

Database Systems I

CMPT 354 Summer 2024

Zhengjie Miao

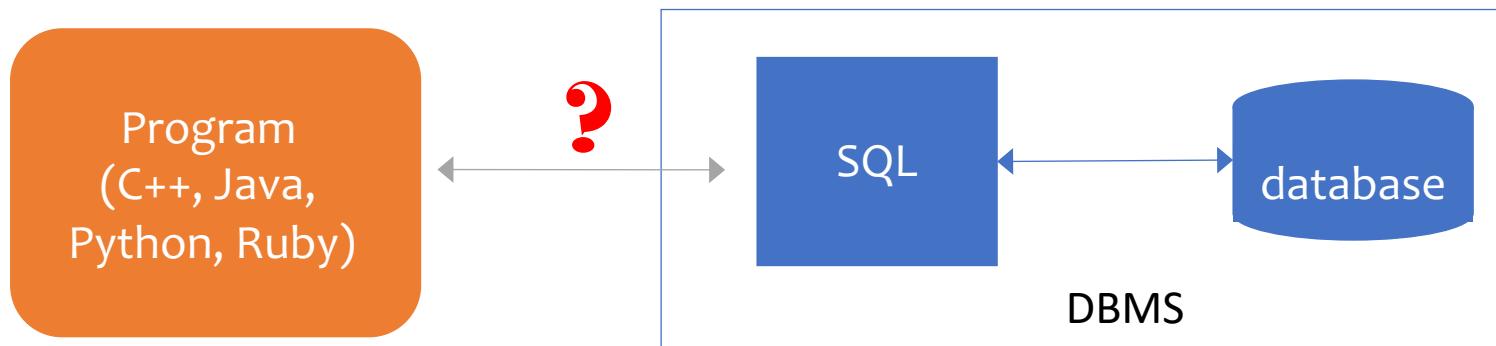
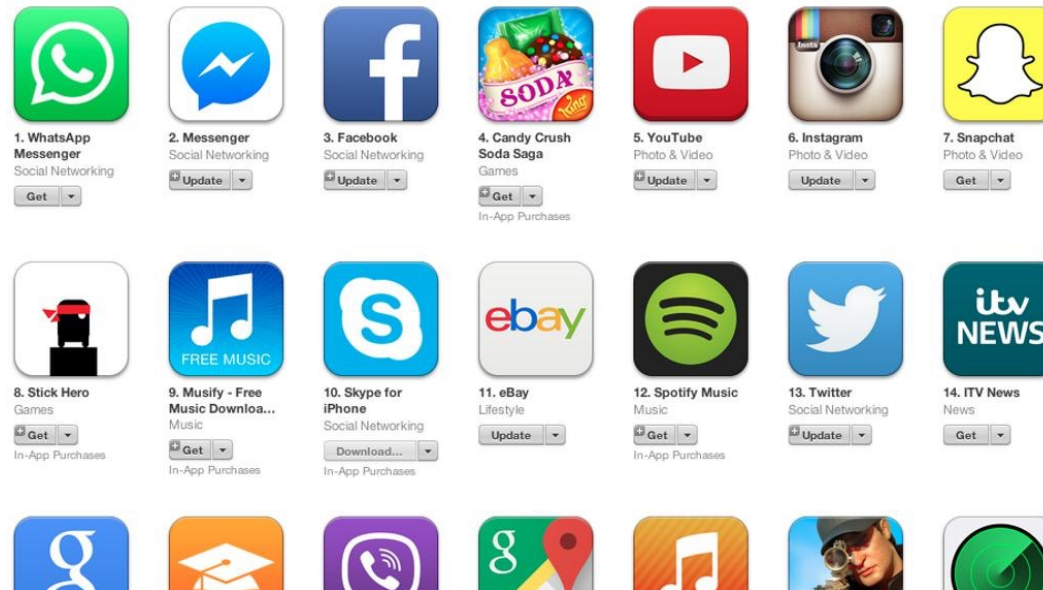
Announcements (Fri. June 21)

- Exam I
 - Solution & grade released
- Schedule
 - Some review lecture?

Why this lecture

- **DB designer:** establishes schema
- **DB administrator:** tunes systems and keeps whole things running
- **Data scientist:** manipulates data to extract insights
- **Data engineer:** builds a data-processing pipeline
- **DB application developer:** writes programs that query and modify a database

Application development



Outline

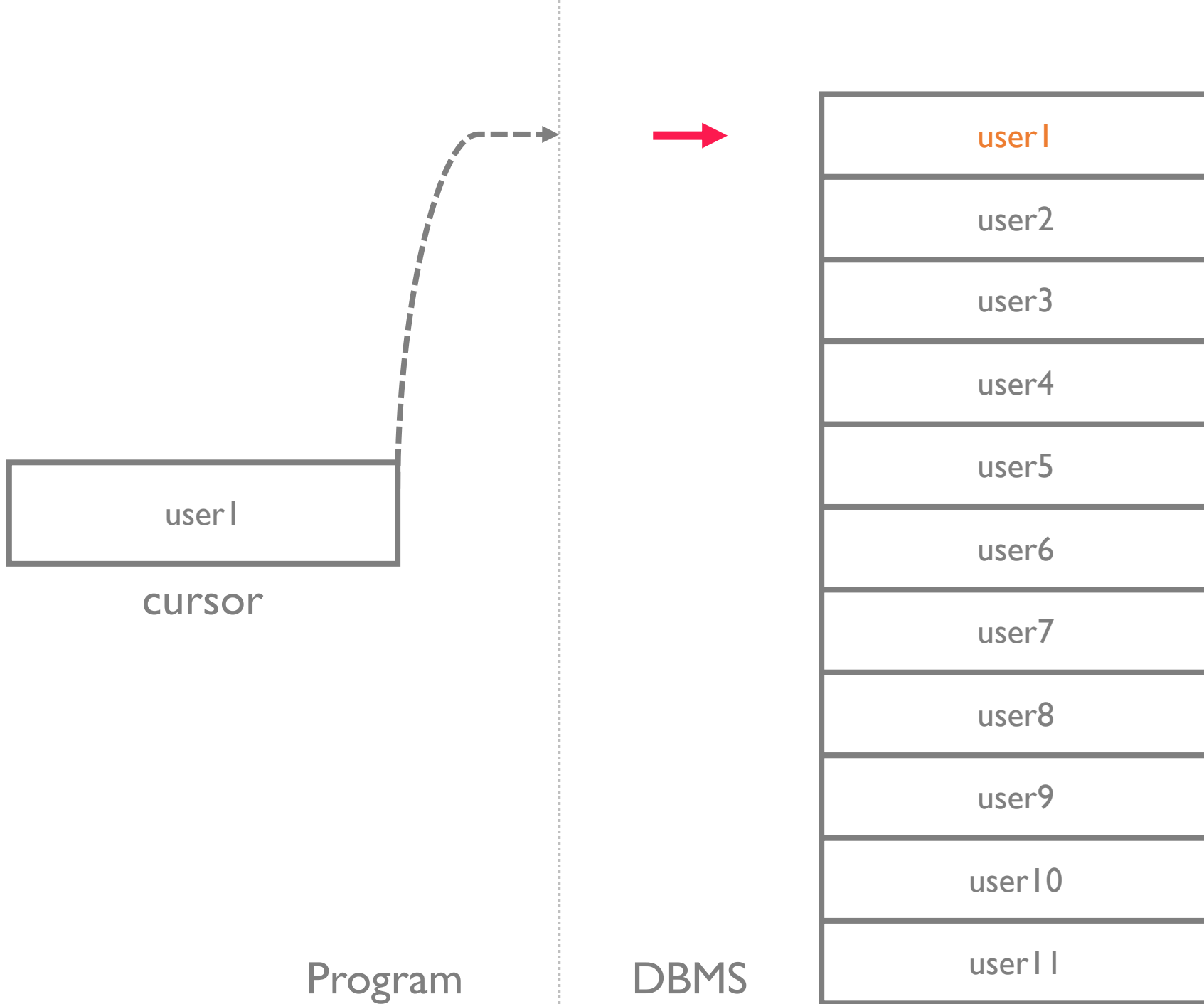
- **SQL Programming**
- Application Architecture

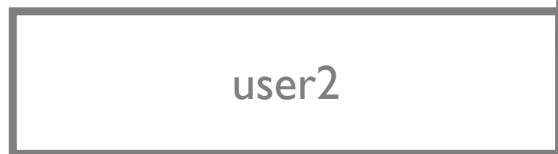
Motivation

- Pros and cons of SQL
 - Very high-level, possible to optimize
 - Not intended for general-purpose computation
- Solutions
 - Augment SQL with constructs from general-purpose programming languages
 - E.g.: SQL/PSM
 - Use SQL together with general-purpose programming languages: many possibilities
 - Through an API, e.g., Python psycopg2 or SQLAlchemy
 - Embedded SQL, e.g., in C
 - Automatic object-relational mapping, e.g.: Python SQLAlchemy
 - Extending programming languages with SQL-like constructs, e.g.: LINQ

An “impedance mismatch”

- SQL operates on **a set of records at a time**
- Typical low-level general-purpose programming languages operate on **one record at a time**
 - Less of an issue for functional programming languages
- Solution: **cursor**
 - **Open** (a result table): position the cursor before the first row
 - **Get next**: move the cursor to the next row and return that row; raise a flag if there is no such row
 - **Close**: clean up and release DBMS resources
 - Found in virtually every database language/API
 - With slightly different syntaxes
 - Some support more positioning and movement options, modification at the current position, etc.





cursor

Program



user1
user2
user3
user4
user5
user6
user7
user8
user9
user10
user11

DBMS

Augmenting SQL: SQL/PSM

- **PSM** = **P**ersistent **S**tored **M**odules
- **CREATE PROCEDURE** *proc_name*(*param_decls*)
 local_decls
 proc_body;
- **CREATE FUNCTION** *func_name*(*param_decls*)
 RETURNS *return_type*
 local_decls
 func_body;
- **CALL** *proc_name*(*params*);
- Inside procedure body:
 SET *variable* = **CALL** *func_name*(*params*);

SQL/PSM example

```
CREATE FUNCTION SetMaxPop(IN newMaxPop FLOAT)
  RETURNS INT
  -- Enforce newMaxPop; return # rows modified.
BEGIN
  DECLARE rowsUpdated INT DEFAULT 0;
  DECLARE thisPop FLOAT;

  -- A cursor to range over all users:
  DECLARE userCursor CURSOR FOR
    SELECT pop FROM User
  FOR UPDATE;

  -- Set a flag upon "not found" exception:
  DECLARE noMoreRows INT DEFAULT 0;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET noMoreRows = 1;

  ... (see next slide) ...

  RETURN rowsUpdated;
END
```

SQL/PSM example continued

```
-- Fetch the first result row:
OPEN userCursor;
FETCH FROM userCursor INTO thisPop;
-- Loop over all result rows:
WHILE noMoreRows <> 1 DO
    IF thisPop > newMaxPop THEN
        -- Enforce newMaxPop:
        UPDATE User SET pop = newMaxPop
        WHERE CURRENT OF userCursor;
        -- Update count:
        SET rowsUpdated = rowsUpdated + 1;
    END IF;
    -- Fetch the next result row:
    FETCH FROM userCursor INTO thisPop;
END WHILE;
CLOSE userCursor;
```

Other SQL/PSM features

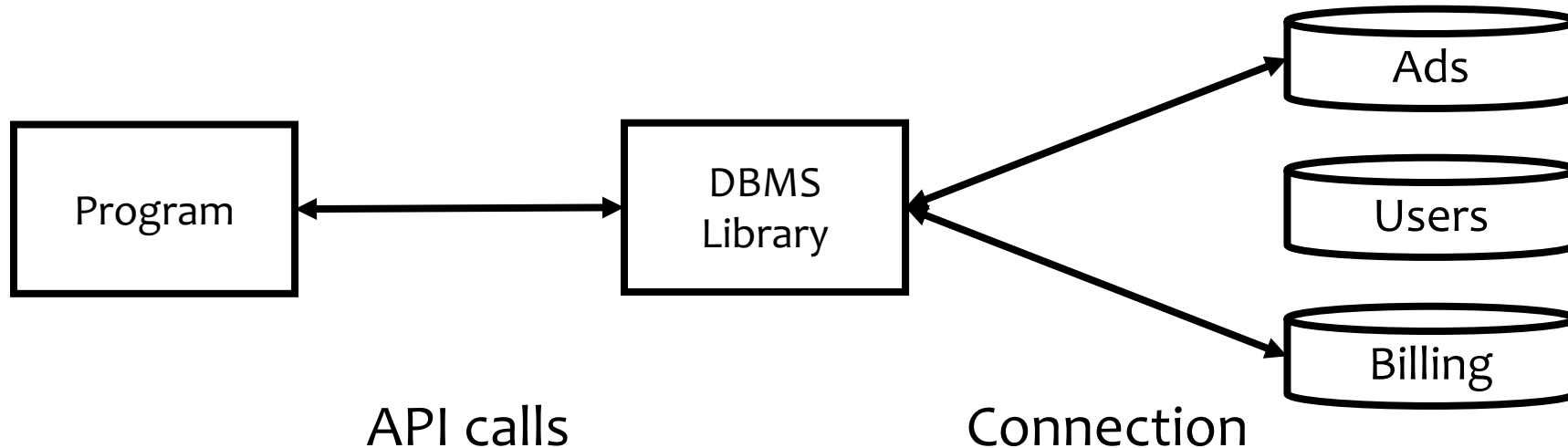
- Assignment using scalar query results
 - `SELECT INTO`
- Other loop constructs
 - `FOR`, `REPEAT UNTIL`, `LOOP`
- Flow control
 - `GOTO`
- Exceptions
 - `SIGNAL`, `RESIGNAL`

...

- For more PostgreSQL-specific information, look for “PL/pgSQL” in PostgreSQL documentation

Working with SQL through an API

- E.g.: Python psycopg2, JDBC, ODBC (C/C++/VB)
 - All based on the SQL/CLI (Call-Level Interface) standard
- The application program sends SQL commands to the DBMS at runtime
- Responses/results are converted to objects in the application program



Example API: Python SQLAlchemy

```
from sqlalchemy import create_engine, text
```

```
engine = create_engine(...)
```

```
with engine.begin() as conn:
```

```
    # list all drinkers:
```

```
    result = conn.execute(text('SELECT * FROM Drinker'))
```

```
    for drinker, address in result:
```

```
        print(drinker + ' lives at ' + address)
```

```
    # print menu for bars whose name contains "a":
```

```
    result = conn.execute\
```

```
        (text('SELECT * FROM Serves WHERE bar LIKE :pattern'),
```

```
        dict(pattern='%a%'))
```

```
    for bar, beer, price in result:
```

```
        print('{} serves {} at ${:,.2f}'.format(bar, beer, price))
```

You can iterate over result
one tuple at a time

Placeholder for
query parameter

Dictionary of parameter values,
one for each placeholder

More SQLAlchemy examples

with engine.connect() as conn:

```
# "commit" each change immediately in this session
conn.execution_options(isolation_level='AUTOCOMMIT')
# ...
```

```
bar = input('Enter the bar to update: ').strip()
beer = input('Enter the beer to update: ').strip()
price = float(input('Enter the new price: '))
```

```
try:
```

```
    result = conn.execute(text('''
UPDATE Serves SET price = :price
WHERE bar = :bar AND beer = :beer'''),
                           dict(price=price, bar=bar, beer=beer))
```

```
    if result.rowcount != 1:
        print('{} row(s) updated: correct bar/beer?' \
              .format(result.rowcount))
```

of tuples modified

```
except Exception as e:
```

```
    print(e)
```

Exceptions can be thrown
(e.g., if positive-price constraint is violated)

More SQLAlchemy examples

Are there any cases where you want reads to be transactions?

Since writes use transactions, does that mean that reads are automatically protected and do not need to use transactions?

One alternative (and recommended) for `index()` in `app.py` in Assignment 3, Problem 2

```
def index():  
    res = []  
    with db.engine.begin() as conn:  
        query_result = conn.execute(text("SELECT * FROM Stock WHERE sym =  
:sym ;"), dict(sym='AAPL'))  
        for sym, price in query_result:  
            res.append([sym, price])  
    return jsonify(res[0])
```

Parameters and
placeholders

May throw exceptions after
checking the # tuples returned

Type Mismatch

- SQL standard defines mappings between SQL and several languages

SQL types

CHAR(20)

INTEGER

SMALLINT

REAL

C types

char[20]

int

short

float

Python types

str

int

int

float

Prepared statements: motivation

```
while True:
```

```
    # Input bar, beer, price...
```

```
    result = conn.execute(text(''UPDATE Serves SET price = :price  
                                WHERE bar = :bar AND beer = :beer''),  
                           dict(price=price, bar=bar, beer=beer))
```

```
    # Check result...
```

- Every time we send an SQL string to the DBMS, it must perform parsing, semantic analysis, optimization, compilation, and finally execution
- A typical application issues many queries with a small number of patterns (with different parameter values)
- Can we reduce this overhead?

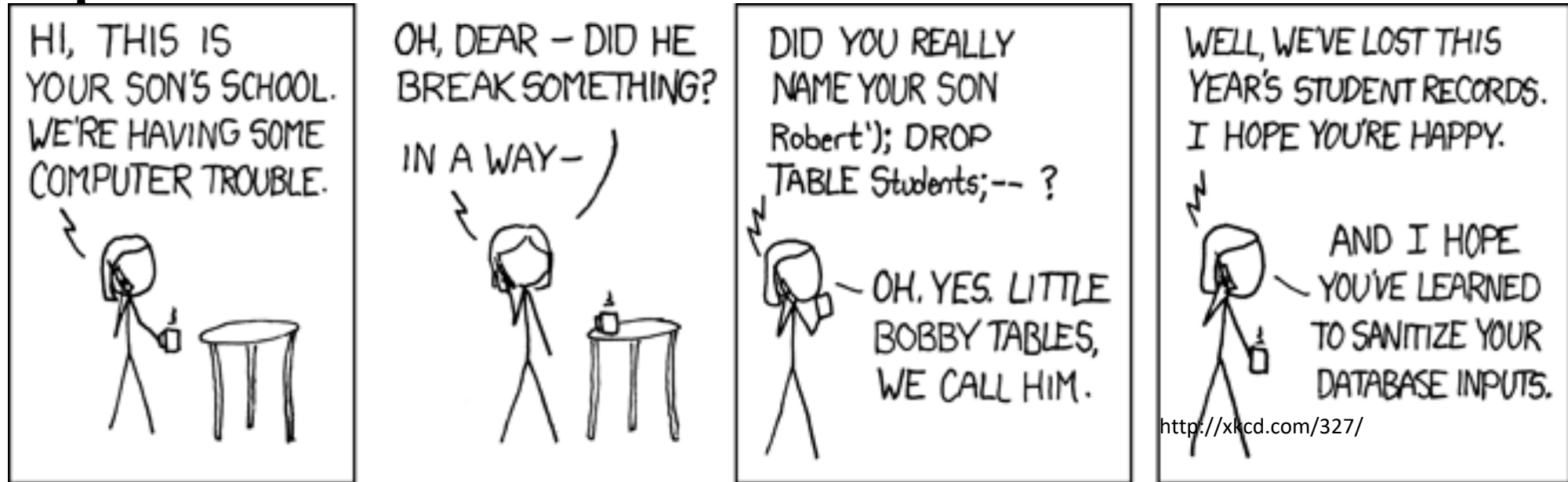
Prepared statements: example

```
conn.execute(text('''
PREPARE update_price AS
UPDATE Serves
SET price = $1
WHERE bar = $2 AND beer = $3'''))
# Prepare once (in SQL)
# Name the prepared plan
# and note the $1, $2, ... notation for
# PostgreSQL parameter placeholders

while True: # Input bar, beer, price...
    conn.execute(text('EXECUTE update_price(:bar, :beer, :price)'),
                  dict(price=price, bar=bar, beer=beer))
    # Note the switch back to : for parameter placeholders
    # Check result...
```

- The DBMS performs parsing, semantic analysis, optimization, and compilation only once, when it “prepares” the statement
- At execution time, the DBMS only needs to check parameter types and validate the compiled plan
- Most other backends/API’s have better support for prepared statements
 - E.g., they would provide a `conn.prepare()` method

“Exploits of a mom”



- The school probably had something like:

```
conn.execute(text("SELECT * FROM Students " + \  
                  "WHERE (name = '" + name + "')"))
```

where **name** is a string input by user
- Called an **SQL injection attack**

Guarding against SQL injection

- Escape certain characters in a user input string, to ensure that it remains a single string
 - E.g., ', which would terminate a string in SQL, must be replaced by " (two single quotes in a row) within the input string
- Luckily, most database API's provide ways to “sanitize” input automatically (if you use them properly)
 - E.g., pass parameter values in sqlalchemy through :’s
- Do **NOT** rely non-DB API/language’s string support
 - E.g., ~~"name = '" + name + "'"~~ or ~~f"name = '{name}'"~~

If one fails to learn the lesson...



*... P.S. To Ashley Madison's Development Team:
You should be embarrassed [sic] for your train
wreck of a database (and obviously security), not
sanitizing your phone numbers to your database
is completely amateur, it's as if the entire site was
made by Comp Sci 1XX students.*

— Creators of CheckAshleyMadison.com

Augmenting SQL vs. API

- Pros of augmenting SQL:
 - More processing features for DBMS
 - More application logic can be pushed closer to data
 - Less data “shipping,” more optimization opportunities ⇒ more efficient
 - Less code ⇒ easier to maintain multiple applications
- Cons of augmenting SQL:
 - SQL is already too big—at some point one must recognize that SQL/DBMS are not for everything!
 - General-purpose programming constructs complicate optimization and make it impossible to guarantee safety

A brief look at other approaches

- “Embed” SQL in a general-purpose programming language
 - E.g.: embedded SQL With embedded SQL, the compiler checks
- Support database features through an object-oriented programming language
 - By automatically storing objects in tables and translating methods to SQL
 - E.g., object-relational mappers (ORM) like Python SQLAlchemy
- Extend a general-purpose programming language with SQL-like constructs
 - E.g.: LINQ (Language Integrated Query for .NET)

Embedding SQL in a language

Example in C

```
EXEC SQL BEGIN DECLARE SECTION;
int thisUid; float thisPop;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE ABCMember CURSOR FOR
    SELECT uid, pop FROM User
    WHERE uid IN (SELECT uid FROM Member WHERE gid = 'abc')
    FOR UPDATE;

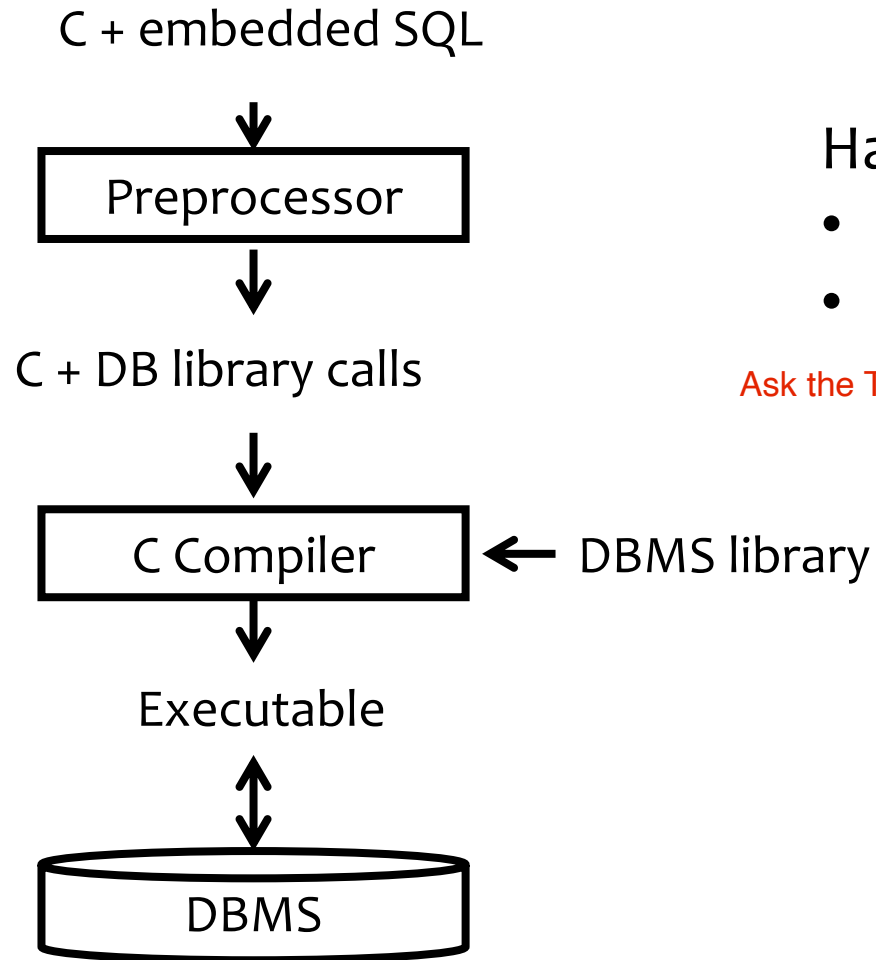
EXEC SQL OPEN ABCMember;
EXEC SQL WHENEVER NOT FOUND DO break;
while (1) {
    EXEC SQL FETCH ABCMember INTO :thisUid, :thisPop;
    printf("uid %d: current pop is %f\n", thisUid, thisPop);
    printf("Enter new popularity: ");
    scanf("%f", &thisPop);
    EXEC SQL UPDATE User SET pop = :thisPop
        WHERE CURRENT OF ABCMember;
}

EXEC SQL CLOSE ABCMember;
```

} Declare variables to be "shared"
between the application and DBMS

→ Specify a handler for
NOT FOUND exception

Embedded SQL



Hard to maintain

- What if SQL evolves?
- What if Compiler evolves?

Ask the TA about these points

Object-relational mapping

- Example: Python SQLAlchemy

```
class User(Base):  
    __tablename__ = 'users'  
    id = Column(Integer, primary_key=True)  
    name = Column(String)  
    password = Column(String)
```

```
class Address(Base):  
    __tablename__ = 'addresses'  
    id = Column(Integer, primary_key=True)  
    email_address = Column(String, nullable=False)  
    user_id = Column(Integer, ForeignKey('users.id'))
```

```
Address.user = relationship("User", back_populates="addresses")
```

```
User.addresses = relationship("Address", order_by=Address.id, back_populates="user")
```

```
jack = User(name='jack', password='gjffdd')
```

```
jack.addresses = [Address(email_address='jack@google.com'), Address(email_address='j25@yahoo.com')]
```

```
session.add(jack)
```

```
session.commit()
```

```
session.query(User).join(Address).filter(Address.email_address=='jack@google.com').all()
```

- Automatic data mapping and query translation
- But syntax may vary for different host languages
- Very convenient for simple structures/queries, but quickly get complicated and less intuitive for more complex situations

Deeper language integration

- Example: LINQ (Language Integrated Query) for Microsoft .NET languages (e.g., C#)

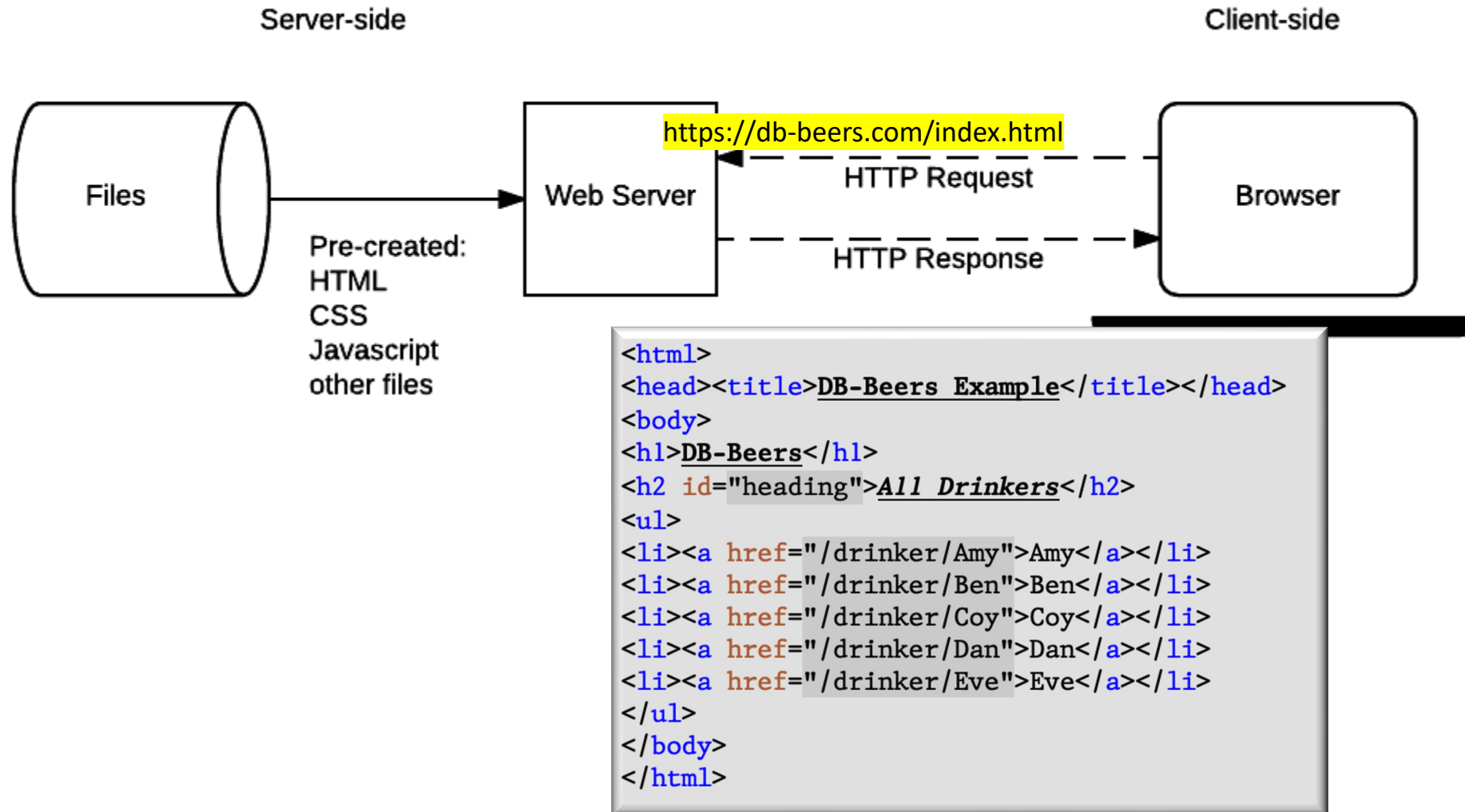
```
int someValue = 5;
var results = from c in someCollection
               let x = someValue * 2
               where c.SomeProperty < x
               select new {c.SomeProperty, c.OtherProperty};
foreach (var result in results) {
    Console.WriteLine(result);
}
```

- Again, automatic data mapping and query translation
- Much cleaner syntax, but it still may vary for different host languages

Outline

- SQL Programming
- **Application Architecture**

A static website



Towards a dynamic website

- Imagine a **function** that dynamically generates the HTTP response
 - Visiting the URL leads to calling the function
 - The function returns the HTML page (as a string)
 - The function can **query the database for content!**

```
def all_drinkers():  
    html_out = '<html>\n'  
    # more HTML...  
    html_out += '<ul>\n'  
    with engine.begin() as conn:  
        result = conn.execute(text('SELECT * FROM drinker'))  
        for name, address in result:  
            html_out += '<li><a href="/drinker/{}>{}</a></li>\n'.format(name, name)  
    html_out += '</ul>\n'  
    # more HTML...  
    html_out += '</html>\n'  
    return html_out
```

*Tedious,
ugly code
though!*

“Refactor” that function!

- Separate data from presentation, e.g.:
 - Data: list of drinker names
 - Presentation: a HTML **template**, with some processing directives to embed data items

```
<html>
<head><title>DB-Beers Example</title></head>
<body>
<h1>DB-Beers</h1>
<h2 id="heading">All Drinkers</h2>
<ul>
{% for drinker in drinkers %}
  <li><a href="{{ url_for('drinker', name=drinker) }}">{{ drinker }}</a></li>
{% endfor %}
</ul>
</body>
</html>
```

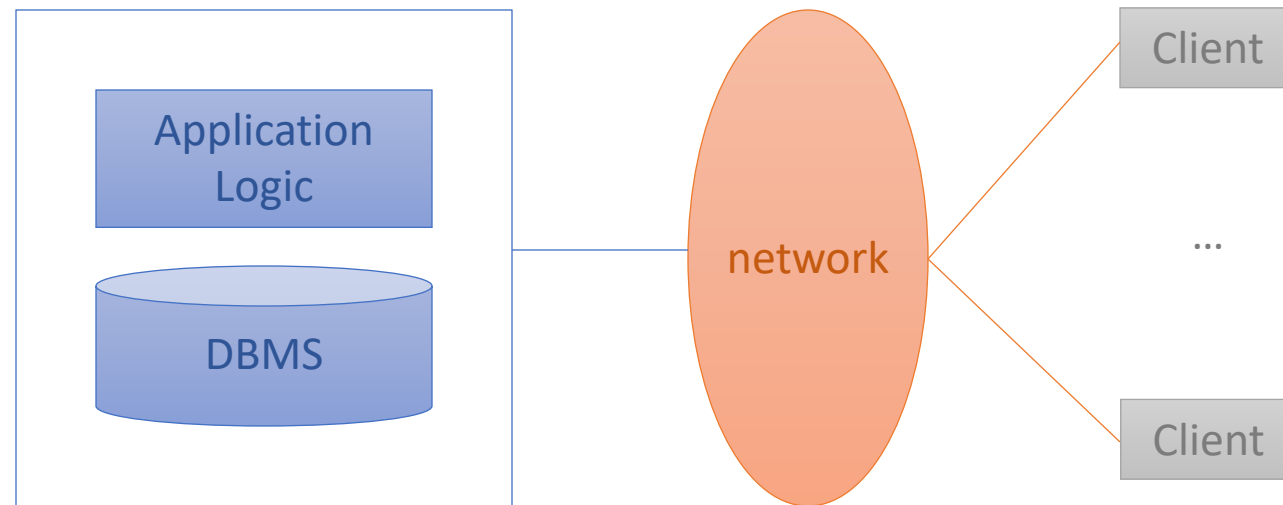
Template for all-drinker listing

```
def all_drinkers():
    with engine.begin() as conn:
        result = conn.execute(text('SELECT * FROM drinker'))
        drinker_names = [name for name, address in result]
        return render_template('index.html', drinkers=drinker_names)
```

Much simplified code/logic

Two-Tier Architecture

- Client/ server architecture
 - The server implements the business logic and data management
- Separate presentation from the rest of the application



Presentation Layer

- Responsible for handling the user's interaction with the middle tier
- One application may have multiple versions that correspond to different interfaces
 - Web browsers, mobile phones, ...
 - Style sheets can assist in controlling versions

Calling these functions

- URL in the HTTP request specifies the function to call
 - AKA routes, endpoints

```
@app.route('/')
def index():
    """
    an alternative implementation of the index() function
    with query parameter and placeholder
    """
    res = []
    with db.engine.begin() as conn:
        query_result = conn.execute(text("SELECT * FROM Stock WHERE sym = :sym ;"),
                                     dict(sym='AAPL'))
        for sym, price in query_result:
            res.append([sym, price])
    return jsonify(res[0])
```

Calling these functions

- URL in the HTTP request specifies the function to call
 - AKA routes, endpoints
- HTTP request also encodes any input parameter values to call the function with
 - Can be part of the URL (GET) or the request body (POST)

```
@app.route('/drinker/<name>')  
def drinker(name):  
    with engine.begin() as conn:  
        result = conn.execute(text('SELECT * FROM drinker WHERE name = :s'),  
                                dict(s=name))  
        name, address = result.fetchone()  
    return render_template('drinker.html',  
                           drinker_name=name,  
                           drinker_address=address)
```

A page showing the details about one drinker

But who calls these functions?

- User usually don't type specific URLs themselves
- Calls are typically embedded in your HTML pages

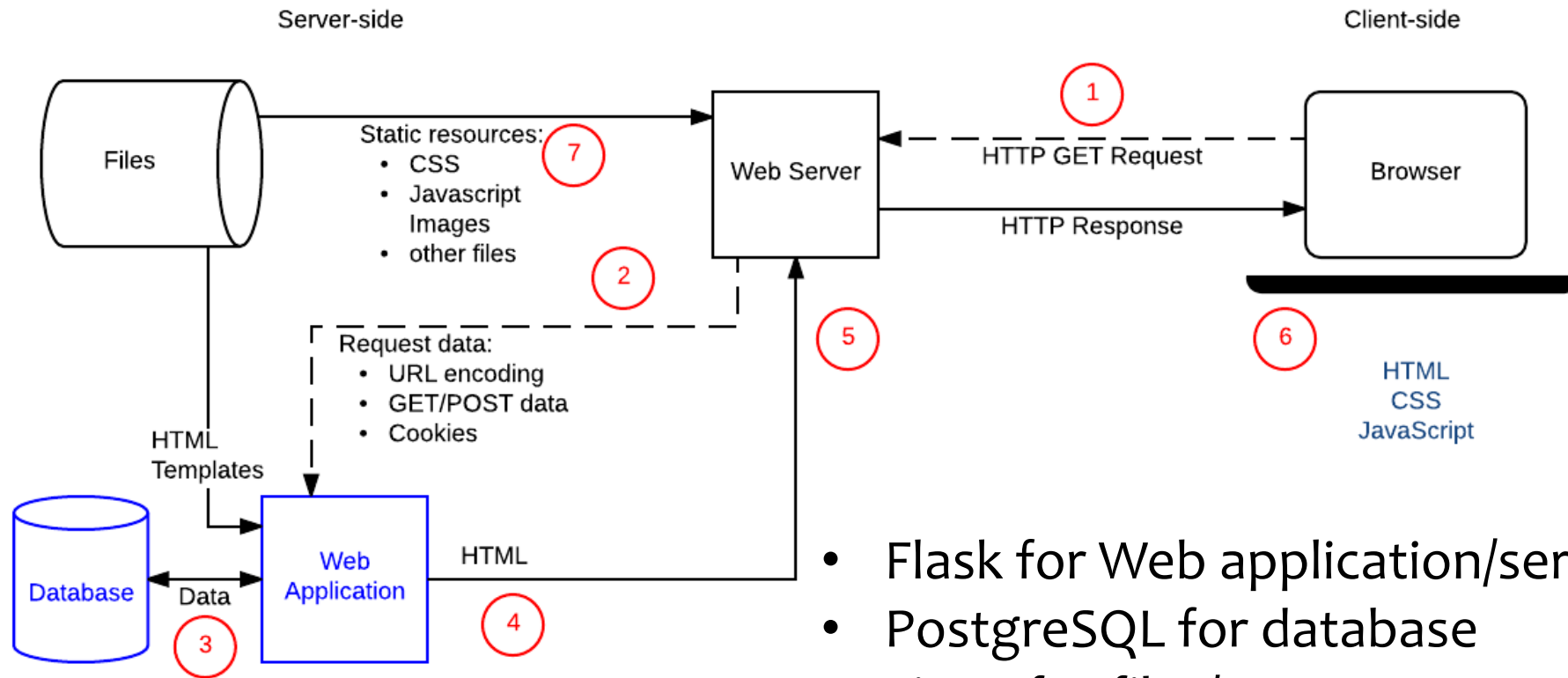
Template for all-drinker listing

```
<html>
<head><title>DB-Beers Example</title></head>
<body>
<h1>DB-Beers</h1>
<h2 id="heading">All Drinkers</h2>
<ul>
{% for drinker in drinkers %}
  <li><a href="{{ url_for('drinker', name=drinker) }}">{{ drinker }}</a></li>
{% endfor %}
</ul>
</body>
</html>
```

Setting up the URL for showing a specific drinker

```
<html>
<head><title>DB-Beers Example</title></head>
<body>
<h1>DB-Beers</h1>
<h2 id="heading">All Drinkers</h2>
<ul>
<li><a href="/drinker/Amy">Amy</a></li>
<li><a href="/drinker/Ben">Ben</a></li>
<li><a href="/drinker/Coy">Coy</a></li>
<li><a href="/drinker/Dan">Dan</a></li>
<li><a href="/drinker/Eve">Eve</a></li>
</ul>
</body>
</html>
```

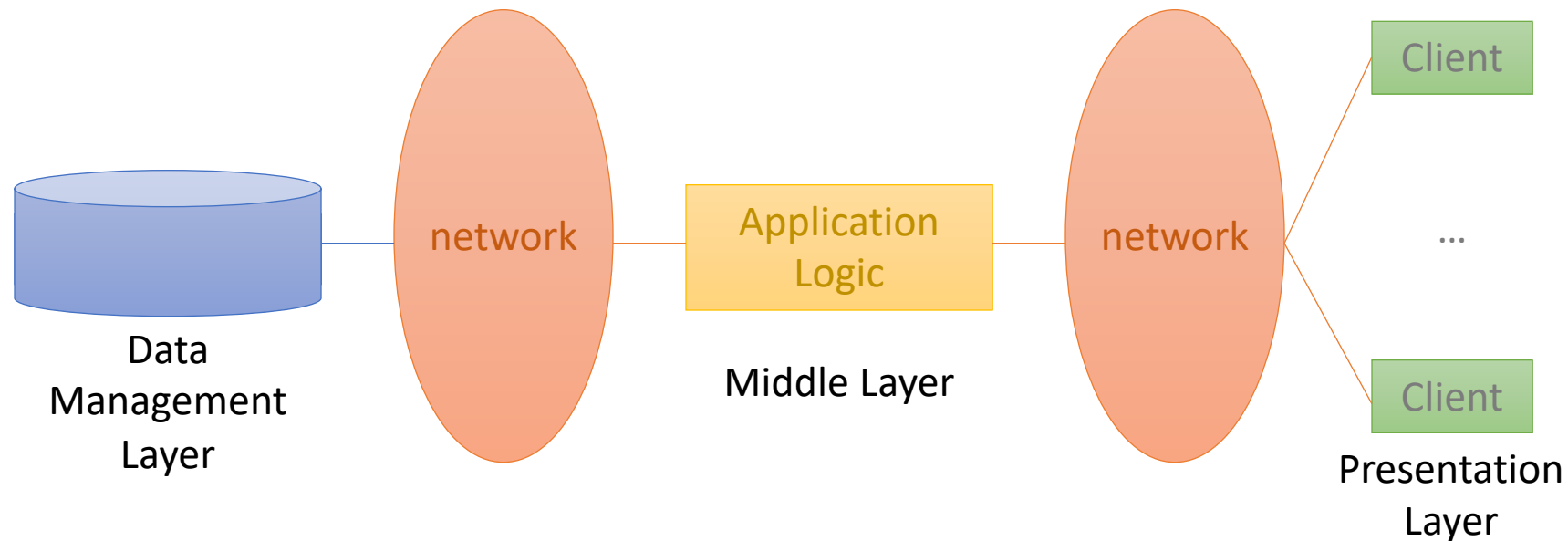
Full-stack



- Flask for Web application/server
- PostgreSQL for database
- Linux for files/server

Three-Tier Architecture

- Separate presentation from the rest of the application
- Separate the application logic from the data management



Business logic Layer

- The middle layer is responsible for running the business logic of the application which controls
 - What data is required before an action is performed
 - The control flow of multi-stage actions
 - Access to the database layer
- Multi-stage actions performed by the middle tier may require database access
 - But will not usually make permanent changes until the end of the process
 - e.g. adding items to a shopping basket in an Internet shopping site

Data Management Layer

- The data management tier contains one, or more databases
 - Which may be running on different DBMSs
- Data needs to be exchanged between the middle tier and the database servers
 - This task is not required if a single data source is used but,
 - May be required if multiple data sources are to be integrated
 - *XML* is a language which can be used as a data exchange format between database servers and the middle tier

Example: RATest

- Consider the three tiers in the RATest website
- Database System
 - Student info, questions, solutions, submissions...
- Application Server
 - Logic to consent recording the history, submit queries
- Client Program
 - Display queries, outputs, and query trees

Example: Course Enrollment

- Student enrollment system tiers
- Database System
 - Student information, course information, instructor information, course availability, pre-requisites, etc.
- Application Server
 - Logic to add a course, drop a course, create a new course, etc.
- Client Program
 - Log in different users (students, staff, faculty), display forms and human-readable output

Summary

- SQL Programming
 - Augmenting SQL
 - DB API
- Application Architecture
 - Three Tier Architecture