Apache CXF, Tika and Lucene
The power of search the JAX-RS way

Apache CXF

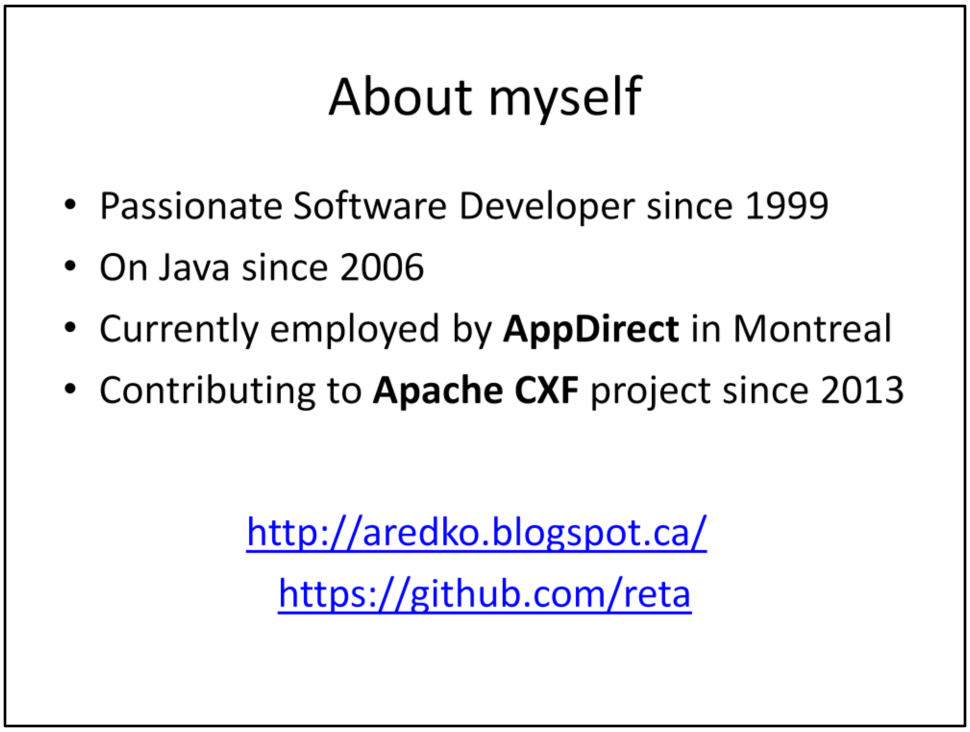
Apache Tika

Lucene

Andriy Redko

AppDirect

Welcome everyone to my talk. This talk is about a quick and easy way of integrating full-fledged search capabilities into typical JAX-RS applications, built on top of excellent Apache CXF framework. This talk assumes that you have some basic knowledge of JAX-RS services and Apache Lucene library.

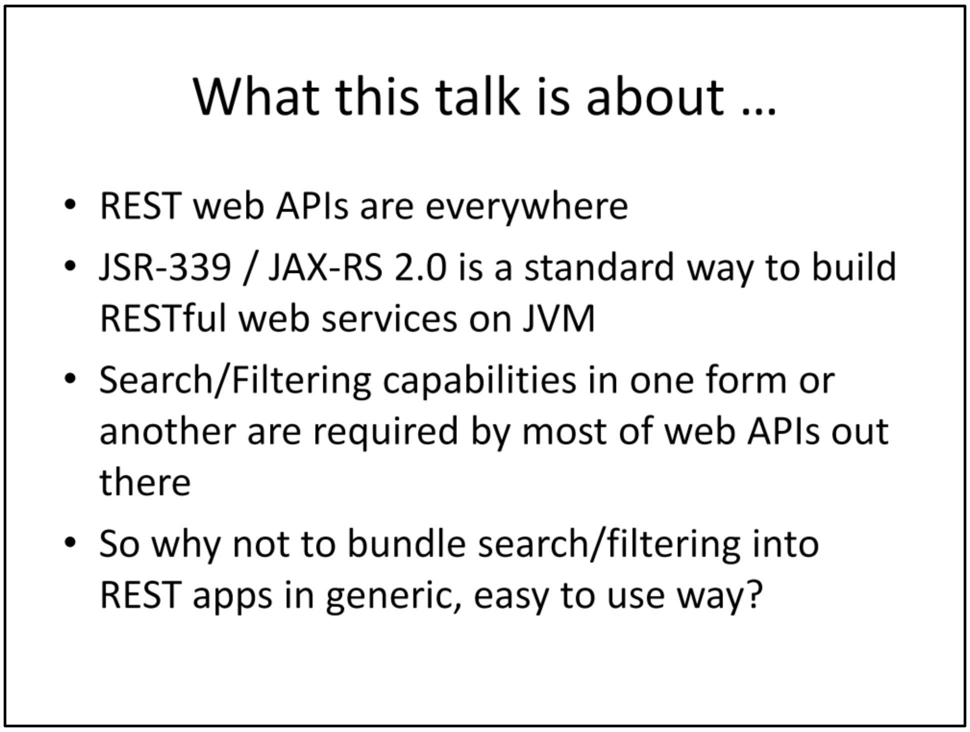


About myself

- Passionate Software Developer since 1999
- On Java since 2006
- Currently employed by **AppDirect** in Montreal
- Contributing to **Apache CXF** project since 2013

http://aredko.blogspot.ca/

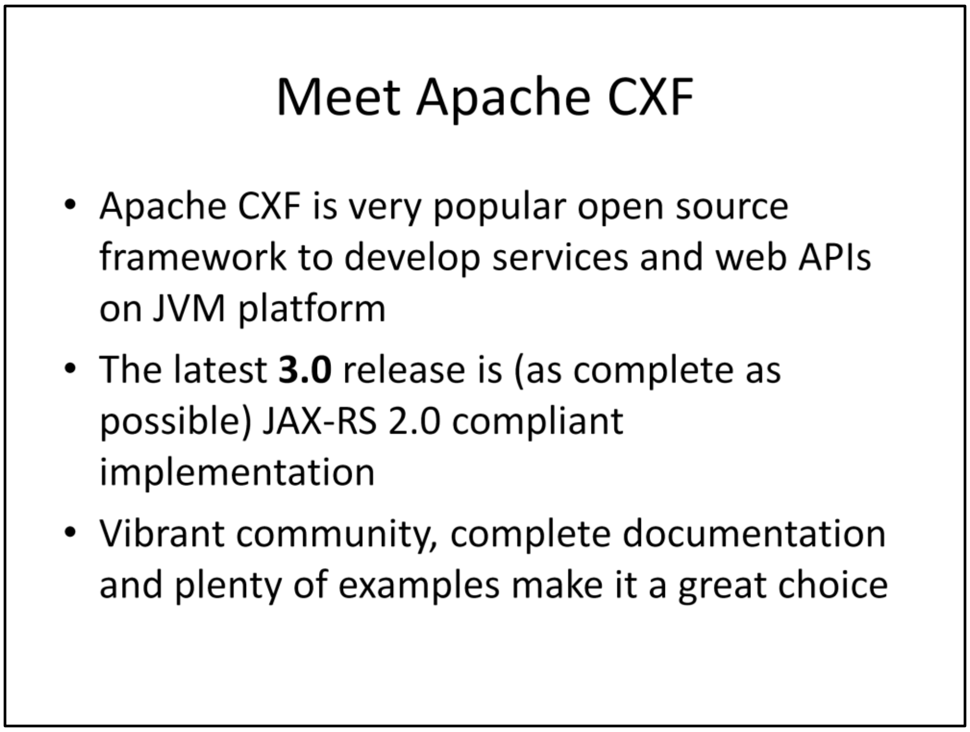
https://github.com/reta



What this talk is about …

- REST web APIs are everywhere
- JSR-339 / JAX-RS 2.0 is a standard way to build RESTful web services on JVM
- Search/Filtering capabilities in one form or another are required by most of web APIs out there
- So why not to bundle search/filtering into REST apps in generic, easy to use way?

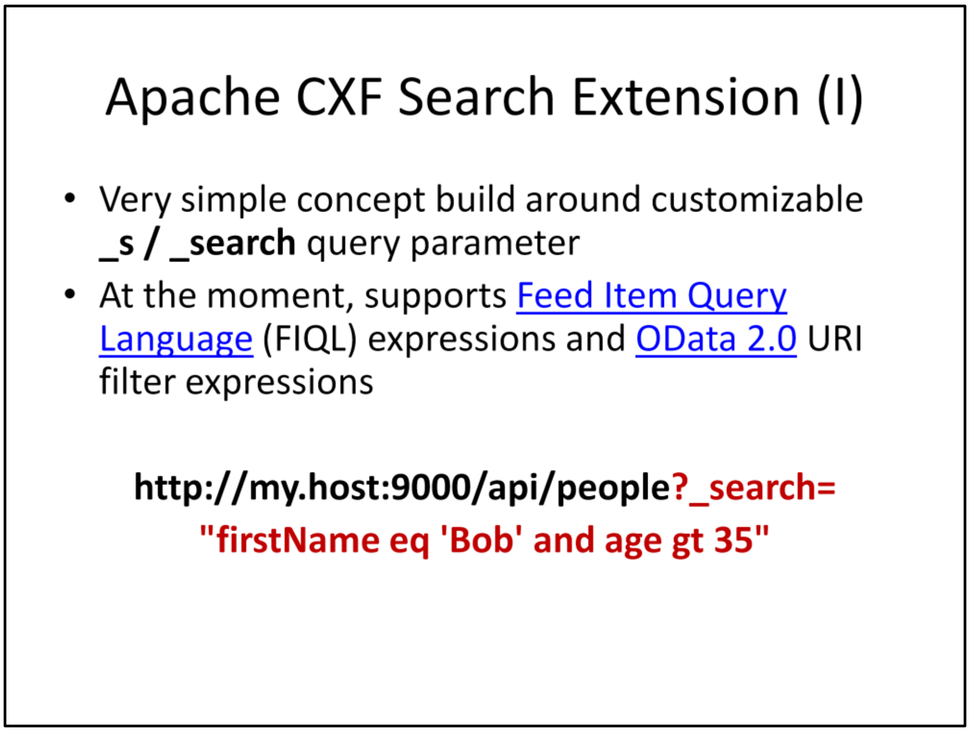
We are going to talk about REST APIs in Java world, more specifically about applications which are using JAX-RS 2.0 / JSR-339 as the foundation. Nowadays, it is quite rare to encounter an API which does not support any kind of search or filtering functionality over the resources it manages (for example, users, customers, invoices, companies, …). For sure, the needs of every API are quite domain specific, however in the essence it narrows down to very similar search / filtering implementations. Why not to come up with a generic, JAX-RS friendly implementation of this feature so any API could immediately benefit from that?



Meet Apache CXF

- Apache CXF is very popular open source framework to develop services and web APIs on JVM platform
- The latest **3.0** release is (as complete as possible) JAX-RS 2.0 compliant implementation
- Vibrant community, complete documentation and plenty of examples make it a great choice

This is where Apache CXF comes on the rescue. Apache CXF is as complete as possible JAX-RS 2.0 compliant but because Apache has no access to TCK, the “full compliance” cannot be claimed. The project is very mature and is actively evolving, adding more and more features with every single release.

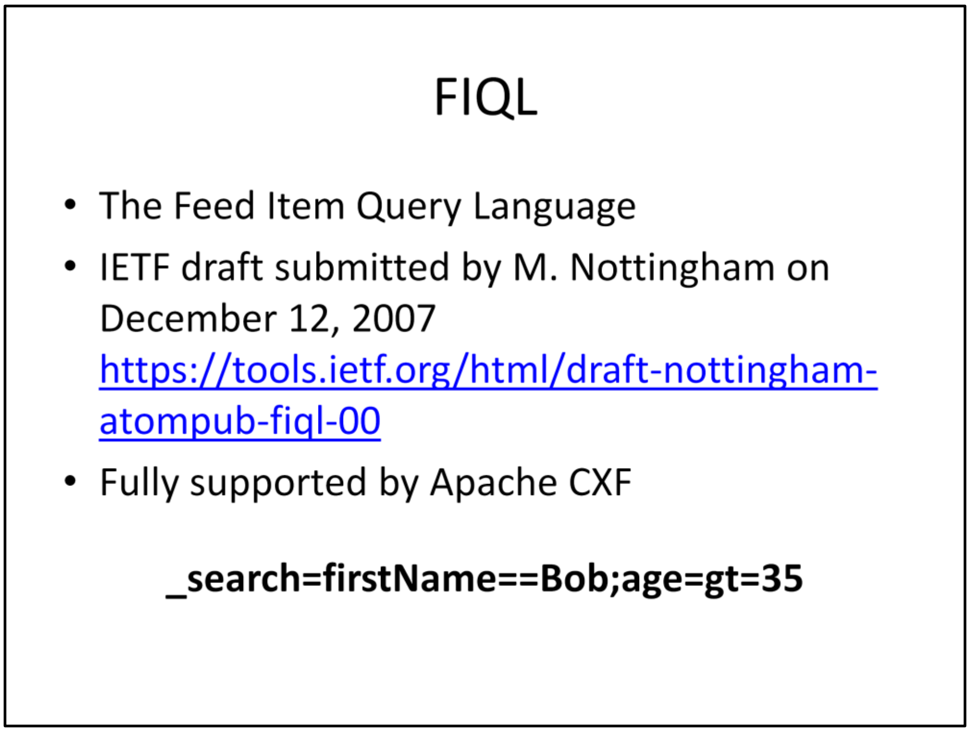


Apache CXF Search Extension (I)

- Very simple concept build around customizable **_s / _search** query parameter
- At the moment, supports [Feed Item Query Language](#) (FIQL) expressions and [OData 2.0](#) URI filter expressions

http://my.host:9000/api/people?_search= "firstName eq 'Bob' and age gt 35"

Apache CXF search extension is a generic search feature implementation, provided and exposed in JAX-RS friendly way. It is build around additional query string parameter which contains an expressions to perform the search over REST resources. Currently, the two expression dialects are support: FIQL and OData 2.0 URI filters. Here is a quick example to taste.

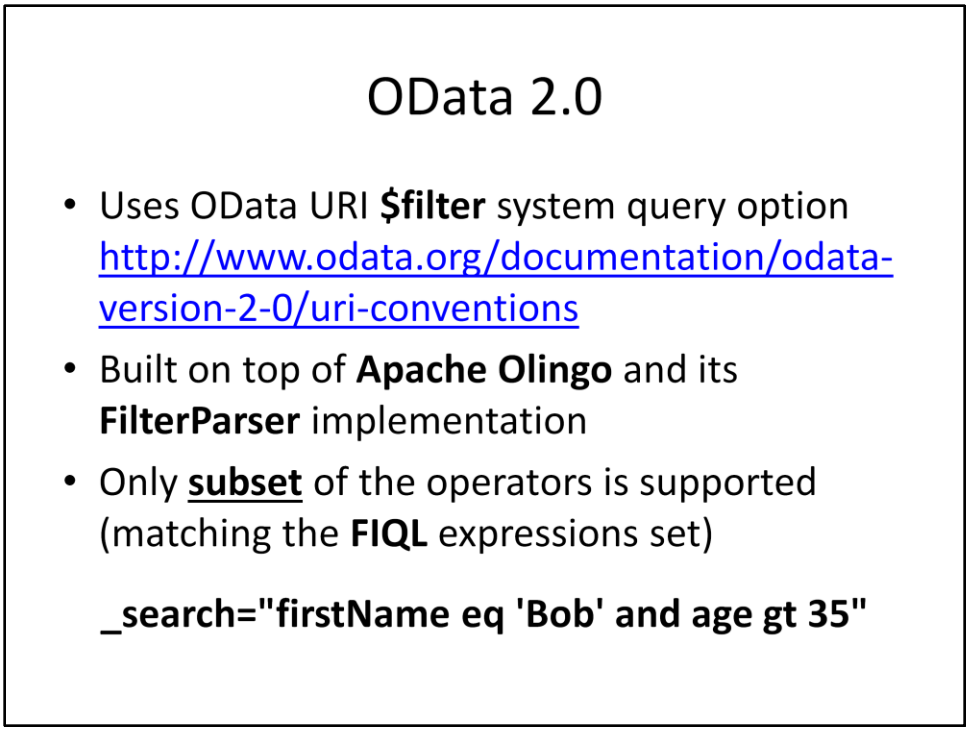


FIQL

- The Feed Item Query Language
- IETF draft submitted by M. Nottingham on December 12, 2007 https://tools.ietf.org/html/draft-nottingham-atompub-fiql-00
- Fully supported by Apache CXF

_search=firstName==Bob;age=gt=35

Few words about FIQL. It is quite simple and uses the ‘equal’ sign to denote operators, ‘;’ as and condition, ‘,’ as or condition and brackets to group expressions.



OData 2.0

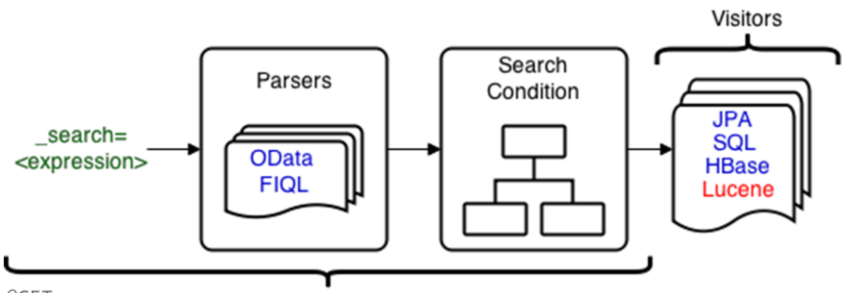
- Uses OData URI **$filter** system query option http://www.odata.org/documentation/odata-version-2-0/uri-conventions
- Built on top of **Apache Olingo** and its **FilterParser** implementation
- Only **subset** of the operators is supported (matching the **FIQL** expressions set)

_search="firstName eq 'Bob' and age gt 35"

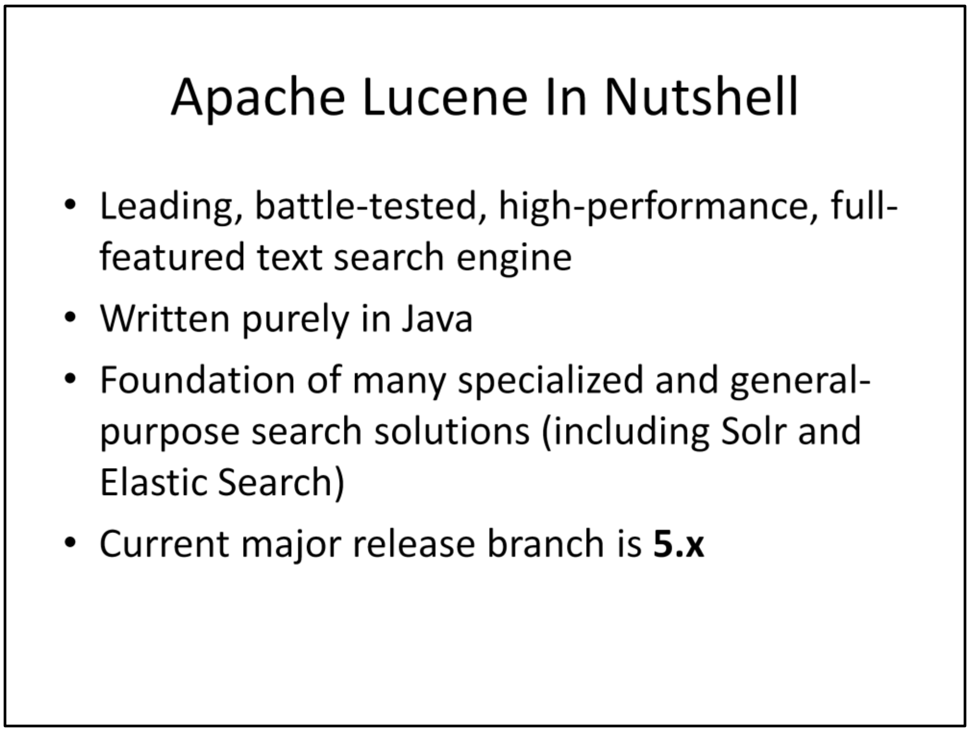
OData is a very well known and widely used RESTful data access protocol (current version is 4). It has a lot of things to offer, but one of the very interesting features is URI query options (like **$orderby**, **$filter**, …). Apache CXF search extension supports **$filter** query option which allows to use subset of OData filtering expressions and operators. At the moment, OData 2.0 version is supported however OData 4.0 is on the list as well.

Apache CXF Search Extension (II)

- Under the hood …



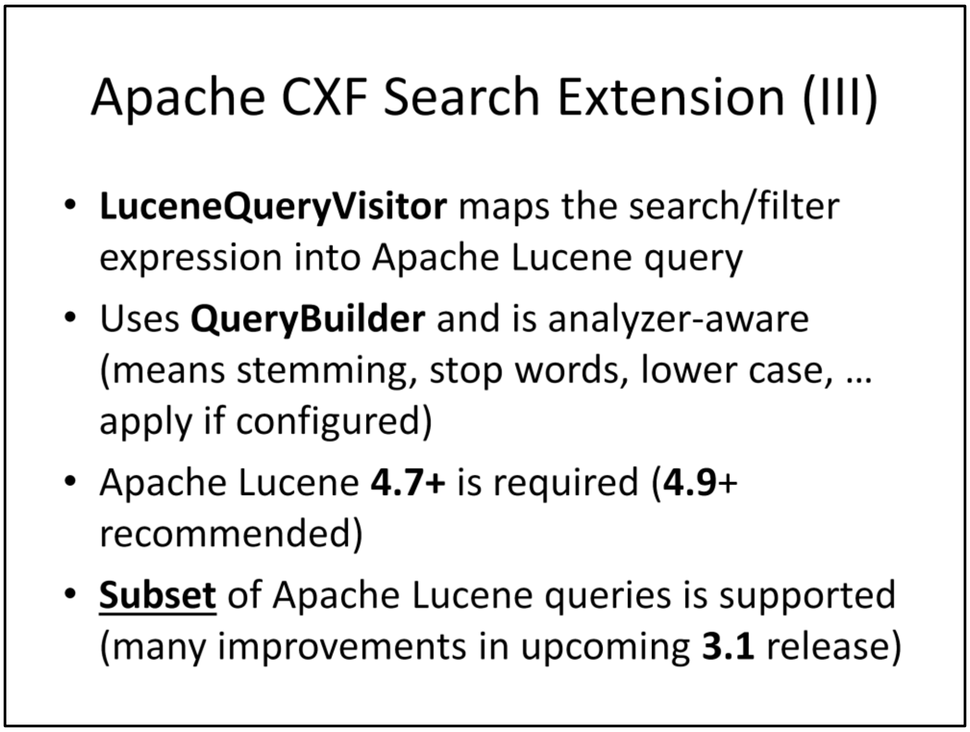
```
@GET
@Produces( { MediaType.APPLICATION_JSON  } )
public Response search(@Context SearchContext context) {
    ...
}
```



Apache Lucene In Nutshell

- Leading, battle-tested, high-performance, full-featured text search engine
- Written purely in Java
- Foundation of many specialized and general-purpose search solutions (including Solr and Elastic Search)
- Current major release branch is **5.x**

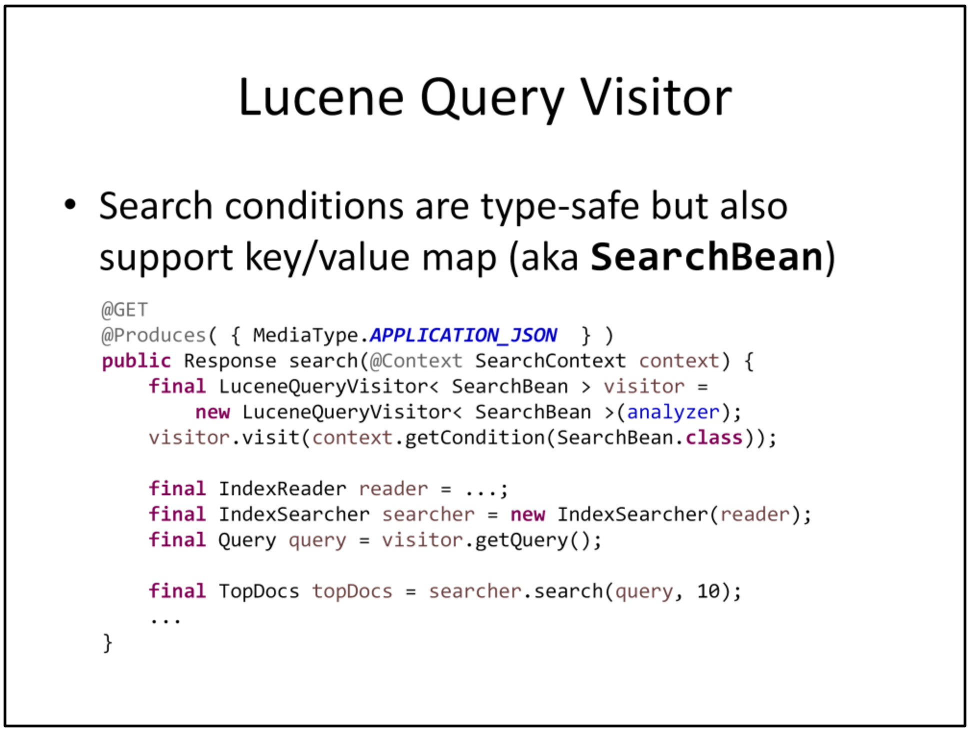
Apache Lucene is the search library for JVM-based applications and services.



Apache CXF Search Extension (III)

- **LuceneQueryVisitor** maps the search/filter expression into Apache Lucene query
- Uses **QueryBuilder** and is analyzer-aware (means stemming, stop words, lower case, … apply if configured)
- Apache Lucene **4.7+** is required (**4.9**+ recommended)
- **<u>Subset</u>** of Apache Lucene queries is supported (many improvements in upcoming **3.1** release)

Knowing how good and popular Apache Lucene, it comes with no surprise that Apache CXF search extension has a visitor implementation which converts the search / filter expression (FIQL, OData) into full-fledge Apache Lucene query. There are so many different types of queries which Lucene supports that only a subset is being support by the visitor. We are going to talk about all those queries shortly.

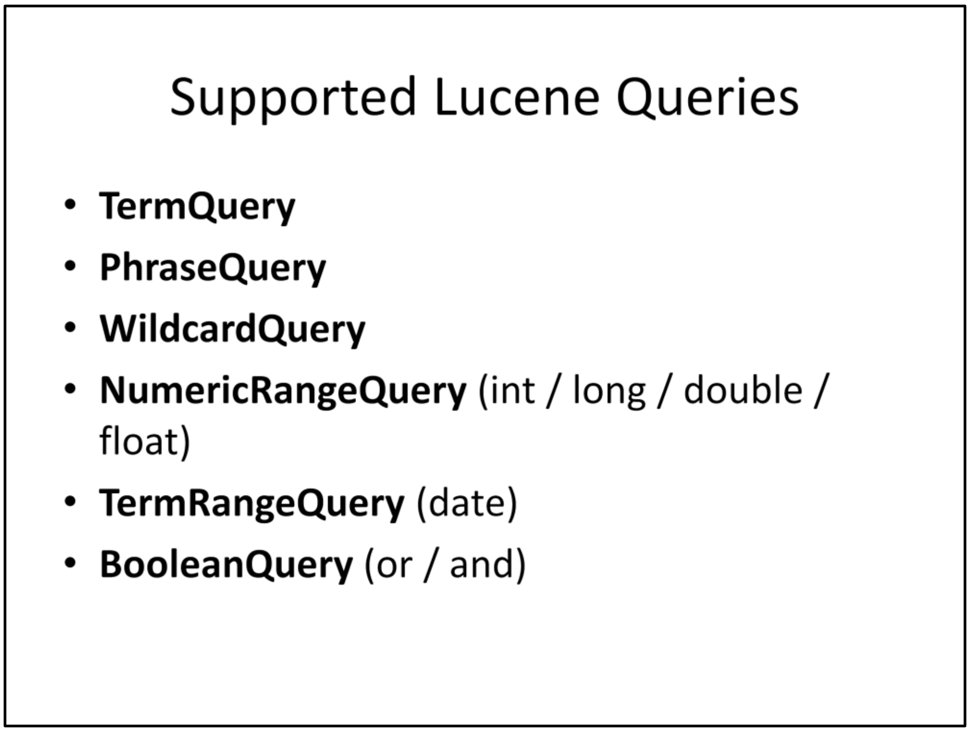


Lucene Query Visitor

- Search conditions are type-safe but also support key/value map (aka **SearchBean**)

```
@GET
@Produces( { MediaType.APPLICATION_JSON  } )
public Response search(@Context SearchContext context) {
    final LuceneQueryVisitor< SearchBean > visitor =
        new LuceneQueryVisitor< SearchBean >(analyzer);
    visitor.visit(context.getCondition(SearchBean.class));

    final IndexReader reader = ...;
    final IndexSearcher searcher = new IndexSearcher(reader);
    final Query query = visitor.getQuery();

    final TopDocs topDocs = searcher.search(query, 10);
    ...
}
```

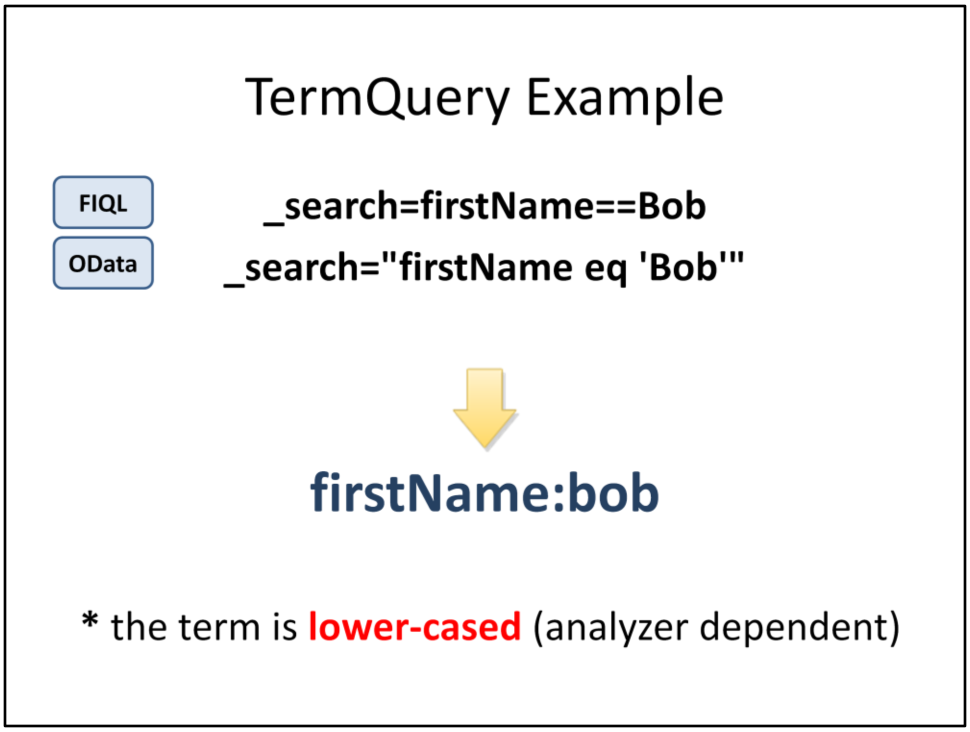
The search conditions implementation is type-safe and can be applied to arbitrary classes and collections (internally the respective property values are retrieved through reflection calls). But regular key/value map (**SearchBean**) is also supported which makes it very easy to project the search conditions to Lucene queries, JPA criteria, SQL, HBASE filters, … without creating tons of supplementary classes.



Supported Lucene Queries

- **TermQuery**
- **PhraseQuery**
- **WildcardQuery**
- **NumericRangeQuery** (int / long / double / float)
- **TermRangeQuery** (date)
- **BooleanQuery** (or / and)

Apache Lucene has a very rich set of query types and their compositions. But Apache CXF search extension uses the subset of the most commonly used ones which nonetheless is quite enough to cover the demands of many applications. On the next couple of slides we are going to see the examples of how FIQL/OData filters are mapped into one of those queries.



TermQuery Example

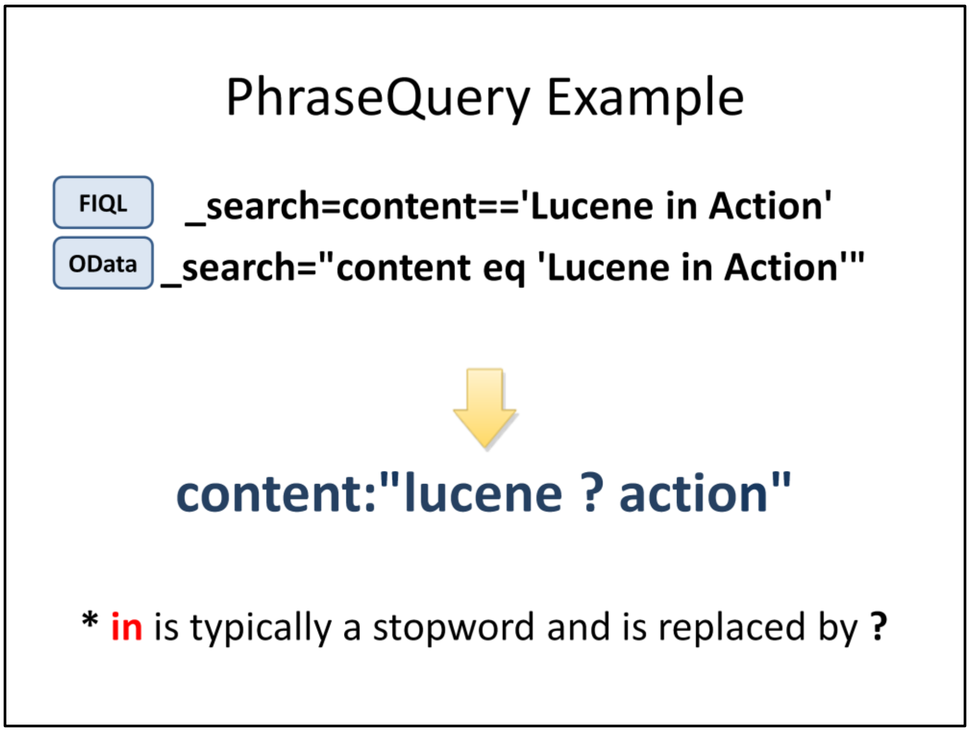
FIQL `_search=firstName==Bob`

OData `_search="firstName eq 'Bob'"`

firstName:bob

* the term is **lower-cased** (analyzer dependent)

TermQuery is the most basic one and looks for documents which contain the single term. In this example, the **StandardAnalyzer** has been used which leads to conversion of the term into lower case while building Apache Lucene query.



PhraseQuery Example

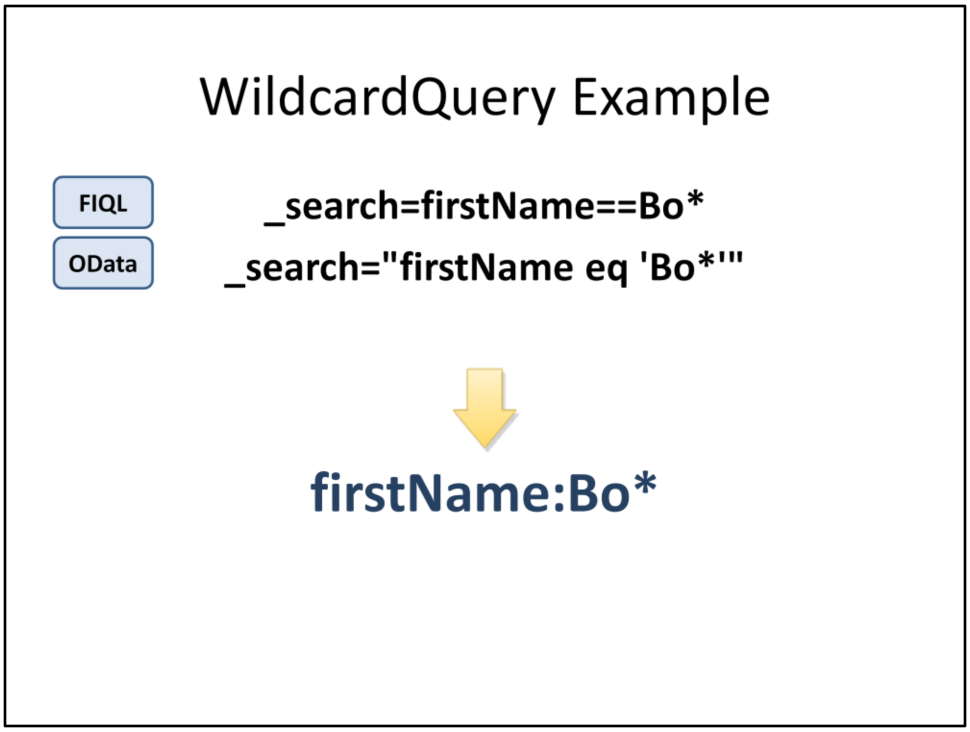
FIQL `_search=content=='Lucene in Action'`

OData `_search="content eq 'Lucene in Action'"`

content:"lucene ? action"

* in is typically a stopword and is replaced by **?**

PhraseQuery is somewhat similar to TermQuery but instead of single term it uses a phrase as a search criteria. In this example, the **StandardAnalyzer** has been used which leads to stop word elimination (and lower-casing) while building Apache Lucene query.



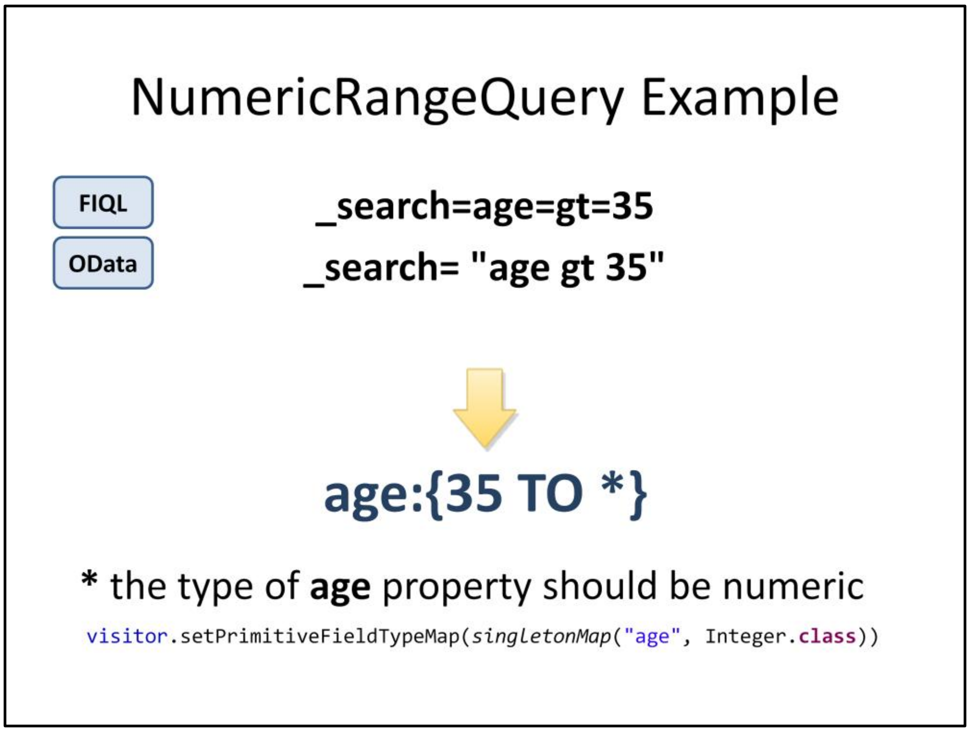
WildcardQuery Example

FIQL `_search=firstName==Bo*`

OData `_search="firstName eq 'Bo*'"`

firstName:Bo*

WildcardQuery searches by partial matches (leading or trailing wildcards). In contrast to other types of queries, they are build “as-is” (analyzer independent) and could be really inefficient.



NumericRangeQuery Example

FIQL
OData

_search=age=gt=35

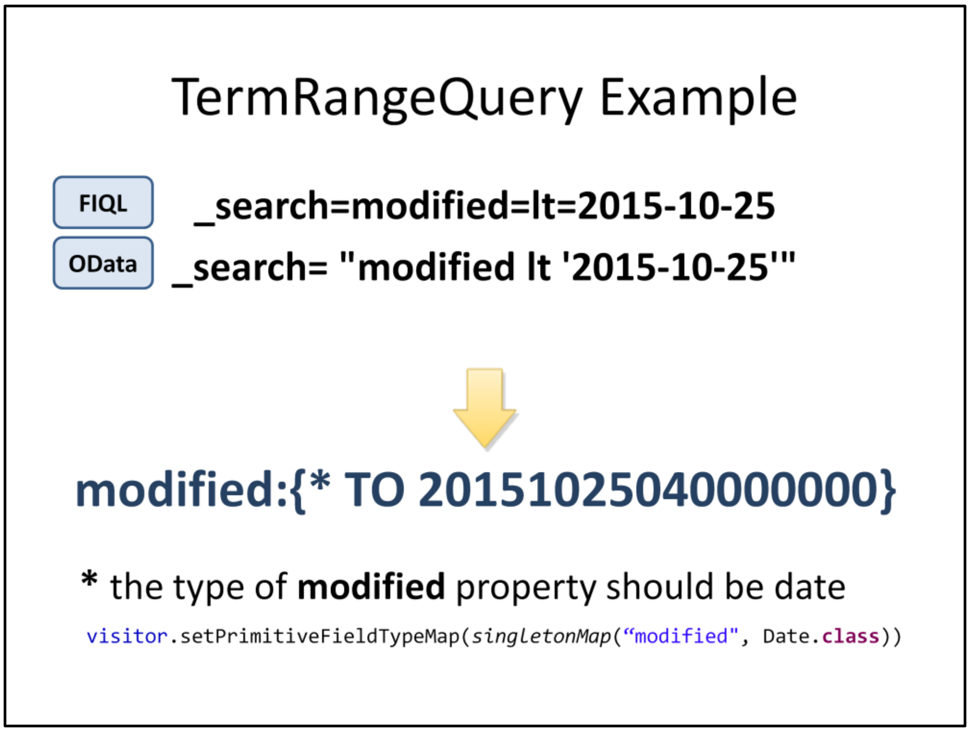
_search= "age gt 35"

age:{35 TO *}

* the type of **age** property should be numeric

```
visitor.setPrimitiveFieldTypeMap(singletonMap("age", Integer.class))
```

NumericRangeQuery is used to perform efficient search for numeric field ranges. As LuceneQueryVisitor is generic and does not have the intrinsic knowledge about types of properties used in FIQL/OData filters, it needs to be hinted about numeric nature and exact types of those. Luckly, it is very easy to do using **`primitiveFieldTypeMap.`**



TermRangeQuery Example

FIQL **_search=modified=lt=2015-10-25**

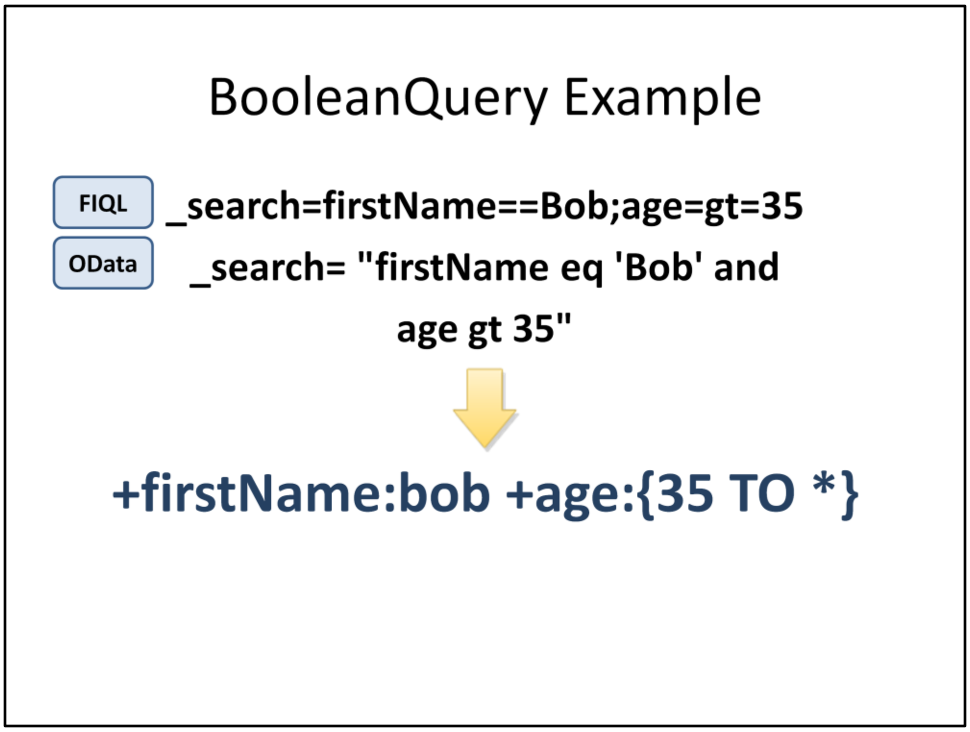
OData **_search= "modified lt '2015-10-25'"**

modified:{* TO 20151025040000000}

* the type of **modified** property should be date

```
visitor.setPrimitiveFieldTypeMap(singletonMap("modified", Date.class))
```

Similarly to NumericRangeQuery, TermRangeQuery is used to perform efficient search across term ranges. It is currently used to search the date fields. The LuceneQueryVisitor also needs to be hinted about temporal nature of the properties using **`primitiveFieldTypeMap`**.



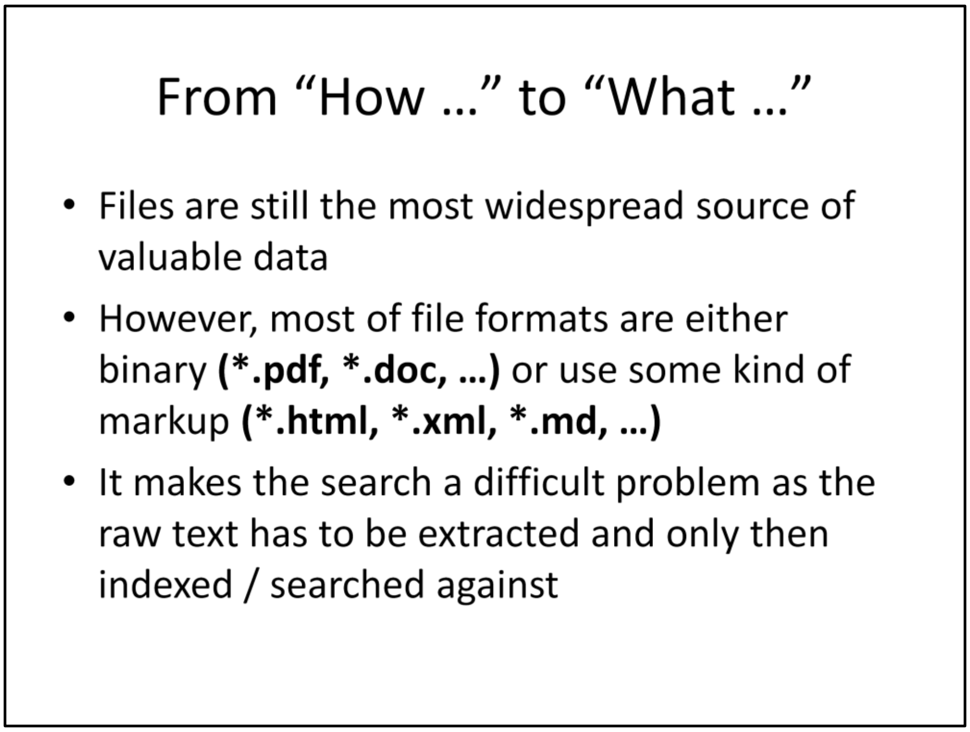
BooleanQuery Example

FIQL `_search=firstName==Bob;age=gt=35`

OData `_search= "firstName eq 'Bob' and age gt 35"`

+firstName:bob +age:{35 TO *}

BooleanQuery is the compositional one, which is used to combine other queries with **and / or** conditions.

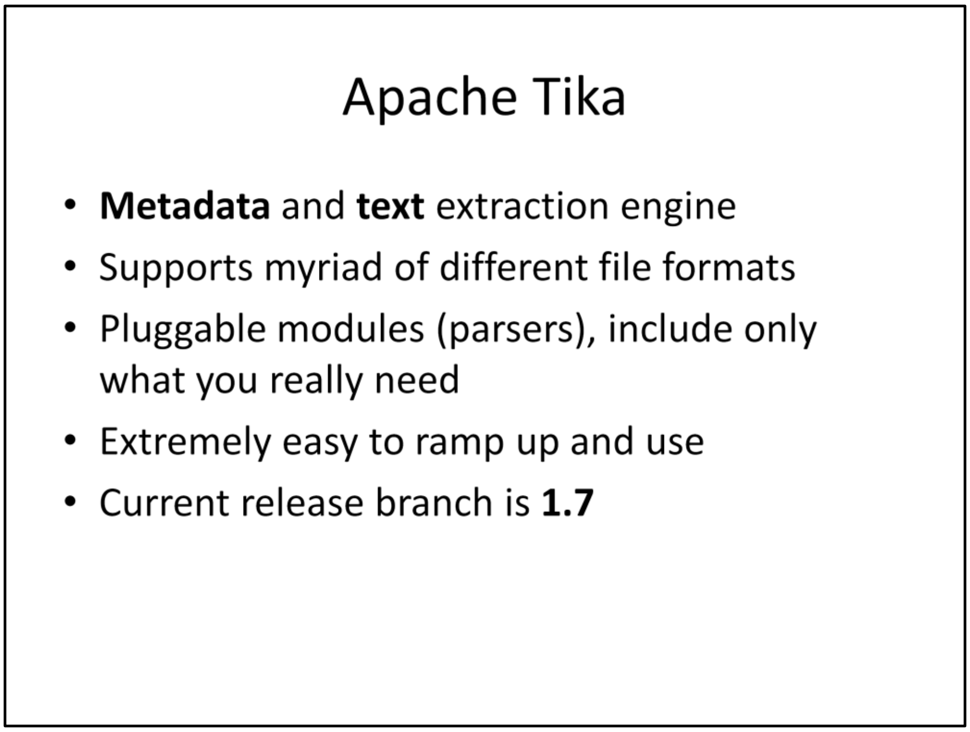


From “How …” to “What …”

- Files are still the most widespread source of valuable data
- However, most of file formats are either binary **(*.pdf, *.doc, …)** or use some kind of markup **(*.html, *.xml, *.md, …)**
- It makes the search a difficult problem as the raw text has to be extracted and only then indexed / searched against

So far we talked about search, different types of queries and their combination. But every search query should be run against the index which is built using the real data. We are switching gears a bit from the “How to search …” topic to “What data we are going to search against …”.

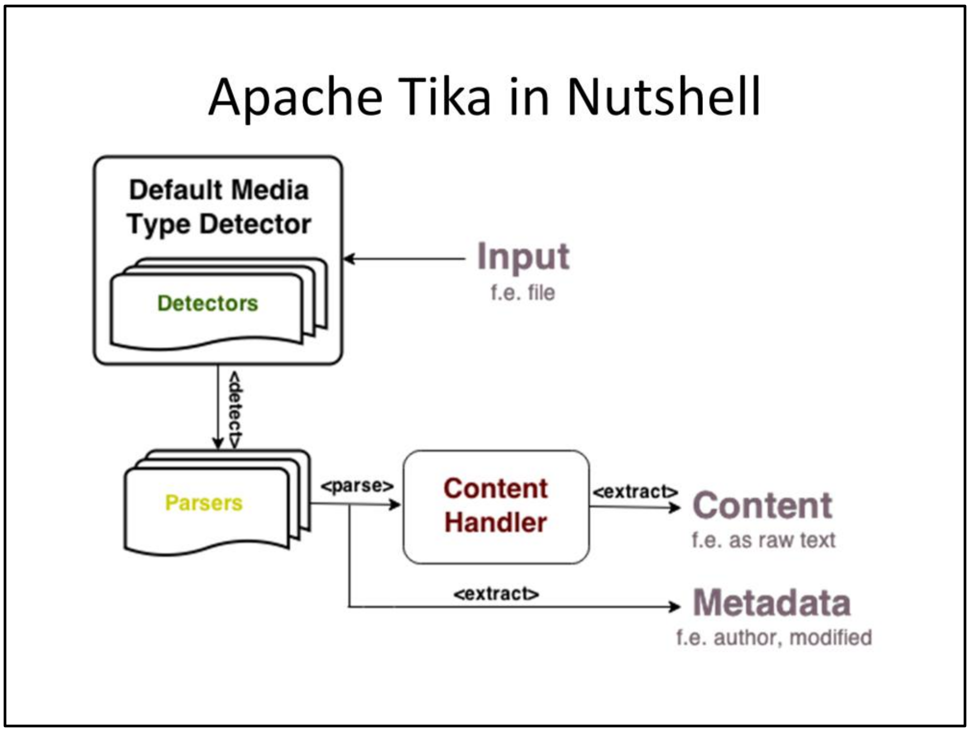
In general, indexing raw binary files as-is is not very good idea. And though files with markup (like XML, HTML) are human-readable and could be indexed as-is, they introduce a lot of noise (tags) which should better be removed before polluting the index.



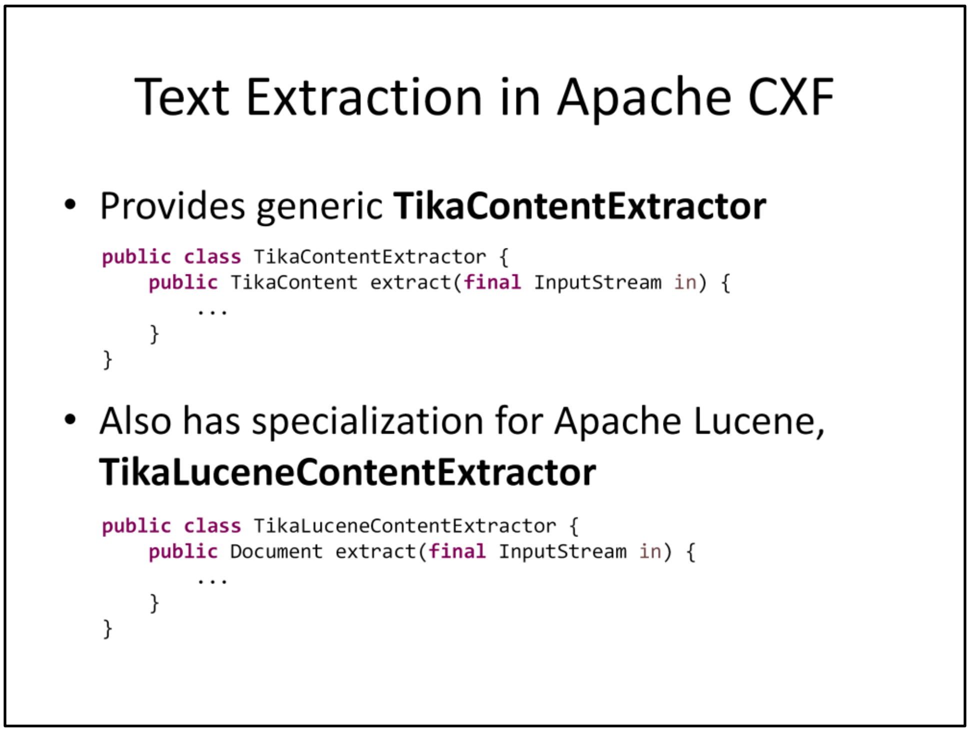
Apache Tika

- **Metadata** and **text** extraction engine
- Supports myriad of different file formats
- Pluggable modules (parsers), include only what you really need
- Extremely easy to ramp up and use
- Current release branch is **1.7**

Apache Tika is metadata and text extraction engine. Its features are not limited only by extraction but this is what we are going to look closely at. The list of supported file formats is very impressive. Due to pluggable parser architecture, you are not required to bundle everything, pick the ones you really need now and easily add more later on.



Media Type Detector is a powerful tool which tries its best to detect the real file format. It uses a combination of techniques, not relying solely on file extension. Once the media type is detected, the respective parser could be pin-pointed (if it is available). Most of the parsers extract content (often just raw text) and also metadata (author, modification/creation dates, number of pages, …).


Text Extraction in Apache CXF

- Provides generic **TikaContentExtractor**

```
public class TikaContentExtractor {
    public TikaContent extract(final InputStream in) {
        ...
    }
}
```

- Also has specialization for Apache Lucene, **TikaLuceneContentExtractor**

```
public class TikaLuceneContentExtractor {
    public Document extract(final InputStream in) {
        ...
    }
}
```

In Apache CXF, the text and metadata extraction using Apache Tika is encapsulated inside **TikaContentExtractor** class (with **TikaContent** class being generic representation of raw text and metadata, extracted from the file). **TikaLuceneContentExtractor** is built on top of **TikaContentExtractor** and returns prepared Apache Lucene document, which could be indexed right away (however document could be customized/modified before).

And finally, indexing …

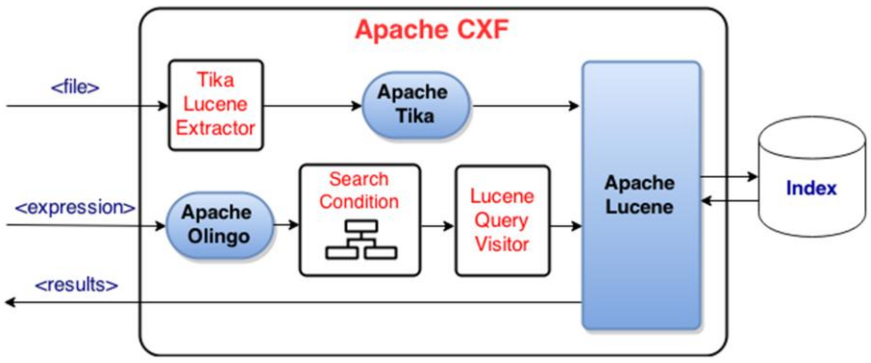
- The **text** and **metadata** extracted from the file could be added straight to Lucene index

```
final TikaLuceneContentExtractor extractor =
    new TikaLuceneContentExtractor(new PDFParser());

final Document document = extractor.extract(in);
final IndexWriter writer = ...;

try {
    writer.addDocument(document);
    writer.commit();
} finally {
    writer.close();
}
```

Demo

https://github.com/reta/ApacheConNA2015

Demo: Gluing All Parts Together …



Apache CXF Search Extension (IV)

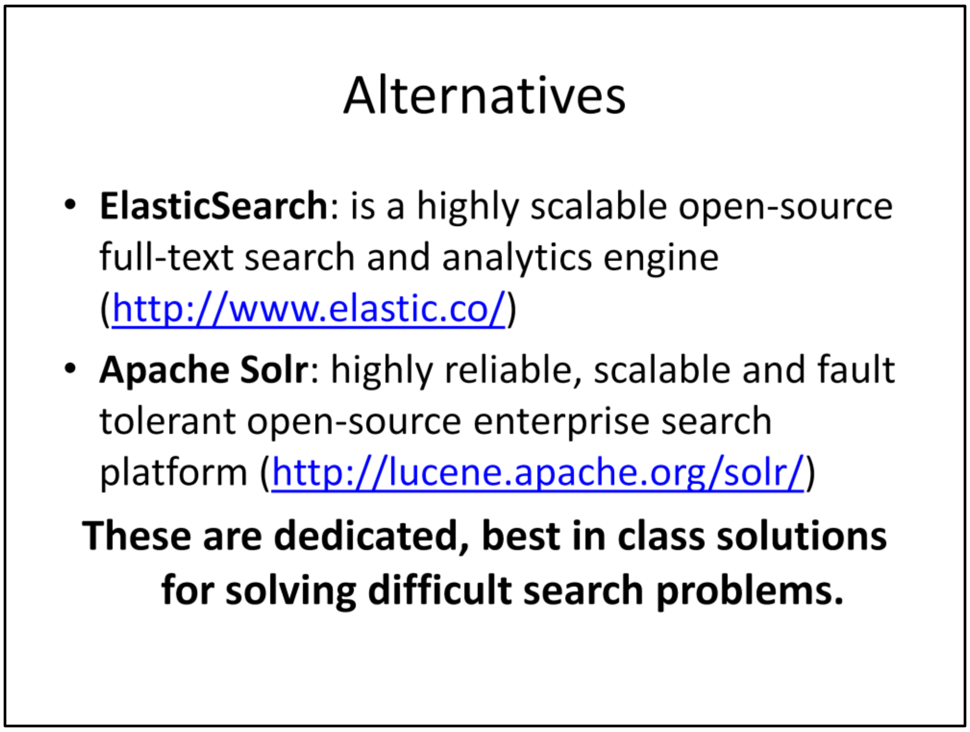
- Configuring expressions parser

  search.parser.class=ODataParser

  search.parser=new ODataParser()
- Configuring query parameter name

  search.query.parameter.name=$filter
- Configuring date format

  search.date-format=yyyy/MM/dd



Alternatives

- **ElasticSearch**: is a highly scalable open-source full-text search and analytics engine (http://www.elastic.co/)
- **Apache Solr**: highly reliable, scalable and fault tolerant open-source enterprise search platform (http://lucene.apache.org/solr/)

These are dedicated, best in class solutions for solving difficult search problems.

[Draft]

Useful links

- http://cxf.apache.org/docs/jax-rs-search.html
- http://lucene.apache.org/
- http://tika.apache.org/
- http://olingo.apache.org/

- http://aredko.blogspot.ca/2014/12/beyond-jax-rs-spec-apache-cxf-search.html

Thank you!

Many thanks to **Apache Software Foundation** and **AppDirect** for the chance to be here