, attributes and relationships using common ontologies and vocabularies. For example, entity *Picture* can be annotated with `http://xmlns.com/foaf/0.1/Image` concept, and its attributes with `http://purl.org/dc/elements/1.1/description` and `http://purl.org/dc/terms/created` coming from FOAF and Dublin Core profiles respectively (Figure 3).

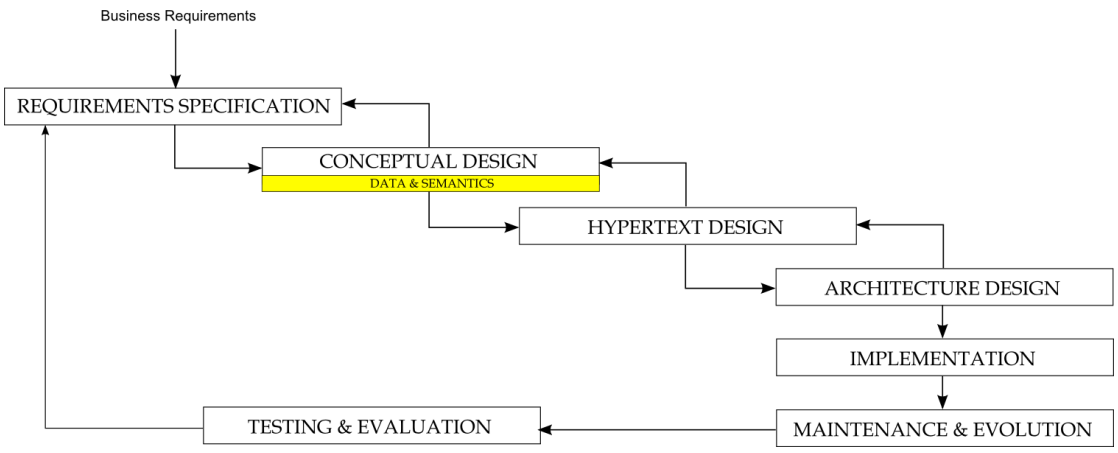

Figure 2 Development cycle of storage-decoupled Web applications

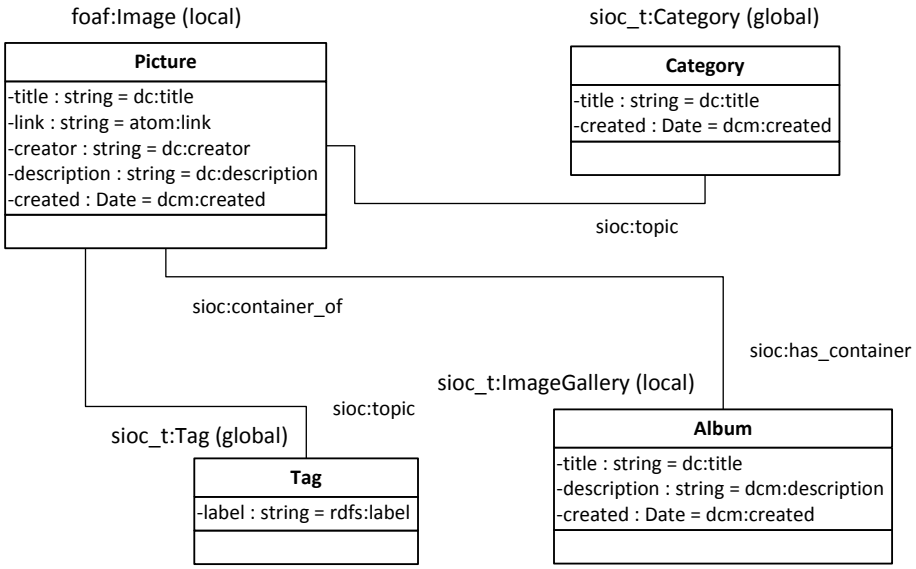

Figure 3 Extended Entity-Relationship model of the example application

By putting more semantics into the ER-model, the storage solutions are empowered to deliver data after the Linked Data principles. Due to the common model and explicit semantics, data stored within user storage can be discovered and reused more efficiently. In our approach, we transform the ER-model into a set of Data Grid Service XML lists, corresponding XML Schemas, relationship objects and transformation stylesheets to produce RDF/XML representation of application content.

The rest of the development process can be executed as in traditional Web applications. In the hypertext design phase we specify data display, input and navigation functions. We apply faceted navigation pattern for category browsing, thumbnails for album and set-based navigation for picture lists [12]. It is desirable that the storage solution has built-in support for these operations, so they perform more efficiently.

Our application has common three-tier architecture [13], where data server is a distributed layer of user-controlled storage solutions. We use WebComposition/Data Grid Service due to its broad support for both binary resources and structured XML content. Annotation and data transformation enables publishing of data after Linked Data principles, so that data created by one application can be seamlessly consumed by another. Furthermore we utilize ASP.NET Model-View-Controller framework [14] as "glue" between user interface and data storage.

To implement application authentication and authorization mechanisms, we apply the same approach as with securing user storages. The security modules described in Section III can be reused and integrated into the application. As a result, users authenticate themselves by presenting a certificate with WebID field, so that application can reuse the profile information stored within the FOAF file. To consume application services, user has to prepare a data space within his storage solution to be used by application and store it in his configuration settings. Corresponding access control rules have to be defined within the storage solution, so that application, identified by WebID as well, can access the required data.

The resulting application is tested and installed in the target environment (Figure 4).

Figure 4 The photo album management application based on Data Grid Service [15]

We notice that the application is loosely coupled with its data storage, which means, that their evolution can take place independently and hence be performed more efficiently.

V. RELATED WORK

Many distributed data storage solutions focusing on scalability, availability and simple data modeling appeared during the last couple of years [16]. In this section, the important contributions in this field of research are analyzed and discussed. The presented approaches can be roughly separated into the three areas: structured, distributed data storages, classical NoSQL databases and publishing tools.

DataWiki [17] is a platform to manage structured data using basic CRUDS operations enabled via a RESTful Web service API. To access or modify data sets stored in the DataWiki, mashups and forms can be created and hosted independently from the platform. All documents available in the DataWiki can be exported as Atom feeds. Information sharing with other systems is supported through built-in federation. Although the DataWiki approach separates data and representation from each other, it lacks of coping with large unstructured data sets, e.g., binary large objects.

data.fm [18] is an open source Web service implementing the REST architectural style, which supports common request methods and various response and media types, e.g., JSON, RSS or Atom, to perform access and retrieval operations on structured data sets available in the internal cloud storage. A graphical Web interface offers a convenient way to create new storage clouds with optionally restricting their access via ACLs. Data within the internal storage is organized in files and directories, which can be adapted from privileged persons through the API or GUI. Like the DataWiki platform, data.fm is well-suited for managing structured data, but compared to our approach, data.fm cannot apply post-transformations, e.g., via XSLT, to responses.

Another representative in this context is OpenLink Virtuoso [19], a structured data cluster providing certain virtualization layers to handle heterogeneous data sources and processing components. Principally, the software consists of an object-relational database accessible through specific database engines, integrated web and application servers. OpenLink Virtuoso provides a rich set of interfaces, e.g., SOAP, REST, XML-RPC, and SPARQL, to query for the uniquely identifiable elements stored in the database. In addition, the software supports protocols, e.g., Blogger and Atom 1.0, to publish data in a suitable form as well as components to interact with many types of application, e.g., wikis and address books. Although Virtuoso is a powerful tool to manage different types of data, it is complicated in installation and administration, which may become a crucial factor in success of user-controlled storage solutions. In contrast, Data Grid Service doesn't require a database or triple store in the backend and is installed using a simple installation wizard.

Similar to our approach, NoSQL solutions can be used as Web components, which support essential CRUD functionality for structured and unstructured data. For example, Apache CouchDB [20] stores schema-free data as name-value pairs, which are accessible through a RESTful Web interface. Like CouchDB, Amazon S3 [21] can be used to store unstructured data and access it through a REST/HTTP or a SOAP interface. In addition, Amazon S3 is often accompanied with Amazon SimpleDB [22], which allows saving structured, but schema-free data sets. NoSQL databases are designed to provide a scalable, fault-tolerant and flexible storage solution for schema-free data. Though NoSQL solutions can handle frequently changing document structures and new file types, the qualified data validation via document schemas is missing. Furthermore, they do not provide any support to centralize user data and enable fine-grained access control.

In conjunction with classic relational databases, the so called publishing tools can be applied to expose user data as Linked Data. The publishing tools automate the tasks of interpreting incoming HTTP requests, dereferencing URIs and converting the data in a proper form. One representative of this kind of tool is D2R server [23], which selectively transforms data from a legacy data source into RDF. The transformation is performed based on the parameters specified in the request. Currently, D2R server supports HTML and RDF browsers and provides a SPARQL endpoint to query the database content. Similar to D2R server, Triplify [24] converts the results of queries transmitted to a relational database into RDF, JSON or Linked Data. However, Triplify just executes a transformation of the output, i.e., queries have to be written in SQL. In contrast to our approach, the publishing tools only perform non-modifying operations on the legacy databases and do not aim to integrate other Web-based data sources.

Though the presented tools facilitate tasks of storing, publishing and linking data on the Web, they do not provide an integrated solution. Data wikis are flexible tools enabling collaborative data acquisition but cannot deal with unstructured data and distributed data spaces. NoSQL databases are scalable Web-based storage solutions, but are not so extensible in the sense of integrating additional data spaces. Finally, publishing tools support users gathering Linked Data out of legacy database. However, they cannot be used to propagate modifications back to the same storage.

VI. Conclusion and Outlook

In this paper, we have presented our approach to engineer Web applications based on user-controlled storage solutions. The separation of applications and data brings many advantages both for the end-user but also for application developers. Users have the full control of their data - they can specify what data should be public or private, what parts the third-party applications are allowed to access and how this data is linked to other resources. Application developers profit from the accessibility of user data and can deliver novel services more easily. Finally, the evolution of storage solutions and applications can take place independently and therefore less coordination and synchronization effort is needed.

We introduced WebComposition/Data Grid Service, a loosely coupled persistence and gateway layer for Web applications. We have shown how Data Grid Service can be used as a Web-based storage solution and how users can define access control using WebAccessControl lists. We presented reusable authentication and authorization modules based on the emerging WebID standard. Finally, we described a systematic approach to develop Web applications for decoupled storage solutions and illustrated it using a simple photo album management example.

In future, we expect the growth of the flexibility and functionality of user-controlled storages. As more and more applications access and change user data, the need to keep provenance information emerges. To meet this demand we

are going to add management and monitoring functionality for Data Grid Service. Especially quota assignment and event logging are important issues to protect the users' data space from malicious use and attacks. To link and synchronize data between storage solutions of different users, we are building dedicated publish/subscribe infrastructures. Finally, we are planning to extend the vocabulary for specification of access control lists and implement additional authorization rules based on the social graph of the user.

VII. REFERENCES

[1] O. Chudnovskyy and M. Gaedke, "Development of Web 2.0 Applications using WebComposition / Data Grid Service," in *The Second International Conferences on Advanced Service Computing (Service Computation 2010)*, 2010, pp. 55-61.

[2] T. Berners-Lee, "Socially aware cloud storage - Design Issues," 2009. [Online]. Available: http://www.w3.org/DesignIssues/CloudStorage.html. [Accessed: 23-Jan-2012].

[3] Manu Sporny, Toby Inkster, Henry Story, Bruno Harbulot, and Reto Bachmann-Gmür, "WebID 1.0 - Web Identification and Discovery," *W3C Editor's Draft*, 2011. [Online]. Available: http://www.w3.org/2005/Incubator/webid/spec/. [Accessed: 23-Jan-2012].

[4] W3C, "WebAccessControl - W3C Wiki," 2011. [Online]. Available: http://www.w3.org/wiki/WebAccessControl. [Accessed: 23-Jan-2012].

[5] O. Chudnovskyy and M. Gaedke, "WebComposition/Data Grid Service v1.0: Demo." [Online]. Available: https://vsr.informatik.tu-chemnitz.de/demo/datagridservice/. [Accessed: 23-Jan-2012].

[6] H.-W. Gellersen, R. Wicke, and M. Gaedke, "WebComposition: An Object-Oriented Support System for the Web Engineering Lifecycle," in *Electronic Proc. of The 6th International WWW Conference*, 1997.

[7] M. Gaedke, D. Härtzer, and A. Heil, "WebComposition / DGS : Dynamic Service Components for Web 2.0 Development," in *Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*, 2008, no. c, pp. 2-5.

[8] E. Wilde and M. Gaedke, "Web Engineering Revisited.," in *BCS Int. Acad. Conf.*, 2008, pp. 41-50.

[9] Y. Raimond, F. Giasson, K. Jacobson, G. Fazekas, T. Gängler, and S. Reinhardt, "Music Ontology Specification," 2010. [Online]. Available: http://musicontology.com/. [Accessed: 23-Jan-2012].

[10] D. Solo, R. Housley, and W. Ford, "RFC 2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile," 1999. [Online]. Available: http://tools.ietf.org/html/rfc2459. [Accessed: 23-Jan-2012].

[11] J. Hollenbach, J. Presbrey, and T. Berners-lee, "Using RDF Metadata To Enable Access Control on the Social Semantic Web," in *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)*, 2009.

[12] G. Rossi, D. Schwabe, and F. Lyardet, "Improving Web information systems with navigational patterns," *Computer Networks*, vol. 31, no. 11-16, pp. 1667-1678, May 1999.

[13] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002, p. 560.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software le*. Addison-Wesley, 1994.

[15] O. Chudnovskyy and M. Gaedke, "DGS Photogallery: Demo." [Online]. Available: https://vsr.informatik.tu-chemnitz.de/demo/photogallery. [Accessed: 23-Jan-2012].

[16] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, p. 12, May 2011.

[17] Google, "DataWiki," 2011. [Online]. Available: http://code.google.com/p/datawiki/. [Accessed: 23-Jan-2012].

[18] Data.fm, "Data Cloud," 2011. [Online]. Available: http://data.fm/. [Accessed: 23-Jan-2012].

[19] OpenLink Software, "Virtuoso Universal Server." [Online]. Available: http://virtuoso.openlinksw.com/. [Accessed: 23-Jan-2012].

[20] The Apache Software Foundation, "Apache CouchDB: The CouchDB Project," 2008. [Online]. Available: http://couchdb.apache.org/. [Accessed: 23-Jan-2012].

[21] Amazon, "Simple Storage Service (Amazon S3)." [Online]. Available: http://aws.amazon.com/de/s3/. [Accessed: 23-Jan-2012].

[22] Amazon, "SimpleDB." [Online]. Available: http://aws.amazon.com/de/simpledb/. [Accessed: 23-Jan-2012].

[23] C. Bizer and R. Cyganiak, "D2R server - publishing relational databases on the Semantic Web," in *Poster at the 5th International Semantic Web Conference (ISWC2006)*, 2006.

[24] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller, "Triplify – Light-Weight Linked Data Publication from Relational Databases," *Proceedings of the 18th international conference on World Wide Web*, pp. 621-630, 2009.