The BHL Portal Abstraction Layer

Specifics of the BHL portal API and BHL metadata

First Draft – Kai Stalmann, MfN Berlin, Sept. 2009

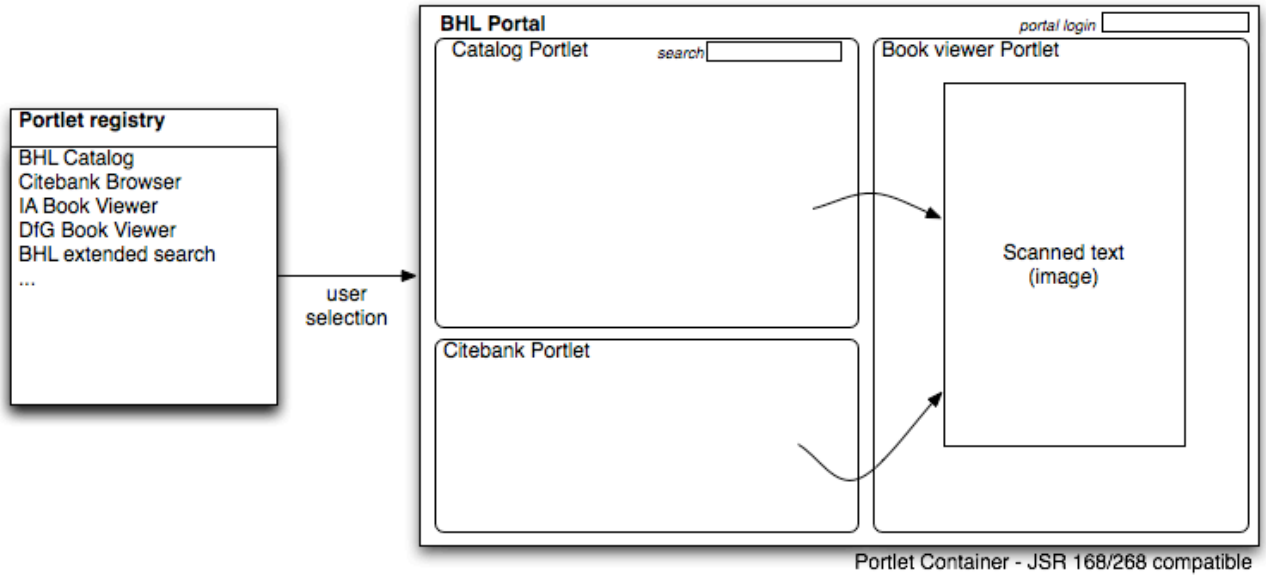
1.) Presentation Layer

BHL-E is meant to be a community project with several partners contributing requirements, code, content, ideas and knowledge. We have taken measures to receive requirements, content, ideas and knowledge from our partners. The development, however, should also be 'crowdsourceable' because distributing the coding tasks on many heads:

- ☒ lowers risks
- ☒ scales better
- ☒ diversifies the team's knowledge

To address this we want use a **Portal Container** and code or recode the BHL application code as **portlets**. A portlet we understand as an industry standard for a defining a functional module for a portal. A portlet can be programmed, translated, deployed, installed and used separatly on any compliant portal container.[1] Using a portal container and portlets makes a difference for the user. On a convential web application it is up to the programmer to decide what the user can use. On a portal the user can personalize his or her portal and select the portlets she wants to see.[2]

While normally in web applications[3] the code (often PHP or JSP) organizes the complete web page as one block[4], a portlet in a portal only organizes model, view and controller for that application (liek the search, the book viewer etc). - Although portlets are somehow selfcontained modules they also are designed to use a set of services that the container must provide.



1 A search for example can be a portlet, a catalog browser can portlet, as can a book viewer or a wiki.

2 Portlets are deployed to a registry. The user selects portlets from the registry an drops them into the container's frame.

3 Web applications are often programmed in PHP or JSP as webpages in a MVC manner on top of a set of technical services and backend systems. The PHP or JSP pages hold all the code that defines the web application from the menu through all tables, forms, controls, links etc that form an entire application. Web applications of this type are usally best managed by a small team working closely together at one place or office. Extending the functionality displayed to the user of the website requires changing the code that produces the HTML and JavaScript scriplets that the user's browser receives in response to the requests the browser sends.

4 The one block that makes the webpage may be structured in 'articles' if a web content management is used to publish the page.

The JSR 168, 268 portal standard

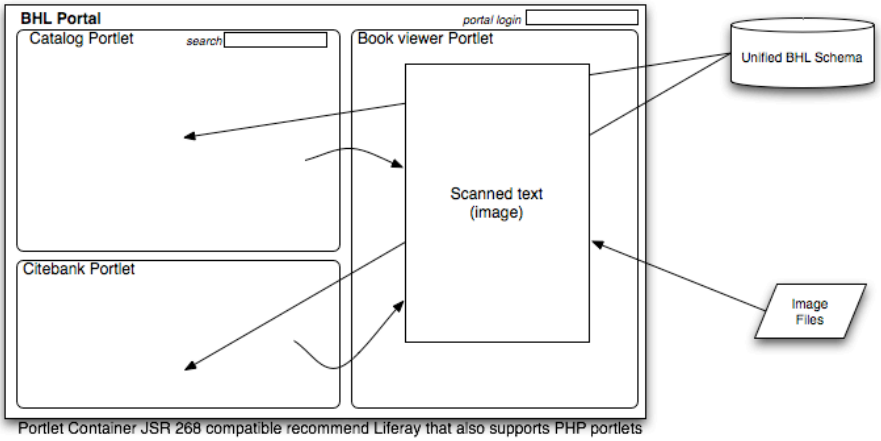
We want to use a JSR 168/268 compliant portal container. This standard is widely used and supported by big vendors (IBM, SUN, Apache, BEA, HP and many others). There are several open source implementations of this standard in place (Pluto, Liferay, eXo, Jetspeed). We'd recommend Liferay because it is mature and widely used. Liferay supports – besides the Java Standards 168 and 268 – also PHP portlets. This would make it much easier to reuse code that BHL already has.

To summarize: Rolling out Liferay as the presentation layer for BHL would have many advantages:

- A portlet container provides a good starting point for BHL. The portal would expose default applications to first-time and anonymous users. Registered users may personalize the portal and install the portlets they really need.
- The container complies to standards: it is easier to find contributers that are willing to support that.
- There is a technical community that we profit from.
- Portlets we build are compliant to JSR 268, or at least to Liferay (PHP portlets): that makes it easier to reuse our code in other projects that may use a similiar container.
- Coders – where ever they reside in europe or even the world – can enrich the portal functionality by providing portlets.
- There are lots of ready made portlets that we or our users may want to use.
- Portlets separate concerns and keep things structured.
- Portal containers can easily be hosted by various institutions. That would help us to achieve scalability.

2.) Data Access Layer

Regardless what kind of infrastructure is used to host the portal (i.e. a IAAS cloud) it is clear that portlets usually need access to backend data. A catalog portlet for example would enable users to search and browse the BHL catalog. Therefore a catalog portlet needs access to the persistent catalog data. A book viewer portlet would display a page of a book selected from the catalog. The image containing the page would be retrieved from the backend system. While the catalog portlet would also work as a controller for the book viewer and select a book, the book viewer is a controller itself and selects and retrieves pages. For retrieving the pages the portal needs to access the persistence layer holding images.



To facilitate accessing backend data from a portlet we need an interface to the backend data. For the application programmer who builds a portlet using BHL platform the API is a crucial part. An API should be[5]:

- ☒ Easy to learn
- ☒ Hard to misuse
- ☒ Easy to read and maintain code that uses it
- ☒ Sufficiently powerful to satisfy requirements (but not more powerful)
- ☒ Easy to evolve
- ☒ Appropriate to the audience

As we invite coders to contribute portlets to our portal these coders firstly need to know how to comply to our presentation layer. We achieve this by using a portal container.
In addition to this we also have to take care that portlet contributers can easily access or backend data. Generic interfaces like SOAP or languages like MARC are good examples for very powerful API's that do not fulfill the criterias above.

Accessing catalog data (content metadata) and for the content itself (images) would be supported in the most appropriate way by simple fulfilling the requirements that an API user might pose.

The requirements however stem from the use cases the portlet programmers implement. Although we cannot forsee all the possible use cases we can anticipate the use cases that are related to BHL data.
Most important use cases are:

- ☒ deduplication of catalog entries
- ☒ searching metadata
- ☒ building a metadata index for indexed search
- ☒ browsing the catalog
- ☒ viewing the images
- ☒ updating metadata or content
- ☒ exploring the metadata for processing semantic calculations
- ☒ enriching metadata
- ☒ bidlists and picklists
- ☒ synchronizing or replicating metadata and data

It is not a surprise that these use case can be brought down to:

- ☒ CRUD operations on single records
- ☒ and LIST operations on sets of records
- ☒ where the notion of RECORDS refers to metadata objects and content objects.

Additionally there may be a need for some convenience operations like deleting a set of records etc.

For the sake of flexibility and maintainance the API functions should rather be implemented as services and not as a fascade. Deploying API functions as services requires an infrastructure for registering, locating, installing service. What we use here has to be defined and will probably be more implementation specific.[6]

---

5 Joshua Bloch, lead architect at Google, on a talk at Google on why an API is important: http://www.youtube.com/watch?v=aAb7hSCtvGw

6 Webservices of course could be used for generic services, but this will be slow and unapproriate for a 'low level' API like the one we discuss. OSGI would be a better choice but is Java specific. Other options are Thrift (language independent marshalling of objects overs sockets), or restlike service on HTTP.

CRUD operations

Create, retrieve, update and delete operations allow API users to basically locate, read and write single records.

LIST operations

List operations work on a set or subset of records. As we expect a huge amount of data these list operations must reflect that be supporting paging directly etc. There may also be iterators or cursors to facilitate navigation in a set.

RECORDS

BHL has to deal with two quite different kinds of records: **content** and **metadata**. Nevertheless there should be a consistent way to handle both object types, and the API functions for these objects should worl seamlessly together.
Accessing content (images) requires an identifier for the requested object. Since we handle media objects in different qualities (resolutions) this also must be determined by the API methods. Even though the underlying operations may be conducted be sophisticated technologies (platform services of a cloud for example) the API calls for accessing content are simple.

API functions for metadata access need more consideration because there is a need for more operations and the structure and semantic of the metadata itself has to be defined.

We differenciate between two kinds of metadata:

Original metadata	any format preserved as is transformed to BHL format accessible via API
BHL metadata	based on Dublin Core parlance extended to meet the BHL needs accessible via API

BHL is going to provide means to map the various metadata formats to the BHL format. While the BHL format is designed to be sufficiently rich to cover all bibliographic levels relevant for BHL we do not care about curational information. But, since we always store the original metadata too we want to allow applications to access original metadata through the API.

However, the main focus of the API is to provide a set of fast performing, easy to use functions for accessing metadata. By providing our metadata format and transforming other formats like MARC, MODS, DC into the BHL format we make sure that the catalog does not suffer from empty fields (sparse data). It is unlikely that the library catalogs of content providers can deliver the data BHL needs. Due to the fact that widly used formats like MARC do not cover fields we need [7] we anticipate that data exported from libraries must be enriched manually. We plan to support and facilitate manual editing by providing a tool that could be run as stand alone application or as a web application. This tool would support semi automatic data import, manual edits and export / upload of the BHL format.
The exact layout of the **BHL Data Provider API** is documented separatly.

3.) BHL metadata and bibliographic levels

7 MARC for example does not cover item information. BHL (US) solves this problem by using an additional service (wonderfetch) to drag item data in during the ingest.

The BHL approach is based on Dublin Core (DC) metadata conventions but enhances DC to meet the BHL requirements.
The BHL catalog – that is formed out of the BHL metadata elements – applies a set of attributes (or fields) to 8 media types that reflect 3 plus 1 different bibliographic levels:

- concept
- title
- item
- page (discussed later)

synthetic	monograph	serial	volume	serial number	item	article	chapter
concept level	**title level**	**title level**	**title level**	**title level**	**item level**	**item level (or title level)**	**item level (or title level?)**
aggregated data from all imports of one title, may be edited, bhl addon	one title per import, grouped under synth. monograph title	one title per import, grouped under synth. serial title	data from one import, grouped under monograph title	data from one import, grouped under serial title	data from one import, grouped under one title or volume	data from one import, grouped under one serial title	data from one import, grouped under one monograph title

While the first three levels are available during the metadata ingest, the fourth is only available after the content has been ingested (either because a digital copy was received, or after scaning and ocr).

Item level

On item level the BHL format supports entities that relate to physical (or at least digital) content in the content providers libraries:

- item: A book or serial
- article: A separate printout of an article (from a serial)
- chapter: A separate printout of a chapter (of a book)

Items usually have a place in a book shelf and can be identified in the library with a barcode or the like. BHL wants to keep track of this information.

Title level

On title level BHL supports entities that relate to catalog entries in content provider libraries:

- monograph: Title of a book, stored as received, grouped under a synthetic title.
- volume: Title of one volume, stored as received, grouped under a synthetic title.
- serial: Title of a journal, stored as received, grouped under a synthetic title.
- serialnumber: Title of a journal issue, stored as received, grouped under a synthetic title.
- article: Title of an article, stored as received, grouped under a synthetic title.
- chapter: Title of a chapter, stored as received, grouped under a synthetic title.

A title may or may not have a physical or digital content object assigned to it. BHL will ingest titles, track the origin and thus build up a catalog containing all titles the content providers send us.

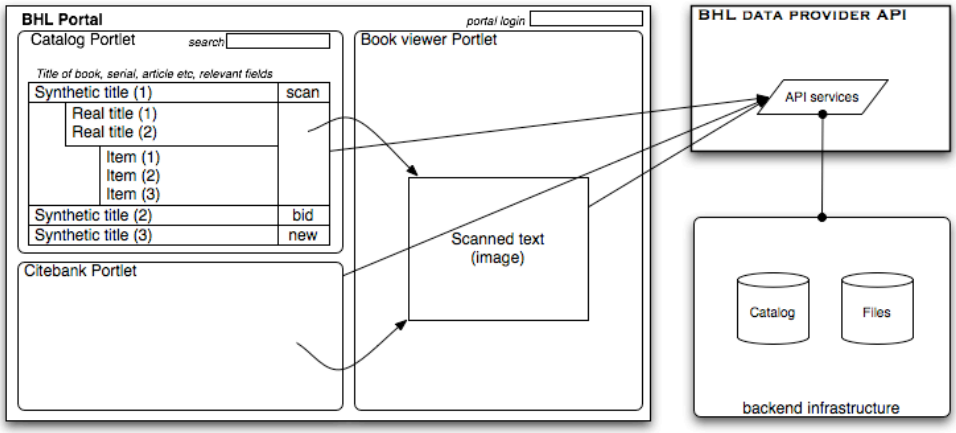
Concept level

BHL introduces the notion of a concept of a title. Titles of this kind are synthetic title, they may be normalized (diacritics being removed), or even edited by the editor of the metadata (BHL does not change original metadata).
One synthetic title subsumes a set of titles that refer to the same concept of a book or serial.
The synthetic title is important for searching, browsing, and we will also use it for deduplication.
The notion to a concept of a book enables us to aggregate metadata from different titles or imports. For example. BHL may receive a title from provider A how maintains OCLC but not ISSN. Another provider who sends us the same title may maintain ISSN and ISBN but not OCLC. Grouping the metadata of this conceptually identical object together give the catalog an additional benefit: we can offer a service that translates identifiers.

Grouping of titles is done during the metadata ingest process: after the deduplication an import is reconciliated. (The **BHL Ingest and Deduplication Process** is documented separatly.)

As already said the fields of the BHL metadata format are the same for all levels. The exact set of fields is be published separatly in the **BHL Metadata Mapping Document**.

Summary



The BHL Portal Abstraction Layer addresses 3 concerns:

- a presentation standard for integrating applications (portlets)
- an API for accessing data
- a metadata convention defining the schema of the catalog

These 3 aspects together form an API that enables code contributers to integrate with the BHL portal. It is also a prerequisite for distributed programming. It is important to note that the API has to be programming language independent, or should at least be decoupled from the API implementation language by a proxy service because it is unlikely that the teams spread over europe (or even the world) will ever speak one programming language – as they don't speak one human language.

Standards, protocols and interfaces can help us getting on with the project, providet these elements

are clear, useful and efficient.

Further information can be found in these documents:

- **BHL Data Provider API**
- **BHL Ingest and Deduplication Process**
- **BHL Metadata Mapping Document**
- **BHL Page Level Metadata Specification**