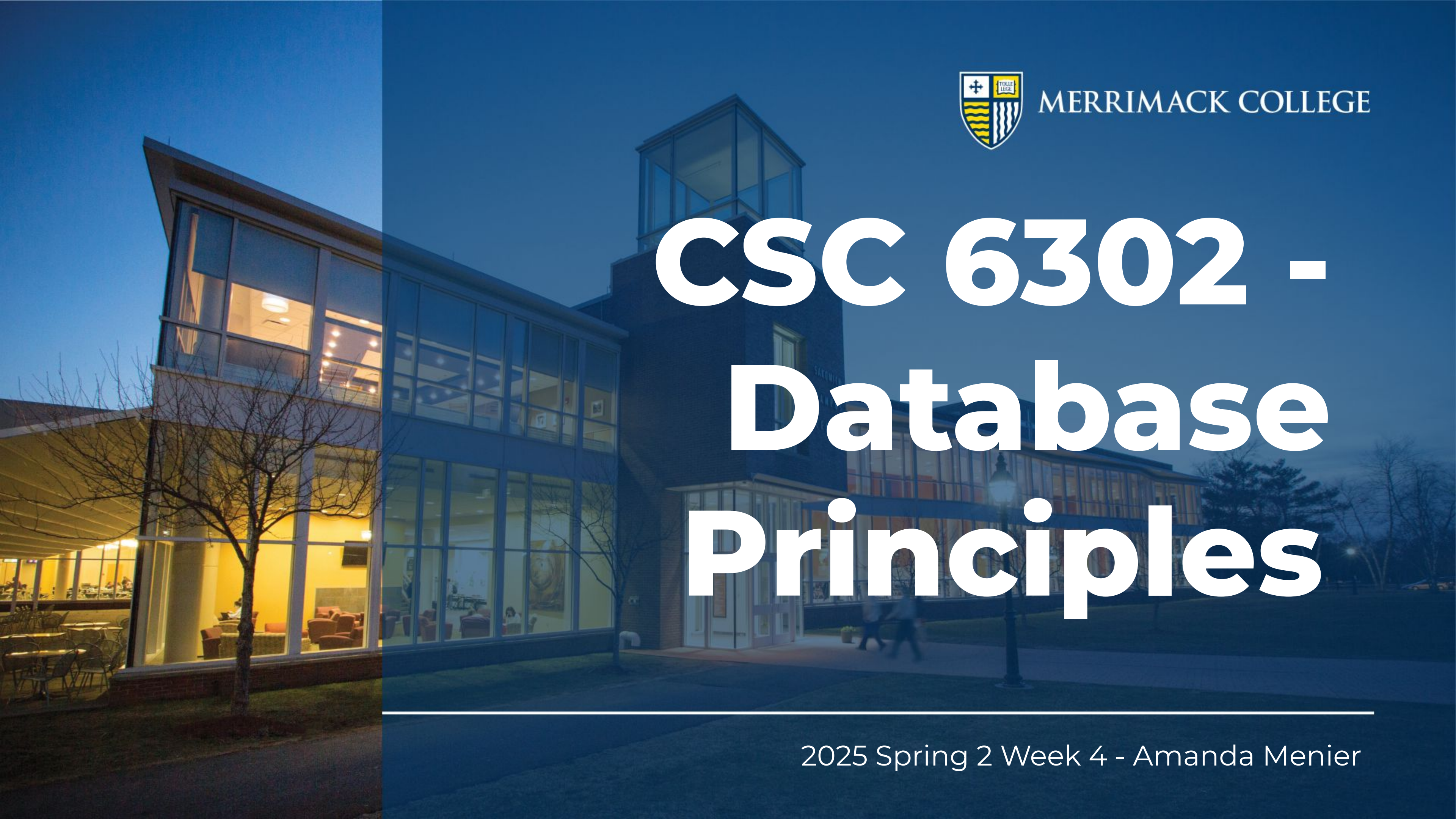


MERRIMACK COLLEGE

CSC 6302 - Database Principles

2025 Spring 2 Week 4 - Amanda Menier

Agenda

Discussion Topics:

- Week 2 Project Notes
- Application Architecture
- Model - View - Controller
- Presentation, Business Logic, Data Access
- Application Architecture Concerns
- Python MySQL Connector
- JDBC
- Node MySQL

MERRIMACK COLLEGE

Project 3 Notes

1) Make dates user friendly
2) Rounding issues

Application s

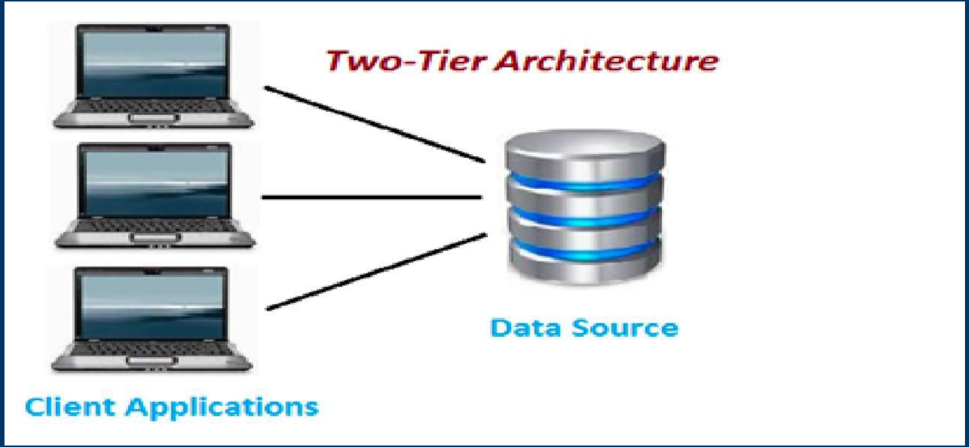
Most database users do not use a query language like SQL.

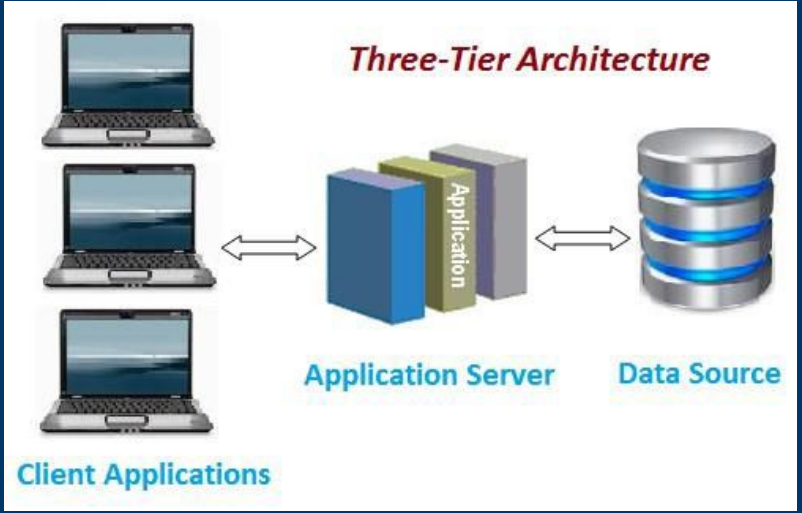
An application program acts as the intermediary between users and the database.

Applications split into:

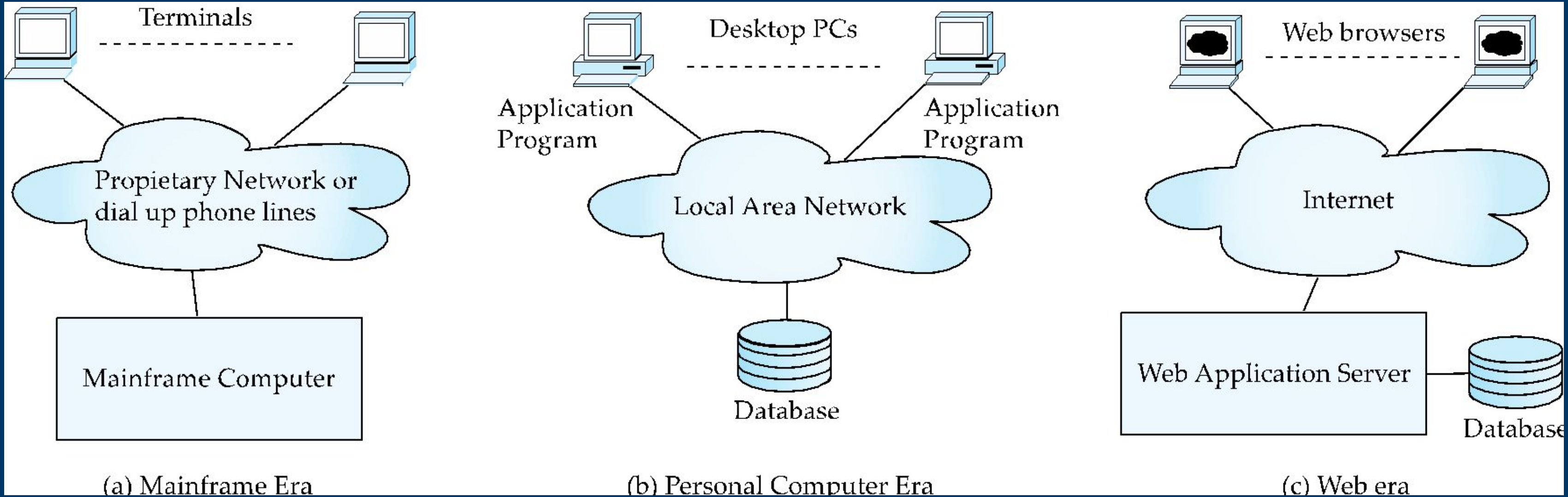
- Front-end:
  - How do I present my data?
    - GUI
    - Websites
    - Mobile applications
- Middle layer:
  - How do I process my data?
    - Business logic
    - Data Access
- Backend:
  - How do I store my data?
    - Database

Two-Tier vs Three Tier Architecture





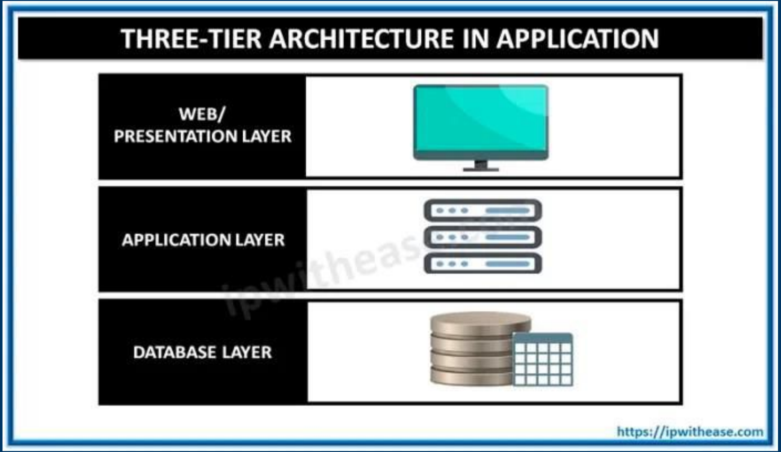
Evolution



(a) Mainframe Era (b) Personal Computer Era (c) Web era

MERRIMACK COLLEGE

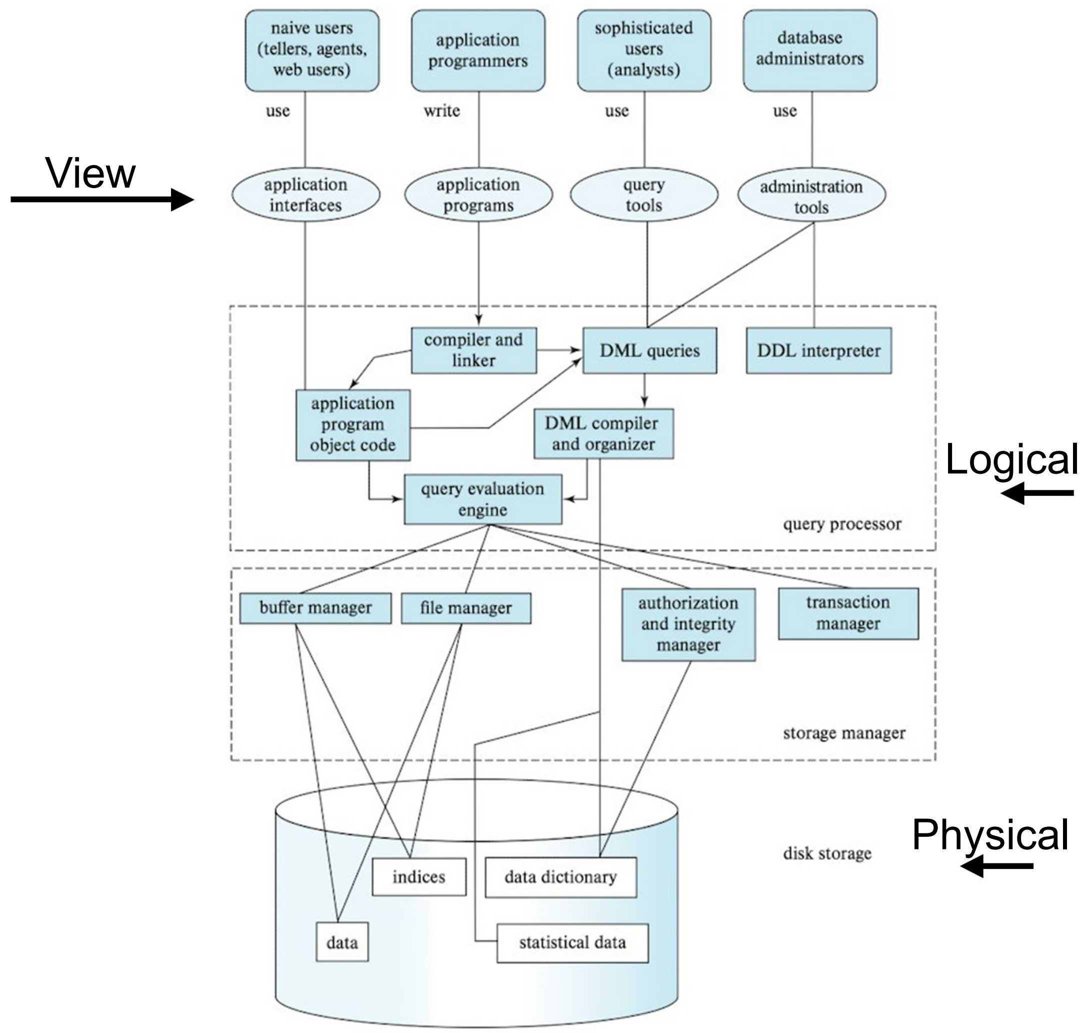
Modern Era



Most applications in the modern era include a varied presentation layer

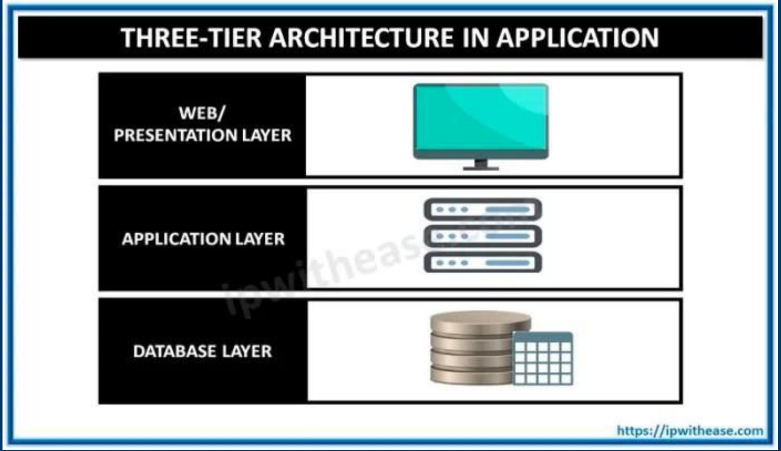
- An application running on a personal computer
  - Requires installation
  - Hardware requirements come into play
- A client application connected to a server
  - Allows multi-user application
  - Handle case when client cannot communicate with server
- A web application in a browser
  - Multi-user application that removes hardware dependencies
  - Allows the use of offsite servers, cloud storage, ect
  - Loss of network connectivity will have an impact
- A mobile application
  - Adds a convenience factor
  - Access anywhere there is a network connection

Application Layers



MERRIMACK COLLEGE

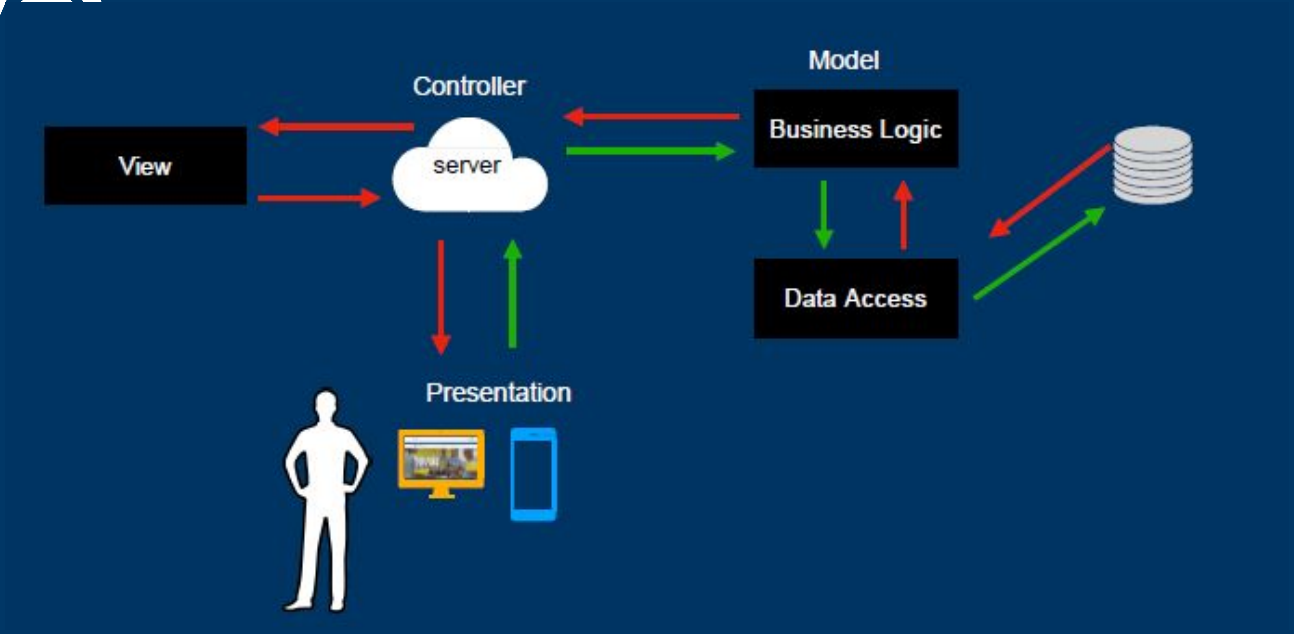
Presentation Layer: Web Wins?



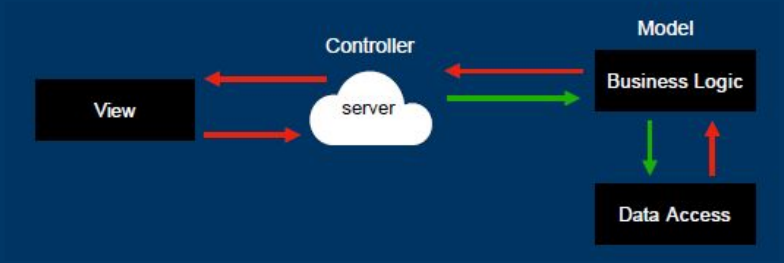
Web browsers have become the de-facto standard user interface to databases:

- Enable large numbers of users to access databases from anywhere
- Avoid the need for downloading/installing specialized code, while providing a good graphical user interface
- Javascript and other scripting languages run in browser, but are downloaded transparently
- Allows for access on mobile devices
- Avoiding the need to download large amounts of code or data is especially important in the mobile environment
- Examples: banks, airline and rental car reservations, university course registration and grading, an so on

Model - View - Controller (MVC)

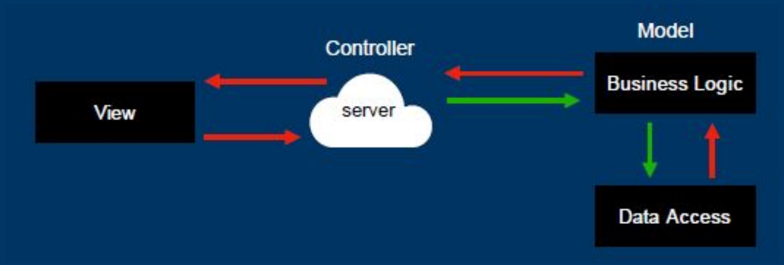


Application Layer



- Business Logic Layer
  - Provides high level view of data and actions on data
  - Often using an object data model
  - Hides details of data storage schema
  - Abstraction of entities (students, courses, etc)
  - Enforce business rules (can only register if prerequisites are met)
  - Supports workflows among multiple participants (handles submitting of application by student and reviewing of application by admissions)
  - Handles sequence of steps to be followed
  - Handles errors or omissions (what if letter of rec is not submitted?)

Application Layer



- Data Access Layer
  - Interfaces between business logic layer and the underlying database
  - Provides mapping from object model of business layer to relational model of database
  - Provides abstraction of the database contents
  - Programmers don’t care about tables and foreign keys or how the underlying database is set up
  - Programmers want to populate their data structures with the data they need for their business logic
  - The database should only be accessed from the DAL
  - Format data from database in a readable way
  - Sometimes we store an integer and want to map it to an enum Sometimes we want to map multiple columns to a single data member

MERRIMACK COLLEGE

Connectivity Concerns

- What happens when:
  - The internet is not available?
  - Mobile networks are not available?
  - A local internet connection goes down?
  - A particular server cannot be reached?
  - A database server goes down?
  - Someone accidentally unplugs something?
- How can we provide redundancy in our application architecture to ensure inevitable problems are manageable?
  - Virtualization and failure techniques
  - RAID
  - Does some data need to be stored locally?
  - How do we keep data consistent if its stored in multiple places?

Performance Concerns

- Performance is an issue for popular Websites and applications
  - May be accessed by millions of users every day, thousands of requests per second at peak time
- Knowing the hardware requirements of your application to fit your requirements
  - Caching techniques used to reduce cost of serving data by exploiting commonalities between requests
  - Caching connections for reuse in a connection pool
  - Caching results of database queries
  - Caching of generated HTML for web pages
- Cached results must be updated if underlying database changes

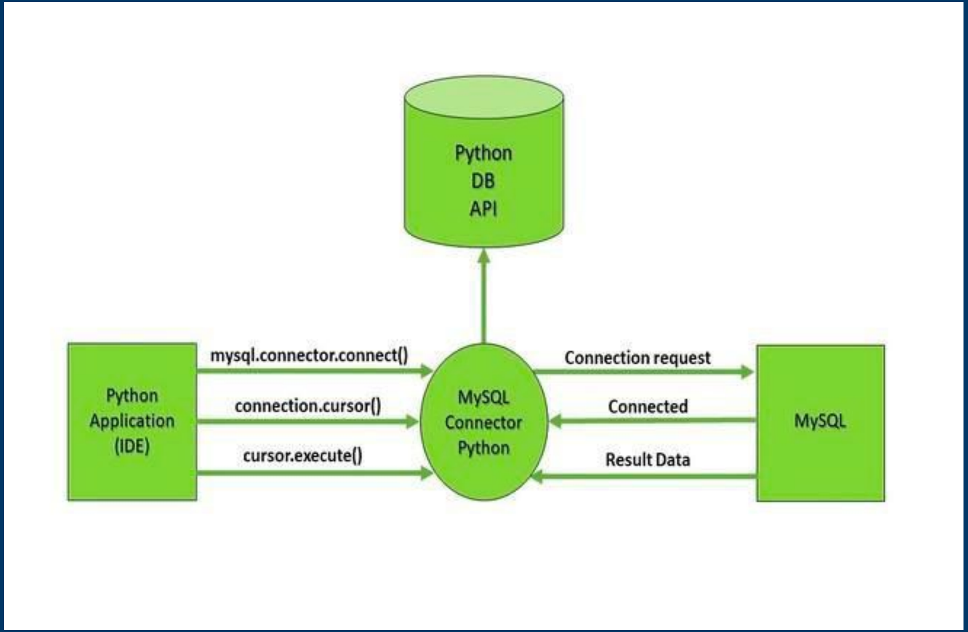
Security Concerns

- Never store passwords, such as database passwords, in clear text in scripts that may be accessible to users
  - On files accessible to a web server
  - In the database itself
- Limit database access to database servers
- Two-factor authentication is more secure than single-factor authentication
  - E.g., password plus one-time password sent by SMS or app
  - Device generates a new pseudo-random number every minute, and displays to user
  - User enters the current number as password
  - Application server generates same sequence of pseudo-random numbers to check that the number is correct.
- Application controls user access to database
  - Application users cannot access database directly

Audit Trails

- Applications must log actions to an audit trail, to detect who carried out an update, or accessed some sensitive data
- Audit trails used after-the-fact to
  - Detect security breaches
  - Repair damage caused by security breach
  - Trace who carried out the breach
- Audit trails needed at
  - Database level, and at
  - Application level
- Exceptionally useful for debugging issue in the field
  - Product logs
  - Database logs
  - Operating system logs

Python MySQL Connector



Python MySQL Connector

- MySQLConnector is a Python Driver for connecting to mysql databases from Python programs
  - Supplies an API with functionality to connect to the database
    - How to connect to a particular database
    - How to run a query against a particular database
- Requires the MySQLConnector to be installed on your machine
  - Ensure that you have MySQL 4.0 or greater installed
  - This shouldn't be a problem
  - There are security concerns with early versions of MySQL that MySQLConnector no longer supports

MySQLConnector Documentation

Connecting

- mysql.connector.connect(host = "localhost",user = "yourusername",password = "your_password", database =" your_database")
  - HostName is the host where your database is installed
  - Database Name is the database you want to connect to
  - There should be a unique connection to each database for simplicity
- MySQL.Connector.Connect returns a database connection object
  - You can request a new connection every single time you query the database.
  - This is more secure, as you don't have open connections, but slow
  - Caching a database connection within the DAL
- Abstraction
  - Business logic requests a connection from the DAL,
    - Doesn't worry about the details of creating connection
- In a larger application, the database server would manage database connections in a connection pool, handling authentication, timeout, security

Queries

- MySQL.Connector.Connect returns a database Connection object
  - A Cursor executes sql statement against a Connection
    - A Cursor can be executed
    - The results of executing the SQL Statement are stored in the cursor
    - A Cursor must be closed when you are done with it!

```
mydb = mysql.connector.connect(host = “localhost",user = “admin_user, password =
"admin1234",database = "SCHOOL")

cursor = mydb.cursor()
cursor.execute("SHOW TABLES”)

for x in cursor:
    print(x)
cursor.Close();
```

FetchOne and FetchAll

Reference Site

- Methods available with using the cursor object
- FetchOne and FetchAll are Python methods used to retrieve data from a mysql database
- FetchOne returns the first row of a query result
  - Similar to using Limit 1 in MySQL
- FetchAll returns all the rows of the query result

```
row = cursor.fetchone()
otherRow = cursor.fetchAll()
```

FetchAll

- Import the MySQL.Connector driver
- Connect to the database using MySql.Connector.Connect
- Create a cursor objector
- Execute a query using the cursor object
- Use a loop with fetch all to retrieve the results from the cursor

—--

```
import mysql.connector

mydb = mysql.connector.connect(host = ‘localhost’, user = ‘yourusername’, password=’yourpass’,
database = ‘yourdatabase’)

 mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM STUDENT;")
result = mycursor.fetchall()
for all in result:
        print(all)
```

Python and Stored Procedures

Tutorial for Using Stored Procedures
Accessing Stored Procedure Results from Cursor

CallProc Documentation

Python and Stored Procedures

- Import the MySQL.Connector driver
- Create a MySQL.Connector.Connection
- Create a cursor object
- Use the CallProc method of the cursor object
- Use FetchOne or FetchAll to retrieve the results
- mydb = mysql. connector. connect(host = " localhost", =

```
user "yourusername",password = "yourpass",database =

"yourdatabase")
mycursor = mydb.cursor()

cursor.CallProc('InsertRecipe')
for result in cursor.stored_results():

   print(result.fetchall())
```

Python and Stored Procedures With Variables

- Method Cursor.CallProc can optionally take in a list of arguments
  - The list supplied to the method must have exactly the same number of arguments expected by the stored procedure
  - The order of the arguments in the list must match exactly to the order of the arguments accepted by the stored procedure
  - The types of the arguments in the list must match exactly to the order of the arguments accepted by the stored procedure

JDBC

What is JDBC?

- Java Database Connectivity
  - A Java API that allows java programs to access your databases
- Requires the JDBC Driver to be installed on your system
  - Classes that implement the JDBC interfaces
    - How to connect to a particular database
    - How to run a query against a particular database
- Ensure that you have:
  - The latest version of Java and JDBC installed on your system
    - The version of JDBC you install must be compatible with your currently installed version of Java
- JDBC Documentation

JDBC: Connection to a Database

- **DriverManager.getConnection(url, databaseName, user, password)**
  - Why a URL?
    - Database Servers are part of the application layer and often manage database connections
    - Client machines may want a connection to the database, so they have to go through the Database Server
      - Specify a url for the database server
    - For small applications, the database is on our local host:
      - "jdbc:mysql://localhost:3306/"
- Database Name
  - JDBC can connect to multiple databases
  - There should be a unique connection to each database for simplicity
- Credentials
  - admin_user
  - admin1234

Caching Database Connections

- **DriverManager.GetConnection** returns a database connection object
  - You can request a new connection every single time you query the database
    - This is more secure, as you don't have open connections
    - This is also slow, as it takes time to establish a connection
  - Caching a database connection within the DAL
    - Abstraction
      - Business logic requests a connection from the DAL,
        - Doesn't worry about the details of creating connection
      - In a larger application, the database server would manage database connections in a connection pool
        - Handle authentication concerns
        - Handle connection timeout concerns
        - Handle security concerns

Querying JDBC Connections

- **DriverManager.GetConnection** returns a database **Connection** object
  - a SQL **Statement** executes against a **Connection**
    - A SQL **Statement** can be executed and returns a **ResultSet**

```
Connection myConnection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/"  +
databaseName, user, password);

 Statement myStatement = myConnection.createStatement();
 String query = "SELECT * FROM STUDENT”;
 ResultSet myRelation = myStatement.executeQuery(query);
```

JDBC Statements

- Statement
  - Use for executing queries against a database
  - Pros:
    - Quick and easy way to get some data from the database
  - Cons:
    - Hard coding SQL in our java code
    - SQL Injection Attacks
    - Escaping SQL special characters, like single quotes, is painful and error prone
- Prepared Statement
  - Executes parameterized queries against a database
  - Allows for the database engine to cache query plans, and reuse them
    - Better performance then a Statement
- Callable Statement
  - Used to execute stored procedures and functions
  - Allows for input and output parameters to be specified/accessed

Statements and Result Set

- Logically a table, with columns and rows, that corresponds to the results of running a query
    - Return type of a statement
- Programmer can iterate through the rows, and access all the data returned by the query.
    - Access data in each row by accessing each individual column

```
Statement myStatement = myConnection.createStatement();
 String query = "SELECT * FROM STUDENT";
 ResultSet myRelation =
    myStatement.executeQuery(query); while
    (myRelation.next())
            {
                    String myRecipeName = myRelation.getString("RecipeName");
                    int myIngredientId = myRelation.getInt("IngredientId");
            }
```

Prepared Statements and Result Set

- Specified with parameters for reuse and improved performance
  - Parameters are indicated with the symbol: ?
  - Parameter values are set with the SetX method
    - SetInt, SetString, ect
    - Need to specify which parameter is being assigned, by position in the query

```
String query = "SELECT * FROM CookBook WHERE IsOnline=? ";
PreparedStatement myPreparedStatement =
myConnection.prepareStatement(query);

myPreparedStatement.SetInt(1,0);
ResultSet myRelation = myStatement.executeQuery(query);
```

Callable Statements

- Used with stored procedures and functions, opposed to queries
  - Need to specify the name of the stored procedure or function
- Need to specify any parameters that are being passed in
  - Similar syntax to PreparedStatements
    - Parameters are indicated with the symbol: ?
    - Parameter values are set with the SetX method
      - SetInt, SetString, ect
      - Need to specify which parameter is being assigned, by position in the query

- Example calling a stored procedure with no parameters

```
CallableStatement myStoredProcedureCall =
myConnection.prepareCall("{Call GetRecipes()}");
        ResultSet myResults = myStoredProcedureCall.executeQuery();
```

Callable Statements

- Example calling a stored procedure with parameters

```
CallableStatement myStoredProcedureCall =
myConnection.prepareCall("{Call getStudentById(?)}");
myStoredProcedureCall.setString(1,1);
            ResultSet myResults =
myStoredProcedureCall.executeQuery()
;
```

Exception Handling

Why is exception handling in the DAL important?

- A Bad connection will throw an exception
  - database doesn’t exist
  - credentials are wrong
- A Bad query will throw an exception
  - database is incorrect
  - tables don’t exist
  - table schema has changed
  - spelling error
  - syntax error
- Disconnected concerns
  - The host hosting the database cannot current be reached
  - MySql is not currently running
- JDBC concerns
  - The JDBC version is not compatible with the Java or MySql version
  - JDBC and MySql have different authentication schemes configured: Common Configuration Error

Data Structures

- RESULT SETS STAY IN THE DAL!
- Abstraction principle
- Result Sets provide more data then the business logic layer needs
  - Meta Data
    - columns names, ordering, types
    - Information about keys and indexes
    - Programmers can add their own meta data to tables and columns
    - Version information
  - Too much data
    - Does the caller need the data, or to know it exists?
    - Does the caller need all the column data, or some?
    - Does the caller want the data processed in Java, because it's more efficient for the task at hand?
      - For example, combining data from various sources (API Calls)

Data Structures Continued

- The DAL provides the data in a format useful for the business layer
- Data structure that contains just what the business layer needs
  - Passed around between layers
  - Sent over the network
    - Define how its serialized
  - Sent to a browser or mobile device
    - Absolute minimal amount of data
      - XML or JSON or the like
- Abstraction
  - removes unneeded information
  - improves performance
- Common to need a return a collection of data structures to the business logic layer
  - Map
  - Array
  - List

Typical Elements of a DAL

- DatabaseMgr class
- Provides an interface for connecting to various databases
- DataProviderClasses
- Takes input from the business layer and returns a data structure to the caller
- Asks the DataMgr for a connection
- Retrieves a Result Set
- Maps the Result Set to the Collection of data structure to return to the business access layer
- Tends to be multiple providers, each “owning” a certain part of the database
- Tends to be based on product features
- Could be limited to only those methods that return a particular type of data structure
- Goal is for the DAL to be designed just like any other part of the product
- Readable and Maintainable

Node mysql

Node JS (A javascript runtime environment for the server)

NPM - Node Package manager

```
npm install mysql
```

Node mysql

Create the connection

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host     : ‘localhost',
  port     : ‘3306’,
  user     : ‘admin_user',
  password : 'admin1234',
  database : ‘SCHOOL’
});

connection.connect();
```

Node mysql

Query and Close

```
connection.query('SELECT 1 + 1 AS solution', func
tion (error, results, fields) {
  if (error) throw error;
  console.log('The solution is: ', results[0].sol
ution);
});


connection.end();
```

Node mysql

Procedure on DB

```
DELIMITER $$

CREATE PROCEDURE filterTodo(IN done BOOLEAN)
BEGIN
    SELECT * FROM todos WHERE completed = done;
END$$

DELIMITER ;
```

Node mysql

Call a Procedure

```
connection.connect((err) => {
  if (err) return console.error(err.message);

  let sql = `CALL filterTodo(?)`;

  connection.query(sql, [false], (error, results, fields) =>
    { if (error) return console.error(error.message);

    console.log(results);
  });

  // close the database connection
  connection.end();
connection.end()
});
;
```

https://www.mysqltutorial.org/mysql-nodejs/call-stored-procedures/

Project 4

We are going to create a command line application that connects to our database and runs our premade views, procedures, and functions we made last week.