VILLANOVA UNIVERSITY

CSC4480: Principles of Database Systems

Lecture 9: Programming APIs /Semi-Structured Data

- API allows programs to connect to databases
- Database access is the same for all database vendors

Typical Programming Procedure

1. Load the database driver
2. Obtain a connection
3. Create and execute statements (SQL queries)
4. Use result sets (tables) to navigate through the results
5. Close the connection

APIs to Connect to Oracle Database

- JAVA: JDBC
- Python: cx_oracle
- C/C++: ODBC, OCI, OCCI

Oracle database Connection using python

```python
# myscript.py

from __future__ import print_function

import cx_Oracle

# Connect as user "hr" with password "welcome" to the "orclpdb1" service running on this co
connection = cx_Oracle.connect("hr", "welcome", "localhost/orclpdb1")

cursor = connection.cursor()
cursor.execute("""
    SELECT first_name, last_name
    FROM employees
    WHERE department_id = :did AND employee_id > :eid""",
    did = 50,
    eid = 190)
for fname, lname in cursor:
    print("Values:", fname, lname)
```

In Class Exercise

- Given the dataset, use one of the APIs listed in slide 4 to print the first name and last name of all of the employee in the EMPLOYEE table.

Semi-structured Data

- Data that does not obey the format of a relational database
  - Contains semantics to denote meaning
  - Lightweight

- Types
  - XML
  - JSON
  - RDF
  - CSV

JSON

- JSON: **J**ava**S**cript **O**bject **N**otation: A Light-weight data-interchange format .
  - Language Independent.
  - Text-based.
  - Easy to parse.
    - Easy to understand, manipulate and generate

JSON is **NOT**

- A document format.
- A markup language.
- A general serialization format
- A programming language

JSON Syntax

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays


```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```


JSON DOCUMENT EXAMPLE

JSON Data Types

- String
- Number
- Object (JSON object)
- Array
- Boolean
- null

JSON Data Types

- Strings
  - Written in quotes “Hello”
- Numbers
  - Integer or floating point
- Object
  - {
    "employee":{ "name":"John", "age":30, "city":"New York" }
    }
- Arrays
  - {
    "employees":["John", "Anna", "Peter"]
    }

JSON Data Types (cont'd)

- Boolean
  - { "sale":true }
- Null
  - { "middlename":null }

More JSON Example

```
{
    "firstName": "John",
    "lastName": "Smith",
    "isAlive": true,
    "age": 27,
    "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
    },
    "children": [“mark”, “James”, “John”],
    "spouse": null
}
```

Parsing JSON via Programming Language

- JSON document structure can be parsed using various programming languages
  - Python - json
  - Java - GSON
  - C/C++ - Jansson

Sample Employee Data in JSON

```
{
  "employees": {
    "employee": [
      {
        "id": "1",
        "firstName": "Tom",
        "lastName": "Cruise"
      },
      {
        "id": "2",
        "firstName": "Maria",
        "lastName": "Sharapova"
      },
      {
        "id": "3",
        "firstName": "James",
        "lastName": "Bond"
      }
    ]
  }
}
```

Python program to parse JSON file.

```
import json

# when performing from a file in disk
with open("employee.json", "r") as readit:
    outer_object = json.load(readit)

inner_object = outer_object['employees']

values = inner_object['employee']

for value in values:
     print("First Name:"+value['firstName']+", "+ "Last Name:"+ value['lastName'])
```

e**X**tensible **M**arkup **L**anguage (XML)

- Designed to store and transport data.
- Designed to be human and machine readable
- An XML document resides in its own file with an '.xml' extension
- Open to define in any way you want
- A Mark up language

e**X**tensible **M**arkup **L**anguage (XML)

- Easy rules, but very strict
- First line is the version and character set used (optional):
  - <?xml version="1.0" encoding="ISO-8859-1"?>
- The rest is user defined tags
- Every tag has an opening and a closing

XML Example

Book Title: My First XML

Chapter 1: Introduction to XML

- What is HTML
- What is XML

Chapter 2: XML Syntax

- Elements must have a closing tag
- Elements must be properly nested

```
<book>
  <title>My First XML</title>
  <prod id="33-657"
  media="paper"></prod>
  <chapter>Introduction to XML
     <para>What is HTML</para>
     <para>What is XML</para>
  </chapter>
  <chapter>XML Syntax
     <para>Elements must have a
     closing tag</para>
     <para>Elements must be
     properly nested</para>
  </chapter>
</book>
```

Components of an XML document

- Elements
  - contains text, other elements, or be empty. Consists of a start and end tag. Although you can have elements that are empty with a single tag.
- Attributes
  - Provides additional information about an element

- Entities
  - Variables used to define common text.
- PCDATA (Parsed Character Data)
  - Parsed Character data
- CDATA (Character Data)
  - Portion of text that is interpreted literally as characters.

XML Example

Book Title: My First XML

Chapter 1: Introduction to XML

- What is HTML
- What is XML

Chapter 2: XML Syntax

- Elements must have a closing tag
- Elements must be properly nested

```
<book>
  <title>My First XML</title>
  <prod id="33-657"
  media="paper"></prod>
  <chapter>Introduction to XML
     <para>What is HTML</para>
     <para>What is XML</para>
  </chapter>
  <chapter>XML Syntax
     <para>Elements must have
  a  closing tag</para>
     <para>Elements must be
     properly nested</para>
  </chapter>
</book>
```

XML Basic Rules

- XML is case sensitive
- All start tags must have end tags
- Elements must be properly nested
- XML declaration is the first statement
- Every document must contain a root element
- Attribute values must have quotation marks
- Certain characters are reserved for parsing

Common Errors for Element Naming

- Do not use white space when creating names for elements
- Element names cannot begin with a digit, although names can contain digits
- Only certain punctuation allowed – periods, colons, and hyphens

Document Type Definition (DTD)

- A Document Type Definition (**DTD**) allows the developer to create a set of rules to specify legal content and place restrictions on an XML file
- If the XML document does not follow the rules contained within the DTD, a parser generates an error
- An XML document that conforms to the rules within a DTD is said to be **valid**

Document Type Definition (DTD)

```
<?xml version="1.0" encoding
="UTF-8"?>
<!DOCTYPE note SYSTEM
"Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this
weekend!</body>
</note>
```

```
<!DOCTYPE note
[
<!ELEMENT note
(to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Document Type Definition (DTD)

- DOCTYPE note
  - Defines that the root element of the document is note
- ELEMENT note
  - Defines that the note element must contain the elements: "to, from, heading, body"
- ELEMENT to
  - Defines the to element to be of type "#PCDATA”
- ELEMENT from
  - Defines the from element to be of type "#PCDATA"
- ELEMENT heading
  - Defines the heading element to be of type "#PCDATA"
- ELEMENT body
  - Defines the body element to be of type "#PCDATA"

XML SCHEMA

- Describes the structure of an XML document, just like a DTD.
- An XML document with correct syntax is called "Well Formed".
- An XML document validated against an XML Schema is both "Well Formed" and "Valid".

ADVANTAGES OF USING XML SCHEMA

- It is easier to describe document content
- It is easier to define restrictions on data
- It is easier to validate the correctness of data
- It is easier to convert data between different data types
- XML Schemas is that they are written in XML:
  - You don't have to learn a new language
  - You can use your XML editor to edit your Schema files
  - You can use your XML parser to parse your Schema files
  - You can manipulate your Schemas with the XML DOM
  - You can transform your Schemas with XSLT

Schema use

- Use of a schema assures that the communicating entities have the same understanding of how data is represented and how it should be interpreted.
- Classic case: dates. Is 02-01-2006 February 1, 2006 or January 2, 2006?
  - Schema specifies how the data values are to be interpreted.
- Use of a schema can help detect errors in use within a well-formed statement.
  - Ex: May include a range of legal values.

Parsing XML via Programming Language

- XML Document Structure can be parsed using various programming languages
  - Python - xml.etree.ElementTree
  - Java - DOM parser
  - C/C++ - Xerces

Parsing XML using Python

```
<foo>
    <bar>
        <type foobar="1"/>
        <type foobar="2"/>
    </bar>
</foo>
```

```
import xml.etree.ElementTree as ET
root = ET.parse('thefile.xml').getroot()
for type_tag in root.findall('bar/type'):
    value = type_tag.get('foobar')
    print(value)
```

prints attributes values

In Class Exercise

- Given the Json file containing a list of books. Parse the json file and print the author's name, and title of each book contained in the file.

- Given the xml file containing list of cars. Parse the xml file and print the year, model and color of each car contained in the file.