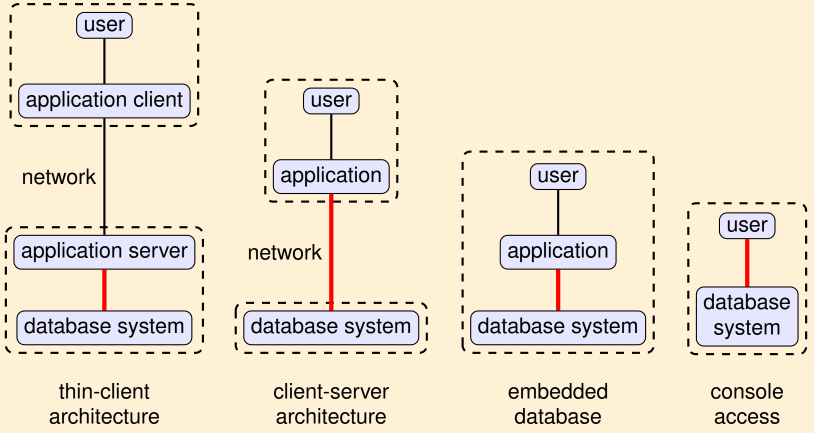
Databases – Application Programming

Jörg Endrullis

VU University Amsterdam

Application Architectures

Various ways of using database technology



How do these applications talk to the database?

How to Talk to a Database?

Database application programming:
how to access a database from an application?

- **Static** embedded queries
  - static SQL (preprocessor-based language extension)
  - inflexible, but syntax checked at compile time
  - e.g. SQLJ, Embedded SQL (C/C++)
- **Dynamic**
  - dynamic SQL (queries constructed at runtime)
  - application programming interface (API)
  - powerful, but error-prone
  - e.g. JDBC, Python DB-API, ODBC, OLE-DB,...
- **Object Relational Mappings (ORM)**, and beyond
  - hide navigational access behind objects
  - e.g. JPA/Hibernate, RubyOnRails, ADO.NET/LinQ

Application Programming :: Dynamic SQL

Dynamic SQL: JDBC

A Java Database Connectivity (JDBC) example:

```
Connection conn = DriverManager.getConnection(url);

Statement stat = conn.createStatement() ;
ResultSet rs = stat.executeQuery(
  "select sid, name from students"
);

while (rs.next()) {
  int sid = rs.getInt("sid");
  String name = rs.getString("name");
  System.out.println(sid + ": " + name);
}
conn.close();
```

fetch results row by row

getInt(...), getString(...) fetch column values by name

Use `rs.wasNull(attribute)` to check if `attribute` is `null`.

The **Impedance Mismatch**: database query language does not match the application programming language.

(Different data models and data types.)

Type (mis)Match

Mapping SQL types to Java Types

SQL type	**Java Type**
char, varchar	String
numerical, decimal	java.math.BigDecimal
bit	boolean
tinyint	byte
smallint	short
integer	int
bigint	long
real	float
float, double	double
binary, varbinary	byte[]
date	java.sql.Date
time	java.sql.Time
timestamp	java.sql.Timestamp

The match is not precise! E.g. `varchar(20)` versus String.

Dynamic APIs: Advantages and Disadvantages

Advantages and Disadvantages of Dynamic APIs

- powerful, flexible, but error-prone
- SQL query given as strings may be incorrect
  - no error checking at development time
  - column names and types unknown at compile time
- risk of SQL injection
- mismatch between SQL and Java types (isNull)

Dynamic SQL: Optimising Applications

Improving Performance of Applications

- **Connection pooling**:
  - keep DB connection open, reduces latency
- **Prepared statements**:
  - SQL calls that are repeated often
  - allows driver to optimise queries (precompiled by DBMS)
  - in JDBC created with **Connection.prepareStatement()**
  - allows parameters: `select * from products where id = ?`
- **Stored procedures** to reduce #query roundtrips
  - written in DB-specific language, not portable ↯
  - in JDBC accessed with **Connection.prepareCall()**
- Use a **driver** that is **bulk**-transfer optimised
  - when retrieving large result sets
  - driver can send several tuples in a single network packet

Application Programming :: SQL Injection

SQL Injection

Website with Login Screen

Name:	Maria
Password:	12345

Server Side SQL

```
String userName = // name that the user has entered
String userPassword = // password that the user has entered

ResultSet rs = stat.executeQuery(
  "select balance from accounts " +
  "where  name = '" + userName + "'" +
  "   and password = '" + userPassword + "'"
);
```

The Resulting SQL Query

```
select balance from accounts
where name = 'Maria' and password = '12345'
```

SQL Injection

Website with Login Screen

Name:	Joe' --
Password:	who cares

Server Side SQL

```
String userName = // name that the user has entered
String userPassword = // password that the user has entered

ResultSet rs = stat.executeQuery(
  "select balance from accounts " +
  "where  name = '" + userName + "'" +
  "    and password = '" + userPassword + "'"
);
```

The Resulting SQL Query

```
select balance from accounts
where name = 'Maria' and password = '12345'
```

SQL Injection

Website with Login Screen

Name:	Joe' --
Password:	who cares

Server Side SQL

```
String userName = // name that the user has entered
String userPassword = // password that the user has entered

ResultSet rs = stat.executeQuery(
  "select balance from accounts " +
  "where  name = '" + userName + "'" +
  "    and password = '" + userPassword + "'"
);
```

The Resulting SQL Query

```
select balance from accounts
where name = 'Joe' -- ' and password = 'who cares'
```

SQL injection is a very common mistake! Very dangerous!

SQL Injection: How to Prevent It?

To Prevent SQL Injection

- **Never build SQL queries with user input using string concatenation!**
- Use the API to fill in the query parameters.

Preventing SQL Injection

```
String userName = // name that the user has entered
String userPassword = // password that the user has entered

PreparedStatement stat = conn.prepareStatement(
  "select balance from accounts " +
  "where  name = ? " +
  "   and password = ? ");

// use the API to fill the name and password
stat.setString(1, userName);
stat.setString(2, userPassword);

ResultSet rs = stat.executeQuery();
```

Application Programming :: Object Relational Mapping

Object Relational Mapping

Database schemas (tables) are not always ideal

- not the same set of constructs and abstractions
- in programming languages: objects, relations, inheritance

In applications we would like to work with

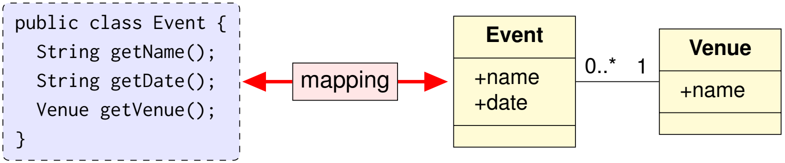
- objects / entities
- inheritance
- relations

Object Relational Mapping

Object Relational Mapping

Maps rows in tables to objects:

- table ≈ class
- row ≈ object
- foreign key navigation ≈ pointers / references



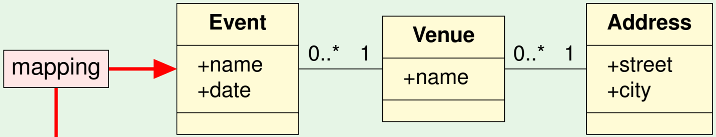
```
public class Event {
  String getName();
  String getDate();
  Venue getVenue();
}
```

Ingredients

- mapping from objects to database (automatic or designed)
- run-time library handles interaction with the database

Many ORM toolkits: Hybernate, RubyOnRails, ADO.NET,...

Object Relational Mapping: JPA/Hibernate



```
public class Event {
  String getName();
  String getDate();
  Venue getVenue();
}
```

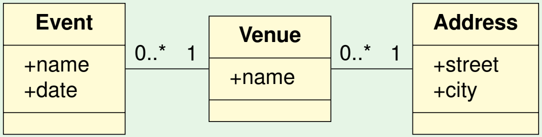
Example:

- `event.getVenue().getAddress().getStreet();`

Under the hood:

- `venue = SELECT * FROM Venues WHERE VenueId = event.venueID`
- `addr = SELECT * FROM Addresses WHERE AddressId = venue.addressID`
- `return addr.getStreet()`

Object Relational Mapping: Dangers



We want all events in Amsterdam:

```
List<Event> eventList = // get all events
for (Event event : eventList) {
  Address address = event.getVenue().getAddress();
  if ("Amsterdam".equals(address.getCity())) {
    System.out.println(event.getName());
  }
}
```

Inefficient!

Instead of loading just the events with city "Amsterdam":

- loads all events, and then iterates through all of them
- also each call to getVenue() will result in an SQL query

JPA/Hibernate: HQL Queries

HQL queries query the object-representation of data:

- Allows member access, e.g. employee.department.name.
- This is not calling methods on the objects!
- Query may return objects (if you are careful).

HQL Query: all events in Amsterdam

```
Query query = em.createQuery("from Events as event
                 where event.venue.address.city = 'Amsterdam'");
List<Event> eventsInAmsterdam = (List<Event>) query.list();
for(Event event : eventsInAmsterdam) {
  ... something ...
}
```

This is a more efficient way to get the events in Amsterdam.
Under the hood translated to SQL with two joins (3 tables).

Many queries do not return a full object!
E.g. what is the type of "select name,date from Events"?

Important Aspects of ORM Toolkits

- Mapping specification:
  - map relational data onto objects
  - can largely be derived automatically
- Query language (e.g. HQL):
  - adds object-oriented features to SQL
  - typically queries as strings (second class citizen)
- Persistence:
  - transaction semantics
  - languages offer start of transactions, commit, abort
- Fetch strategies
  - danger of implementing queries in Java ↯
  - object caching

Challenges of ORMs

- ORMs introduce an additional level of complexity
  - can be difficult to debug
- Performance analysis is problematic because:
  - database queries are under the hood
  - sometimes very complex SQL queries are generated
  - difficult to understand what caused the complex queries

ADO.NET Entity Framework

ADO.NET Entity Framework

- Different applications can have different views on the data.
- **Views entirely implemented on the client side**.
  - Avoid polluting DB schema with per-application views.
  - No added maintenance on the database side.

  (ANSI-SPARC model has views on server side)
- Powerfull
  - Broad set of views that are updatable.
  - Updatability can be statically verified.

ADO.NET Entity Framework

Entity Data Model (EDM)

Data representation on client side: Entity Data Model.

- **Entity type** = structured record with a key
- **Entity** = instance of an Entity Type
- Entity types can inherit from other entity types

Object-relational mapping

The EDM is then mapped to the logical database schema.

- can be queried similar to HQL
- can be queried similar to JDBC

Can we do better?

LinQ

LinQ

LinQ stands for Language INtegrated Query. Allows developers to query data structures using an SQL-like syntax.

Advantages of LinQ

- **Queries are first-class citizens** (not strings).
- **Full type-checking and error checking for queries.**
- Allows to query all collection structures.
  (lists, sets, . . . ; not restricted to databases)

Problem

LinQ is not portable! Only available for C# and Visual Basic.

Luckily. . . similar frameworks in other programming languages.

LinQ

LinQ: Querying an array

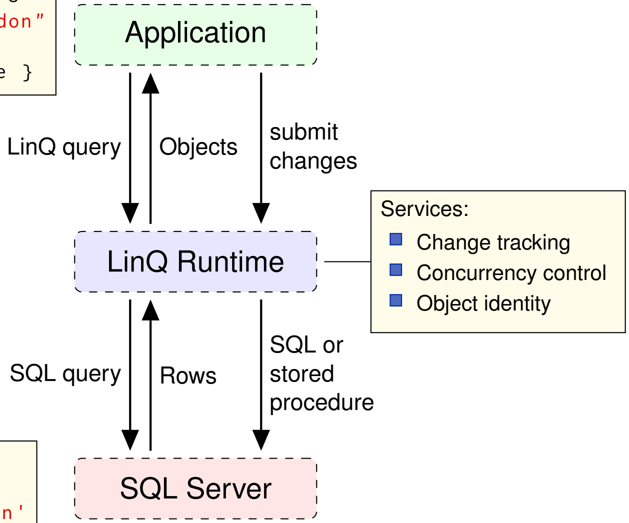
```
//Create an array of integers
int[] myarray = new int[] { 49, 28, 20, 15, 25, 23, 24, 10, 7 };

//Create a a query for odd numbers,
var oddNumbers = from i in myarray where i \% 2 == 1 select i;

//Odd numbers in descending order
var sorted = from i in oddNumbers orderby i descending select i;

//Display the results of the query
foreach (int i in oddNumbers)
  Console.WriteLine(i);
```

LinQ allows query various kinds of data sources:

- LinQ to DataSet (querying data sets like lists)
- LinQ to XML
- LinQ to SQL (interact with logical database model)
- **LinQ to Entities** (interact with conceptual/object model)

LinQ: What the Runtime Module Does

```
from c in db.Customers
where c.City == "London"
select
new { c.Name, c.Phone }
```



Services:

- Change tracking
- Concurrency control
- Object identity

```
select Name, Phone
from customers
where city = 'London'
```

LinQ: Under the Hood

Syntactic sugar...

```
var contacts =
  from c in customers
  where c.State == "WA"
  select new { c.Name, c.Phone };
```

Syntactic sugar for an expression with lambda expressions:

Query operations with lambda expressions

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c => new{c.Name, c.Phone});
```

LinQ: Under the Hood

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c => new{c.Name, c.Phone});
```

Here customers is of type IEnumerable<Customer>.

IEnumerable<...> provides methods for querying:

```
public static IEnumerable<T>
         Where<T>(this IEnumerable<T> src,
                  Func<T, bool>> p);
```

`Func<T, bool>> p` converted on-the-fly in an **expression tree** (a delegate). This is then translated into an SQL expression...

Database APIs

After this lecture, you should be able to:

- Explain the problem of **impedance mismatch.**
- Be able to classify DB application interfaces:
  - static, dynamic, object-relational mapping
- Discuss advantages and disadvantages of an API in terms of object **navigation** and complex **query execution.**
- Understand object-relational mappings:
  - **Hibernate** for Java
  - **Entity Framework** for .NET

  Relate these to the ANSI SPARC 3-layer model and the concepts of logical and physical data independence
- Explain advantages of **LinQ** and how it relates to impedance mismatch.