Chapter 15

Database Application Architectures

Database Applications

- Most users do not interact directly with a database system
  - The DBMS is hidden behind application programs
- There are lots of different systems implementing database applications
- At an abstract level, however, these systems all have a component for the
  - presentation
  - application logic
  - resource management

Presentation Layer

- This component is responsible for communicating with the outside world:
  - Presenting information to external entities (may be humans or other systems)
  - Allowing the external entities to interact with the system
- Examples:
  - Grapical user interfaces
  - Web servers (offering an HTML interface or Web services)

Application Logic Layer

- Usually a system does not just
  - throw raw data at a user
  - and write input directly to the database
- The data is processed in some way before presenting or inserting it
- This is the job of the application logic

Resource Management Layer

- Clearly, a system needs data to work with
- This data can come from different sources such as databases, file systems, or other repositories
- In our case we focus on databases, not general data resources
- In this context, the resource management layer is often referred to as *data access layer*

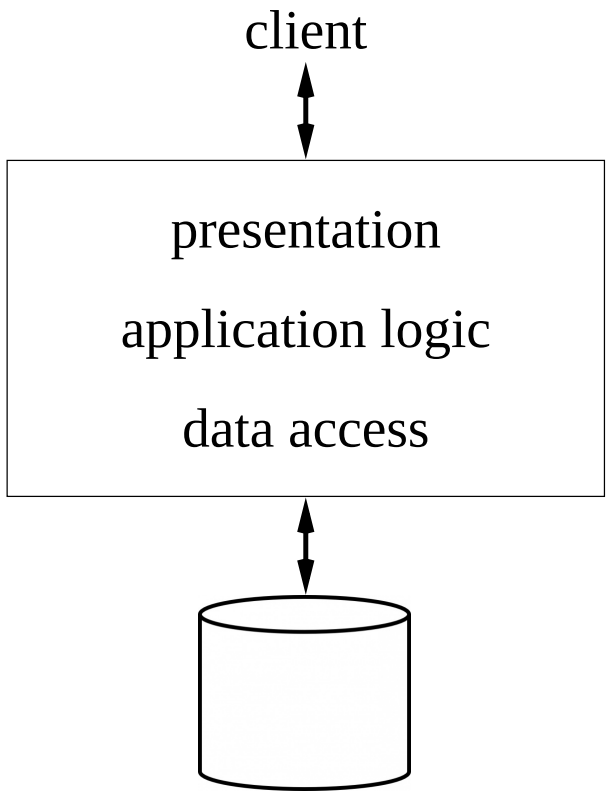
How Does It All Fit Together?

- Historically, the architecture of systems went through different phases
- We are going to look at
  - one-tier
  - two-tier
  - three-tier
  - n-tier

  architectures

One-Tier Architecture

- Oldest model is a one-tier architecture, i.e., all layers are integrated into a single tier:

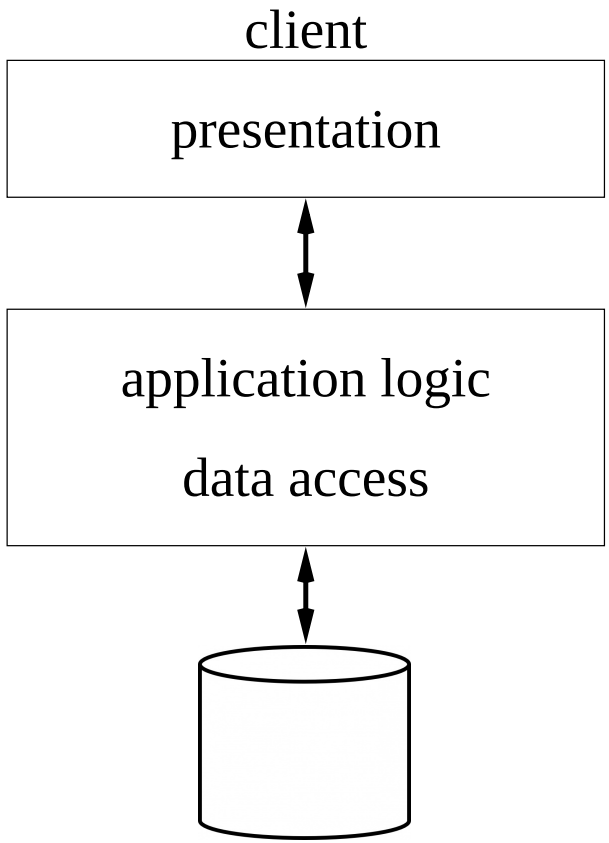


One-Tier Architecture (2)

- A one-tier system tends to be very monolithic
- Often the different layers are not clearly separated, but knitted together for optimization
- A typical example of a one-tier system is a mainframe with dumb terminals

Two-Tier Architecture

- Next up is the two-tier architecture with smarter clients

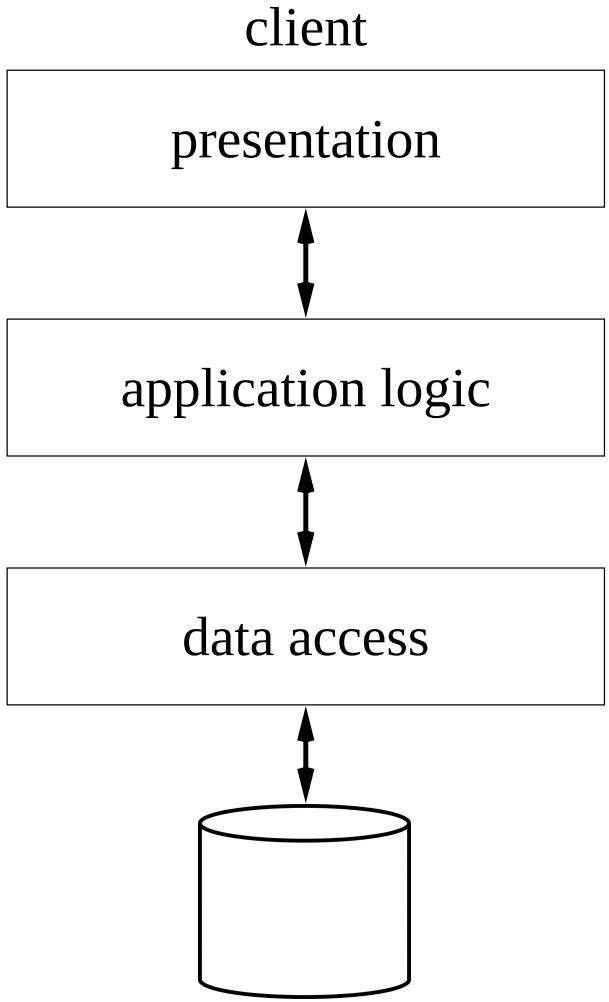


Two-Tier Architecture (2)

- With the emergence of PCs, it was possible to move the presentation layer to the client
  - This frees computing resources on the main system
  - The presentation can be adapted to different needs

Three-Tier Architectures

- Three-tier architecture systems are still developed today

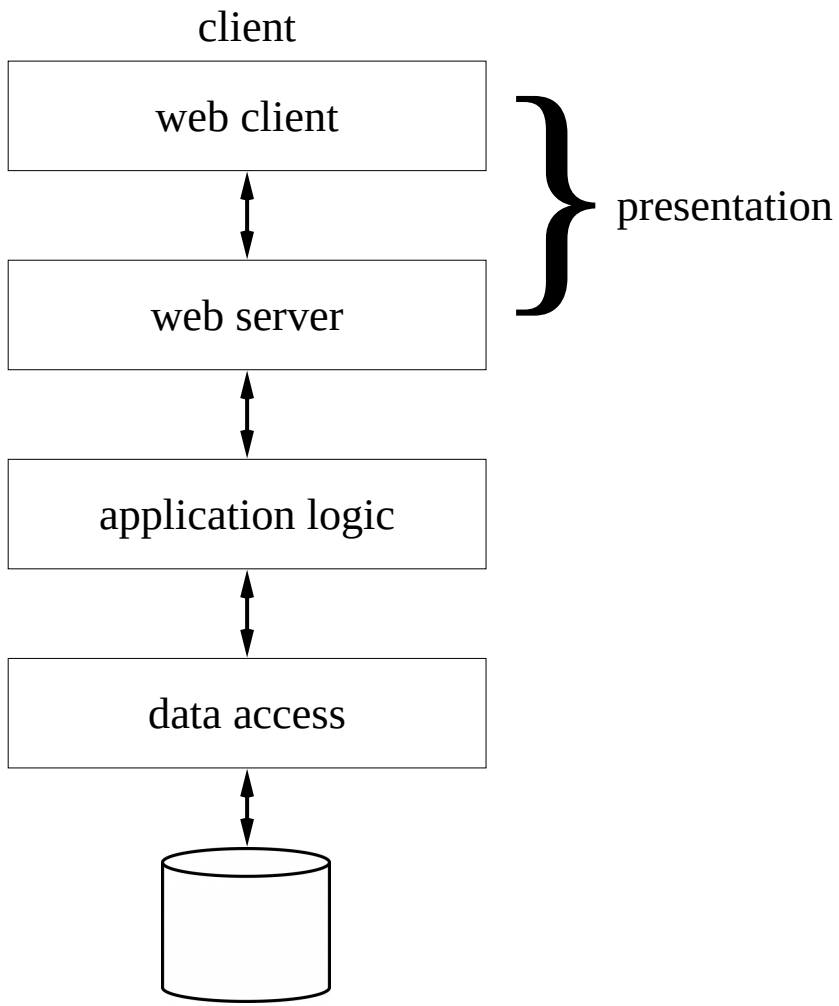


Three-Tier Architectures (2)

- Three-tier architectures clearly separate all three layers from each other
- This has several advantages:
  - Can be used to integrate different resources into a system
  - Application logic can be moved to a different data access layer
  - It is possible to scale the application logic and data access layers independently
- There is a disadvantage, though:
  - Communication overhead between application logic and data access layer

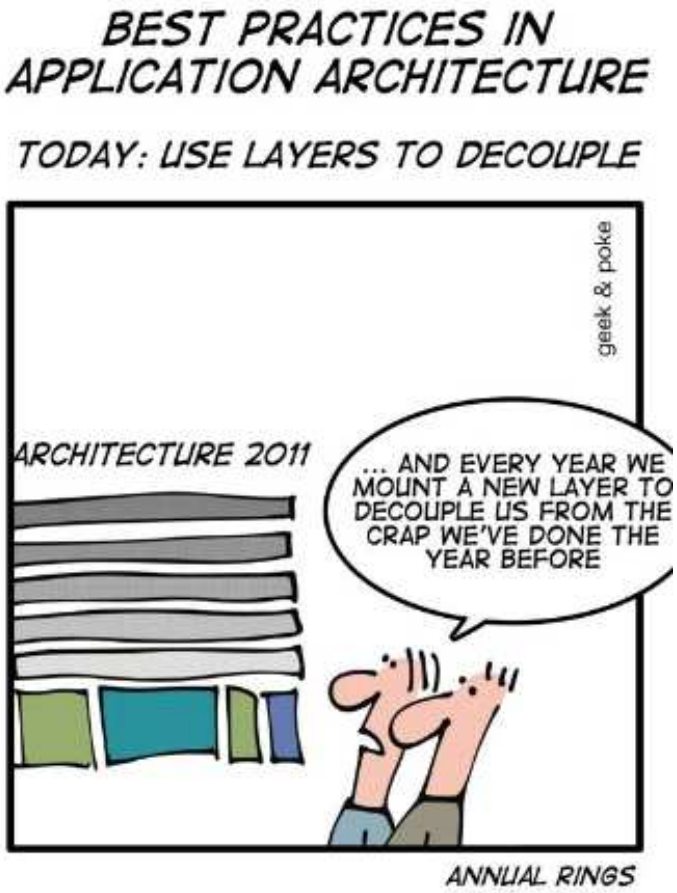
N-Tier Architectures

- Finally, we have n-tier architectures (generalization of three-tier architectures):



N-Tier Architectures (2)

- N-tier architectures allow the system to be split up to a finer granularity
- In the previous diagram, the presentation layer was divided up into a web server and web client
- Another example would be to allow two- or three-tier systems as components in the resource management layer

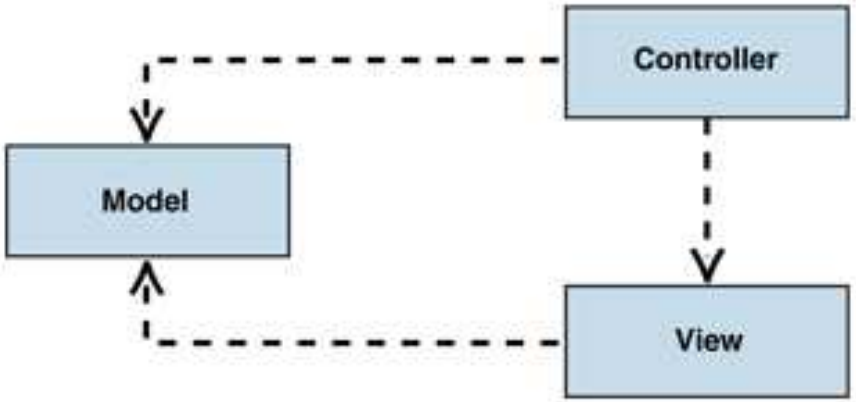


Model-View-Controllers

- For applications built on smaller scales, three- or n-tier architectures may be an overkill
- A well-known design pattern used for these cases is a *model-view-controller* (MVC):
  - Model: a representation of the data + defining the behavior
  - View: generates output information by displaying the data
  - Controller: reacts to user input and actions

Model-View-Controllers (2)

- The following diagram shows the dependencies:



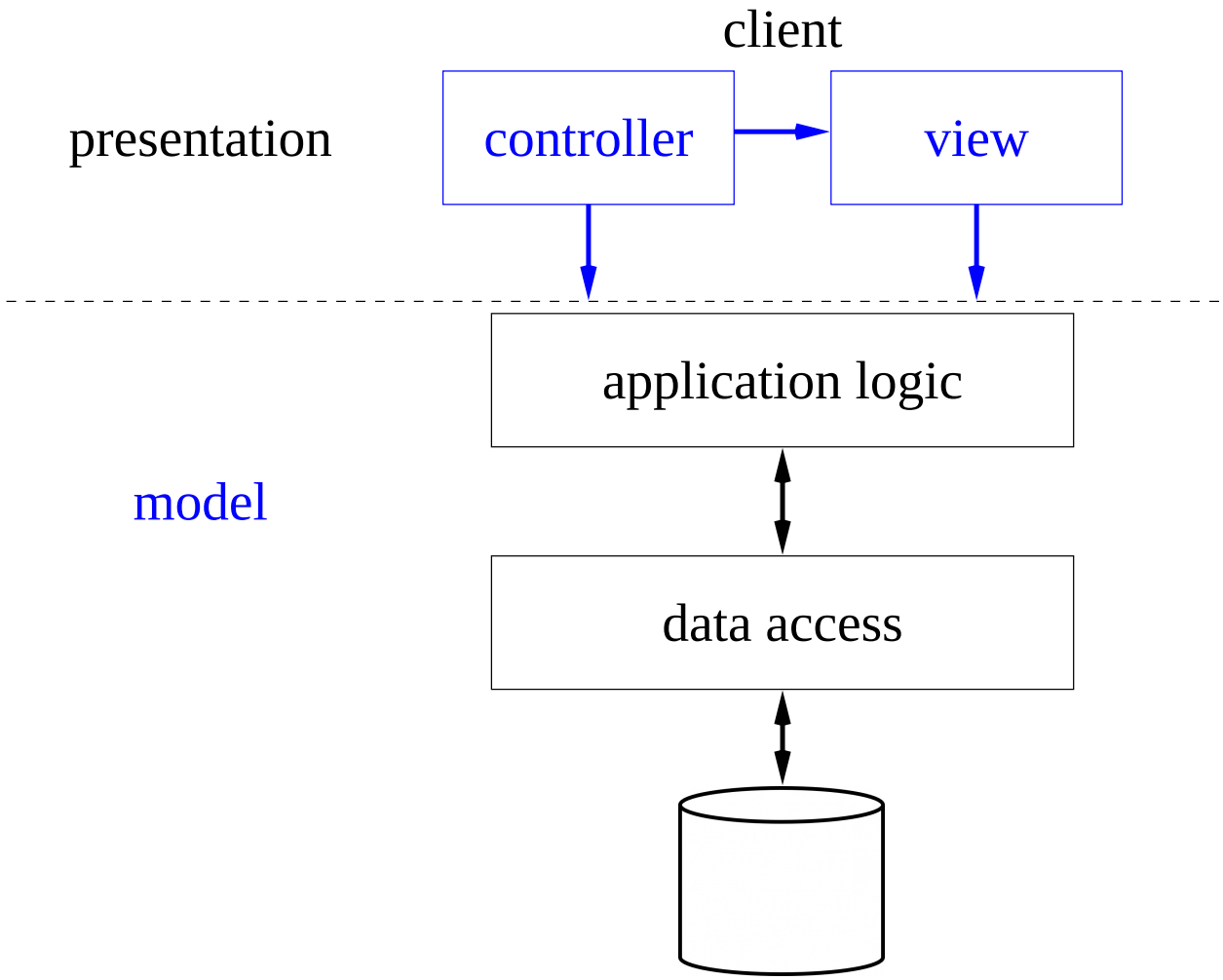
- Put simply:
  - The view is responsible for displaying stuff on a screen
  - The model is responsible for retrieving and updating state
  - The controller is responsible for handling the interaction between the user, view, and model

MVC vs. Three-Tier Architecture

- MVCs and three-tier architectures may look similar
  - However, there are some major differences
- Strengths and weaknesses of MVC:
  - \+ Great at building and maintaining user interfaces
  - – Not so great at building big scalable systems
- Strengths and weaknesses three-tier architecture:
  - \+ Great at building scalable server infrastructure
  - – Not so great at creating user interfaces across multiple platforms

MVC vs. Three-Tier Architecture (2)

- Nevertheless, the two can be combined in a single architecture (best of both worlds):



Data Access Layer

- After having a look at the big picture, we are now focusing on the data access layer
- A low-level approach of integrating a relational database into an object-oriented application is JDBC
- This is just an API for executing SQL and collecting the results:
  - Relatively simple to use
  - Does not help in bridging the gap between “relational” and “object-oriented”
- What we really want is an *Object-Relational Mapping* (ORM)

Object-Relational Mapping

- An (almost) automatic ORM that (almost) always works is not at all straightforward
- The object-oriented and the relational data model are two data models with completely different philosophies
- The incompatibility of data models is also called *impedance mismatch*

Encapsulation/Information Hiding

- One central aspect of OO is to hide implementation details
  - Allows us to swap parts of the implementation without affecting the rest of the system
- This concept is not really present in the relational model
  - Views could be seen as a very rudimentary form of “information hiding”
- Mapping objects to relational tables exposes their internal, hidden state

Encapsulation/Information Hiding (2)

- Additionally, objects encapsulate data and operations
- This is not possible in the relational model, there is only a set of standard operators
- Workarounds don’t accomplish much:
  - Storing code in binary large objects is not very interactive
  - Re-implementing the methods as stored procedures would create a lot of redundancy

Inheritance

- Another central aspect of the object-oriented model is inheritance
  - This allows code re-usage
- This concept is completely missing from the relational model
  - Compare to the mapping of generalizations from ER-diagrams into relational schemas

Object Identity

- Every object is identified by a unique ID, usually a reference (i.e., memory address)
  - Two objects having the exactly same content are still different
- Identity in the relational model is defined purely on content
  - Can be solved by introducing artificial surrogate keys

Navigation

- Objects are connected via references, forming a graph
  - Following a reference is a trivial operation
- In the relational model every operation is set-oriented
  - In order to connect data, expensive join operators are necessary

Structural Differences

- Objects can be composed of other objects, resulting in an arbitrary deep nesting
- Can also be structured in different ways, e.g.
  - lists of objects
  - hashmaps of objects
  - sets of objects
- The relational model has only one native composition: tuples in relations
  - This composition is always set-based

Transactions

- Concurrency works at a much finer granularity when using objects
  - Critical sections are usually about assigning a couple of values in individual objects
- In the relational model there is a much coarser granularity
  - Queries potentially handling a very large number of tuples can be bundled into a single transaction

How Do We Solve This?

- First important point: there is no perfect solution to ORM
- In general, no framework will be able to automatically get 100% of the mapping correct
- So, what do we do know? Look the other way or ignore ORM?

Avoiding the Problem

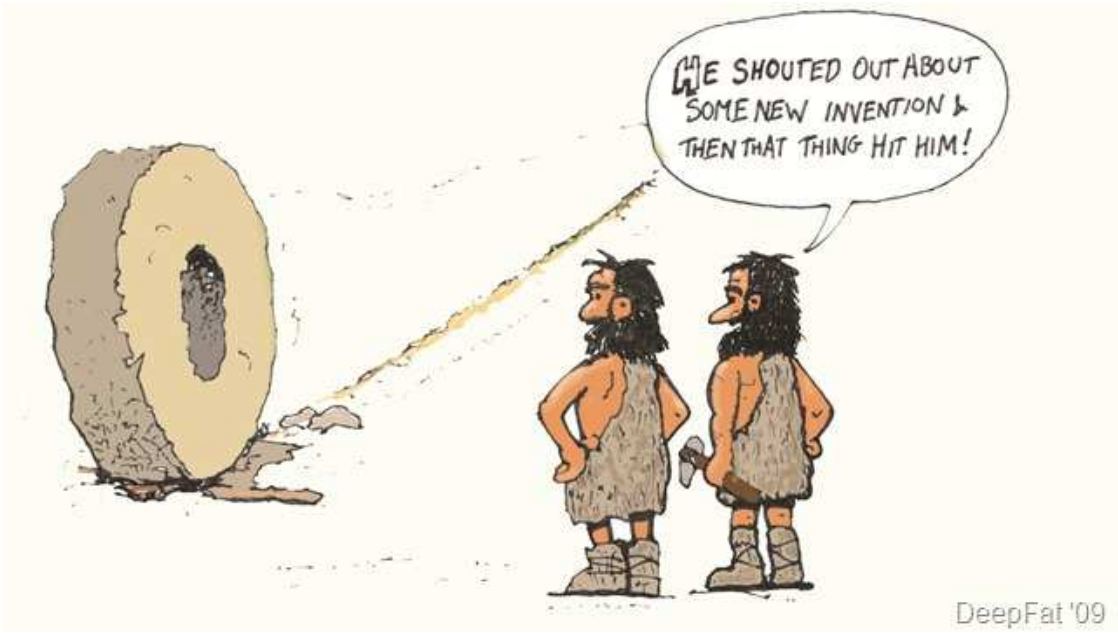
- The most extreme approach would be to abandon one of the sides
- Get rid of the oo-side, i.e., use a non-object-oriented programming language/architecture
  - If we have no objects, there is no OR impedance mismatch
  - May be possible to do for smallish applications
  - The code is centered around the relational model, using JDBC to communicate with the database

Avoiding the Problem (2)

- Get rid of the relational side, i.e., use an object-oriented DBMS
  - While pure OODBMS certainly exist, their market share is tiny (compared to RDBMS)
  - OODBMS never reached the performance and maturity of RDBMS
  - Mostly used for niche applications such as CAD/CAM
- With the advent of NoSQL systems, it may be possible to find a model that fits better

Facing ORM

- It's not always possible to bypass the problem
- Very often you have an object-oriented application connected to an RDBMS
- One option you always have is: do the mapping yourself
  - Involves a lot of work. . .
  - Not too exciting, often you'll be re-inventing the wheel

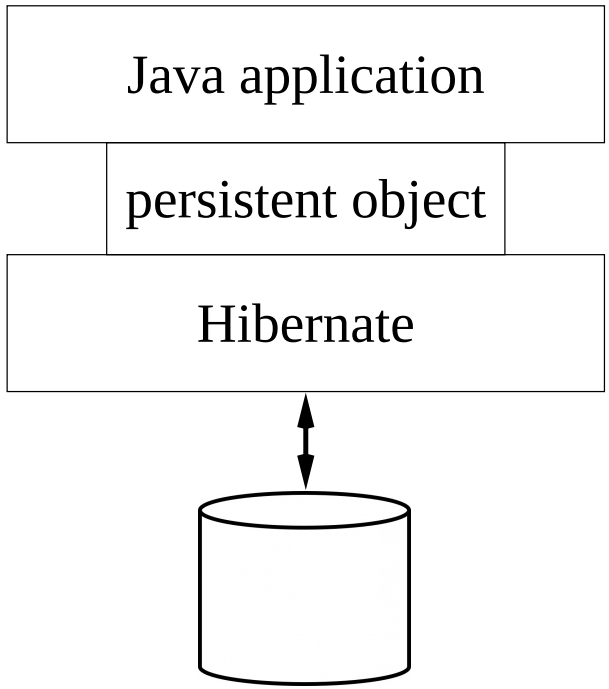


Facing ORM (2)

- Your best option is probably to use an ORM-framework
- A framework will take care of most of the mappings in a semi-automatic way
- We have a brief look at three ORM-frameworks
  - Hibernate
  - MyBatis
  - ActiveRecords

Hibernate

- Hibernate is an ORM library for Java
- It is freely available under a GNU license
- High level overview:



Hibernate (2)

- On the application level you build POJO classes
  - POJO = Plain Old Java Object, an ordinary Java object not extending special classes
- Classes should follow JavaBeans conventions, though
- To connect objects to database tables, mapping configuration files in XML have to be created

Example

- Let's look at a POJO class:

```
public class Employee {
  private int id;
  private String firstName;
  private String lastName;
  private int salary;

  public Employee() {}
  public Employee(String f, String l, int s) {
    ...
  }

  public int getId() {...}
  public void setId(int id) {...}

  and all the other getters and setters
```

Example (2)

- The corresponding table on the relational side could look like this:

```
create table EMP (
  id INT NOT NULL PRIMARY KEY,
  first_name VARCHAR(30),
  last_name  VARCHAR(30),
  salary     INT
);
```

Example (3)

- Now we need to tell Hibernate how the two are connected:

```
<hibernate-mapping>
  <class name="Employee" table="EMP">
    <meta attribute="class-description">
      Our example mapping
    </meta>
    <id name="id" type="int" column="id">
      <generator class = "native"/>
    </id>
    <property name="firstName"
              column="first_name"
              type="string"/>
    <property name="lastName"
              column="last_name"
              type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```

Example (4)

- The example on the previous slides was a pretty straightforward case
- Hibernate can also handle more complex mapping scenarios such as
  - associations
  - subclasses
  - collections
- There are also tools for generating standard XML-mapping files

Running Hibernate

- (Most of) the SQL code to interact with the database is generated automatically by Hibernate
- Access to database tables is (mostly) via the Hibernate API
- The tasks are organized in sessions
- During a session the state of an object is synchronized with data in a table
  - Changing the state of/inserting/deleting an object will trigger the corresponding SQL updates/insertion/deletions on the table

MyBatis

- MyBatis (formerly known as iBatis) is a persistence framework for Java and .NET
- It is free software available under the Apache license
- MyBatis does not map objects to tables, but methods to SQL statements
  - Similar to Hibernate, the mapping can be defined via XML

Example

- Let's assume we want to retrieve some blog entries stored in a database:
- A possible XML mapping could look like this

```
<mapper namespace="org.mybat.ex.BlogMapper">
  <select id="selectBlog"
          parameterType="int"
          resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

Example (2)

- Interactions with the database are organized in sessions
- For example, retrieving the information out of the blog database could be as simple as

```
BlogMapper mapper =
  session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

More On MyBatis Mappers

- There are also constructs for telling MyBatis what to do with the result of a query
- For example, setting `resultType` equal to `hashmap` will map the result values to the keys of a HashMap
- More useful is probably mapping them to POJOs
  - The value of a `resultType` can be a class name
  - If the column names match the attribute names, the mapper will be created automatically
- Update, insert, and delete elements can also be used within XML mapper files

More On MyBatis Mappers (2)

- Again, this was a rather straightforward example
- MyBatis, like Hibernate, can handle complex mapping scenarios as well
- This includes constructs such as
  - associations
  - collections
  - constructors

Active Record

- There are several implementations of the Active Record pattern:
  - ColdFusion
  - .NET
  - PHP
  - Ruby
  - Java
  - and a couple of other languages
- We are going to look at the Ruby version

Active Record in Ruby

- Active Record is implemented in the Ruby on Rails framework
- Let's stick with the blog example for a little longer
- Assume we have a table `posts` with the attributes `content`, `name`, and `title`
- Then creating the actual model on the Ruby side can be as simple as

```
class Post < ActiveRecord::Base
  attr_accessible :content, :name, :title
end
```

Example Cont'd

- Getting a post is also not very hard to do:

```
post = Post.find(10)
```

  is equivalent to the following SQL statement:

```
SELECT *
FROM   posts
WHERE  posts.id = 10
```

  assuming that the primary key of `posts` is `id`

- Changing the title of the second post is also not a big problem:

```
p = Post.find(2)
p.title = "Me too!"
p.save
```

Connecting Two Models

- If a post can be accompanied by multiple comments, we can model this in the following way:

```
class Post < ActiveRecord::Base
  attr_accessible :content, :name, :title
  has_many :comments
end

class Comment < ActiveRecord::Base
  attr_accessible :date, :text
  belongs_to :posts
end
```

Advanced Querying

- Querying this association is also not difficult:

```
p = Post.find(1)
p.comments.each do |comm|
    puts "#{comm.text}"
end
```

- Also allows you to go deeper into the SQL query:

```
p = Post.find(:where => 'date > 2013-03-10',
              :limit => 10)
```

Simplicity

- Active Records in Ruby on Rails is able to get rid of lots of code
- Does so by assuming that certain conventions are followed (e.g. table names are plurals of model names)
- There has been some criticism that this couples the application logic very tightly to the database

ORM Summed Up

- Object-relational mapping is inherently difficult
- So don’t expect any miracles:
  - The available tools are not known for their elegance
  - In general, they will not solve 100% of all the issues, more likely 80% - 90%
  - The remainder has to be done semi-automatically by someone who knows the object and relational models
  - Nevertheless, the tools will save you from writing a lot of boilerplate code

Summary

- Building (large) applications involving database systems is not easy
- For maintainability and scalability it is usually a good idea to organize the systems in layers
  - Databases tend to have longer lifespans than the applications running on top of them
  - Someone may have to implement a new application using your data access layer
- Development of database applications is further complicated by data model incompatibilities