Conception of Information Systems
Lecture 4: Web Access to Databases

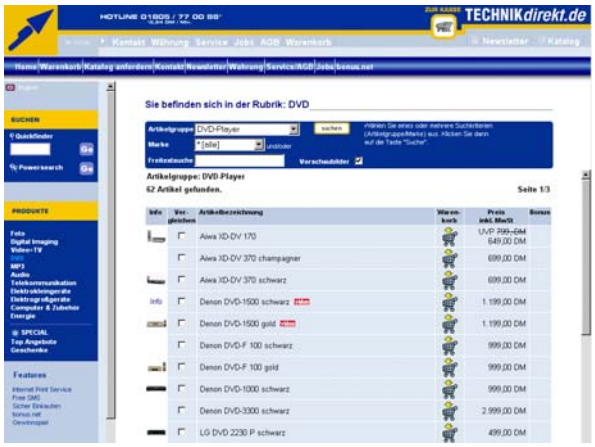
5 April 2005
http://lsirwww.epfl.ch/courses/cis/2005ss/

Outline

- Introduction
- JDBC
- Interfacing applications and the Web
  - Web Servers
  - Common Gateway Interface (CGI)
  - Applets
  - Servlets
  - Java Server Pages (JSP)
- Access to databases over the Web
  - CGI-based access
  - Applet-based access
  - Servlet/JSP-based access
- Summary
- References

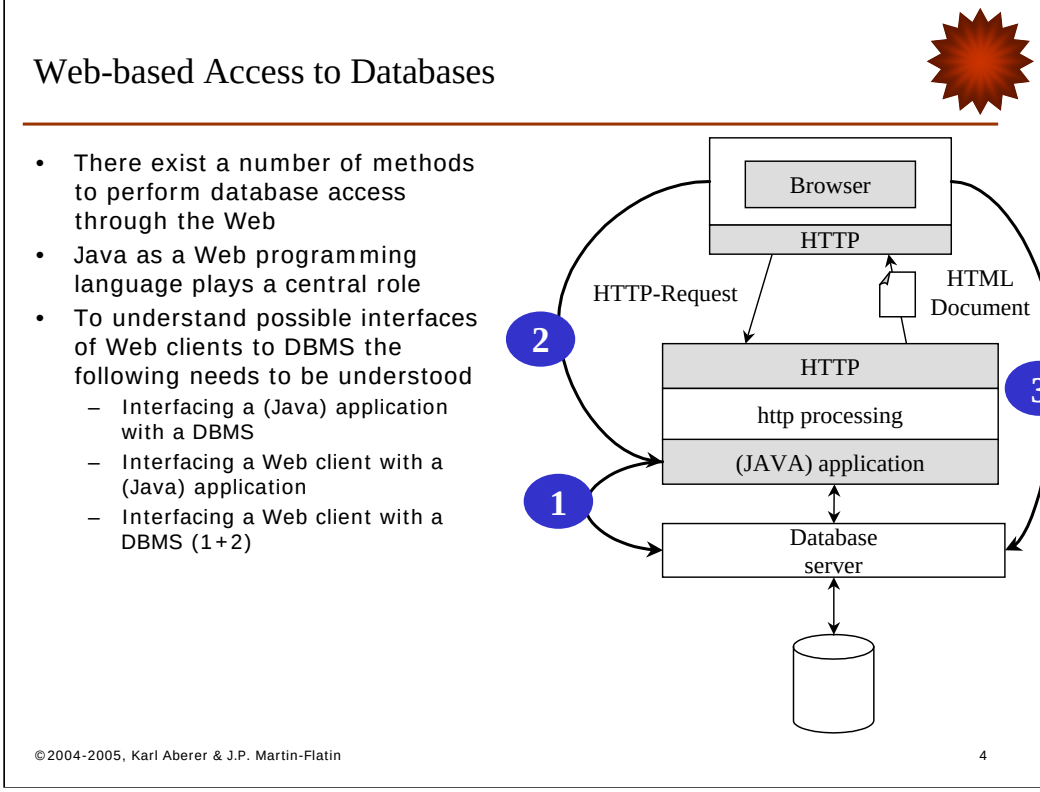
1. Introduction

- Most Web pages today are not static HTML (or XML) but are dynamically created, e.g. from databases
  - Electronic shops, information services etc.



Most Web pages that are available today over the Web are not statically stored HTML files, but dynamically generated, typically by using data from underlying databases. Typical examples for that are catalog data that is presented in an electronic shop or query results that are presented as part of an information service. In this part of the lecture we will study elementary methods of how contents of databases can be published over a Web page. Solutions to that problem were the earliest examples of dynamic Web applications, which are nowadays frequently implemented using modern application servers. Application servers will be the main topic of the subsequent parts of the lecture. However, since the original approaches to data publishing on the Web are still used in many applications and they can be considered as the predecessors of application servers, we will study them now in more detail.

Web-based Access to Databases



- There exist a number of methods to perform database access through the Web
- Java as a Web programming language plays a central role
- To understand possible interfaces of Web clients to DBMS the following needs to be understood
  - Interfacing a (Java) application with a DBMS
  - Interfacing a Web client with a (Java) application
  - Interfacing a Web client with a DBMS (1+2)

What does "web-enabling" of database servers exactly mean ? We know that a Web browser interacts with a Web server via the http protocol by issueing http requests and receiving html documents. In order to web-enable a database server therefore the Web server must be enabled to access the database server. This can be done by interfacing the Web server with the database server through an application, typically a JAVA application, that knows of how to access the API of a database server. Therefore the question of web-enabling a database server can be decomposed into the following three subproblems:

- How can an application access the database API (interfacing of applications with DBMS). This question is actually independent of the question of web-enabling and is of general interest. We will look at it from the perspective of accessing DBMS from the JAVA programming language.
- How can a Web client interact with an application, that is accessible at the Web server (not necessarily executed at the Web server!). This question is in fact independent of the problem of database access and again of more general interest.
- Which combinations of two solutions to the above questions result in feasible architectures and which are the trade-offs among the different alternatives.

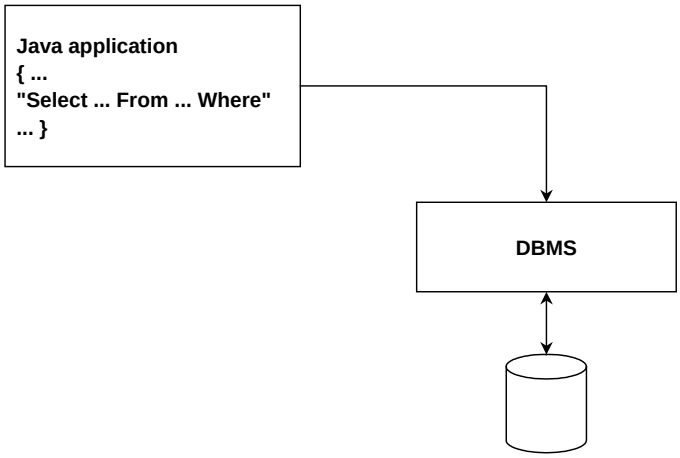
Note that one can observe in the architecture depicted on the slide that the Web-server constitutes an intermediate system layer between the web-browser (client) and the database server, and therefore represents a simple form of middleware. Question: which of the fundamental problems of information systems integration (distribution, heterogeneity, autonomy) are addressed by this middleware system ?

2. JDBC

- Database Gateways
- JDBC: the Big Picture
- Establishing a Connection
- SQL Instructions
- Result Processing
- Error Handling
- Metadata

Database Gateways

- Problem: accessing DBs from within a programming language (e.g. Java)



We turn our attention to the first problem, the access to a database through a programming language. Solutions to that problem are frequently called "database gateways". The problem is very simple: how can we embed an DDL/DML instruction, e.g. an SQL query, into a programming language, e.g. Java, such that the instruction is executed by the DBMS and the (possible) results can be further processed in the application.

Static vs. Dynamic Database Access

- Static (Embedded SQL, SQLJ for Java)
  - SQL statements are determined at compile time
  - Advantages
    - Simple programming
    - Syntactic and semantic errors are detected at compile time
  - Disadvantages
    - Query plans are generated at compile time and are possibly no longer valid at runtime
- Dynamic (Dynamic Embedded SQL, JDBC, ODBC)
  - SQL statements are dynamically constructed at runtime
  - Advantages
    - More flexibility: the application can generate and execute SQL statements at runtime
    - Query plans are up to date
  - Disadvantages
    - Overhead caused by the translation of SQL statements at runtime
    - Possible runtime errors

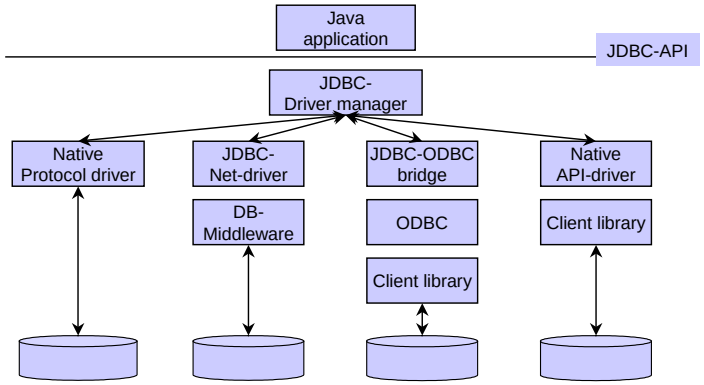
There exist two fundamentally different ways of building a database gateway: static and dynamic.

With *static access*, the database instructions are embedded into the application program code, using some special syntax, and are preprocessed by a pre-compiler. The pre-compiler replaces the statements it finds by calls to function libraries that implement the access to the DBMS. While doing so, it can perform certain checks (syntax, type checks) and recognize errors. Complex statements, such as SQL expressions, are also preprocessed (and therefore type-checked). During that, execution plans for executing the queries are constructed. Since the plans take into account the actual data distribution in the database and optimize them for execution accordingly, it can happen that at runtime the plans become out-dated when the distributions change. Static access is also less flexible as certain parameters, e.g. the names of tables in queries, cannot be dynamically assigned.

With *dynamic access*, the database instructions are also embedded into the application program code, using some special syntax, but are considered like data values (strings) and are interpreted at runtime. So no static type-checking takes place, which makes this type of embedding more error-prone. However, since the database statements are represented as data values, they can be manipulated at runtime and can thus be dynamically constructed, which is not possible with static access. Queries are interpreted at runtime, therefore the query plans are also constructed at runtime and up to date. The price to be paid for this increased flexibility is a larger runtime overhead.

Java DataBase Connectivity (JDBC)

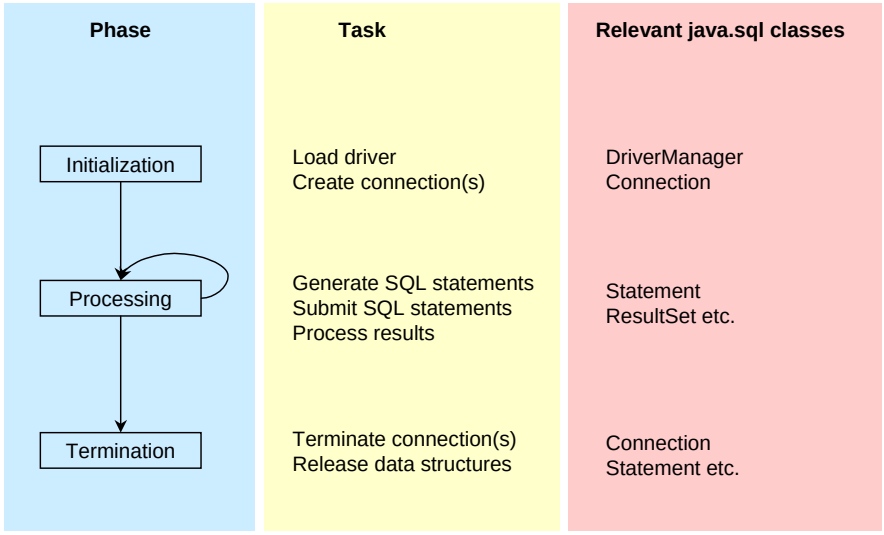
- **The** database programming API for the Java platform
  - Low-level programming API and class library
  - Dynamic integration of SQL into Java
- Implementations of the JDBC API are called JDBC drivers



Due to their flexibility approaches to dynamic access have become more popular in practice. In the following we have a look at the standard way of accessing relational DBMS from Java, which is the JDBC API.

JDBC is the standard interface from JAVA to relational DBMS, providing a programming API, a class library and different standard architectures that are proposed for implementation. It has been derived from Microsoft's ODBC (Open Database Connectivity) standard. There exist 4 standard implementations of the JDBC API, which are called JDBC drivers. We give here just a short overview of those, more details can be found in the appendix 1. Native protocol drivers implement the JDBC API by directly accessing the functions within a specific database kernel implementation and are thus dependent on the specific implementation of a DBMS. In contrast, Native API drivers depend also on a specific DBMS implementation, but instead of directly accessing the DBMS software they access it through the standard DBMS client API that is provided by the DBMS. The two other driver types make use of a middle layer. JDBC-ODBC bridges reuse existing ODBC implementations and simply translate the JDBC API to the ODBC API, which is straightforward since JDBC has been derived from ODBC. JDBC-Net drivers are implemented as part of an application server, that provides the necessary functionality in order to access a DBMS.

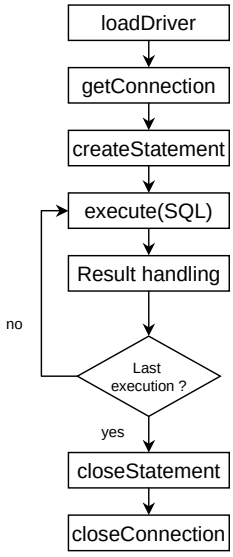
Using JDBC in an Application



The basic pattern for running a JDBC application is the following:

- During the initialization phase, the necessary initializations are performed; they are required in order to access the DBMS. The JDBC drivers are loaded and connections to the DBMS are established.
- Then the actual processing of DDL/DML statements to be executed in the DBMS can start. In this phase, the statements are generated and submitted to the DBMS and results are collected and provided within Java data structures for further processing in the application.
- Once the processing has completed, the application needs to be closed down by terminating the connections to the database(s) and release all data structures that have been instantiated.

Example: JDBC/ODBC Bridge




```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Connection conn =
   DriverManager.getConnection("jdbc:odbc:mydb",
        "username","passwd");

Statement stmt = conn.Statement (
   "SELECT author FROM book WHERE title = xxx");

ResultSet rset = stmt.executeQuery();
while (rset.next()) {
   ...
   }


stmt.close();

conn.close();
```


This example shows the process described before in more detail, together with the corresponding Java statements.

loadDriver: in the first step a JDBC driver is loaded. In that case it is a JDBC-ODBC bridge.

getConnection: once the driver is loaded it provides a DriverManager object that can open connections to a database. This is done here, by supplying also the necessary authentication information.

createStatement: for a specific connection a database statement can be created, in this example an SQL query. One can see that the query is passed as a string parameter, and therefore it could have been arbitrarily manipulated and constructed beforehand.

execute(SQL): the statement is executed next, by invoking the executeQuery method on it. Since it is a query it produces a result which is returned to a specific data structure ResultSet. This data structure provides the necessary methods in order to access it from Java applications. Since query results are typically sets, it provides in particular a possibility to access a set, by means of the iterator concept. An iterator can be considered as a pointer that points to one specific element of the set and that can be moved by invoking navigation methods on the result set. In the example next() is a method that moves the pointer to the next result element.

Result handling: Within the loop the application can access the result elements and do whatever is needed, e.g. display it or perform some computation on it.

Last execution ? : the iterator returns upon the invocation of a navigation method a value that indicates whether the last element has been reached (or some other condition is met), which enables the application to terminate the processing of the result set.

closeStatment: once the processing of a statement is completed the statement is closed in order to release the corresponding data structures (statement, result set)

closeConncetion: once the application has completed processing the connection to the database is released in order to release the data structures.

More details on the JDBC interface are given in the following.

Establishing a Connection

- Loading of driver
  - Creates an instance of the driver
  - Registers driver in the driver manager
  - Explicit loading: `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
  - Several drivers can be loaded and registered
- Connecting to database
  - A connection is a session with one database
  - Databases are addressed using a URL of the form
    " jdbc:<subprotocol>:<subname>"
  - Examples
    - JDBC-ODBC bridge (Type 1): `jdbc:odbc:mydb`
    - JDBC-Native (Type 4): `jdbc:oracle:thin:@dombach:1526:poem`
  - Connection is closed: `conn.close();`

JDBC - SQL Instructions

- SQL instructions are encapsulated in `Statement` objects
- Three types of statements
  - `java.sql.Statement`
    simple SQL instruction without parameter
  - `java.sql.PreparedStatement`
    pre-compiled SQL instruction with parameters
  - `java.sql.CallableStatement`
    calling stored procedures
- `PreparedStatement`: SQL statement is sent to DBMS and compiled at creation time
  - No additional compilation overhead when executed
  - Can be executed multiple times with different parameters
- Example

```
PreparedStatement stmt = con.prepareStatement (
   "SELECT price FROM book WHERE ordernr = ?");
stmt.setInt (1, ordernr);
ResultSet rset = stmt.executeQuery();
```

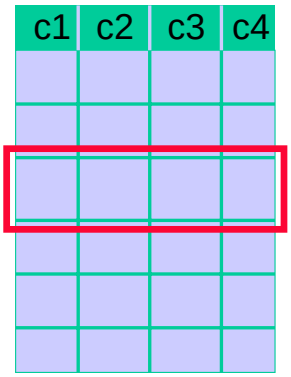
- Closing the statement object: `stmt.close();`

Executing SQL Queries

- Three kinds of execution instructions
  - SELECT statements ⇒ `executeQuery`
    ```
    ResultSet executeQuery(String sql) throws SQLException;
    ```
  - DDL/DML statements ⇒ `executeUpdate`
    ```
    int executeUpdate(String sql) throws SQLException;
    ```
  - Unknown type ⇒ `execute`
    ```
    boolean execute(String sql) throws SQLException;
    ```
- Return value of `executeUpdate`
  - DDL statement: always 0
  - DML statement: number of tuples
- Return value of execute
  - true, if result of a SELECT statement
  - false, if result of a DML/DDL statement
- Execution parameters
  - Timeout, maximal number of result tuples and field size

```
// java.sql.Statement
int getMaxFieldSize() throws SQLException;
void setMaxFieldSize (int max) throws SQLException;
int getMaxRows() throws SQLException;
void setMaxRows (intmax) throws SQLException;
int getQueryTimeout () throws SQLException;
void setQueryTimeout (int secs) throws SQLException;
```

Processing the Result (1/2)



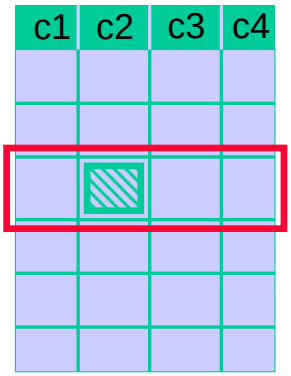
- Getting the result

```
// java.sql.Statement
ResultSet getResultSet() throws SQLException;
```

- Access to tuples
  - The ResultSet object manages a cursor for tuple access
  - Example

```
Statement stmt=con.createStatement();
ResultSet rset=stmt.executeQuery(
                                    "SELECT …");
   while (rset.next()) {
     …
   }
rset.close();
```

Processing the Result (2/2)



- Access to columns of a tuple
  - Using column index or column name
  - `findColumn`: finds the column index for a column name
  - Example

```
while (rset.next()) {
   int cust=rset.getInt(2);
   String name=rset.getString("name");
   String city=rset.getString(
                rset.findColumn("city"));
     ...
}
```

Error Handling

- Each SQL statement can generate errors
  - Thus each SQL method should be put into a try-catch block
- Exceptions are reported as instances of class `SQLException`
- Example

```
import java.sql.*;
public class JdbcDemo1 {
  public static void main(String[] args) {
    String driverClass = "sun.jdbc.odbc.JdbcOdbcDriver");
    try {
          Class.forName(driverClass);  // Load JDBC driver
    } catch (ClassNotFoundException e) {
          System.out.println(e.getMessage());
    }
    try {
          Connection conn = DriverManager.getConnection("jdbc:odbc:mydb","tux","penguin");
          Statement stmt = conn.createStatement();
          ResultSet rs = stmt.executeQuery("SELECT title,price,stock FROM book");
          while (rs.next()) {
             String s = rs.getString(1); String d = rs.getDouble(2); int i = rs.getInt(3);
             System.out.println(s + " " + d + " CHF, " + i);
          }
    } catch (SQLException e) {
          System.out.println("SQLException: " + e.getMessage()); } } }
```

JDBC Metadata

- Metadata allows to develop generic (e.g. schema independent) applications for databases
  - Generic output methods
  - Type dependent applications
- Two types of metadata are accessible
  - `java.sql.ResultSetMetaData`
    describes the stucture of a result set object
  - `java.sql.DatabaseMetaData`
    provides information about the database (schema etc.)
- Information about a `ResultSet` object
  - Names, types and access properties of columns
- Information about the database
  - Name of database
  - Version of database
  - List of all tables
  - List of supported SQL types
  - Support of transactions

Example of JDBC Metadata

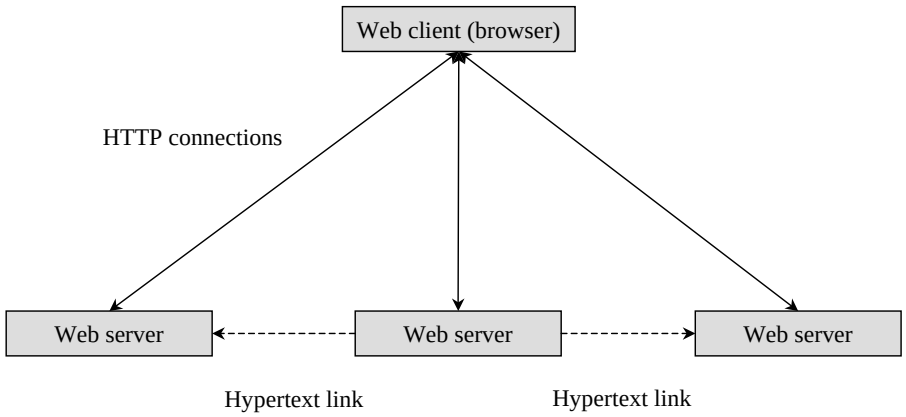
```
ResultSet rset =
     stmt.executeQuery("SELECT * FROM customer");
ResultSetMetaData rsmeta = rset.getMetaData();
int numCols = rsmeta.getColumnCount();
for (int i=1; i<=numCols; i++) {
     int ct     = rsmeta.getColumnType(i);
     String cn  = rsmeta.getColumnName(i);
     String ctn = rsmeta.getColumnTypeName(i);
     System.out.println("Column #" + i + ": " + cn +
       " of type " + ctn + " (JDBC type: " + ct + ")");
   }
```

3. Interfacing Applications and the Web

- Web servers
- CGI
- Applets
- Servlets
- JSP

Now we turn the attention to question 2 related to the problem of Web access to databases, the interfacing of applications with Web servers.

3.1 Web Servers



We recall the basic architecture of the WWW:

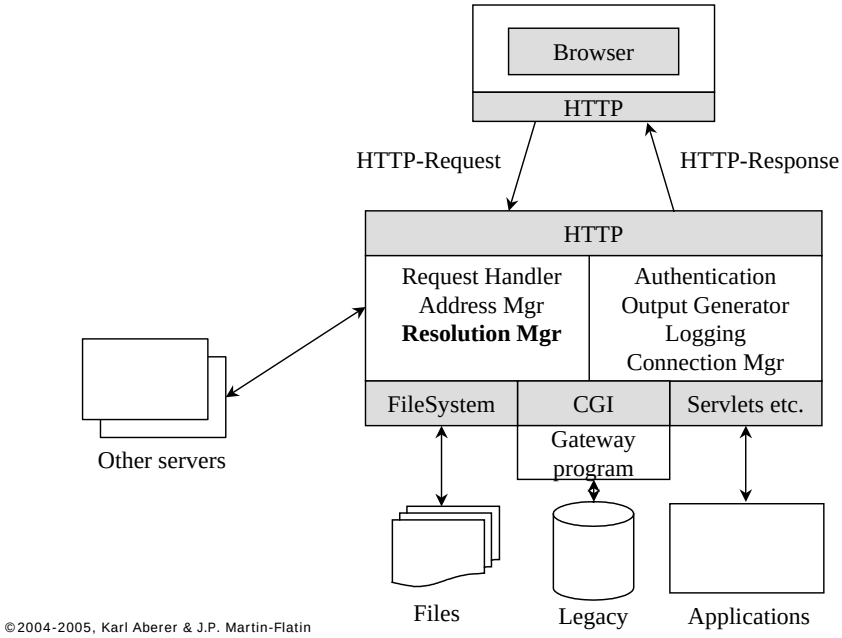
The WWW is a synchronous, distributed, multiple-client, multiple-server hypermedia system. It relies on two types of software:

1. Web browsers that implement the client functionality.

2. Web servers that manage access to the hypermedia documents.

By means of hyperlinks that are embedded in HTML documents users can navigate in between different web servers.

Web Server Architecture



A Web server works like a simple database management system: it receives requests (in HTTP) for which it produces result data in the form of HTML documents. For performing this processing it has a number of components that take care of different tasks:

-request handler: it creates the process to handle the request and provides the necessary resources

-Address manager: it handles redirections to other servers and supports rewrites of the addresses requested

-Resolution manager: it determines the type of the request and the type of processing required. This is the most interesting part for us, since it allows the web server not only to access HTML documents from the file system, but also to invoke instead programs that dynamically generate HTML code. We will describe the different possibilities for doing that in more detail subsequently.

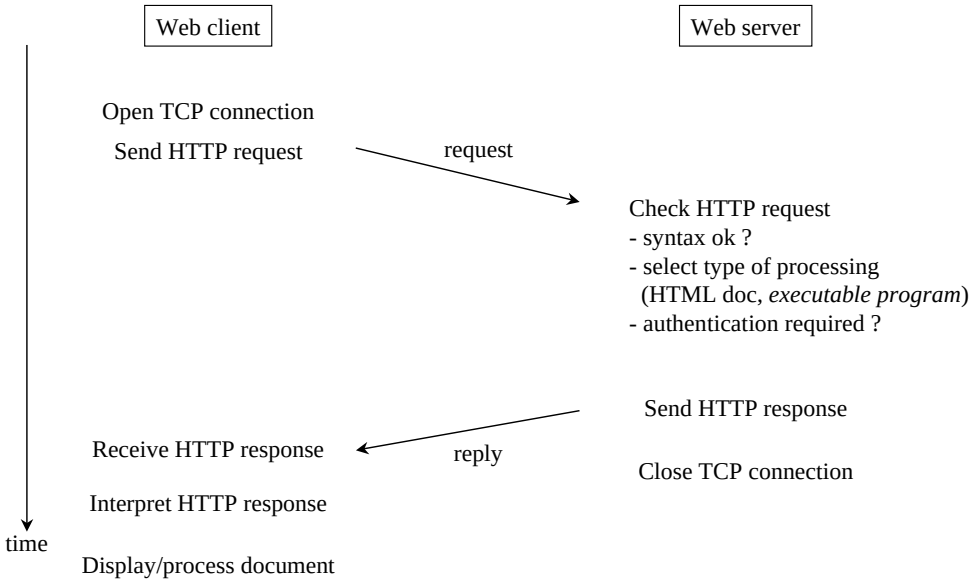
-Authentication: it request from the user authentication information if required

-Output generator: it composes the response message

-Logging: logs all the accesses to the Web server

-Connection manager: terminates connections, e.g. also by means of time-out

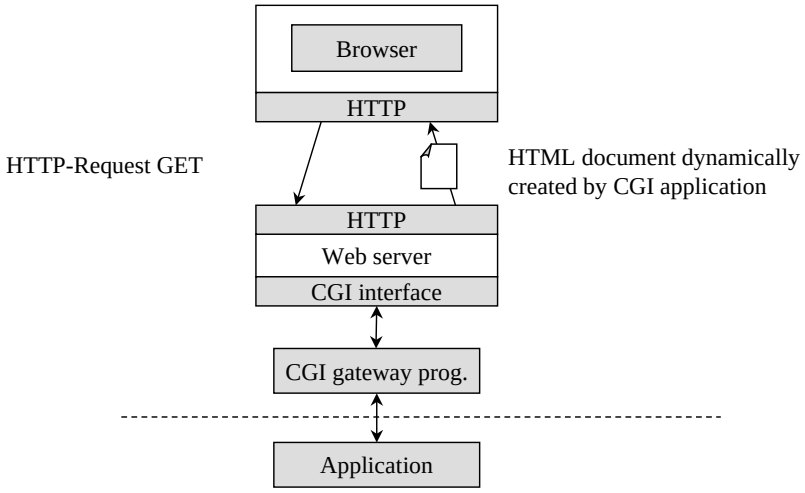
Reminder: HTTP Protocol



This figure illustrates the HTTP protocol that is executed between the Web client and Web server. We see that on the client side the essential activity takes place once the response arrived, and the Web browser has to interpret and display the reply message. On the server side the essential step is the composition of response message, either by retrieving it from the files system or by composing it on the fly by invoking an application. More details on the http protocol and its processing are given in the appendix.

3.2 Good Old Days: Common Gateway Interface (CGI)

- Standard for interfacing external applications with Web servers
- CGI programs execute on the Web server and produce dynamic output
- Language independent (Perl, C, Java, TCL, ...)



The problem of interfacing applications with web servers was recognized very early and the first standard approach provided by web servers to that extend was the Common Gateway Interface (CGI). It is completely language-independent since it is based on the execution of an arbitrary executable program that is found in the file system. This program, which is called the CGI gateway program, then can be used in order to interface with any application.

CGI Example

MyScript.html

```
<FORM METHOD="GET" ACTION="/cgi-bin/MyScript.pl">
   Parameter 1: <INPUT TYPE="TEXT" NAME="par1" SIZE="10"><P>
   Parameter 2: <INPUT TYPE="TEXT" NAME="par2" SIZE="10">
   <INPUT TYPE="SUBMIT" VALUE="Send">
</FORM>
```

MyScript.pl

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
While (($key, $val) = each %ENV)
   {print "$key = $val<BAR>\n";}
```

Result document

```
GATEWAY_INTERFACE = CGI/1.1 <BAR>
SERVER_PROTOCOL = HTTP/1.0 <BAR>
REQUEST_METHOD = GET <BAR>
QUERY_STRING = par1=hello&par2=world <BAR>
...
```


This simple example illustrates of how the CGI Gateway is used. A CGI Gateway application consists of two parts:

- An HTML document from which the gateway program can be invoked
- The gateway program itself

The invocation of a gateway program from an HTML document is performed by using an active HTML element, such as FORM. When the user submits the form an action is invoked (for processing the form input, this seems to be the only reasonable thing to do with a form). The action that is executed is a program that is deposited in a special cgi-bin directory. It is started by the web server as soon as the Web client sends the GET message upon issuing the action by the user. A special mechanism is used in order to pass also parameters from the HTML form to the program. The web server stores the parameters in an environment variable QUERY_STRING after receiving them as part of the GET message. In this example the value would be $QUERY_STRING="par1=hello&par2=world" and the program invoked by the web server would be a perl script, MyScript.pl

The perl script in turn generates a result document dynamically. In this example it reads the contents of all environment variables, separated by a <BAR> element. The resulting (HTML) document is also shown.

This example also demonstrates one of the weaknesses of the CGI approach: the security is very weak ! Reading the contents of the OS environment, containing probably confidential information, is not necessarily desirable, and avoiding such blunders is completely under the responsibility of the programmer.

CGI Properties

- Availability: Platform and vendor independent
- Coupling: HTTP is stateless
  - no sessions for multiple interactions with a CGI program
  - Cookies: limited in size
  - Hidden (input) fields
- API: Awkward parameter handling
- Performance: Each call starts a process (inefficient)
- Security: CGI script are potential security holes
  - Leak information about the server system
  - Program code may be introduced by user input

The major advantages of the CGI approach are its simplicity, its generality (no limitations on the programming languages), and its general availability as part of the web server infrastructure.

The CGI approach has however four major weaknesses:

The most important is related to the fact that the HTTP protocol is stateless, and thus multiple interactions that take place between a web server and client, as for example required in an electronic commerce application, cannot be supported directly. There exist various work-arounds for this problem, the best known being the use of Cookies. Cookies are files that a web server can deposit in a special directory on the file system of the Web client. This approach is insofar problematic as it intrudes into the security of the client (the client is not in charge of controlling what is deposited on its machine) and cookies are also rather limited in size. Another workaround are hidden input fields, such that the complete state information is passed back and forth as part of the generated HTML documents and the GET messages. This approach is inefficient as it leads to unnecessary transfers of state information and is also awkward to implement.

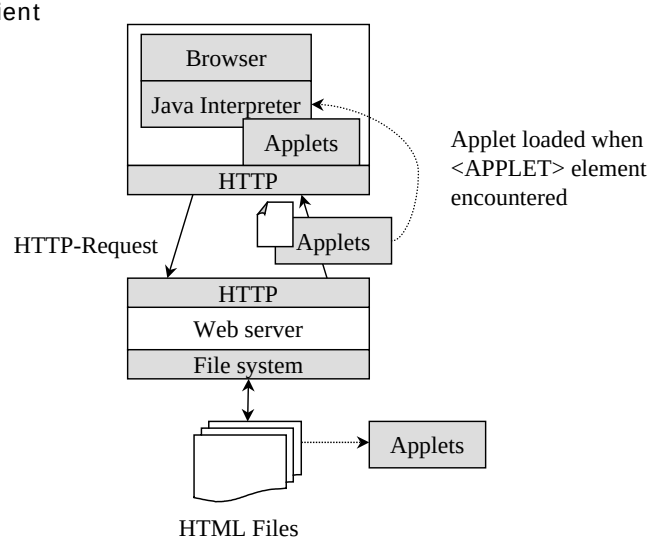
A second weakness is the awkward mechanism that is used in order to pass parameters. In addition to being awkward this mechanism also offers no type safety for the parameters, which thus requires extensive type checking on the server side.

A third problem is that each request from a client to a server requires the invocation of a separate process, instead of using the same process for multiple interactions that are related. Starting a process is a fairly expensive operation and maintaining a large number of processes at the same time in the OS can quickly exhaust the available resources (e.g. memory).

And finally, as already mentioned CGI also introduces serious security problems, one example being the application we have discussed before. Another potential problem is the introduction of program code through user input into the programs executed as CGI applications.

3.3 Applets

- Standard for loading and running client-side Java applications
- Applets execute on the client



Applets are Java programs that can be dynamically loaded into a Web browser and executed on the client. The applet mechanism has been originally introduced in order to allow the execution of programs at the client side in order to support complex applications that implement interactions with users. Thus the programming is running on the client, and not on the server as CGI applications, which has important consequences for database access. The applet (a JAVA application) is stored in the file system. When the browser encounters an <APPLET> tag in an HTML document, it loads the applet from the server to the client. In order to run the applet the browser needs to include a JAVA interpreter. When the browser exits a Web page it also discards the applet.

Applet Example

MyApplet.java

```
import java.awt.*;
import java.applet.*;                        // packages required
public class MyApplet extends Applet         // generic applet class
{
   public void init()                        // executed upon loading
   {     message = getParameter("message");}
   public void paint(Graphics g)
   {     g.set Color(new Color(255,0,0));
         g.fillRect(0,0,200,20);
         g.setColor(new Color(255, 255, 255));
         g.drawString(message, 10, 15);;}
}
```

MyApplet.html

```
<HTML><HEAD></HEAD>
<BODY>
   <APPLET CODE=MyApplet.class WIDTH=350 HEIGHT=30>
         <PARA; NAME="message" VALUE="Message from MyApplet">
   </APPLET>
</BODY></HTML>
```


This is an example applet application. It consists of a Java class, representing the applet code and an HTML page for invoking the applet. The applet class MyApplet is derived from a generic applet class. It has a standard method init(), inherited from the applet class, that can for example be used to obtain parameters. In this example the applet implements a method paint() that produces some graphics at the user interface. The invocation of the applet is shown in the document MyApplet.html. The applet is invoked when the html file is loaded at the client and the web browser detects the <APPLET> element. It obtains from the web server the class (MayApplet.class) and loads into its JVM. While doing that the applet is assigned a window at the user interface of which the size is specified. Parameters that are passed to applet methods are declared within hte <APPLET> element using a <PARA> element. The init() method of the MyApplet class shows of how the parameters may be accessed.

Applet Properties

- Availability: Platform and vendor independent
- Coupling: Applets can establish a session outside the HTTP protocol
  - using RMI
- API: Accessing server resources requires RMI
- Performance: Exploits client resources
- Security: Sandbox concept
  - Applications can only access resources within "sandbox"
  - Code verification
  - Signed applets can access some local files
  - Improved since JDK 1.4

Applets are distributed as Bytecode. Bytecode is the closest to processor instructions preserving portability among platforms. Before the applet is run it is checked by a Bytecode verifier for security reasons, then it is executed by a run-time interpreter.

In order to establish a coupling that supports sessions consisting of multiple interactions between the client and server a communication mechanism that lies outside HTTP needs to be used. Java RMI (remote method invocation) is the standard approach to achieve that. With RMI also resources at the server (like databases) can be accessed that would otherwise not be accessible from within the applet.

One of the main concerns in the design of the applet mechanism is to relieve the server from extensive computations that are related to the user interface support and delegate those to the client. This can help to improve server performance.

A main concern of the applet mechanisms is security. Applets are executed in a "sandbox". Concretely this means that

-Java does not support pointers and memory locations are resolved by the Java interpreter (no addresses outside the allowed memory are accessible)

-The verifier recognizes tampered code

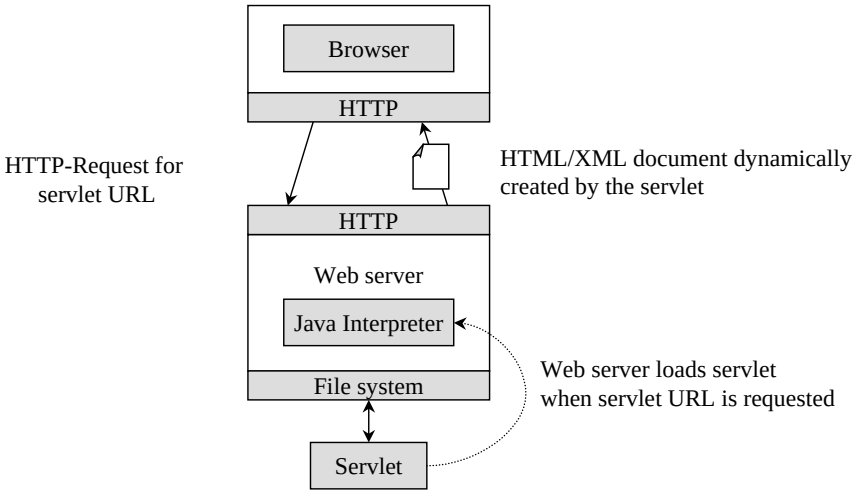
-Classes can only access objects from within its namespace

-File access is controlled by access control lists with restrictive defaults

-Network access can be restricted to various levels: none, originated host, outside firewall

3.4 Servlets

- Java objects that run on Java-enabled Web servers and generate HTML/XML dynamically

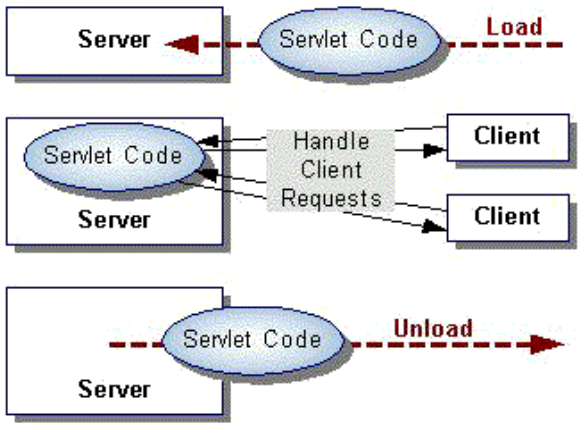


With servlets in a certain sense the roles of the HTML document and the JAVA application are reversed as compared to applets. It is not the HTML document that leads to the loading of a JAVA application to the client, but it is the JAVA application that generates the HTML document that is loaded to the client. Consequently a servlet is an application that runs at the Web server. Therefore web servers supporting servlets must be servlet-enabled and contain a JVM, which is called also the servlet engine.

The basic servlet mechanism consists of loading a servlet into the servlet engine as soon as a URL pointing to a servlet is requested by the client (via an HTML link, for example). Then the servlet starts to dynamically create HTML documents, which can enable further accesses to the servlet.

Servlet Engine

- Execution environment embedded in Web server (e.g., Tomcat)
- Servlets called through URLs
- Java Servlet API specification and package



The servlet engine is the execution environment of the JAVA application that constitute the servlet. In contrast to CGI applications a servlet can maintain a state throughout multiple interactions with the client. To that end a servlets support methods in order to handle multiple requests by clients and to produce the corresponding responses. Only after the tasks of a servlet are completed, it is unloaded from the servlet engine.

Servlet Software Architecture

- Two packages: **`javax.servlet`** and **`javax.servlet.http`**
- **`Servlet`** interface
  - **`init()`** and **`destroy()`** manage resources that are held for the lifetime of the servlet
  - **`getServletInfo()`** is used by the servlet to provide information about itself
- **`ServletRequest`** and **`ServletResponse`** interfaces
  - To handle client requests and responses
- **`HttpServlet`** class
  - Inherits from **`GenericServlet`** class
  - **`doGet()`**, **`doPost()`**, **`doPut()`**, **`doDelete()`** to handle HTTP requests
  - parent class for servlets actually used in program
- **`More info:`**
  - http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html

In order to develop a servlet the corresponding JAVA classes need to support specific methods in order to handle the interactions with the clients as well as the lifecycle of the servlet itself. The corresponding software architecture consists of a set of classes, specializing from a generic servlet class to specific servlet classes supporting different protocols. Currently the HTTP protocol is the only one of interest.

The root of this class hierarchy is the Servlets class. It provides methods for managing the lifecycle of the servlet (creation and destruction) and methods to access information about the servlet.

The Generic Servlets class provides the basic methods (interfaces) for handling clients requests and creating responses. For that purpose it provides the ServletRequest and ServletResponse interfaces.

The httpServlet class specializes these methods to methods that are required to specifically handle the requests and responses that are required for supporting the HTTP protocol.

Servlet Example

```
public class SimpleServlet extends HttpServlet
{
// Handle the HTTP GET method by building a simple web page
public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
throws ServletException, IOException
{
        PrintWriter out;
        String title = "Simple Servlet Output";
           // set content type and other response header fields first
        response.setContentType("text/html");
           // then write the data of the response
        out = response.getWriter();
        out.println("<HTML><HEAD><TITLE>");
        out.println(title);
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>This is output from SimpleServlet.");
        out.println("</BODY></HTML>");
        out.close(); } }
```


This simple example illustrates the use of servlets: the user-defined servlet SimpleServlet extends the base HttpServlet class by defining a method that implements the doGet method of HttpServlet. This method is executed upon a HTTP GET request by the client. We see that the response is created by using a response object and writing to it the HTML code in string format. This creates the HTML page that needs to be generated as part of the response. The required processing for creating the HTTP response message and sending it to the client is handled by the built-in Servlet classes.

Servlet Properties

- Availability: requires servlet-enabled Web servers
  - e.g., Apache -> Tomcat
  - Portable code
- Coupling: Supports sessions
  - Not a separate process
  - Stays in memory between requests
- API: Awkward generation of HTML code
- Performance: Better than CGI
  - Single servlet answers multiple requests
  - Uses server resources (as opposed to applets that use client resources)
- Security: Servlet runs in sandbox
  - only HTML code is transferred to the client

Servlets are more restricted in their applicability since they require a Servlet-enabled Web server (which is no more a substantial restriction today).

They offer compared to CGI the big advantage that they support sessions with the client. This has not only the advantage that the state of the sessions is kept in memory between requests (and needs not to be maintained by the workarounds as with CGI), but also that the servlet runs as a single process, which gives a substantial performance gain.

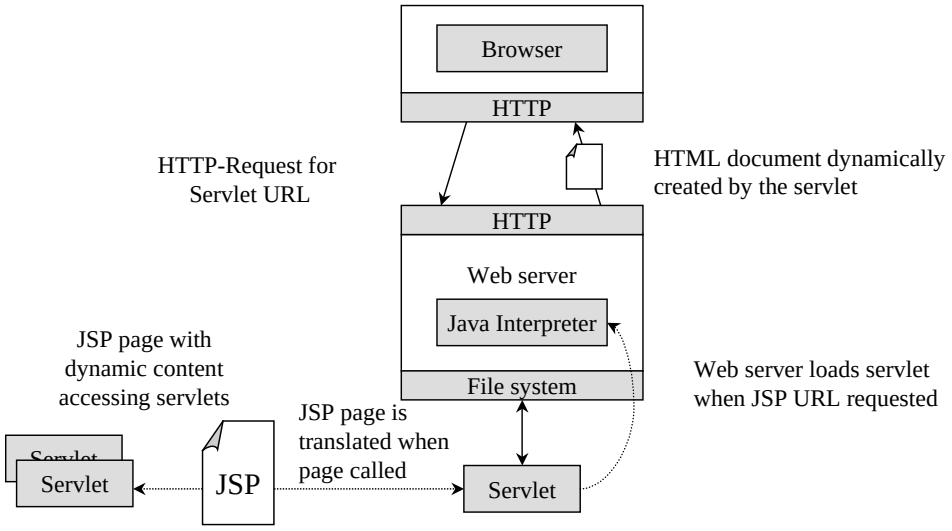
The generation of HTML code through imperative programming is awkward and maybe the biggest disadvantage of the servlet mechanism.

Servlets offer also further performance advantages as compared to CGI, since the same servlets can be used to handle requests from different clients (sharing). On the other hand, since servlets are running on the server, they can create there more likely performance bottlenecks as if the application would be running on the client as an applet.

From the security perspective servlets are also running in a sandbox (which is however not so important as for applets, since the servlet belongs to the server anyway). More important is the fact that no program code is transferred to the client, and thus servlets are more secure from the client perspective.

3.5 Java Server Pages

- A JSP page is an HTML/XML document with embedded instructions to dynamically create its content



Java server pages (JSP) address one of the main weaknesses of servlets without sacrificing their advantages: they substantially simplify the development process of dynamically generated web pages, by using for the specification of a dynamically generated web pages a document model. They extend the servlet technology by turning the servlets "inside-out": rather than using JAVA code in order to generate an HTML (or XML) page, they allow to directly specify an XML page and to embed the dynamically generated parts within the HTML code. Within the dynamic parts of a JSP page it can access other servlets. Therefore JSP pages are easier to author than servlets and development tools are easier to provide for the JSP authoring process.

The basic working of JSP pages is as follows:

- Translation phase: when a JSP page is accessed (by accessing its URL) it is translated by the "JSP container" into a servlet, that provides exactly the functionality as specified by the JSP page. The translation can be done both in advance, after each change of the page (precompiled JSP pages) or at invocation time.
- Request phase: once the JSP page is translated into a servlet it is loaded into the Servlet engine of the web server and starts to process requests from the client.

JSP Architecture

- JSP container
  - Handles lifecycle of JSP page
- JSP scripting elements
  - Java declarations, expressions and scriptlets (= Java code)
  - File inclusion
  - Call other components
- TAG libraries
  - Encapsulate functionality
  - Reusability
  - http://java.sun.com/products/jsp/taglibraries/
- Support for XML and HTML syntaxes
- Properties
  - like servlets except improved programming support

The main elements of the JSP architecture are

-The JSP container that handles the lifecycle including translation of JSP pages and request handling (thus including the servlet engine)

-The JSP scripting elements that are used to include the dynamic programming elements into HTML code and that are resolved by the JSP container into servlet code through translation. These scripting elements can consists of Java declarations, expressions and scriptlets (=Java code) for computation and direct output, of file inclusion in order to include other JSP pages, and of calls to other components, including Beans, Servlets, EJBs (Enterprise Java Beans).

-TAG libraries that allow to define JSP related tags in order to encapsulate functionality for reuse

-Support for both XML and HTML syntax.

Properties of JSP pages are comparable to those of Servlets with one exception: the generation of Web pages is substantially simplified since the developer can use a document model in order to specify the desired output.

JSP Example

```
<HTML>
<HEAD> <TITLE>Using JavaServer Pages</TITLE> </HEAD>
<BODY>
<P> Some dynamic content created using various JSP mechanisms:
<UL>
<LI><B>Expression.</B><BR>
   Your hostname: <%= request.getRemoteHost() %>.
<LI><B>Scriptlet.</B><BR>
   <% out.println("Attached GET data: " + request.getQueryString()); %>
<LI><B>Declaration (plus expression).</B><BR>
   <%! private int accessCount = 0; %> Accesses to page since server
   reboot: <%= ++accessCount %>
<LI><B>Directive (plus expression).</B><BR>
   <%@ page import = "java.util.*" %> Current date: <%= new Date() %>
</UL>
</BODY>
```


This example demonstrates some of the capabilities of the JSP mechanism. First we observe that the JSP page looks like an ordinary HTML page, with the exception that it contains some special JSP tags. The JSP tags are identified by the % symbol. We see four kinds of these tags:

- The first is for the evaluation of a JAVA expression and the returning of its result as text on the page enclosed in the tag <%= %> (expression evlauation). We see that in the expression the same JAVA classes are used to access the request object as we have seen for servlets.
- JSP tags can also contain JAVA code fragments that generate text on the output, exactly as with the servelt mechanism. This is illustrated in the second example of a scriptlet. Scriptlets are enclosed in the tag <% %>.
- The third example shows of how JAVA variables can be declared and used. The declaration is enclosed in <%! %> (declarations) and is used later for the evaluation of an expression. We also see that variables of non-text type are converted automatically to text when included into the HTML code.
- The fourth example shows the inclusion of another JSP page enclosed in the tag <%@ %> (directives)

Servlet Corresponding to the JSP

```
import = "java.util.*";
private int accessCount = 0;

out.println("<HTML>");
out.println("<HEAD> <TITLE>Using JavaServer Pages</TITLE> </HEAD> ");
out.println("<BODY> ");
out.println("<P> Some dynamic content created using various JSP mechanisms: ");
out.println("<UL> ");
out.println("<LI><B>Expression.</B><BR> ");
out.println("Your hostname:" + request.getRemoteHost() + ".");
out.println("<LI><B>Scriptlet.</B><BR> ");
out.println("Attached GET data: " + request.getQueryString());
out.println("<LI><B>Declaration (plus expression).</B><BR> ");
out.println("Accesses to page since server reboot: + ++accessCount %>);
out.println("<LI><B>Directive (plus expression).</B><BR> ");
out.println("Current date: +  new Date());
out.println("</UL> ");
out.println("</BODY>");
```


This is a servlet code fragment that corresponds to the JDP page given in the previous example.

4. Access to Databases on the Web

- CGI-based database access
- Applet-based database access
- Servlet/JSP-based database access

In the last section of this part of the lecture we combine the database access method (via JDBC) with the different methods to interoperate an application with a web server and discuss the resulting architectures for accessing a DBMS from a Web client.

Any of the application integration approaches that we have seen can be used

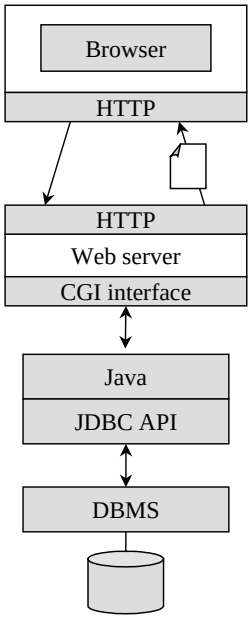
-CGI access to an application accessing database, where we will be using JAVA as the programming language and using JDBC at the server side

-Applet based access using JAVA RMI to establish a communication from the client to the DBMS and using JDBC at the client side

-Servlet/JSP based access using JDBC at the server side

In addition there exist database vendor specific solutions , such as Oracle Web Application Server or Informix Web DataBlades.

4.1 CGI based Database Access



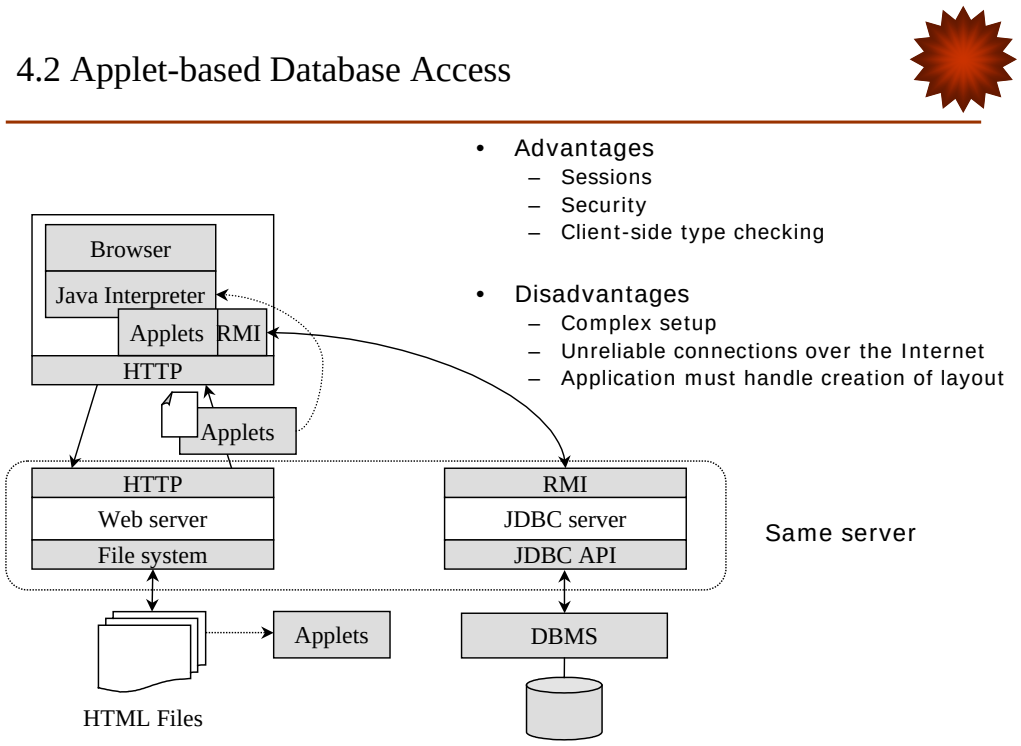
- Advantages
  - Minimal browser technology
  - Minimal server technology
- Disadvantages
  - Separate process for each access (expensive database login)
  - Stateless interaction
  - No checking of input at client

CGI based access works by invoking a Java application, that supports the database access and generates the necessary HTML pages for the user interfaces, through the CGI interface. The main advantage are the minimal technological requirements both on the server side and the client side. Neither must the server support a servlet/JSP container nor must the client's browser support JAVA applets. As time proceeds, these advantages however diminish, since JAVA support becomes practically universally available.

The main problems of this approach are performance and stateless interaction. The performance problem is aggravated by the fact that not only a new OS process is started for each interaction, but also a new database login is required, which is particularly costly.

Finally, since the input is provided by means of an HTML form, no client-side type checking of the inputs can be performed. All input is passed as strings and needs to be checked within the JAVA application that is providing the gateway to the DBMS.

4.2 Applet-based Database Access



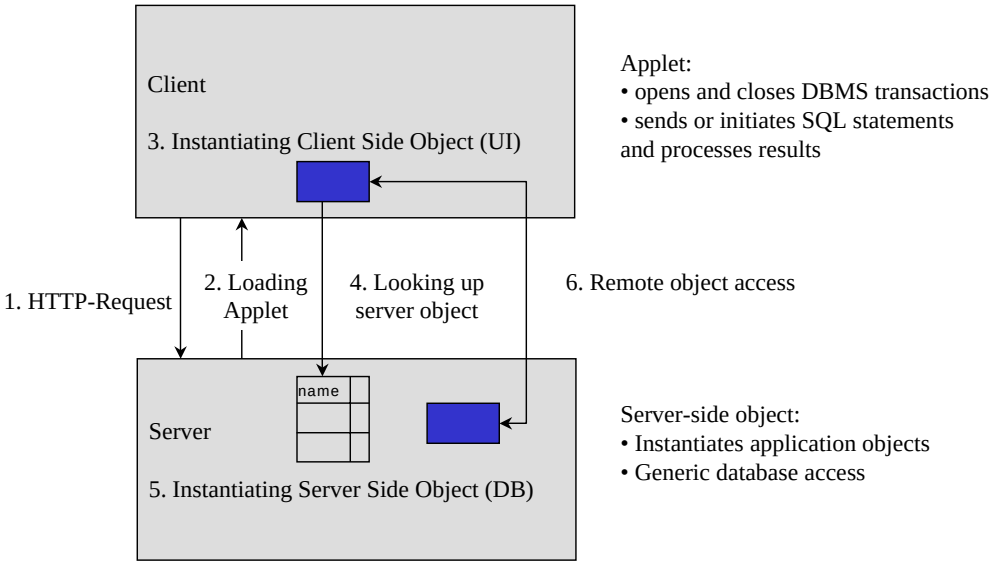
Applet-based access works by loading an applet to the client side that provides the necessary functionality for database access. The database access are, once the applet is loaded, performed by a synchronous communication mechanism outside the HTTP protocol, using RMI. To that end a Server application, the JDBC server is communicating with the client-side applet. More details on this interaction will be given on the next slide.

The main advantages of this approach are the possibility to establish a session and the increased security. As long as the applet is loaded it can perform multiple interactions with the DBMS server and maintain a session state. Security is enhanced as compared to CGI, which has a number of problems as discussed earlier.

On the downside the architectural setup is fairly complex, and the development of this type of database access, requires the use of the RMI mechanism. Using a synchronous communication mechanism over unreliable Internet connections is problematic. It may leave, for example, a database transaction in an undefined state Therefore the developer has to take measures for handling disconnections and interruptions of sessions, which further complicates the development. Also the user interfaces are generated as graphical user layout, which is again more complex than using the more declarative approach of HTML. Generating the user interface at the client has however the advantage the user inputs can be type checked before they are submitted to the DBMS and thus unnecessary client-server communications in the case of erroneous input are avoided.

Using an applet to access the database via RMI has also the restriction that only database can be accessed through a JDBC server that is running on the same server where the web server is running (this is a restriction of the applet mechanism)

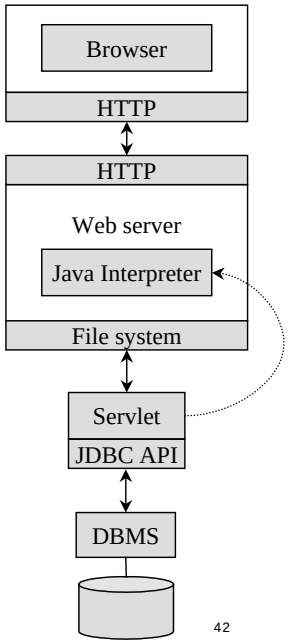
Applet-based Access: Basic Interaction



The detailed client-server interaction using applets and RMI works as follows: Once the applet is loaded the communication of the applet with the server takes place through synchronous communication over TCP/IP with the JDBC server via the RMI mechanisms (remote method invocation). In order to do this the application object (for database access this is the object that performs database accesses via JDBC) is instantiated at the server side. At the client side a "proxy" object is instantiated, that forwards method calls to the server-object from the client to the server. In that way the client can issue database accesses (transactions, SQL statements) to the database server. The applet on the client side generates the statements to be processed on the database and processes and displays the results.

4.3 Servlet/JSP-based Database Access

- Advantages
  - Server-side processing
  - HTTP-based communication
  - All general advantages of servlet/JSP technology (performance, development, security, sessions, etc.)
- Disadvantages
  - Servlet/JSP-enabled Web server required
  - No checking of input at client



With sevlet/JSP based access the database access functionality is executed directly within the servlet container of the web server and thus can benefit from the advantages of the servlet mechanisms, like sessions or increased performance.

Using servlet, and preferably JSP, based database access, one can combine the advantages of having server-side processing and purely http based communication, as compared to applet-based access, with the general advantages of the servlet technology for server-side processing, as compared to CGI-based access. The requirement is that the web server supports servlets/JSP. Since the interaction with the client is http based and html forms are used to capture user input, no type checking can be performed at the client side.

Summary

- Web-access to databases involves three issues
  - Accessing a database from within a programming language
  - Accessing a program from a Web browser
  - Combination of the two methods
- In addition to the approaches demonstrated, there exist many vendor-specific solutions
- Choice of the right method depends on
  - Performance requirements
  - Functional requirements
  - Development requirements
  - Available infrastructure
- The architectures for Web access to databases are another example of middleware
  - They are also a predecessor to todays *application servers*

Data Integration Checklist

- Task: integrate distributed, heterogeneous (and probably autonomous) data sources via Web access at the user interface

- Abstract Model ✓
  - HTML/HTTP
- Embedding ✓
  - JDBC + CGI/Servlets/JSP/Applet
- Architectures and Systems ✓
  - Web Servers + extensions

- Methodology ✗
  - Ad-hoc integration

References

- Books
  - R. Orfali, D. Harkey, J. Edwards:
    Client/Server Survival Guide, third edition, John Wiley, New York, 1999.
- Websites
  - JDBC http://java.sun.com/products/jdbc/
  - JDBC Tutorial http://java.sun.com/docs/books/tutorial/jdbc/index.html
  - Servlets http://java.sun.com/products/servlet/index.html
  - JSP http://java.sun.com/products/jsp/index.html
  - WebDataBlade http://examples.informix.com/frameset.html