



# Living with *Linked Data*

Usage in a "live" production setting



# About NXP Semiconductors

- Net Revenue: \$4.82 billion (2013)
- Established: 2006 (former division of Philips)
- 55+ years of experience in semiconductors
- Headquarters: Eindhoven, The Netherlands
- Businesses
  - High Performance Mixed Signal: Automotive, Identification, Infrastructure & Industrial, Portable & Computing
  - Standard Products
- <http://www.nxp.com>



# About Semaku

- › Established: 2013
- › Headquarters: Eindhoven, The Netherlands
- › Consultancy services and software products
- › Areas of expertise
  - Product Information Management
  - Linked Data
  - Content Strategy
- › <http://semaku.com>
- › [info@semaku.com](mailto:info@semaku.com)

**SEM**AKU



# Linked Data at NXP

## So far, so good...

- › Canonical data source for marketing master data
  - Provide the linking data
  - Unambiguous identifiers
- › Using stored SPARQL SELECT queries to expose REST APIs
  - Results in tabular XML, JSON or CSV/TSV format
  - Easy to manage queries
  - Extremely quick to set up new APIs (15 – 30 mins)
- › Able to answer previously unanswerable questions
- › Minimal investment compared to traditional BW/BI projects

# But we want more

- Faster, faster, more, more...
  - Daily RDF dumps not frequent enough
  - Do not contain all the data
- Phase out legacy systems
  - Reduce maintenance effort/costs
  - Manage data natively as RDF



Adding a new data source

# PRODUCT LIFECYCLE MANAGEMENT



# What is product lifecycle management

- › “product lifecycle management (PLM) is the process of managing the entire lifecycle of a product ... PLM integrates people, data, processes and business systems and provides a product information backbone for companies and their extended enterprise.” – [Wikipedia](#)
- › Uses same basic EAV model that RDF is built upon
- › Massively interlinked

# Rationale

- PLM system is “closed”
  - No web services / API
  - Weekly reports with flattened data (CSV)
  - Changes sent as incremental messages over ESB (ASCII/CSV/XML)
- Need to migrate from a legacy ASCII message to new XML message
  - Flat message
  - Implicit links
- We need more data, but didn’t know exactly what
  - Wanted to be able to flexibly query the data leveraging the conceptual structure NOT the explicit structure in a particular serialization
- Point-to-point integration does not make sense
  - Specific mappings would have to be defined in the channel
  - Not robust and more effort to maintain



# Why use Linked Data

- › Consensus that a canonical model is a long term goal
- › Challenge to define RDB and XML schema
  - Many classes
  - Reuse of properties across classes
  - Heterogeneous data
- › Prototype with querying XML message using XSLT, XPath
  - Over 1 minute to build explicit tree structure from single 'root' item
  - A single message can contain over 1000 root items
  - Does not allow to traverse links in reverse direction
- › Prototype with generating RDF (Turtle) and query with SPARQL
  - 200 ms to convert message AND run query (Jena in memory)
  - Using arbitrary length property paths

# Modeling as RDF

- › Bottom-up approach
- › Model is already defined in source system
- › Start with the instance data
- › Data first, model later
  - RDF Schema is used to describe the data, not constrain
  - We didn't even get round to making the RDF Schema yet!

# Mapping to RDF (1)

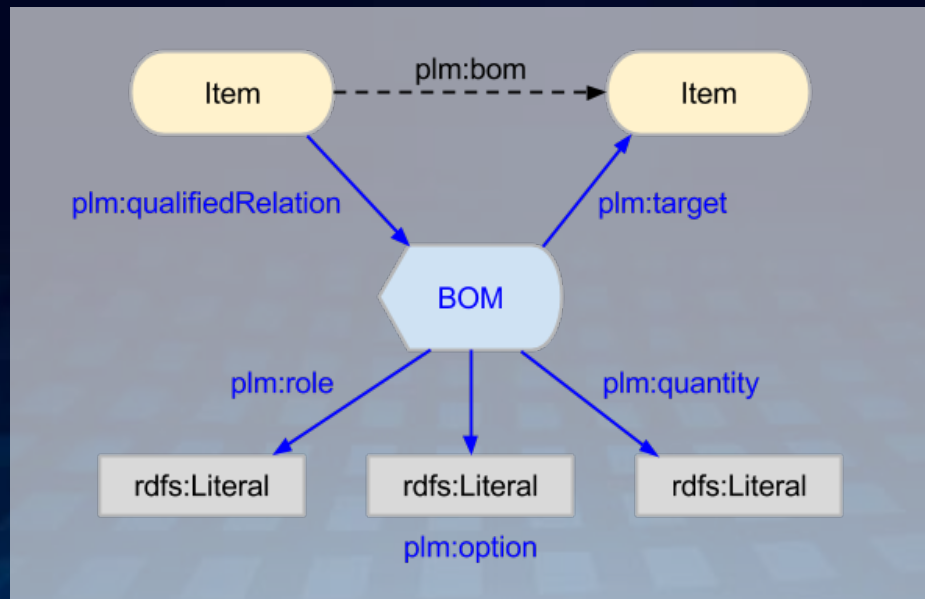
- › Map class and property names converted to CamelCase in URIs
- › Follow convention that classes start with initial caps, properties lowercase

Name	Slug	CURIE
Sales Item	SalesItem	plm:SalesItem
CEPT	Cept	plm:Cept
Orderable Part Number	orderablePartNumber	plm:orderablePartNumber
Status	status	plm:status

- › For instances we combine the class name and key id to build URI
  - `http://example.com/id/{class}/{id}`
- Result
  - `http://example.com/id/salesItem/1234567890`
- › Guarantees uniqueness

# Mapping to RDF (2)

- › Values do not have a datatype in the source XML
- › By default map values to plain literals
- › Specific match (regex) for timestamps and map to `xsd:dateTime`
- › Could be extended in future
- › Qualified links get reified



# Example item description



Visualization created with W3C RDF Validator (<http://www.w3.org/RDF/Validator/>)

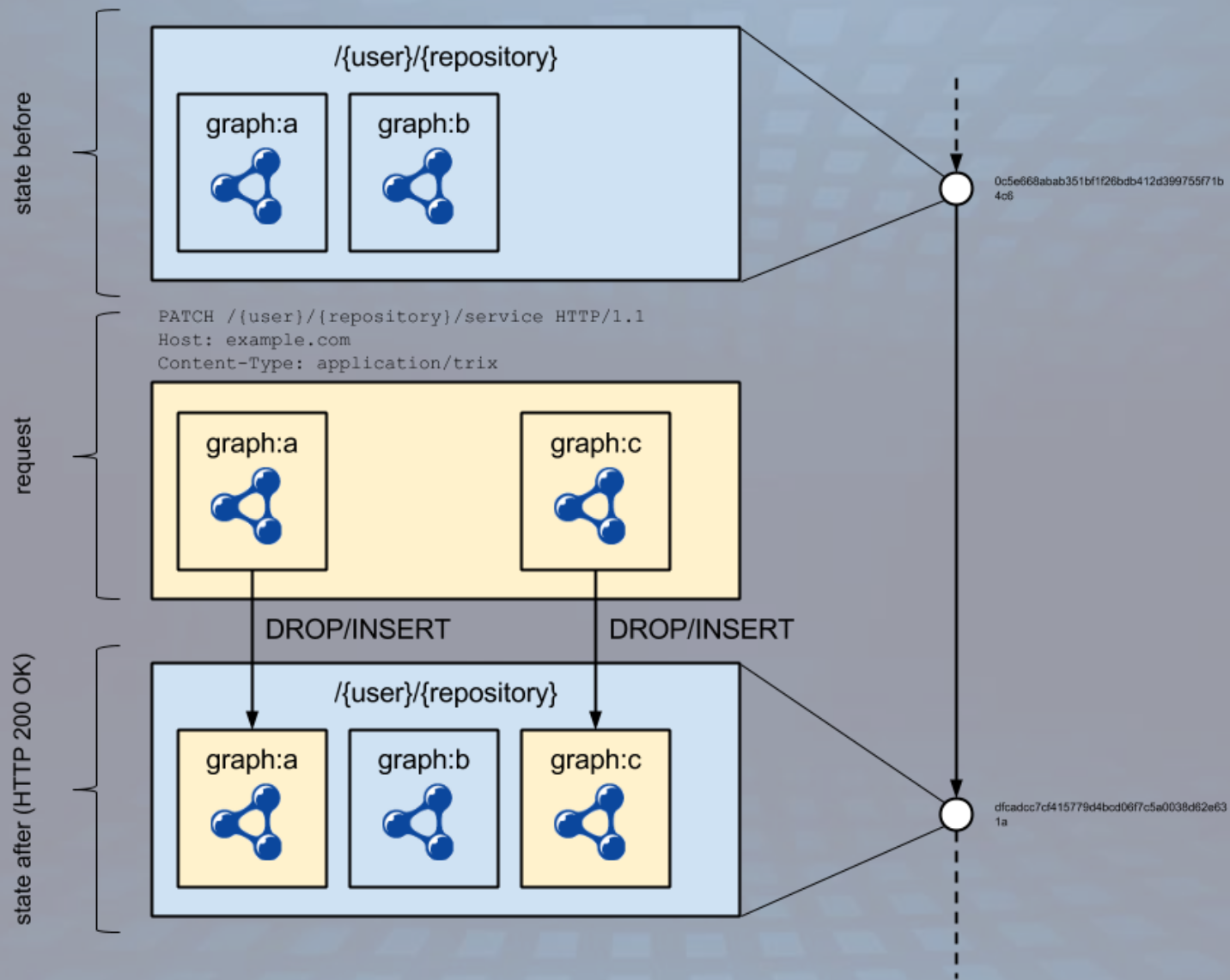
# Data management

- PLM system distributes  $\Delta$  updates via XML messages over ESB
- A single message contains description of multiple items
- Messages can contain overlapping content
- Simplest approach is “Graph Per Resource” data management pattern
  - <http://patterns.dataincubator.org/book/graph-per-resource.html>
- Enables HTTP operations (GET, PUT) to manipulate individual resource descriptions



# Apply changes to RDF graph store using Quads and HTTP PATCH

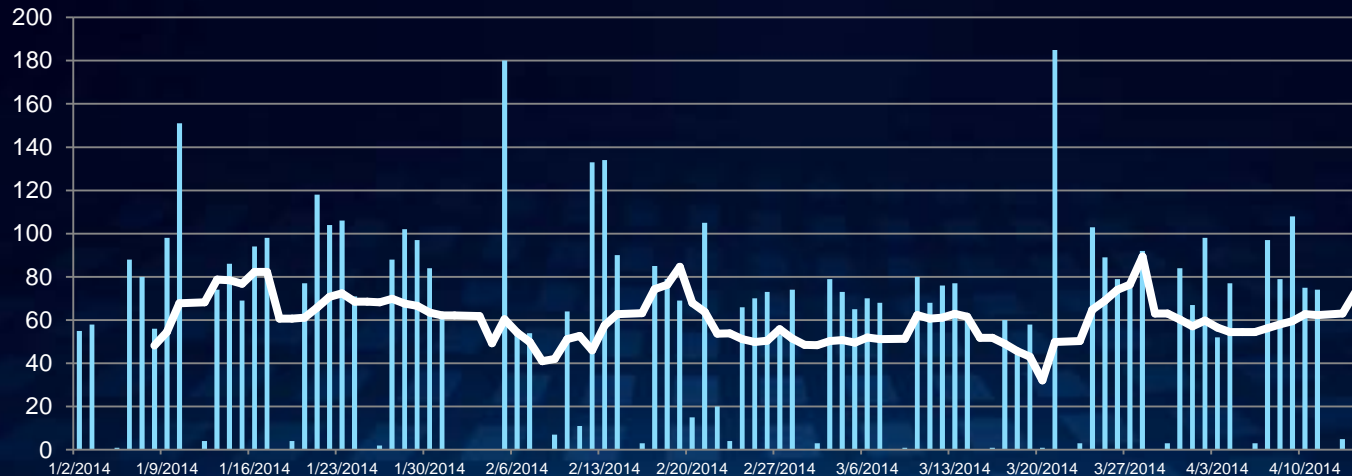
- › We define HTTP PATCH using Quad data as equivalent to:
  - DROP SILENT operation on each named graph in payload, followed by
  - INSERT DATA operation on each named graph in payload
- › Enables update of multiple resource descriptions (named graphs) in a single ACID transaction (i.e. HTTP request)
  - Same granularity as original message
  - No ambiguity about state of store, change is never partially applied
- › Operation is carried out against the graph store service endpoint
  - Same endpoint as used with SPARQL 1.1 Graph Store HTTP Protocol
  - No graph or default parameter is passed with request
  - Content-Type header used to specify MIME type of the Quads format
    - application/n-quads
    - application/trix
    - application/trig
  - Quad data passed as message payload



# ETL pipeline

- PLM system distributes XML messages over ESB
- Generic XSLT transformation to TriX
- Load TriX to graph store using HTTP PATCH method
- Typically 3-4 seconds to transform and load a message

**Messages per day**



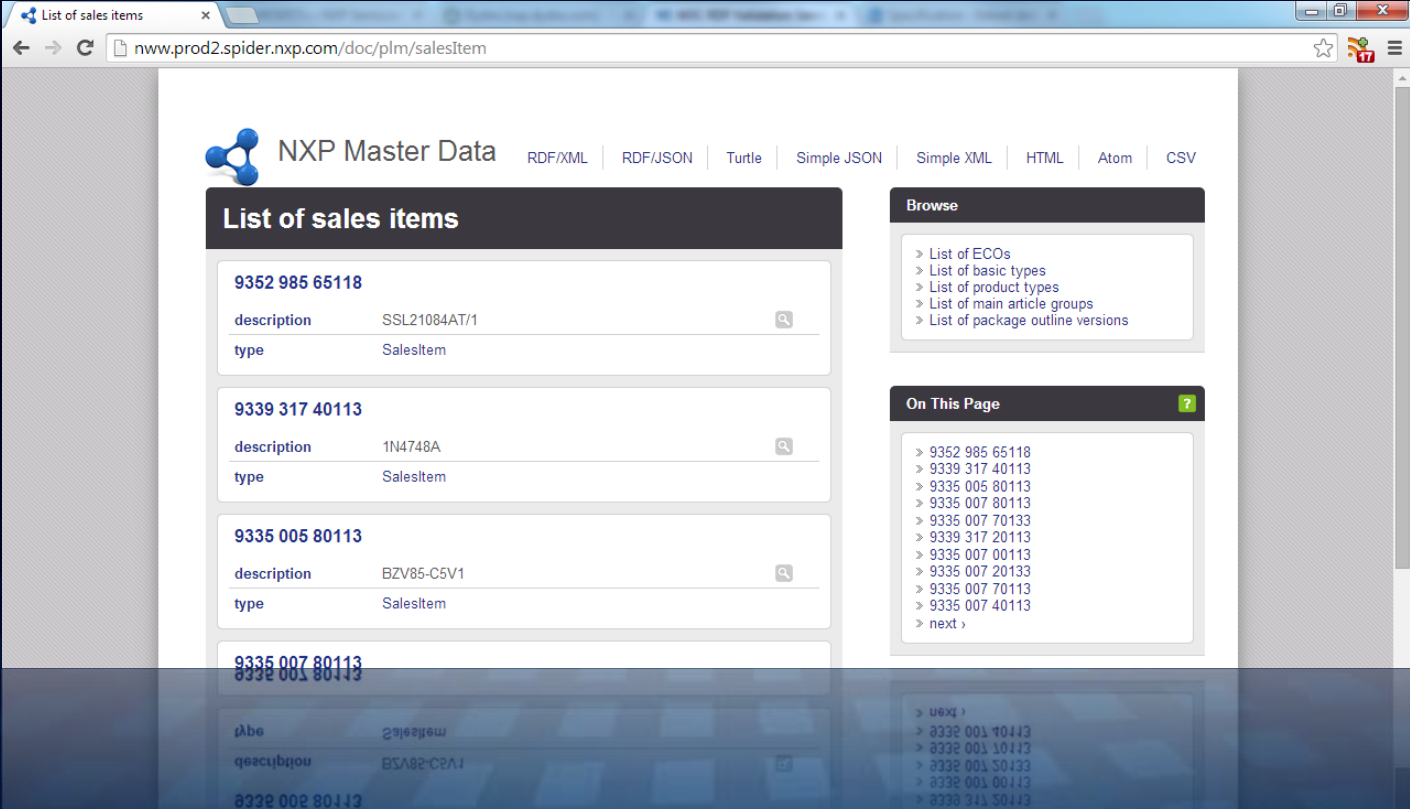
# Facts and figures

- › ~14M triples in dataset
- › Describing ~0.5M items
- › ~1.7M links between items

Per day	Typical	Max
Messages	60	185
Changed items	530	5,300
Sent items	14,000	77,000
Triples loaded	500,000	2,685,554

# Publish as Linked Data

- › Used Linked Data API to make data browseable as HTML
- › Also provides simple API for XML, JSON and CSV



The screenshot shows a web browser window displaying the 'List of sales items' page from the NXP Master Data application. The page features a navigation menu with options: RDF/XML, RDF/JSON, Turtle, Simple JSON, Simple XML, HTML, Atom, and CSV. The main content area displays a list of sales items, each with a unique ID, a description, and a type. The 'Browse' sidebar offers links to various data categories, and the 'On This Page' sidebar provides a list of links to specific items on the page.

ID	description	type
9352 985 65118	SSL21084AT/1	SalesItem
9339 317 40113	1N4748A	SalesItem
9335 005 80113	BZV85-C5V1	SalesItem
9335 007 80113		

**Browse**

- > List of ECOs
- > List of basic types
- > List of product types
- > List of main article groups
- > List of package outline versions

**On This Page**

- > 9352 985 65118
- > 9339 317 40113
- > 9335 005 80113
- > 9335 007 80113
- > 9335 007 70133
- > 9339 317 20113
- > 9335 007 00113
- > 9335 007 20133
- > 9335 007 70113
- > 9335 007 40113
- > next >

<https://code.google.com/p/puelia-php/>







Building a Linked Data application

# LINK MANAGEMENT







# Product placement

- › Products need to be placed (linked) to product categories
- › Products defined in PLM data set, categories in SKOSjs
- › Different role and users to tree manager
- › Decided to implement a simple application
- › Use cases:
  - Find unassigned products
  - Search for products and categories
  - Add links
  - Remove links

# Let's Play



- › Decided to build application using Java version of Play Framework
- › Approx 4-6 weeks to develop and test
- › Application makes queries against graph store direct from client browser
  - Using SPARQL 1.1 Federated Query
  - Originally by populating pre-defined SPARQL templates and executing these against SPARQL endpoint
  - Now queries are stored in database and exposed as an API
    - Initial variable bindings can be passed as request parameters
- › Results are in SPARQL Query Results JSON Format

# SPARQL Query Results JSON Format

- ▶ Simple tabular results format for SELECT and ASK queries
- ▶ Content-Type: application/sparql-results+json
- ▶ Example response

```
{
  "head": {
    "vars": [ "btn", "desc", "csi", "psi", "status", "statusDate" ]
  },
  "results": {
    "bindings": [
      {
        "btn": { "type": "literal", "value": "BZL3615AHN" },
        "csi": { "type": "literal", "value": "No" },
        "psi": { "type": "literal", "value": "No" },
        "status": { "type": "literal", "value": "Development" },
        "statusDate": {
          "type": "literal",
          "value": "2014-02-03",
          "datatype": "http://www.w3.org/2001/XMLSchema#date"
        }
      },
      ... more bindings ...
    ]
  }
}
```

# Example 1: Get unassigned products

- SELECT query with no initial bindings

```
GET /nxp/marketing-tree/pp_get_unassigned_products.srj HTTP/1.1
```

btn	desc	csi	psi	status	statusDate
BTH3415TIO		No	No	Development	2014-02-03
BTM4500TIO		No	No	Development	2014-01-31
CHB3131		Yes	No	Development	2014-03-04
CHV7072		No	No	Development	2014-03-12
CHV7075		No	No	Development	2014-03-12
CHV811		No	No	Qualification	2014-01-23
CHV8M1		No	No	Qualification	2014-01-23
CHV8O1		No	No	Qualification	2014-01-23
CMD8H21MT-160B		No	No	Development	2014-04-07
CMG6H22MT-180Q		No	No	Production	2014-03-20



# Example 2: Get products filtered

- › SELECT query with a single binding for search string
- › Initial bindings passed as parameter `$searchString`
- › Literal values enclosed in quotes "2n7002"
- › Parameter name and value URL encoded

```
GET /nxp/marketing-  
tree/pp_get_products_filtered.srj?&%24searchString=%222n7002ck%22  
HTTP/1.1
```

btn	desc	csi	psi	status	statusDate
2N7002CK	60 V, 0.3 A N-channel Trench MOSFET	No	No	Production	2011-10-28



# Example 3: Link product to category

- › INSERT query with binding for product and category
- › Initial bindings passed as parameters `$productId $categoryId`
- › Literal values enclosed in quotes `"2n7002"`
- › Parameter names and values URL encoded

```
GET /nxp/marketing-tree/pp_link_product_to_category.srj?  
%24productId=%22BFU915F%22&%24categoryId=%22208%22 HTTP/1.1
```

Product 2N7002CK

www.prod2.spider.nxp.com/productplacement/product/2N7002CK

PRODUCT PLACEMENT Product 2N7002CK nlv01111 Logout Help

Name: 2N7002CK  
 Descriptive title: 60 V, 0.3 A N-channel Trench MOSFET  
 Published to web: Yes   
 CSI: No  
 PSI: No  
 Status: Production  
 Status date: 2011-10-28  
 SSPD: small signal MOSFET

Category:

*Categories with product 2N7002CK*

Category name	Category ID	TriMM ID
Automotive MOSFETs	130	50933
Standard MOSFETs	67	48014

First Previous Next Last Page:   Rows:

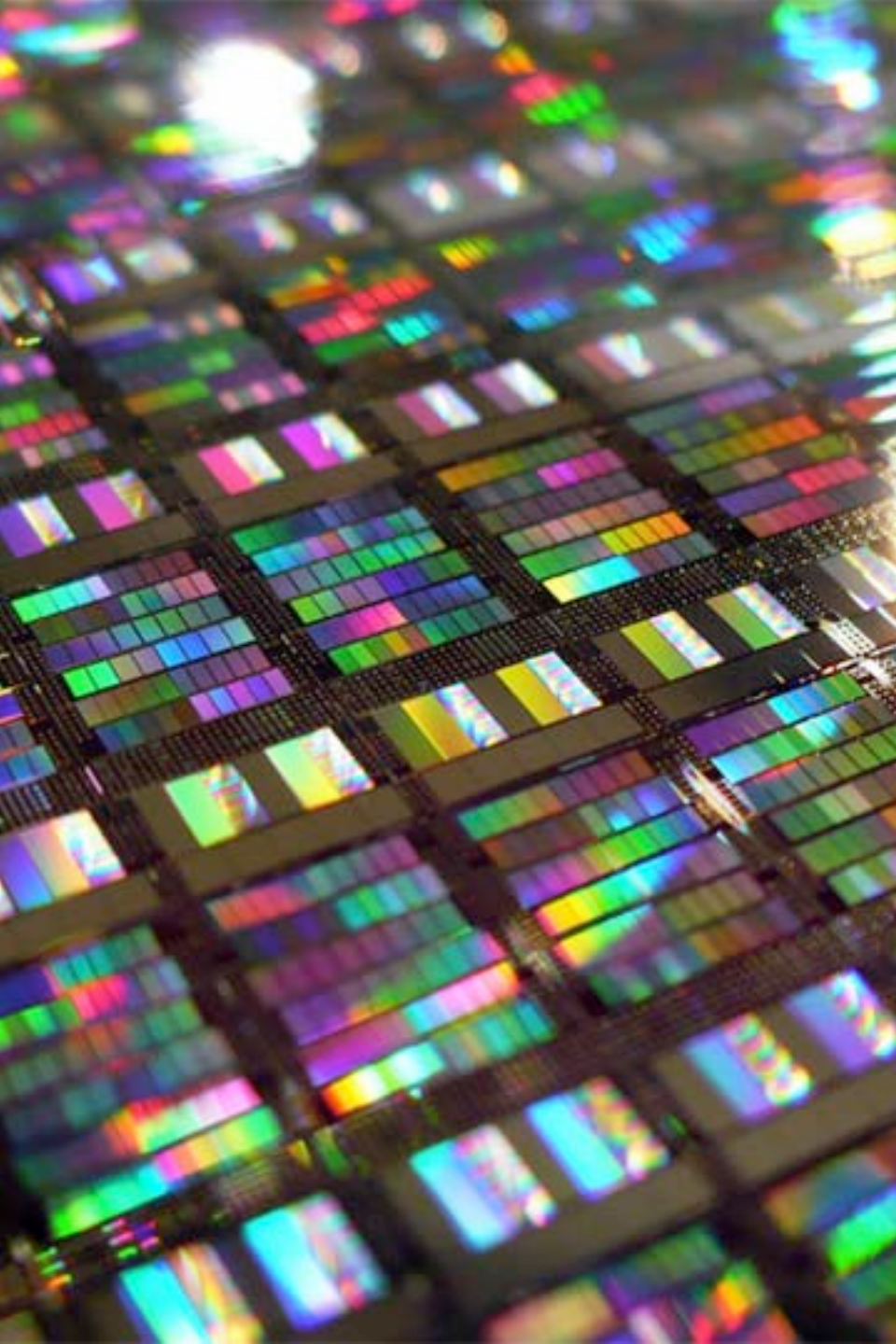
*Categories without product 2N7002CK*

Category name	Category ID	TriMM ID
(De)multiplexers, data and clock recovery, limiting amplifiers	208	42486
0-1600 MHz (HF / VHF / ISM)	672	53511
0-500 MHz (HF / VHF / ISM)	1254	71254
0.4 - 1.0 GHz transistors	1190	42804
1 GHz - 2 GHz (L band)	1805	71805
1 GHz -> + 2 GHz (L band)	1699	71699
1 GHz -> + 2 GHz (L band)	654	30921
1.3 - 1.7 GHz transistors	1417	71417
1.8 - 2.0 GHz transistors	1191	42805
10-500 MHz (HF / VHF / ISM)	673	53512

First Previous Next Last Page:   Rows:

# Lessons learned

- › Building a simple Linked Data application is easy
- › SPARQL Federation is very useful
- › Stored queries is a good approach to expose API
  - Front end developer doesn't see SPARQL
  - Works for read and write



Publishing Linked Data

# LINKED DATA API



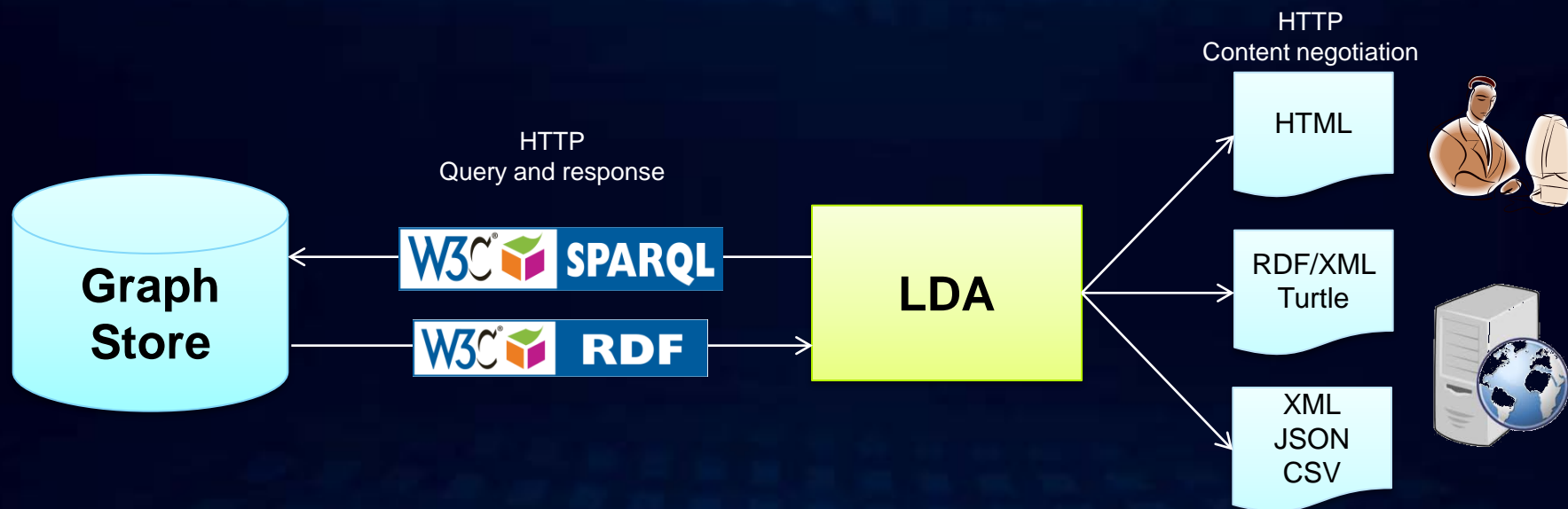


# Linked Data API (LDA)

- › LDA defines a vocabulary and processing model for a configurable API layer intended to support the creation of simple RESTful APIs over RDF triple stores.
- › The API layer is intended to be deployed as a proxy in front of a SPARQL endpoint to support:
  - Generation of documents (information resources) for the publishing of Linked Data
  - Provision of sophisticated querying and data extraction features, without the need for end-users to write SPARQL queries
  - Delivery of multiple output formats from these APIs, including a simple serialization of RDF in JSON syntax

# LDA Architecture

- › Use W3C web standards RDF, SPARQL for portable solution
- › Work with any RDF graph store with only minor configuration





# LDA is open source

- › Open source specification:

<http://code.google.com/p/linked-data-api/>

- › Open source implementations:

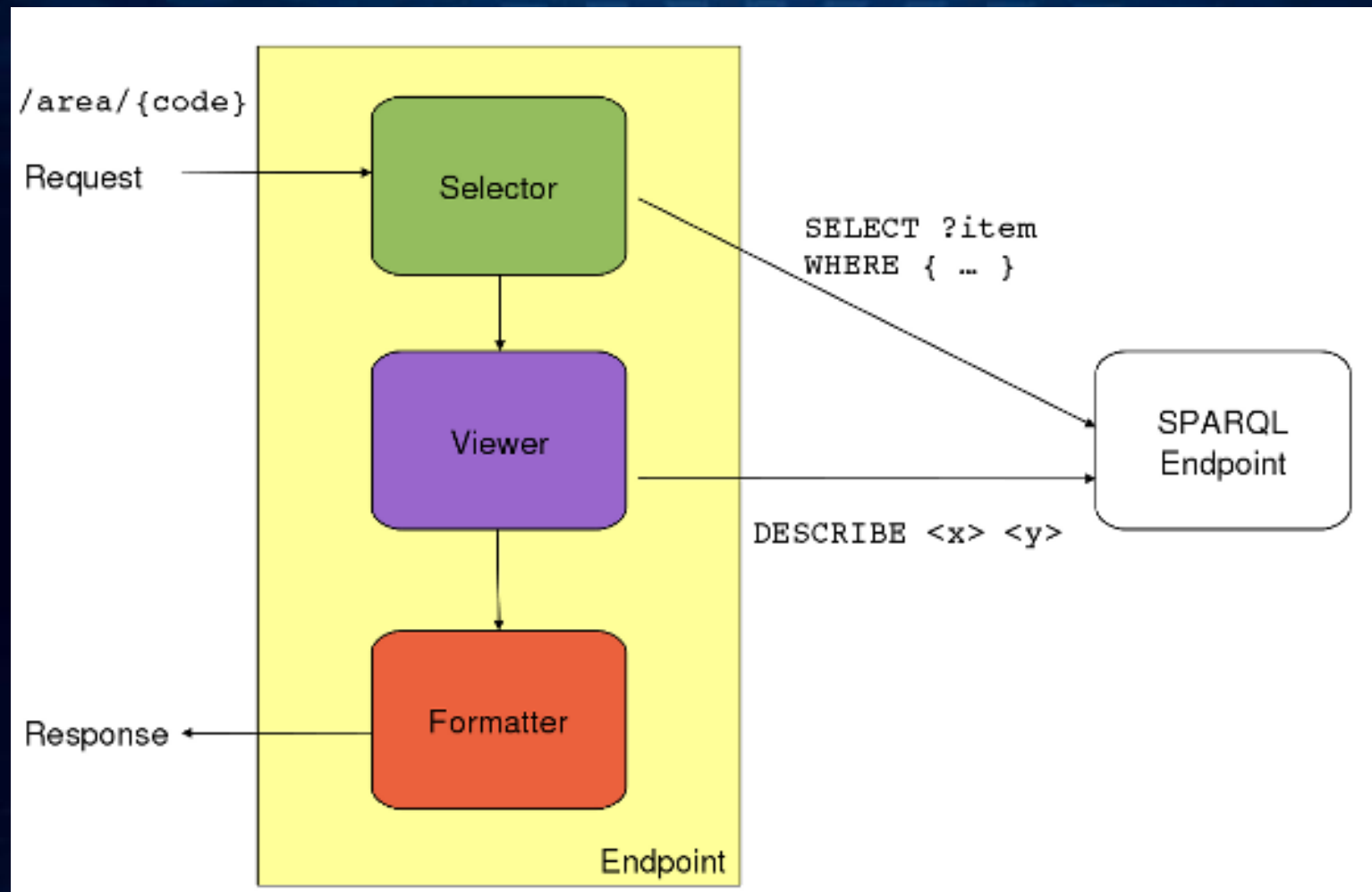
- Puelia (PHP): <http://code.google.com/p/puelia-php/>

- Elda (Java): <https://github.com/epimorphics/elda>

# LDA Processing Model

- › **Identifying an Endpoint** -- GET request made to a particular URI is mapped to an **Endpoint** that describes further processing logic
- › **Binding Variables** -- the API creates a number of variable bindings based on the structure and parameters of the incoming request, and the Endpoint configuration. These variables as well as the available API configuration and request metadata describe the **Context** for the execution
- › **Selecting Resources** -- the API identifies a sequence of items whose properties are to be returned. Usually this will be based on a **Selector** that describes how to identify a single item or an ordered list of resources, in concert with the available bindings
- › **Viewing Resources** -- the API retrieves the desired properties of the identified resource, constructing an RDF graph of the results. This process is described by a **Viewer** that identifies the relevant properties of the resources
- › **Formatting Graphs** -- the API identifies how to serialize the resulting RDF graph to the client. This process is defined by a **Formatter**

# LDA Processing Model



# LDA configuration

- › LDA config files are declarative description of the API
- › Describes
  - List and item endpoints
  - Selectors
  - Viewers
  - Formatters
- › Written in Turtle
- › Read by application at run time





Thank you

