STARDOG (/)
↩ Tutorials (../)

RDF Graph Data Model

Learn about the RDF graph model used by Stardog.

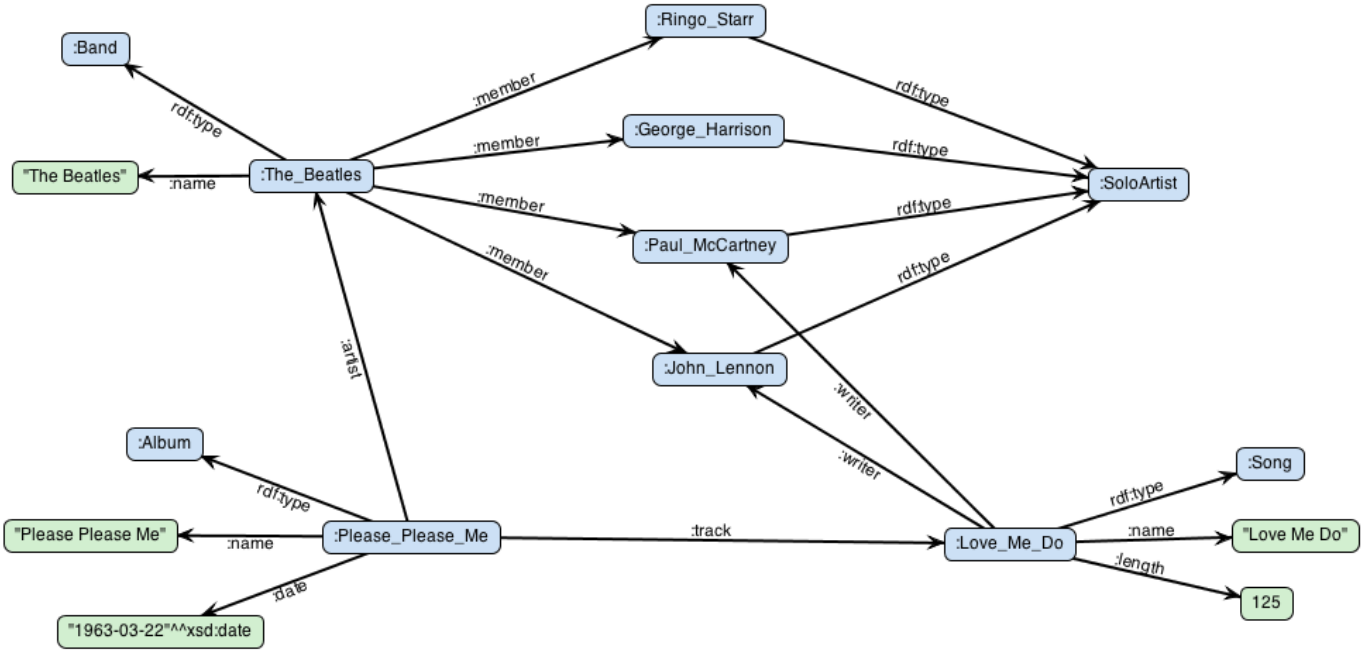
Last update: 12 October 2018

RDF Graphs

Stardog Knowledge Graph supports a graph data model based on RDF, a W3C standard for exchanging graph data. More specifically, Stardog's data model is a directed semantic graph. We will talk about the "semantic" part in an upcoming tutorial; for now let's talk about the "directed" part.

A directed graph is a set of objects, usually just called "nodes", connected together by lines, usually just called edges, that are "directed" from one node to another. So really the edges are directional or you can think of them as arrows, i.e., the edges aren't really lines but they have a pointer like this: →. An edge points from one node to another node.

Here's a picture that shows an RDF graph about The Beatles:



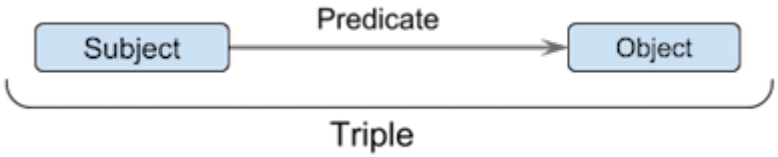
This graph shows several nodes that represent entities such as the Beatles band and one of their studio albums. Each edge has an identifier that tells us what relationship holds between those nodes. For example, the `:member` edge links bands to its members. The `rdf:type` edge represent a special kind of relationship that we will explain below, in the RDF Schema section.

In this graph we also have nodes representing datatype values (i.e., "literals") such as strings, numbers, dates. These edges are sometimes called "attributes" of the node and are often used to represent the characteristics of the nodes.

This simple, flexible data model has a lot of expressive power to represent complex situations, relationships, and other things of interest, while also being appropriately abstract, i.e., it does not expose very much implementation detail in the same way as, say, the relational data model used with SQL.

RDF Terminology

RDF has a special nomenclature for naming nodes and edges in a graph. Consider this figure:



An edge is called a `triple`, the source node is called a `subject`, the edge name is called a `predicate`, and the target node is called an `object`. Note that in a graph a node can be in the subject position in one triple and in the object position in another triple.

Based on this terminology an RDF graph is defined as a set of RDF triples. This definition is not much different than the informal definition given above with one slight change: you cannot have an RDF node without any edges. In other words, the nodes of an RDF graph are not declared separately, rather, they are determined by the edges in that graph.

RDF Nodes

There are three different kinds of RDF nodes:

- *IRI*[1]: An IRI is a unicode string for identifying nodes and edges in an unambiguous way. IRIs are internationalized versions of URIs which are generalizations of URLs. Using IRIs helps avoid name clashes and promotes distributed naming, that is, to

avoid requiring a centralized naming authority.

- *Blank node*: Nodes without a user-visible identifier are called blank nodes ("bnode" for short). A blank node is appropriate when the node does not need to be referenced directly. Blank nodes can be reached by following its incident edges from other nodes.

- *Literal*: Literals are concrete values used to represent datatypes like strings, numbers, and dates.

In the example Beatles graph, we have several IRIs: `:The_Beatles` and `:John_Lennon` are two examples. (They're abbreviated as explained in the IRI serialization section below.) Literals in our example include the string `"The Beatles"`, the date `"1963-03-22"`, and the integer `125`.

Unlike IRIs and bnodes, literals can only appear in the object position of a triple. This means a literal can only have incoming edges and cannot have any outgoing edges. For this reason, the IRIs and blank nodes are jointly called *resources*. Resources are described in the graph by their relationships to other resources and their attributes as denoted by their connection to literals.

RDF Syntax

Node-and-link visualization of graphs is convenient and easy to understand on a small scale, but it is not very useful for exchanging data between systems or loading graph data into a system like Stardog. There are several syntaxes to serialize an RDF graph as text including syntaxes based on XML and JSON. In this tutorial, we will use the Turtle syntax which is also the basis of the SPARQL query language.

The serialization of the Beatles graph in Turtle syntax looks like this:

```
PREFIX : <http://stardog.com/tutorial/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

:The_Beatles      rdf:type  :Band .
:The_Beatles      :name     "The Beatles" .
:The_Beatles      :member   :John_Lennon .
:The_Beatles      :member   :Paul_McCartney .
:The_Beatles      :member   :Ringo_Starr .
:The_Beatles      :member   :George_Harrison .
:John_Lennon      rdf:type  :SoloArtist .
:Paul_McCartney   rdf:type  :SoloArtist .
:Ringo_Starr      rdf:type  :SoloArtist .
:George_Harrison  rdf:type  :SoloArtist .
:Please_Please_Me rdf:type  :Album .
:Please_Please_Me :name     "Please Please Me" .
:Please_Please_Me :date     "1963-03-22"^^xsd:date .
:Please_Please_Me :artist   :The_Beatles .
:Please_Please_Me :track    :Love_Me_Do .
:Love_Me_Do       rdf:type  :Song .
:Love_Me_Do       :name     "Love Me Do" .
:Love_Me_Do       :length   125 .
:Love_Me_Do       :writer   :John_Lennon .
:Love_Me_Do       :writer   :Paul_McCartney .
```

The serialization is simple. A triple is serialized by writing the subject, the predicate, and the object separated by whitespace[2] and terminated with a period `.` at the end. We will talk about the the details of serializing RDF terms and what the `PREFIX` declarations mean, but first let's talk about some ways to simplify the triple serialization.

As seen above, repeating the subjects and the predicates for every triple can be verbose. For this reason, Turtle introduces some syntactic sugar:

- Multiple predicates: If two triples share the same subject then the first triple can be terminated with `;` and the subject of the second triple can be omitted.
- Multiple objects: If two triples share the same subject and the same predicate the objects can be separated with `,` without repeating the subject or the predicate.
- Types: The letter `a` can be used in place of `rdf:type`; you can read this as "is a", basically. For example, "Love Me Do is a song."

Using these shortcuts the serialization becomes more concise and elegant:

```
:The_Beatles      a :Band ;
                  :name "The Beatles" ;
                  :member :John_Lennon , :Paul_McCartney , :George_Harris
:John_Lennon      a :SoloArtist .
:Paul_McCartney   a :SoloArtist .
:Ringo_Starr      a :SoloArtist .
:George_Harrison  a :SoloArtist .
:Please_Please_Me a :Album ;
                  :name "Please Please Me" ;
                  :date "1963-03-22"^^xsd:date ;
                  :artist :The_Beatles ;
                  :track :Love_Me_Do .
:Love_Me_Do       a :Song ;
                  :name "Love Me Do" ;
                  :length 125 ;
                  :writer :John_Lennon , :Paul_McCartney .
```

Let's dive into the details of serializing different kinds of RDF nodes.

IRIs

IRIs, just like the URIs and URLs they generalize, are long strings that are not easy to read or write. A full IRI can be serialized by simply enclosing it in angle brackets:

```
<http://stardog.com/tutorial/The_Beatles>
```

However, it is more common and much more convenient to write IRIs as ***prefixed names***. A ***prefix*** is a short name that is mapped to a long ***namespace***. A prefixed name is the sequence of the prefix and the ***local name*** separated by a colon `:`. The empty string is a valid prefix and called the ***default namespace*** for a graph.

In our example serialization above we have three prefix declarations:

```
PREFIX : <http://stardog.com/tutorial/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

The first declaration is a user-defined default namespace used to identify almost all the nodes and the edges in the graph. Default namespace means the empty or bare prefix, which is why those IRIs all start with the `:` symbol. The exception in the example is the `rdf:type` IRI which belongs to the built-in RDF namespace (we'll talk more about `rdf:type` in the RDF Schema section) and the `xsd:string` IRI which

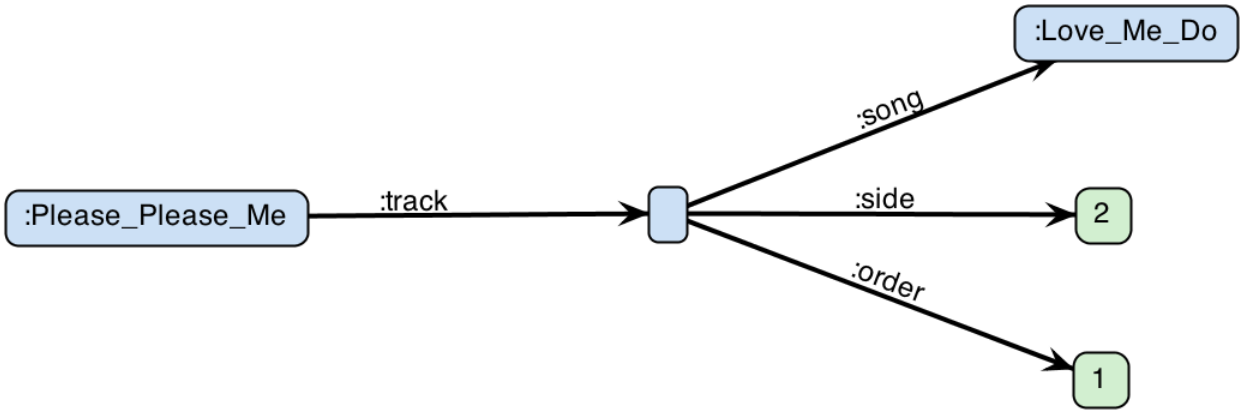
belongs to the built-in XML Schema namespace (see the Literals section for more info).

When working with an RDF graph there are typically only a handful of namespaces one needs to use. Stardog allows these prefix mappings to be stored in the database metadata so you do not need to repeat the prefix declarations in every file, database, or query. The commonly used namespaces like RDF and XSD are included by default so in practice you may not need to use a single prefix declaration.

> **NOTE:** All the IRIs we use in this tutorial are regular URLs. This is a common source of confusion since URLs are used to locate web pages and many common IRIs in fact reference web pages that provide additional information about the corresponding resource. However, using URLs as identifiers is not a requirement and there are many kinds of IRIs that are not URLs. For example, we could have used a tag IRI (https://en.wikipedia.org/wiki/Tag_URI_scheme) like `tag:stardog.com,2018:tutorial#The_Beatles` in our RDF graph. There is also no requirement that the IRIs point to active web pages even if they are well-formed URLs. For example, the `http://stardog.com/tutorial/The_Beatles` IRI we used does not refer to an actual document. The key takeaway is that IRIs are designed be be globally unique which makes it the easier to unify knowledge graphs created by different teams without worrying about naming clashes.

Blank Nodes

Suppose we extend our example. We have a new use case that requires us to capture, for all the tracks in an album, which side they belong to–when applicable for albums released on media with "sides", i.e., vinyl, cassettes, etc.–and the order of the song on the album/side. We can introduce a blank node between the album and the song to attach this data:



Blank nodes do not have globally unique identifiers so when they are serialized a locally unique, non-persistent label is used. These blank node names are serialized after the special prefix `_:` . In Turtle serialization we can avoid using these names completely by using the `[]` abbreviation. When a `[` symbol is used in the object position it denotes a new bnode which will be the subject of a subsequent predicate list until the matching `]` symbol.

The following table shows two different ways of serializing bnodes:

Short form	Long form
`:Please_Please_Me :track [ :side 2 ; :order 1 ; :song :Love_Me_Do ] .`	`:Please_Please_Me :track _:s2 _:s2n1 :side 2 ; :order 1 ; :song :Love_Me_Do .`

Bnodes are optional and using them makes most sense when there is no identifier that can be used to distinguish nodes from one another. But one can always substitute an IRI that includes a UUID in place of a bnode. It is considered good practice to avoid bnodes as much as possible so that all nodes can be referenced directly in queries and transactions.

Literals

Literals are serialized as their lexical value in double quotes followed by the datatype after double carets (^^). The datatype is typically a built-in datatype from XML Schema Datatypes (XSD) (https://www.w3.org/TR/xmlschema-2/) that defines many commonly used datatypes but custom datatype IRIs can also be used. Some of the XSD datatypes can be serialized without the explicit datatype or the quotes. The following table shows examples of serializing different datatypes:

Serialization	*Datatype*	*Description*
“The Beatles”	`xsd:string`	Datatype can be omitted for string values

Serialization	*Datatype*	*Description*
“The Beatles”^^xsd:string	`xsd:string`	Same as the previous literal but with an explicit datatype
“The Beatles”@en	`rdf:langString`	An optional language tag (https://en.wikipedia.org/wiki/IETF_language_tag) can be added after the `@` symbol. These literals have the special type `rdf:langString` which is different than `xsd:string`
“Los Beatles”@es	`rdf:langString`	Another literal with a language tag
”””The Beatles were an English rock band formed in Liverpool in 1960. With members John Lennon, Paul McCartney, George Harrison and Ringo Starr, they became widely regarded as the foremost and most influential music band in history.”””	`xsd:string`	Multi-line strings can be enclosed in triple quotes
“1963-03-22”^^xsd:date	`xsd:date`	Date value
125	`xsd:integer`	The datatype and double quotes can be omitted for integer values
3.0	`xsd:decimal`	Arbitrary-precision decimals can be written without the datatype and quotes too. Existence of `.` in the number makes it a decimal.

Serialization	*Datatype*	*Description*
3.2E4	`xsd:double`	Double-precision floating point values can be written in scientific notation with the symbol `E` separating the mantissa from the exponent
true	`xsd:boolean`	Lowercase strings `true` and `false` can be used for boolean values

Named Graphs

Sometimes it is useful to assign a name to an RDF graph for the purposes of sane data management, access control, or to attach metadata to the overall graph rather than to individual nodes. The notion of named graphs in RDF allows us to do that. To give a careful definition, we say that an "RDF Dataset" is a collection of RDF graphs defined as:

- Exactly one default graph: The default graph does not have a name and may not contain any triples.
- Zero or more named graphs: Each named graph is a pair consisting of a resource (IRI or a blank node), which is the the graph name, and an RDF graph.

So the example we have been looking at so far was technically an RDF dataset with just a default graph and no named graphs. If we decide to separate the artists, albums, and songs into separate named graphs, then we can group the triples under a `GRAPH` block in the serialization[3].

The next example shows this along with some metadata attached to the graph names in the default graph (triples outside the `GRAPH` blocks are in the default graph):

```
:Artist :description "This is the graph that contains information about a

GRAPH :Artist {
   :The_Beatles a :Band ;
              :name "The Beatles" ;
              :member :John_Lennon , :Paul_McCartney , :George_Harrison ,

   :John_Lennon      rdf:type  :SoloArtist .
   :Paul_McCartney   rdf:type  :SoloArtist .
   :Ringo_Starr      rdf:type  :SoloArtist .
   :George_Harrison  rdf:type  :SoloArtist .
}

:Album :description "This is the graph that contains information about all

GRAPH :Album {
  :Please_Please_Me rdf:type :Album ;
                  :name "Please Please Me" ;
                  :date "1963-03-22"^^xsd:date ;
                  :artist :The_Beatles ;
                  :track :Love_Me_Do .
}

:Song :description "This is the graph that contains information about ind

GRAPH :Song {
  :Love_Me_Do rdf:type :Song ;
             :name "Love Me Do" ;
             :length 125 .
}
```

Note that it is the triples that are separated into named graphs, not the nodes, and different named graphs can share some common nodes, e.g. `:Please_Please_Me` node appears both in the `:Artist` graph and the `:Album` graph. So it is possible to traverse the edges starting from one named graph and continue into another named graph via these shared nodes. It is through this sharing of nodes across named graphs that the collection of named graphs (conceptually) constitute a larger unified graph.

Each graph in a dataset is still a set of triples which means there can be no duplication of triples within a graph. However, there is no similar requirement across multiple graphs, so the same triple may appear in multiple graphs and each occurrence is considered a distinct triple. Another way to think of named graphs is as a set of quads where the fourth component added to the triple is the name of the graph, which is possibly empty for the default graph.

RDF Schema

Stardog supports a schema-flexible data model. This means in an RDF graph you can go from having an implicit schema (which is what we have been doing in our examples so far) to an explicit schema about how the nodes can be connected to each other, datatypes, valid labels or identifiers, and so on. Every point along the spectrum from implicit to explicit schema is permissible. Some part of an application can be explicit and rigid while other parts may be implicit and flexible. RDF itself does not actually provide any schema-related vocabulary. The RDF Schema (RDFS) and Web Ontology Language (OWL) specifications provide this functionality.

RDF and RDFS has some similarities to object-oriented models but some differences too. For example, the schema information for an RDF graph is represented as part of the graph. Named graphs may be used to separate the schema triples into a dedicated graph but this is not required.

There are two main concepts in RDFS: Class and Property.

Classes

Classes represent categories of nodes with similar characteristics. Nodes that belong to this category are linked to the class using the `rdf:type` (short hand: `a`) property. Classes themselves are identified by the meta-class `rdfs:Class`.

```
:Band a rdfs:Class .          # declaration of a class
:The_Beatles a :Band .        # declaring an instance of a class
```

Since classes are declared in the same way as data, syntactically, schemas are part of the graph data and can be queried with all the same tools and mechanisms. This also means that unlike other data models the distinction between 'data' and 'metadata' is quite fluid and informal in RDF. In the real world the difference between these is always relative to some particular use case or requirement, that is, it's is not a very rigid or universal distinction.

Classes can be organized in a hierarchy by relating them to each other via `rdfs:subClassOf`. For example, it is reasonable to define a generic superclass `Artist` for the more specific `Band` and `SoloArtist` classes we have used so far:

```
:Artist a rdfs:Class .
:Band rdfs:subClassOf :Artist .
:SoloArtist rdfs:subClassOf :Artist .
```

We will talk about class hierarchies and reasoning in more detail in an upcoming tutorial.

Properties

Property is a relation between subjects and objects. We have already seen many examples of properties; `:album`, for example. We can use the `rdf:Property` class to declare properties:

```
:track a rdf:Property .
:length a rdf:Property .
```

One unique feature of the RDF data model is that a class is not defined in terms of the properties its instance may have. Rather, properties are defined in terms of the kind of subjects and objects they relate. This is done via domain and range definitions:

```
:track a rdf:Property ;
       rdfs:domain :Album ;
       rdfs:range :Song .

:length a rdf:Property ;
       rdfs:domain :Song ;
       rdfs:range xsd:integer .
```

As this example shows, RDF does not syntactically differentiate if a property relates subjects to IRI objects or to literal objects. But we can observe this difference semantically.

The range of the `track` property is defined to be a `Song` class, so the objects should be resources that are instances of this class. The range of the `length` property, on the other hand, is defined to be the built-in datatype `xsd:integer` so the objects should be integer literals. The domain declaration should always be a class as it defines the types of subjects used with the property and subjects can't be literals.

Even though RDFS allows one to define domain and range declarations for properties, there is no builtin concept of validation similar to other schema languages like XML

Schema. That is, it would be legal in RDFS, though nonsensical in practice, to use the `track` property to link an album to another album.

But of course data quality matters a lot and so Stardog allows a data validation capability to actually enforce such constraints. We will explore this capability (https://docs.stardog.com/data-quality-constraints) in detail in another tutorial.

Metadata

RDFS also provides two properties that can be used to provide metadata about nodes, classes, and properties:

- `rdfs:label` : provides a human-readable name for a resource
- `rdfs:comment` : provides a human-readable description of a resource

```
:length rdfs:label "length (in seconds)" ;
        rdfs:comment "The length of a song expressed in seconds".
```

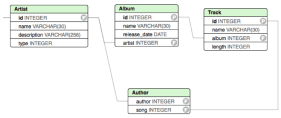
Using these RDFS properties for human-readable names and descriptions is considered a good practice but this is not a requirement. For example, in our example above, we used the `:name` property instead of `rdfs:label` for human readable names. But since these RDFS properties are well-established, widely used in many RDF datasets and has special support in UIs there is no good reason to invent new properties for the same purpose either.

---

1. IRI stands for Internationalized Resource Identifier. [return]
2. Whitespace is insignificant and indentation in our example is just for legibility. [return]
3. Technically Turtle syntax does not have support for named graphs. The RDF syntax that extends Turtle with named graphs is called TriG (https://www.w3.org/TR/trig/). TriG is a strict superset of Turtle so every valid Turtle document is also a valid TriG document. [return]

Foundational, Data Model

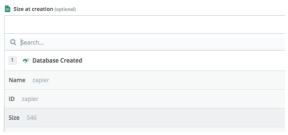
Read Next



(/tutorials/data-mappings/)

Virtual Graph Mappings (/tutorials/data-mappings/)

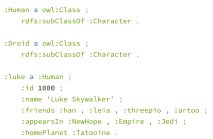
Learn how to unify relational data in Stardog at query time.



(/tutorials/stardog-and-zapier/)

Using Stardog with Zapier (/tutorials/stardog-and-zapier/)

Learn how to automate Stardog operations with Zapier.



(/tutorials/how-to-bulid-react-app-in-stardog/)

Build a React App (/tutorials/how-to-bulid-react-app-in-stardog/)

Learn how to build a React app in front-end of Stardog.

(/)

2101 Wilson Boulevard
Suite 800, Arlington, VA 22201
(https://goo.gl/maps/3womRuFYjJFx783u9)

(https://twitter.com/StardogHQ) (https://www.linkedin.com/company/stardog-union/) (https://www.youtube.com/channel/UCtVgpMy8P1HwV7iyN9Mxx4Q) (https://github.com/stardog-union)

PRODUCT (/PLATFORM)

How it Works (/how-knowledge-graphs-work/)

Enterprise Knowledge Graph Platform (/platform)

Stardog Cloud (/stardog-cloud)

Stardog Studio (/studio)

Connectors (/platform/connectors/)

Pricing (/pricing)

SERVICES (/PROFESSIONAL-SERVICES/)

Professional Services (/professional-services)

Support Packages (/support)

USE CASES (/USE-CASES/DATA-FABRIC/)

Analytics Modernization (/use-cases/analytics-modernization/)

Data Fabric (/use-cases/data-fabric)

Data Lake Acceleration (/use-cases/data-lake-acceleration/)

Drug Discovery (/use-cases/drug-discovery)

Operational Risk (/use-cases/operational-risk)

Semantic Search (/use-cases/semantic-search-for-the-enterprise)

Supply Chain (/use-cases/supply-chain)

INDUSTRIES (/INDUSTRIES/LIFE-SCIENCES/)

Financial Services (/industries/financial-services/)

Life Sciences (/industries/life-sciences/)

Manufacturing (/industries/manufacturing/)

RESOURCES (/RESOURCES)

Blog (/blog)

Resource Hub (/resources)

Events & Webinars (/events)

COMPANY (/COMPANY/ABOUT)

Customers (/company/customers)

Partners (/company/partners)

Careers (/company/careers)

About Us (/company/about)

Contact Us (/company/contact)

LEARN STARDOG (/LEARN-STARDOG)

Learning Portal (/learn-stardog)

Documentation (https://docs.stardog.com)

Stardog Labs (/labs)

Community (https://community.stardog.com/)

Trainings (/trainings)

Tutorials (/tutorials)

TRY STARDOG (/GET-STARTED)

Start a Trial (/get-started)

Stardog Express (/stardogexpress)

Keep up with Stardog:

your email