Apache Jena

jena.apache.org

André Henrique Dantas Neves Cordeiro

Conteúdo

- O que é o Jena?
- Capacidades do Jena
- Noções básicas
- Conceitos RDF no Jena
- Armazenamento
- Gerenciamento de Ontologias
- Raciocínio
- SPARQL

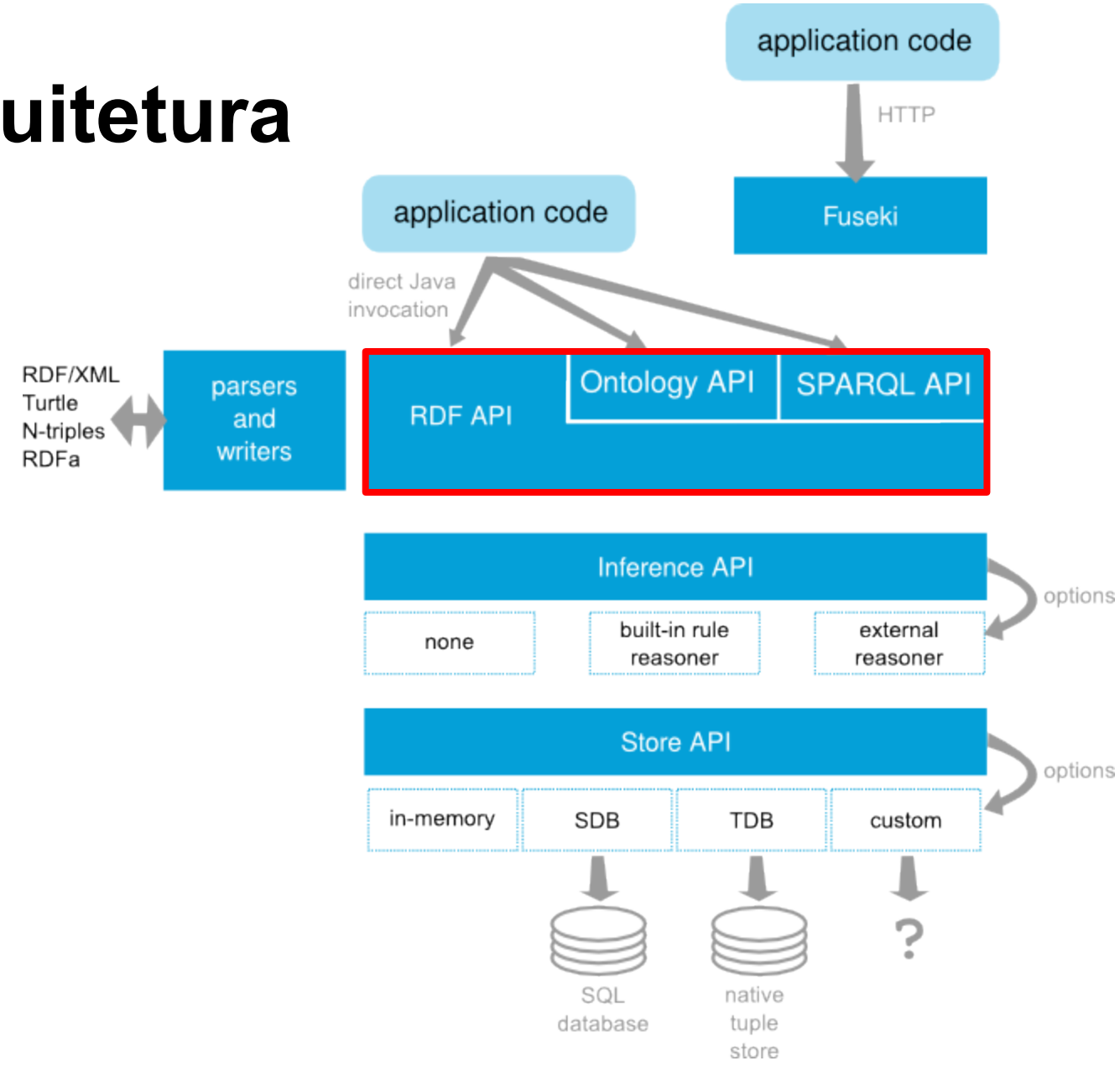
O que é o Jena?

- Jena é um framework em Java para construir aplicações para a web semântica
- Suporte para RDF, RDFS, RDFa, OWL e SPARQL
- Incorpora um motor de inferência para OWL e RDFS
- Desenvolvdo por pesquisadores do laboratório de pesquisa da HP, o HP Labs
- Open-source \o/
- Incubabo pela Apache em 2010 e se tornou um projeto top-level em Abril de 2012
- Fornece classes e interfaces para criação e manipulação de RDF e Ontologias baseadas em OWL.

Capacidades do Jena

- API para RDF
- Ler e escrever em RDF/XML, N3 ou externos
- API para OWL
- Armazenamento em memória, arquivo, banco de dados ou externos
- Motor de consulta SPARQL
- Interface de comunicação através do protocolo HTTP

Arquitetura



Conceitos RDF

- Recursos(Resources), Propriedades(Properties), Literais (Literals), Sentençcas(Statements) (Triplas: <suj pred obj>)
- Um conjunto de sentenças constitui um grafo RDF
- A API do Jena para RDF contém classes e interfaces para todos os aspectos importantes da especificação RDF
- Pode ser utilizado para contruir grafos do zero, ou editar grafos existentes
- Estas classes e interfaces estão no pacote "com.hp.hpl.jena.rdf.model" *
- No Jena a interface Model é usada para representar os grafos RDF
- Através do Model, Statements podem ser obtidas, criadas, removidas, avaliadas, etc.

RDF API - Exemplos

```
// Criando um modelo vazio
Model model = ModelFactory.createDefaultModel();
String ns = new String("http://www.exemplo.com.br/exemplo#");
// Criando dois recursos
Resource joao = model.createResource(ns + "João");
Resource maria = model.createResource(ns + "Maria");
// Criando a propriedade "temIrmao"
Property temIrmao = model.createProperty(ns, "temIrmao");
// Associa Maria a João através da propriedade "temIrmao"
maria.addProperty(temIrmao, joao);
// Criando a propriedade "temIrma"
Property temIrma = model.createProperty(ns, "temIrma");
// Associa João a Maria através de uma sentença com a propriedade
"temIrmao"
Statement sentencaIrma = model.createStatement(joao, temIrma,
maria);
model.add(sentencaIrma);
```

Leitura e escrita de modelos

- Modelos RDF podem ser obtidos de fontes externas (arquivos, banco de dados)

Exemplo de um modelo obtido de um arquivo

```
// O local do arquivo é especificado através do prefixo "file:"
String caminhoArquivo = “file:meuRDF.rdf”;
// Um modelo em branco é criado
Model futuroModeloDoArquivo = ModelFactory.createDefaultModel();
// O modelo obtém as definições do arquivo informado
futuroModeloDoArquivo.read(caminhoArquivo);
```

Exemplo de um modelo sendo escrito para a saída padrão (console)

```
// O destino e o formato da saída é especificado
Model.write(System.out, “RDF/XML”)
```

Leitura de banco de dados

- O projeto "jena-sdb" contém as classes necessárias para o uso de modelos com armazenamento em banco de dados

Acessando um modelo em um banco de dados MySQL

```
JDBC.loadDriverHSQL();
// Criando uma conexão com o banco de dados
SDBConnection conn = new SDBConnection("jdbc:hsqldb:mem:jenadb",
"sa", "");
StoreDesc storeDesc = new StoreDesc(LayoutType.
LayoutTripleNodesHash, DatabaseType.HSQLDB);
// Obtendo um Store para a configuração
Store dbStore = SDBFactory.connectStore(conn, storeDesc);
```

Jena OWL API

- OWL é uma extensão do RDF. Essa relação é refletida no Jena
  - Classes e interfaces relacionadas a OWL extendem ou usam classes ou interfaces da API RDF do Jena
- Propriedade (Property) > Tipo de dados (DatatypeProperty), Objeto (ObjectProperty), Simetrica (SymmetricProperty), Funcional (FunctionalProperty), Inversamente funcional (InverseFunctionalProperty)...
- Resource > OntResource > OntClass, Individual
- Herança (OntClass.isHierarchyRoot)
- Equivalência (OntResource.sameAs) e Disjunção (OntResource. differentFrom)
- Restrições em propriedades (AllValuesFromRestriction, CardinalityRestriction, etc)
- A API do Jena para OWL contém classes e interfaces para todos os aspectos importantes da especificação OWL
- Estas classes e interfaces estão no pacote "com.hp.hpl.jena.ontology" *
- OntModel é a interface mais utilizada para manipular ontologias

Jena OWL API

- OntModel
  - Contém as sentenças da ontologia
  - Pode ser utilizado para obter os recursos existentes na ontologia (Classes, indivíduos, propriedades, etc) ou criar novos
- Classes são representadas por OntClass
  - A classe OntClass possui métodos para percorrer instâncias (indivíduos), superclasses, subclasses, restrições, etc, de uma classe em particular
- OntClass possui métodos para avaliar hierarquia e relações entre classe<>instância
- Classes podem ser apenas "textos" que caracterizam os indivíduos, mas também podems ser mais complexos
  - UnionClass, IntersectionClass, EnumeratedClass, ComplementClass, Restriction
  - A API para OWL do Jena possui métodos para verificar se a classe se enquadra em uma das categorias acima
  - OntModel possui método para criar tais classes

Jena OWL API

- Propriedades são representadas por OntProperty
  - OntProperty possui métodos que definem o domínio e o contradomínio de propriedades, assim como determinar o tipo de propriedade
  - DatatypeProperty, ObjectProperty, SymmetricProperty, FunctionalProperty...
  - Pode determinar relações de sub-propriedade ou super-propriedade
- Propriedades podem ser definidas por conta própria(ex., elas não são "presas" a certas classes, como acontece em sistemas de frames)
- Entretanto, as vezes é necessário obter uma propriedade de classe específica. Isso significa encontrar uma propriedade cujo domínio contenha a classe específica. Jena fornece método para tais tarefas.

OWL API Exemplo: Classes

```
// Criando um modelo de ontologia vazio
OntModel ontModel = ModelFactory.createOntologyModel();
String ns = new String("http://www.exemplo.com/onto1#");
String uri = new String("http://www.exemplo.com/onto1");
Ontology onto = ontModel.createOntology(uri);
// Criando as classes "Pessoa", "Masculino", "Feminino"
OntClass pessoa = ontModel.createClass(ns + "Pessoa");
OntClass masculino = ontModel.createClass(ns +
"Masculino");
OntClass feminino = ontModel.createClass(ns + "Feminino");
// Definindo Masculino e Feminino como subclasses de
Pessoa
pessoa.addSubClass(masculino);
pessoa.addSubClass(feminino);
// Definindo Masculino e Feminino são disjuntos
masculino.addDisjointWith(feminino);
feminino.addDisjointWith(masculino);
```

OWL API Exemplo: Propriedades Tipo de Dados

```
// Criando o tipo de dado "temIdade"
DatatypeProperty temIdade =
  ontModel.createDatatypeProperty(ns + "temIdade");
// "temIdade" utiliza contradomínio de valores inteiros, então o
"range" é "XSD.integer"
// Tipos de dados básicos(comuns) estão definidos no pacote
"vocabulary"
temIdade.setDomain(pessoa);
temIdade.setRange(XSD.integer);
// com.hp.hpl.jena.vocabulary.XSD
// Criando indivíduos
Individual joao = masculino.createIndividual(ns + "João");
Individual maria = feminino.createIndividual(ns + "Maria");
Individual jose = masculino.createIndividual(ns + "José");
// Criando sentença definindo "João temIdade 20"
Literal idade20 = ontModel.createTypedLiteral("20", XSDDatatype.
XSDint);
Statement joaoTem20 = ontModel.createStatement(joao, temIdade,
idade20);
ontModel.add(joaoTem20);
```

OWL API Exemplo: Propriedades Objeto

```
// Criando uma propriedade do tipo objeto "temIrmao"
ObjectProperty temIrmao = ontModel.createObjectProperty(ns
+ "temIrmao");
// Domínio e contradomínio são "Pessoa"
temIrmao.setDomain(pessoa);
temIrmao.setRange(pessoa);
// Adicionando sentenças "João temIrmao Maria" e "Maria
temIrmao João"
Statement joaoIrmaoMaria = ontModel.createStatement(joao,
temIrmao, maria);
Statement mariaIrmaoJoao = ontModel.createStatement(maria,
temIrmao, joao);
ontModel.add(joaoIrmaoMaria);
ontModel.add(mariaIrmaoJoao);
```

OWL API Exemplo: Propriedades restritivas

```
// Criando uma propriedade do tipo objeto "temConjuge"
ObjectProperty temConjuge = ontModel.createObjectProperty(ns +
"temConjuge");
temConjuge.setDomain(pessoa);
temConjuge.setRange(pessoa);
Statement joseConjMaria = ontModel.createStatement(jose, temConjuge,
maria);
Statement mariaConjJose = ontModel.createStatement(maria, temConjuge,
jose);
ontModel.add(joseConjMaria);
ontModel.add(mariaConjJose);
// Criando uma restrição do tipo "AllValuesFromRestriction" em temConjuge:
Masculino temConjuge SOMENTE Feminino
AllValuesFromRestriction soFeminino =
ontModel.createAllValuesFromRestriction(null, temConjuge, feminino);
// Criando uma restrição do tipo "MaxCardinalityRestriction" em temConjuge:
Masculino só pode ter no máximo 1 temConjuge
MaxCardinalityRestriction maximo1Conjuge =
ontModel.createMaxCardinalityRestriction(null, temConjuge, 1);
// Restringir Masculino com as duas restrições
masculino.addSuperClass(soFeminino);
masculino.addSuperClass(maximo1Conjuge);
```

OWL API Exemplo: Classes Definidas

```
// Criando a classe "PessoaCasada"
OntClass pessoaCasada = ontModel.createClass(ns + "PessoaCasada");
MinCardinalityRestriction minimo1Conjuge = ontModel.
createMinCardinalityRestriction(null, temConjuge, 1);
// Definindo "PessoaCasada": Pessoa & com no mínimo 1 cônjuge
// Uma lista precisa ser criada para conter a classe "Pessoa" e a restrição
RDFNode[] restricoesArray = { pessoa, minimo1Conjuge };
RDFList restricoes = ontModel.createList(restricoesArray);
// As duas classes são combinadas em uma class de interseção
IntersectionClass ic =
ontModel.createIntersectionClass(null, restricoes);
// "PessoaCasada" é declarada como equivalente a interseção definida
pessoaCasada.setEquivalentClass(ic);
```

Raciocínio

- Jena foi concebido de tal forma que motores de inferência podem ser plugados em Modelos e processá-los
- Jena já possui um pacote "com.hp.hpl.jena.reasoner" com as classes relacionadas
  - Todos os raciocinadores devem implementar a interface Reasoner
- Jena possui alguns motores de inferência, apesar de possuírem capacidade limitada
  - Os raciocinadores conhecidos estão na classe ReasonerRegistry
- Uma vez que o raciocinador seja obtido é preciso registrar ao modelo. Isso é feito modificando a especificação do modelo(OntModelSpec)

Raciocínio

- A especificação do modelo (OntModelSpec) é composta dos seguintes objetos
  - Armazenamento
  - Motor de inferência
  - Tipo de linguagem (RDF, OWL-Lite, OWL-DL, OWL Full, DAML)
- Jena possui algumas especificações predefinidas para tipos de modelos básicos
  - A especificação OntModelSpec.OWL_DL_MEM é uma especificação de modelos OWL-DL, armazenados em memória, sem motores de inferência
  - Motores de inferência também podem ser anexados:

```
// Obtendo o motor de inferência a partir da API Pellet
Reasoner raciocinador = PelletReasonerFactory.theInstance().create();
// Obtendo a especificação OWL-DL e anexando o raciocinador Pellet
OntModelSpec especRacioc = OntModelSpec.OWL_DL_MEM;
especRacioc.setReasoner(raciocinador);
// Criando o modelo com a especificação com suporte a inferência
OntModel ontModel = ModelFactory.createOntologyModel(especRacioc);
```

Raciocínio

- Fora a referência ao motor de inferência na especificação do modelo não é preciso fazer mais nada para ativar o raciocínio
- Modelos sem raciocinadores irão responder a consultas usando apenas as sentenças relacionadas, enquanto modelos com raciocinadores irão inferir novas sentenças, sem a interação do programador

```
// PessoaCasada não tem nenhuma instância relacionada
// Contudo, se um motor de inferência for usado, 2 dos 3
indivíduos serão reconhecidas como PessoaCasada
OntClass pessoaCasada = ontModel.getOntClass(ns + "PessoaCasada");
ExtendedIterator casados = pessoaCasada.listInstances();
while(casados.hasNext()) {
  OntResource casado = (OntResource) casados.next();
  System.out.println(casado.getURI());
}
```

Processamento de consultas com SPARQL

- Jena utiliza o motor ARQ para o processamento de consultas SPARQL
  - A classes da API ARQ podem ser encontradas no pacote "com.hp.hpl.jena.query"
- Classes básicas no ARQ:
  - Query: Representa uma simples consulta SPARQL
  - Dataset: A base de conhecimento que as consultas irão ser executadas (equivalente aos modelos RDF)
  - QueryFactory: Utilizado para criar objetos Query a partir de strings SPARQL
  - QueryExecution: Fornece métodos para a execução das consultas
  - ResultSet: Contém o resultado obtido da execução
  - QuerySolution: Representa uma linha do resultado
    - Se tiver várias respostas para a consulta, o ResultSet é retornado, dentro dele estão vários QuerySolution

Exemplo: Processamento de consultas com SPARQL

```
// Preparando a string da consulta
String stringConsulta =
"PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n" +
"PREFIX : <http://www.exemplo.com/onto1#>\n" +
"SELECT ?casado ?conjuge WHERE {" +
"?casado rdf:type :PessoaCasada.\n" +
"?casado :temConjuge ?conjuge." +
"}";
// Usando o modelo para criar um Dataset
// Nota: Neste cenário, caso o motor de inferência esteja
desativado, nenhum resultado será obtido (PessoaCasada não tem
nenhum instância relacionada)
Dataset dataset = DatasetFactory.create(ontModel);
// Fazendo o parse da string da consulta e criando o objecto Query
Query consulta = QueryFactory.create(stringConsulta);
// Executando a consulta e obtendo o resultado
QueryExecution qexec = QueryExecutionFactory.create(consulta,
dataset);
ResultSet resultado = qexec.execSelect();
```

Exemplo: Processamento de consultas com SPARQL

```
// Imprimindo os resultados
while(resultado.hasNext()) {
  // Cada linha contém dois campos: "casado" e "conjuge", assim foi definido na string da consulta
  QuerySolution linha = (QuerySolution) resultado.next();

  RDFNode casado = linha.get("casado");
  System.out.print(casado .toString());

  System.out.print(" é casado com ");

  RDFNode conjuge = linha.get("conjuge");
  System.out.println(conjuge.toString());
}
```

Observações

- Jena pode ser utilizado para controlar ontologias existentes ou criar novas do zero
  - Independente do modo de armazenamento
  - É necessário um certo conhecimento de ontologias representadas em Triplas e/ou XML, uma vez que conceitos complexos como restrições, listas e classes definidas podem ser necessárias a criação, caso contrário podem aparecer inconsistências
- Inferir sobre dados já existente para obter novos conhecimentos
  - Motores de inferência devem seguir a interface Java especificada
  - Para ontologias complexas a inferência pode diminuir a performance da aplicação
  - É importante saber quando um motor de inferência é realmente necessário

Bibliografia

- **APACHE JENA**. Online. Disponível em <jena.apache.org>. Acesso em: 05/03/2013.
- **W3C SPECIAL INTEREST GROUP**. Online. Disponível em <www.w3.org/2001/sw>. Acesso em: 05/03/2013.
- **TÓPICOS AVANÇADOS EM INTELIGÊNCIA ARTIFICIAL SIMBÓLICA**, Ontologias e a Web Semântica. Online. Disponível em <www.cin.ufpe.br/~in1099/122>. Acesso em: 05/03/2013.
- **SEMANTICWEB**. Online. Disponível em <http://answers.semanticweb.com/tags/jena/>. Acesso em: 05/03/2013.
- **STARDOG**. Online. Disponível em <stardog.com>. Acesso em: 05/03/2013.

FIM