Apache Jena

- Home
- Download
- Learn
    - Tutorials
    - Overview
    - RDF core API tutorial
    - SPARQL tutorial
    - Manipulating SPARQL using ARQ
    - Using Jena with Eclipse
    - How-To's
    -
    - References
    - Overview
    - Javadoc
    - RDF API
    - RDF I/O
    - ARQ (SPARQL)
    - RDF Connection - SPARQL API
    - Elephas - tools for RDF on Hadoop
    - Text Search
    - TDB
    - SDB
    - SPARQL over JDBC
    - Fuseki
    - Permissions
    - Assembler
    - Ontology API
    - Inference API
    - Command-line tools
    - Extras
- Javadoc
    - Jena Core
    - ARQ
    - TDB
    - Fuseki
    - Elephas
    - Text Search
    - Spatial Search
    - Permissions
    - JDBC
    - All Javadoc
- Ask
- Get involved
    - Contribute
    - Report a bug
    -
    - Project
    - About Jena
    - Roadmap
    - Architecture
    - Project team

  - Related projects
  -
  - ASF
  - Apache Software Foundation
  - License
  - Thanks
  - Become a Sponsor
  - Security
- Improve this Page

1. DOCUMENTATION
2. FUSEKI2
3. FUSEKI MAIN

Fuseki : Main Server

Fuseki main is a packaging of Fuseki as a triple store without a UI for administration.

Fuseki can be run in the background by an application as an embedded server. The application can safely work with the dataset directly from java while having Fuseki provide SPARQL access over HTTP. An embedded server is useful for adding functionality around a triple store and also for development and testing.

The main server does not depend on any files on disk (other than for databases provided by the application), and does not provide the Fuseki UI or admins functions to create dataset via HTTP.

See also Data Access Control for Fuseki.

Running as a configured deployment or development server

The artifact `org.apache.jena:jena-fuseki-server` is a packaging of the "main" server that runs from the command line. Unlike the UI Fuseki server, it is only configured from the command line and has no persistent work area on-disk.

```
java -jar jena-fuseki-server-$VER.jar --help
```

The arguments are the same as the full UI server command line program. There are no special environment variables.

The entry point is `org.apache.jena.fuseki.main.cmds.FusekiMainCmd` so the server can also be run as:

```
java -cp jena-fuseki-server-$VER.jar:...OtherJars... \
    org.apache.jena.fuseki.main.cmds.FusekiMainCmd ARGS
```

Application Use

The application can safely access and modify the data published by the server if it does so inside a transaction using an appropriate storage choice. `DatasetFactory.createTxnMem()` is a good choice for in-memory use; TDB is a good choice for a persistent database.

To build and start the server:

```
Dataset ds = ...
FusekiServer server = FusekiServer.create()
  .add("/rdf", ds)
  .build() ;
server.start() ;
```

then the application can modify the dataset:

```
// Add some data while live.
// Write transaction.
Txn.execWrite(dsg, ()->RDFDataMgr.read(dsg, "D.trig")) ;
```

or read the dataset and see any updates made by remote systems:

```
// Query data while live
// Read transaction.
Txn.execRead(dsg, ()->{
Dataset ds = DatasetFactory.wrap(dsg) ;
try (QueryExecution qExec = QueryExecutionFactory.create("SELECT * { ?s  ?o}", ds) ) {
    ResultSet rs = qExec.execSelect() ;
    ResultSetFormatter.out(rs) ;
  }
}) ;
```

The full Jena API can be used provided operations (read and write) are inside a transaction.

Dependencies and Setup

To include an embedded Fuseki server in the application:

```
<dependency>
  <groupId>org.apache.jena</groupId>
  <artifactId>jena-fuseki-main</artifactId>
  <version>3.x.y</version> <!-- Set the version -->
</dependency>
```

This brings in enough dependencies to run Fuseki. Application writers are strongly encouraged to use a dependency manager because the number of Jetty and other dependencies is quite large and difficult to set manually.

This dependency does not include a logging setting. Fuseki uses slf4j. See section "Logging" for details.

If the application wishes to use a dataset with a text-index then the application will also need to include jena-text in its dependencies.

Logging

The application must set the logging provided for slf4j. Apache jena provides helpers for the JDK-provided java logging and for Apache Log4j v1.

Using the JDK-provided java logging:

```
LogCtl.setJavaLogging()
```

and a dependency of:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-jdk14</artifactId>
  <version>1.x.y</version>
</dependency>
```

For Apache log4j: - this is how the full Fuseki server sets its logging:

```
FusekiLogging.setLogging();
```

and dependencies:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.x.y</version>
</dependency>

<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.x.y</version>
</dependency>
```

See Fuseki Logging.

To silence logging from Java, try:

```
LogCtl.setLevel(Fuseki.serverLogName,  "WARN");
LogCtl.setLevel(Fuseki.actionLogName,  "WARN");
LogCtl.setLevel(Fuseki.requestLogName, "WARN");
LogCtl.setLevel(Fuseki.adminLogName,   "WARN");
LogCtl.setLevel("org.eclipse.jetty",   "WARN");
```

Building a server

A `FusekiServer` is built by creating a configuration, building the server, then running it. The application needs to start the server.

The default port for a Fuseki embedded server is 3330. This is different for the default port for Fuseki running as a standalone server or as a webapp application.

Examples of embedded use

Example 1

Create a server on port 3330, that provides the default set of endpoints for an RDF dataset that can be updated via HTTP.

```
DatasetGraph ds = DatasetFactory.createTxnMem() ;
FusekiServer server = FusekiServer.create()
    .add("/ds", ds)
    .build() ;
server.start() ;
```

```
...
server.stop() ;
```

URLs:

Service	Endpoint
SPARQL Query	`http://host:3330/ds/query`
SPARQL Query	`http://host:3330/ds/sparql`
SPARQL Update	`http://host:3330/ds/update`
File upload	`http://host:3330/ds/update`
GSP read-write	`http://host:3330/ds/data`
Read-write quads	`http://host:3330/ds`

"GSP" = SPARQL Graph Store Protocol

Example 2

Create a server on port 3332, that provides the default set of endpoints for a data set that is read-only over HTTP. The application can still update the dataset.

```
Dataset ds = ... ;
FusekiServer server = FusekiServer.create()
    .setPort(3332)
    .add("/ds", ds, true)
    .build() ;
server.start() ;
```

Service	Endpoint
SPARQL Query	`http://host:3332/ds/query`
SPARQL Query	`http://host:3332/ds/sparql`
GSP read-only	`http://host:3332/ds/data`
GET quads	`http://host:3332/ds`

Example 3

Different combinations of services and endpoint names can be given using a `DataService`.

```
DatasetGraph dsg = ... ;
DataService dataService = new DataService(dsg) ;
dataService.addEndpoint(OperationName.Quads_RW, "");
dataService.addEndpoint(OperationName.Query, "");
dataService.addEndpoint(OperationName.Update, "");

FusekiServer server = FusekiServer.create()
   .setPort(3332)
   .add("/data", dataService)
   .build() ;
server.start() ;
```

This setup puts all the operation on the dataset URL. The `Content-type` and any query string is used to determine the operation.

Service	Endpoint
SPARQL Query	`http://host:3332/ds`
SPARQL Update	`http://host:3332/ds`
GSP read-only	`http://host:3332/ds`
GET/POST quads	`http://host:3332/ds`

Example 4

Multiple datasets can be served by one server.

```
Dataset ds1 = ...
Dataset ds2 = ...
FusekiServer server = FusekiServer.create()
    .add("/data1", ds1)
    .add("/data1-readonly", ds1, true)
    .add("/data2", ds2)
    .build() ;
server.start() ;
```