

Document publishing in the Daisy CMS

Cocoon GetTogether
October 4, 2006
Amsterdam

Bruno Dumon

bruno@outerthought.org



<http://www.daisycms.org/>



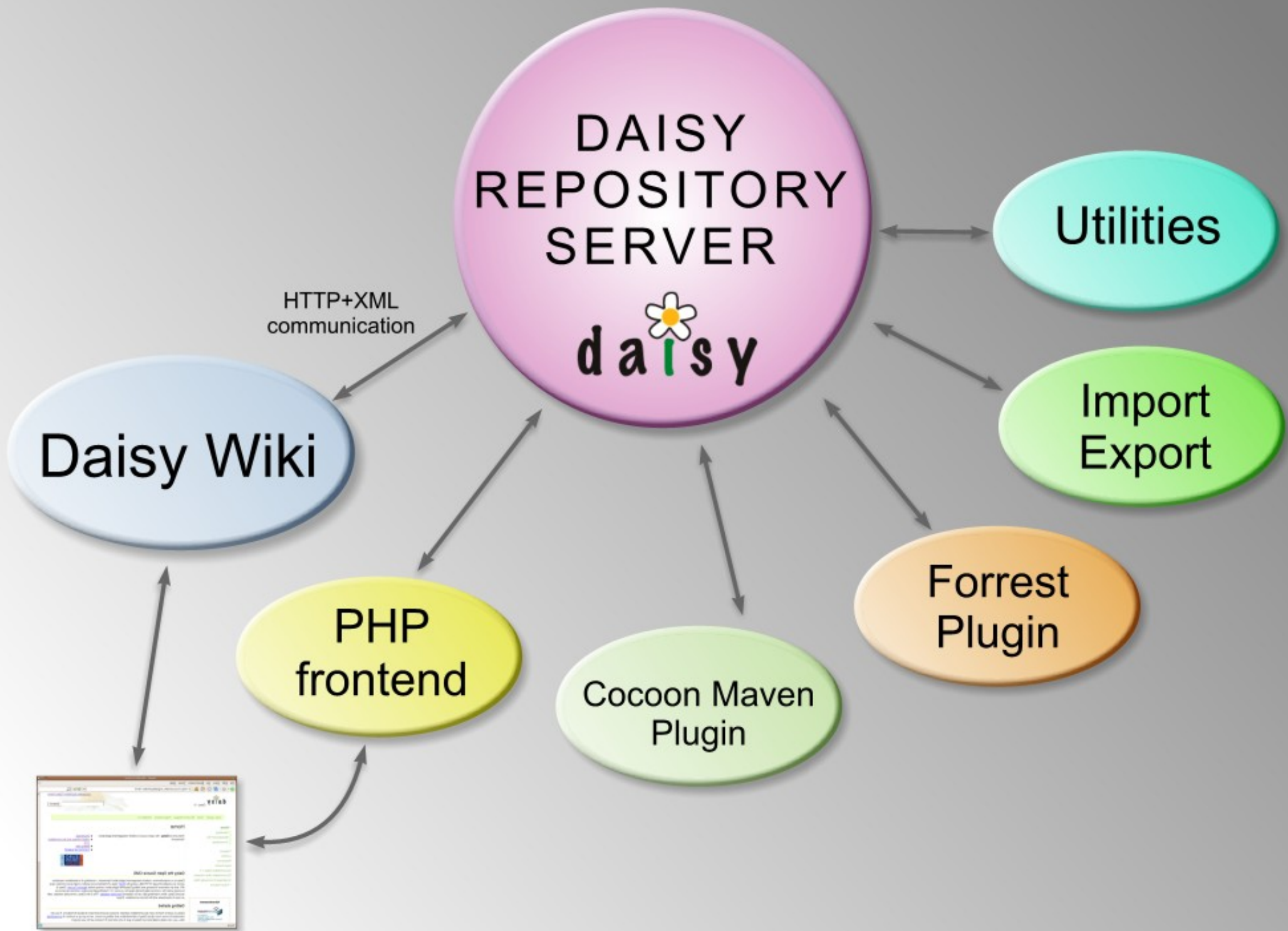
<http://www.outerthought.org/>

What is Daisy?

- CMS = Content Management System
- Java-based, frontend build on Cocoon (2.1)
- Open source project
- Daisy 1.0 released October 12, 2004
Current release: Daisy 1.5, Daisy 2.0 on the way.
- *Used by Cocoon for its documentation*

Agenda

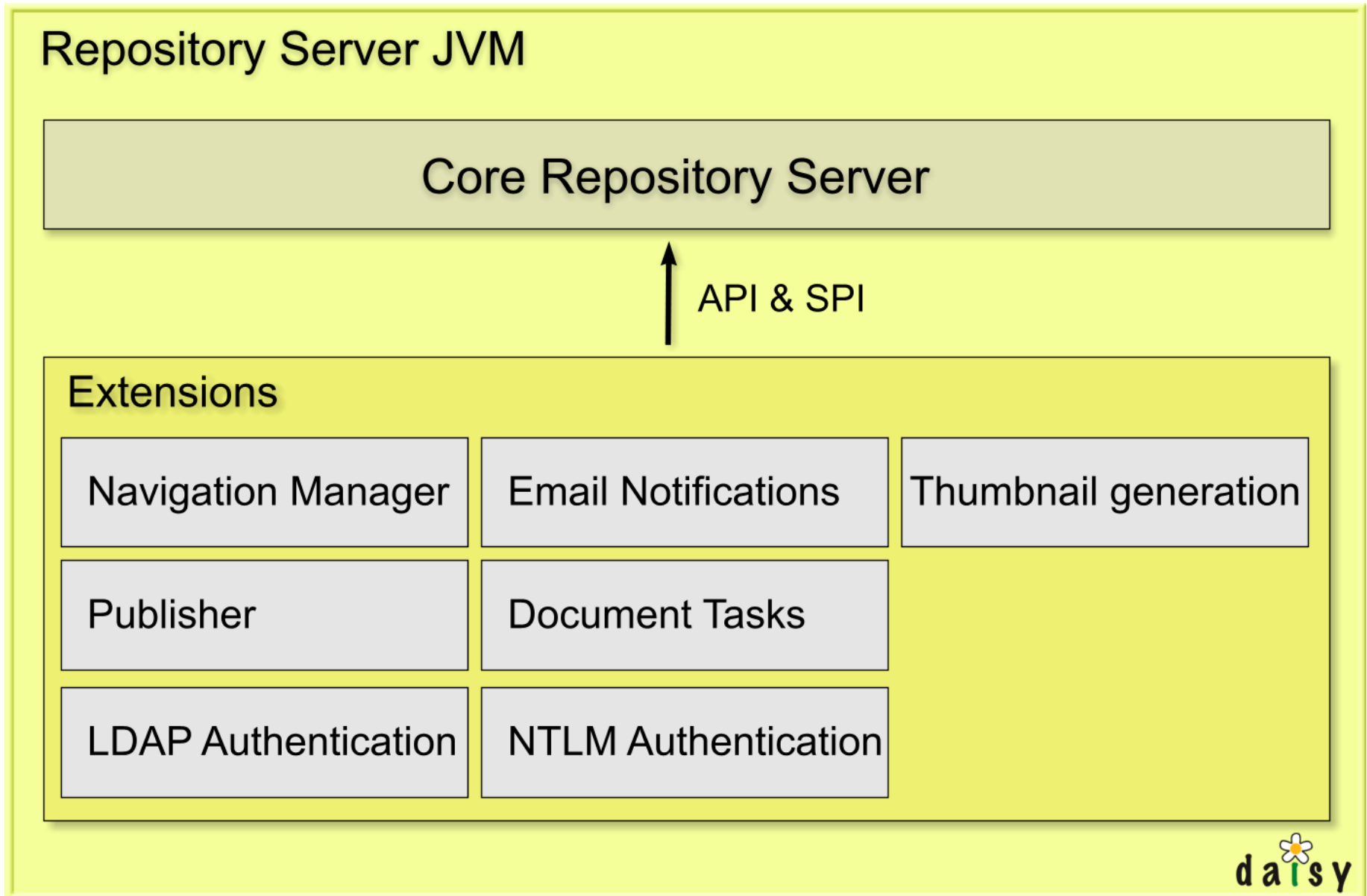
- General Daisy overview
- Demo of Daisy document publishing features
- Daisy Wiki overview
- Delve into how the document publishing works



Core repository server features


- Manages 'documents'
 - identified by an ID (Daisy 2.0: namespaced)
 - parts and fields (defined by a schema)
 - language and branch variants
 - flat structure (no directories)
- Versioning
- Locking
- Access control
- Link extraction
- Daisy Query Language (fulltext/structured search), faceted browsing
- JMS notifications
- APIs: native: Java, remote access: HTTP+XML, Java
- Persistence: SQL database + filesystem + lucene index
- Backup solution

Repository server extensions



(demo)

The Daisy Wiki

- A Cocoon-based application 
- Much of the tough work is done by the repository, Wiki can focus on end-user interaction and styling.
- Can be viewed as:
 - a ready-to-use application
 - a front-end platform

Daisy Wiki customisation

- Customisation possibilities:
 - Skinning (custom layout)



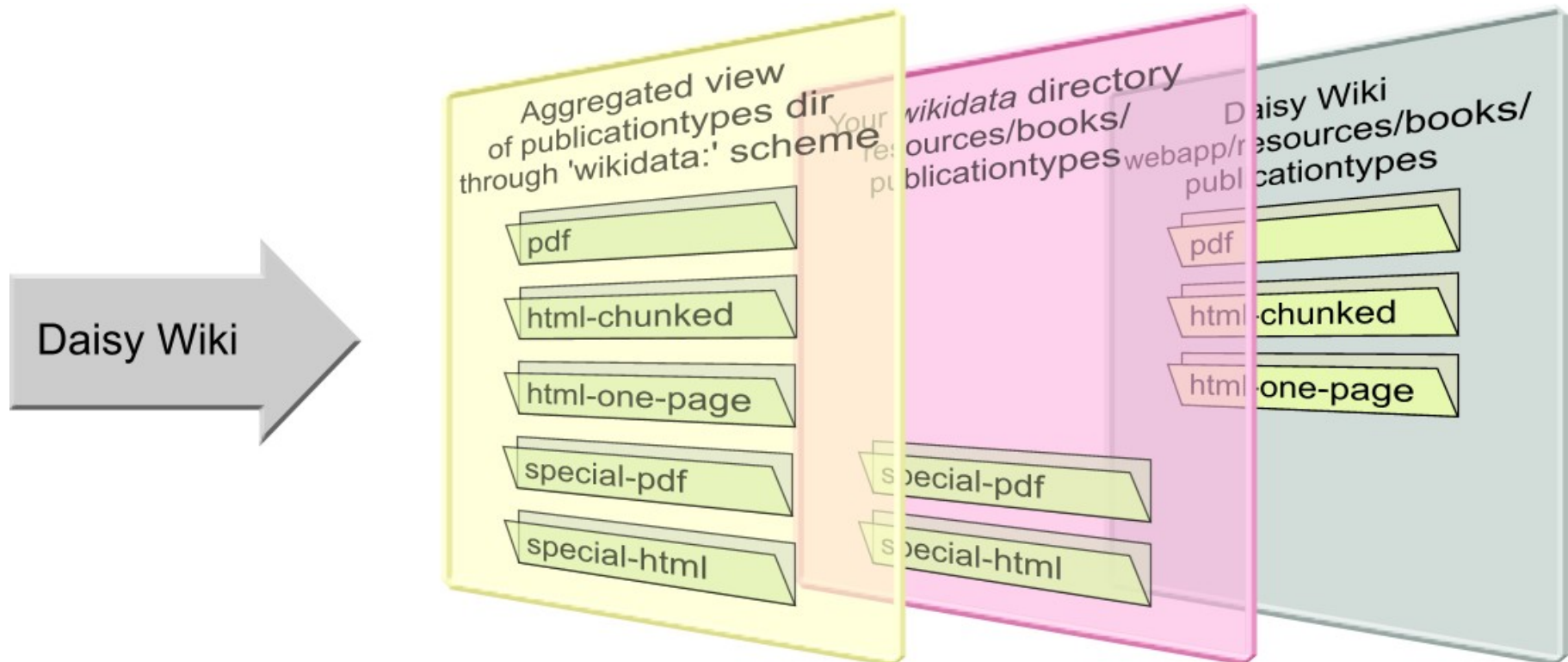
- Document type and query styling
- Extensions
 - `/ext/**` are forwarded to custom sitemap.xmap
 - flowscript API to access Daisy context: repository API etc.
 - can make use of global layouting possibilities

Wiki Data directory

- A directory containing all user customisations of the Daisy Wiki:
 - general config
 - site definitions
 - custom skins
 - custom document & query styling
 - custom book publication types
 - extensions
 - except for custom JARs (-> Cocoon 2.2)

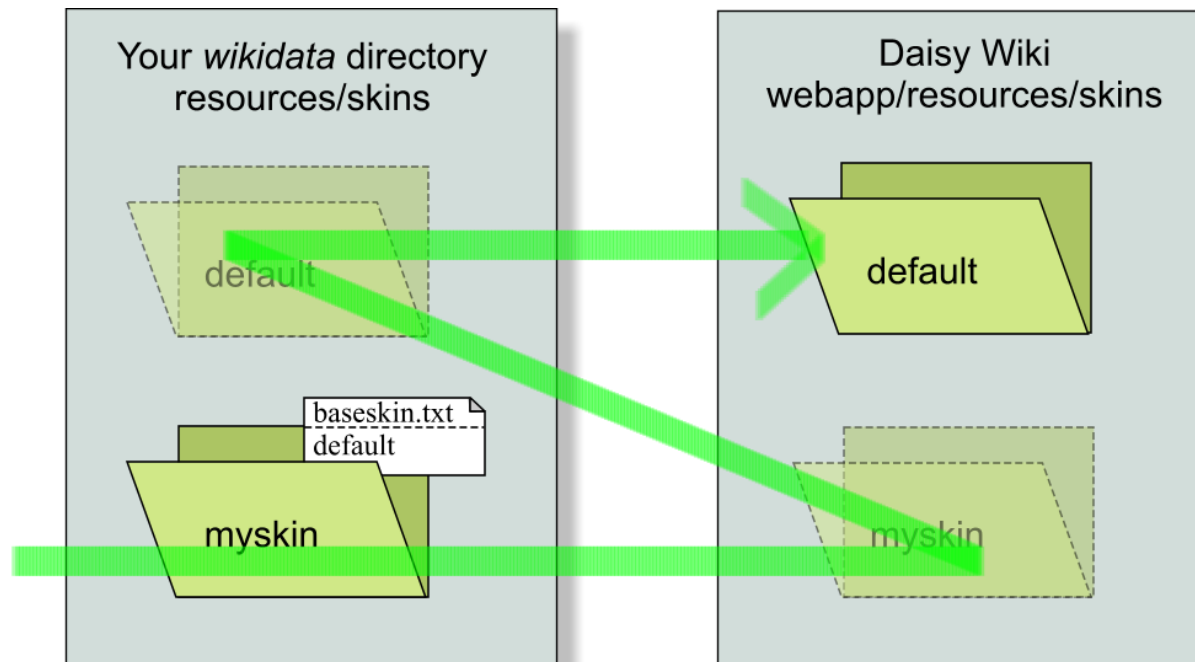
The wikidata source

- A 'fallback' source: first searches for a file in the wikidata directory, then in the webapp
- Provides an aggregated view on directories



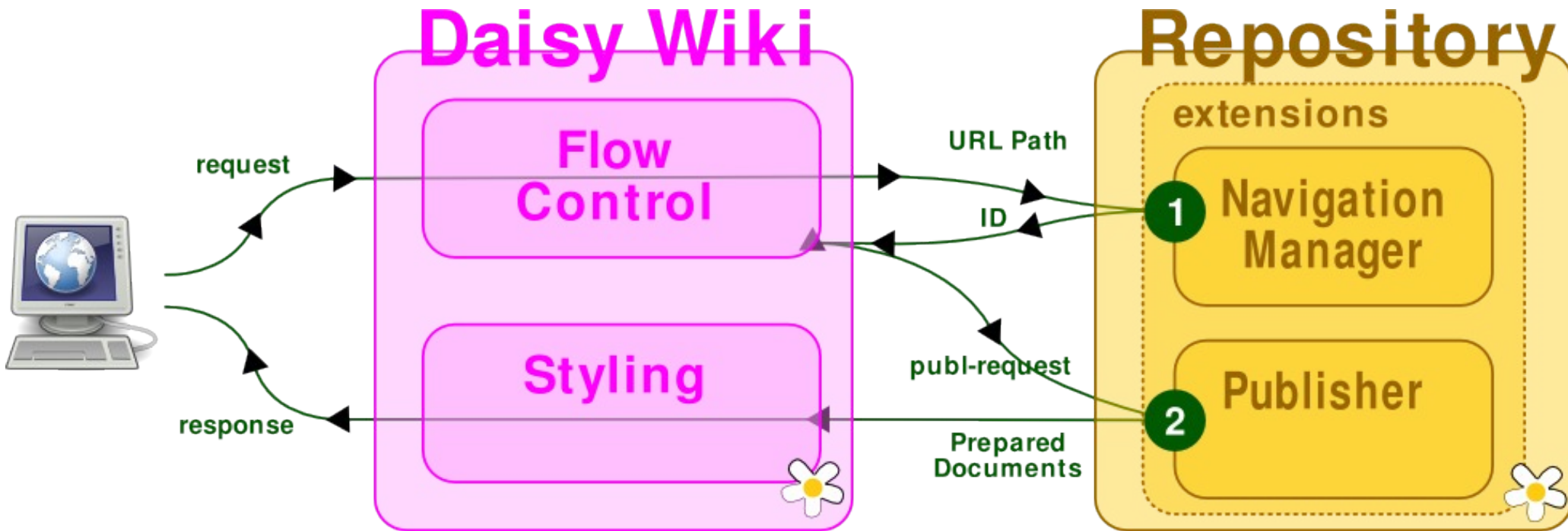
The daisyskin source

- The problem: don't want to duplicate all skin resources in each skin
- Therefore: fallback between skins, but also between wikidata dir and webapp
- XSLs themselves can also use daisyskin source



Document publishing

What happens between clicking a link to view a document and getting back the published HTML page?



The publisher

- Extension component inside repository server which performs a part of the publishing work.
- Aggregates data requested via 'publisher request' into one XML response
- Purpose:
 - speed: local access to all data (+repo caches)
 - basic publishing work reusable by other frontends

Publisher request example

```
<p:publisherRequest  
  xmlns:p="http://outerx.org/daisy/1.0#publisher"  
  locale="en-US" versionMode="live">
```

```
<p:document id="...">
```

```
<p:acInfo/>
```

```
<p:subscriptionInfo/>
```

```
<p:comments/>
```

```
<p:availableVariants/>
```

```
<p:annotatedDocument/>
```

```
<p:preparedDocuments applyDocumentTypeStyling="true">
```

```
</p:preparedDocuments>
```

```
</p:document>
```

```
<p:navigationTree>
```

```
<p:navigationDocument id="..."/>
```

```
</p:navigationTree>
```

```
</p:publisherRequest>
```

Information
about
a
document

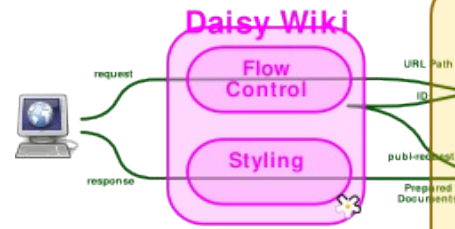
Navigation
tree

Prepared Documents

- Retrieves a 'prepared-for-publishing' expansion of a document

= XML representation of a document with content of HTML parts embedded and enriched with annotations

- The publisher also uses Cocoon-like SAX-based pipelines for this



Repository

extensions

Publisher

1. Field Annotation

2. Part Annotation

3. Content Inclusion
(for certain parts)

3a. Link Annotation

3b. Image Annotation

3a. Query Expansion

3a. QueryInclude Exp.

3e. Include Processing

1.

2.

3.

4. Response with
PreparedDocuments



1. Field Annotation

```
<d:document [...]>
[...]
<d:fields>
  <!-- Example annotation on basic string field -->
  <d:field typeId="5" valueType="string" name="MyStringField" label="My string field">
    <d:string valueFormatted="String value">String value</d:string>
  </d:field>

  <!-- Example annotation on date field -->
  <d:field typeId="6" valueType="date" name="MyDateField" label="My date field">
    <d:date valueFormatted="9/9/06">2006-09-09+02:00</d:date>
  </d:field>

  <!-- Example annotation on field with selection list-->
  <d:field typeId="7" valueType="string" name="AnotherStringField">
    <d:string valueFormatted="Belgium">BE</d:string>
  </d:field>

  <!-- Example annotation on link field -->
  <d:field typeId="19" valueType="link" name="MyLinkField" label="My link field">
    <d:link documentId="84" target="daisy:84" valueFormatted="My document"/>
  </d:field>

</d:fields>
```

2. Part Annotation

```
<d:document [...]>
  [...]

  <d:parts>
    <d:part typeId="13" size="1997" mimeType="text/xml" daisyHtml="true"
      name="CountrySummary" label="CountrySummary" inlined="true">
      <html>
        <body>
          <p>foo bar</p>
        </body>
      </html>
    </d:part>
  </d:parts>

</d:document>
```

3a. Link Annotation

```
<html>
  <body>
    <p><a href="daisy:2-DSY">
      <p:linkInfo documentName="test" documentType="SimpleDocument">
        <p:linkPartInfo id="7" name="SimpleDocumentContent" fileName="..." />
      </p:linkInfo>Example link</a>.
    </p>
  [...]
```

Note: whitespace added for readability

3b. Image Annotation

```
<p>  
  
  <p:linkInfo documentName="road" documentType="Image">  
    <p:linkPartInfo id="1" name="ImagePreview"/>  
    <p:linkPartInfo id="6" name="ImageThumbnail"/>  
    <p:linkPartInfo id="11" name="ImageData" fileName="road.jpg"/>  
  </p:linkInfo>  
</img>  
</p>
```

3c. Query Expansion

```
<html>  
<body>
```

```
[...]
```

```
<pre class="query">select name where documentType = 'Country' order by name option  
style_hint='bullets'</pre>
```

```
</body>  
</html>
```

Query gets replaced by its result



```
<d:searchResult styleHint="bullets">  
  <d:titles>  
    <d:title name="name">Name</d:title>  
  </d:titles>  
  <d:rows>  
    <d:row documentId="3-DSY" branchId="1" languageId="1">  
      <d:value>Belgium</d:value>  
    </d:row>  
    <d:row documentId="5-DSY" branchId="1" languageId="1">  
      <d:value>Burkina Faso</d:value>  
    </d:row>  
  </d:rows>  
</d:searchResult>
```

[...]

3d. QueryInclude Exp.

```
<html>  
<body>
```

```
[...]
```

```
<pre class="query-and-include">select name where documentType = 'Country' order by name  
option style_hint='bullets'</pre>
```

```
</body>  
</html>
```

Query-include gets replaced by include instructions



```
<pre class="include">daisy:3-DSY</pre>  
<pre class="include">daisy:5-DSY</pre>  
<pre class="include">daisy:6-DSY</pre>  
<pre class="include">daisy:4-DSY</pre>
```

3e. Include Processing

```
<html>  
<body>
```

```
[...]
```

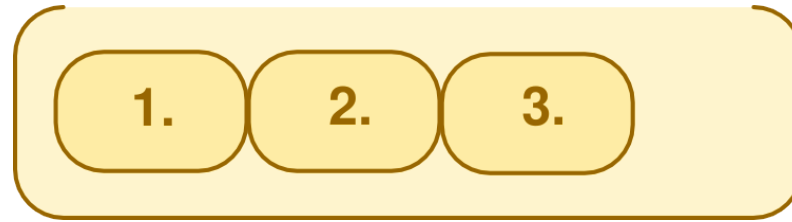
```
<pre class="include">daisy:5-DSY</pre>
```

```
</body>  
</html>
```

Include:

- * Included document is 'prepared' in the same way as the main document
- * result is outputted not in-place but in parallel (next slide)

```
<p:daisyPreparedInclude id="2"/>
```



<p:preparedDocuments>

```
<p:preparedDocument id="1">  
  <d:document>  
    [...] <p:daisyPreparedInclude id="2"/> [...]  
  </d:document>  
</p:preparedDocument>
```

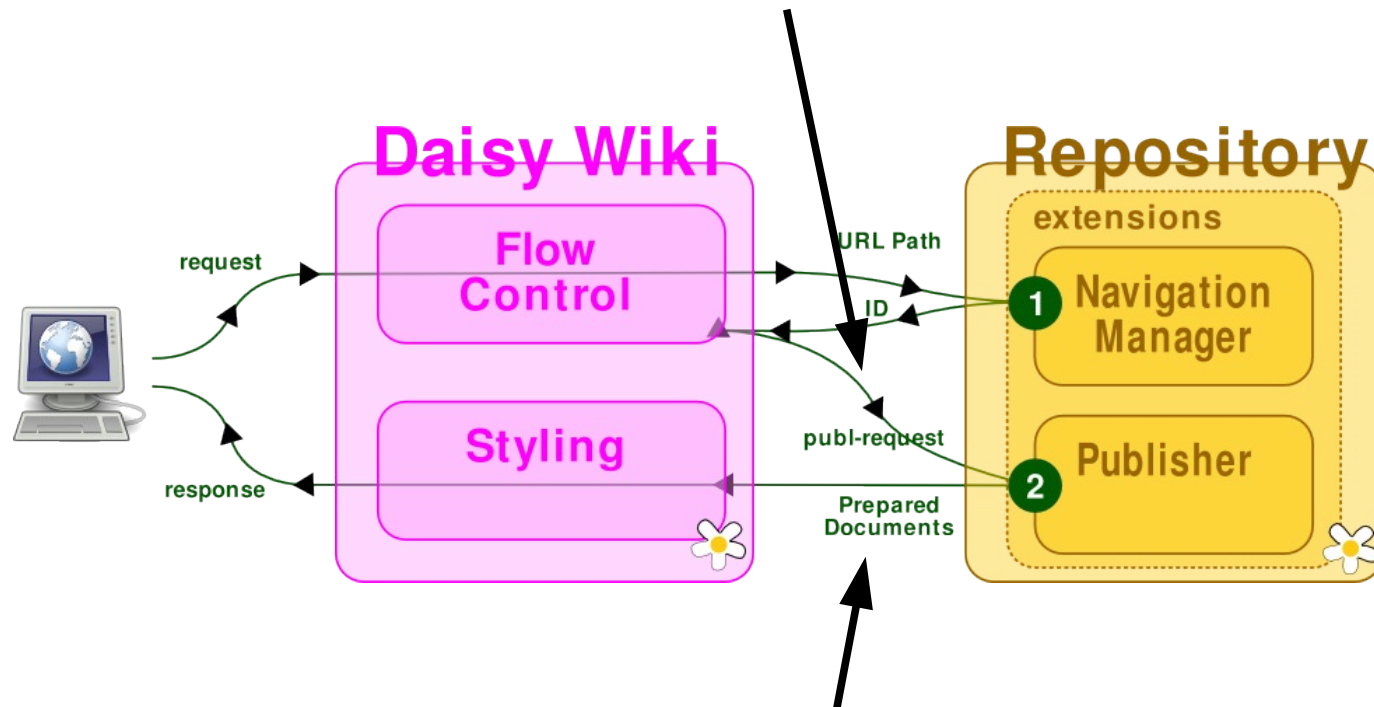
```
<p:preparedDocument id="2">  
  <d:document>  
    [...] <p:daisyPreparedInclude id="3"/> [...]  
  </d:document>  
</p:preparedDocument>
```

```
<p:preparedDocument id="3">  
  <d:document>  
    [...]  
  </d:document>  
</p:preparedDocument>
```

</p:preparedDocuments>

So where were we?

Request asked for a prepared document and a navigation tree



Response contains XML response with expanded/contextualized navigation tree and the prepared document.

Now, it's up to the Daisy Wiki / Cocoon to style the publisher response to a nice HTML page (or a PDF).

Daisy wiki

Cocoon's sitemap

1 Call flow

Pipeline to style an individual document

prepared doc

...

serializer

Pipeline to style the complete result

publisher resp.

...

serializer

"Show Document" flow controller

1. Navigation Path Resolving

2. Build & Execute Publisher Request

4 Publication Helper

1. call publisher

2. parse publisher response

3. foreach prepared doc

3a. Apply document styling

3b. Store result (Request)

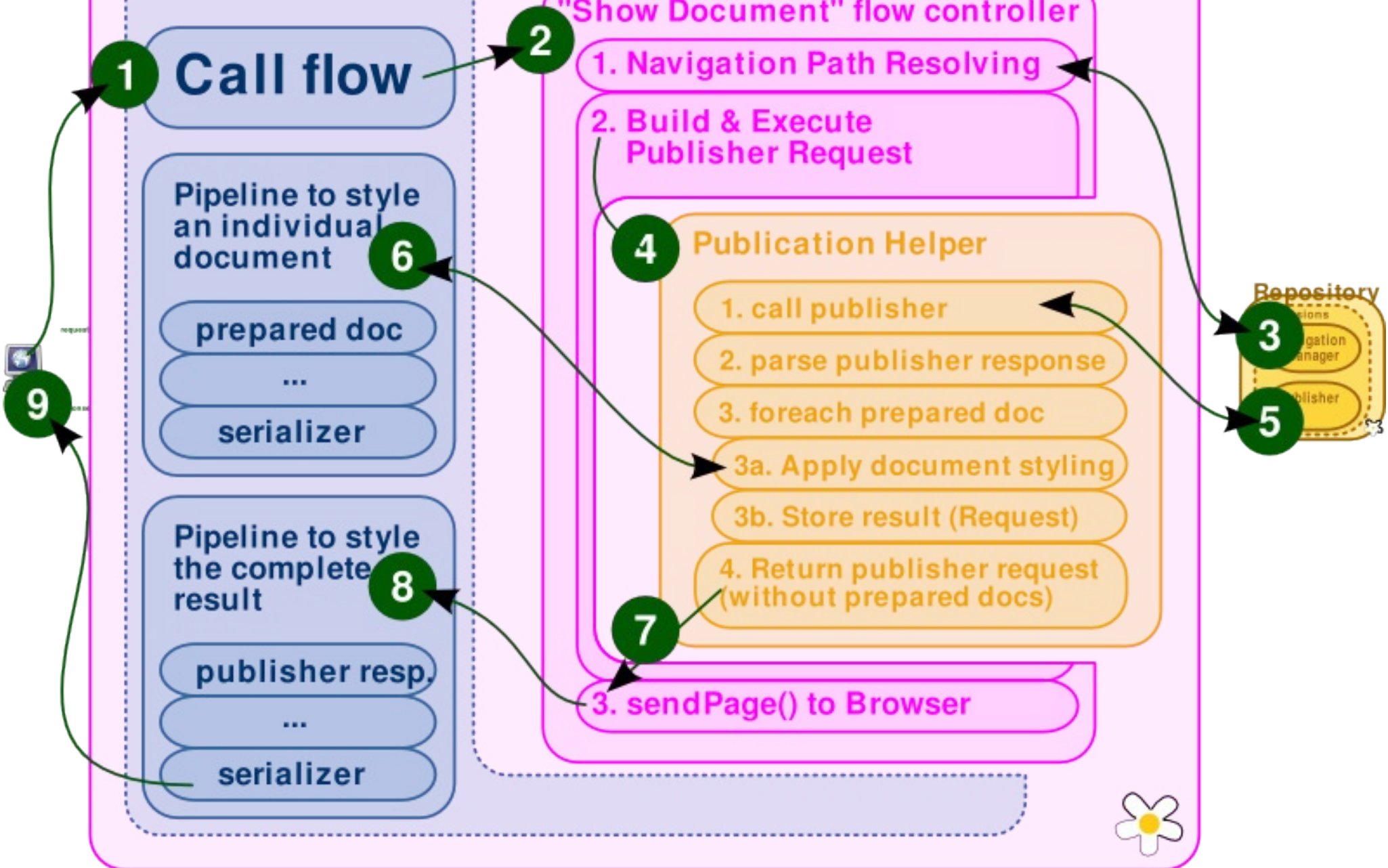
4. Return publisher request (without prepared docs)

3. sendPage() to Browser

Repository

Navigation manager

Publisher



Wiki: publisher response handling

```
<p:publisherResponse>  
  <n:navigationTree>  
    [...]  
  </n:navigationTree>
```

```
<p:document id="...">  
  [unannotated doc XML, available variants,  
  comments, subscription status, ACL info]
```

```
<p:preparedDocuments applyDocumentTypeStyling="true">  
...  
</p:preparedDocuments>
```

```
</p:document>
```

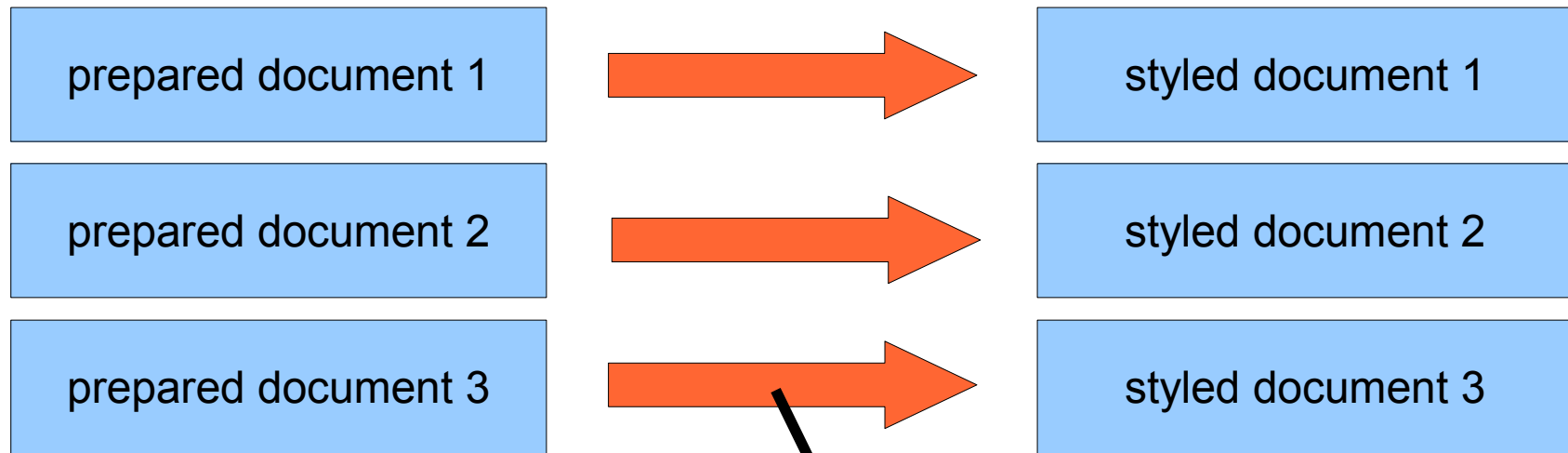
```
</p:publisherResponse>
```

- Prepared documents are extracted from the publisher response and styled, result is stored somewhere else (request attribute).
- An empty preparedDocuments tag is inserted instead, with an ID referring to the results.
- Later the ID can be used to retrieve the styled results

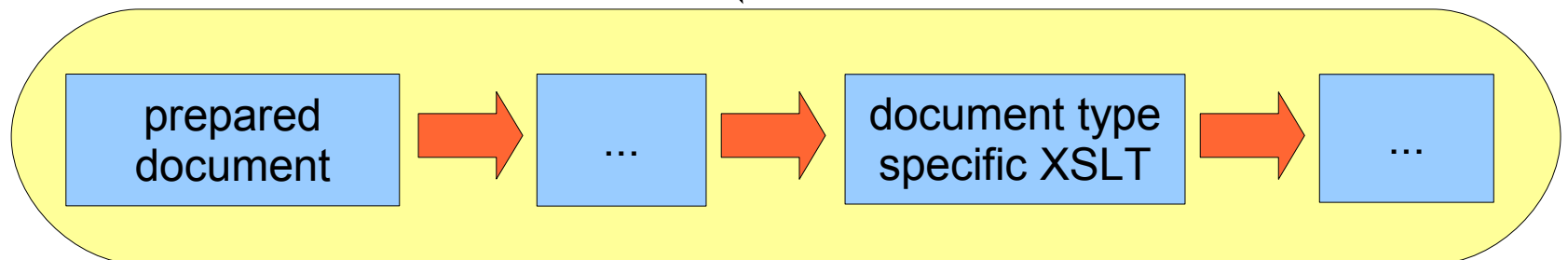
```
<p:preparedDocuments styledResultsId="..." />
```

Styling of individual documents

Each prepared document is styled individually by a Cocoon pipeline



Document styling consists of a pipeline containing a document type specific XSLT



The authoring of a document type specific XSLT is quite simple, as it only needs to be concerned with styling one document.

Final document display

publisher response



documentlayout.xsl



layout.xsl



Merge prepared results

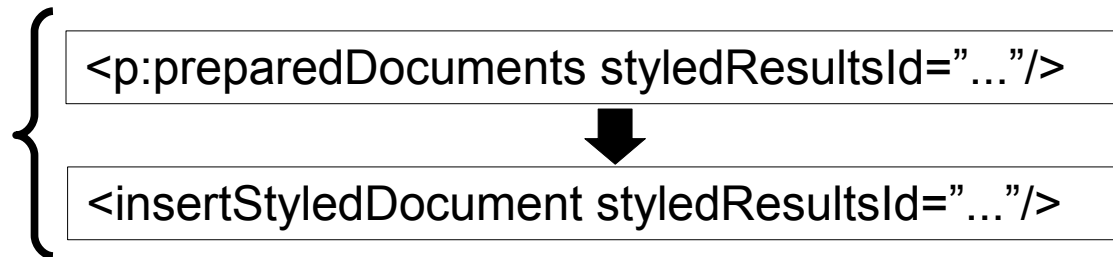


Process non-*"daisy:"* includes



Serialize

this is the publisher response with the preparedDocuments removed



Note that document content doesn't go through layout.xsl (could be a lot of data when having many includes or in case of e.g. document basket or RSS feeds)

Inserts previously styled documents on the location of the insertStyledDocument tag. Included documents are nested in their parent documents.

After this, only streaming transformations follow

Note: a couple of transformers have been left out (e.g. i18n)

Document styling conclusions

The good

- document type specific styling is simple and also works for included documents
- no needless data copying through numerous XSLTs
 - even when rendering an aggregated 1000 documents, only one document at a time goes through an XSL
- document publishing is very fast (without caching), considering everything that is happening
- XML/SAX-based nature of Cocoon leans itself very well to document publishing, SaxBuffer has been very useful
- Everything discussed today is already present since Daisy 1.0

Document styling conclusions *problems / challenges*

- Each document styling needs to produce an embeddable piece of HTML, but for web applications more free layouting is desired.
- XSLT: high barrier to entry (?)
- Repository/wiki separation drove us to publisher requests:
 - nice/easier because it separates data aggregation and publishing stages
 - but also more difficult because users need to decide up front what they need (no pull access to data in templates)
- Cached publishing

Thanks for listening

Questions?



<http://www.daisycms.org/>



<http://www.outerthought.org/>