Week 7 Lecture 1

Class	BSCCS2001
Created	@October 19, 2021 1:11 AM
Materials	
Module #	31
Type	Lecture
Week #	7

Application Design & Development: Architecture

Application Programs: Internet/Web or Mobile

- **Financial**
  - **Netbanking** → SBI, PNB, BoB, Canara, HDFC, ICICI
  - **Share Market** → ICICIDirect, Sharekhan, HDFCDirect
  - **Insurance & Investment** → LICI, PolicyBazaar, NSDL, NPS
  - **Payment Gateway** → PayTM, GPay, Bhim UPI, PhonePe
  - **e-Commerce** → Amazon, Flipkart, eBay, BigBazaar, BigBasket
- **Travel & Tourism**
  - **Travel Reservations** → IRCTC, Airlines, MakeMyTrip, Yatra
  - **Accommodation** → Booking, OYO, AirBnB, Fabhotels, Treebo
  - **Transportation** → Uber, Ola Cab, Mega Cab, Meru Cab
  - **Navigation** → Google Maps, MapQuest, Apple Maps
  - **Food & Delivery** → Zomato, Swiggy, UberEats, Dunzo
- **Communication**
  - **Live Interaction** → Zoom, Google Meet, Teams, Webex, Skype
  - **Intermittent Interaction** → WhatsApp, Telegram, Signal, Skype
  - **Mail** → Gmail, Yahoo, Hotmail, Rediffmail, Enterprise Mail

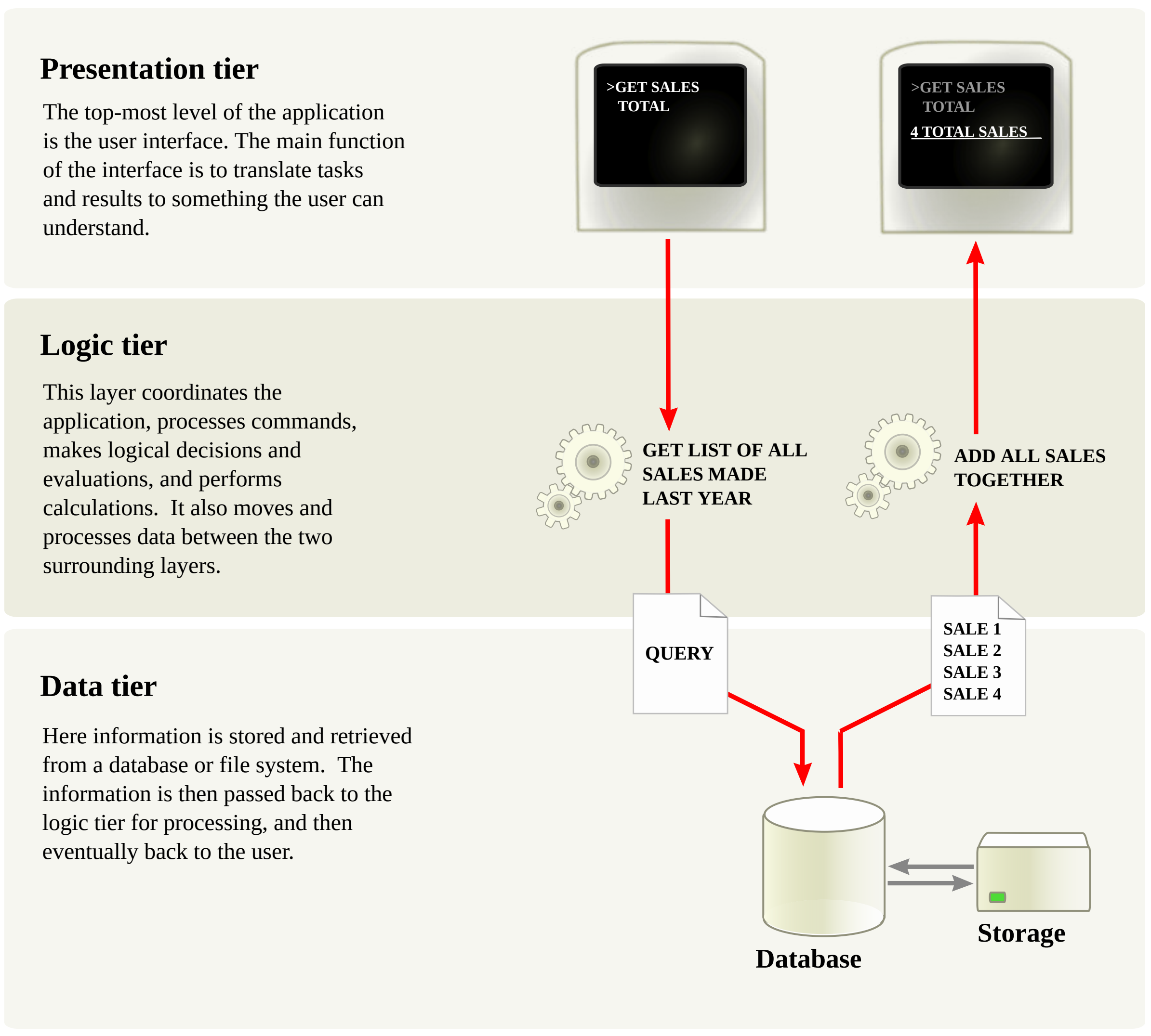
    - **Social Media** → Facebook, Instagram, Twitter, YouTube
- **Knowledge Discovery**
    - **Static** → Google, Yahoo, Bing, Wikipedia, Encyclopedia.com
    - **Q&A** → Quora, ASKfm, Yahoo Answers, Reddit, Digg
- **Sports**
    - **Cricket** → Cricbuzz, CricViz, Cricket-21, Cricket Exchange
    - **Tennis** → ATP, ITF, SwingVision, TennisPAL, Tennis Clash
- **Software Engineering**
    - **Issue Tracking** → Jira, Bugzilla, GitHub, GitLab
    - **VCS** → GitHub, GitLab, Bitbucket, SourceForge
    - **Online IDE** → OnlineGDB, Codechef, Ideone
- **Library**
    - **Digital Library** → National Digital Library of India
    - **Archives** → Internet Archive, arXiv, Nextpoint
- **Education**
    - **eLearning** → BYJU's, IGNOU, NIIT, Edukart
    - **MOOCs** → SWAYAM, edX, Coursera, Udemy
- **Document Processing**
    - **Editing** → Overleaf, Google Docs, Spreadsheet
    - **Website, Blog** → Google sites, WordPress, Weebly
- **Health**
    - **Telemedicine** → MDLIVE, Doctor on Demand
    - **National** → Aarogya Setu, CoWin, NACO App
- **Organizational ERP** (Intranet)
    - **Institutions** → Students, Faculty, Course
    - **Hospital** → Patient, Doctor, OPD, IPD, Pharmacy
    - **Manufacturing** → Suppliers, Inventory, Customers
    - **Bank** → Customers, Accounts, Lockers, Deposits
    - **Courier** → Customers, Parcels, Delivery Agents

Characteristics of Application Program

- **Diversity** → These applications widely differ in their
    - *Domain, functionality, user base, response time, scale, daily hit*, and many more
- **Unity** → Yet, these have a lot in common
    - Most use an RDBMS like Oracle, DB2 MySQL, PostgreSQL, etc. for managing data
    - Applications are functionality split into the *frontend layer, middle layer, backend layer*
        - **Frontend or Presentation Layer/Tier**
            - Interacts with the users → Display/View, Input/Output
            - *Choose item, Add to cart, Checkout, Pay, Track order*
            - Interfaces may be *Browser based, Mobile app or custom*
        - **Middle or Application/Business Logic Layer/Tier**
            - Implements the functionality of the application → Links the frontend & backend
            - *Authentication, Search/Browse logic, Pricing, Cart management Payment handling (gateway), Order management (mail, SMS, internal actions), Delivery management*

        - Support functionality based on frontend interface
    - **Backend or Data Access Layer/Tier**
        - Manages persistent data, large volume, efficient access, security
        - *User, Cart, Inventory, Order, Vendor DBs*

Characteristics of Application Programs: Architecture

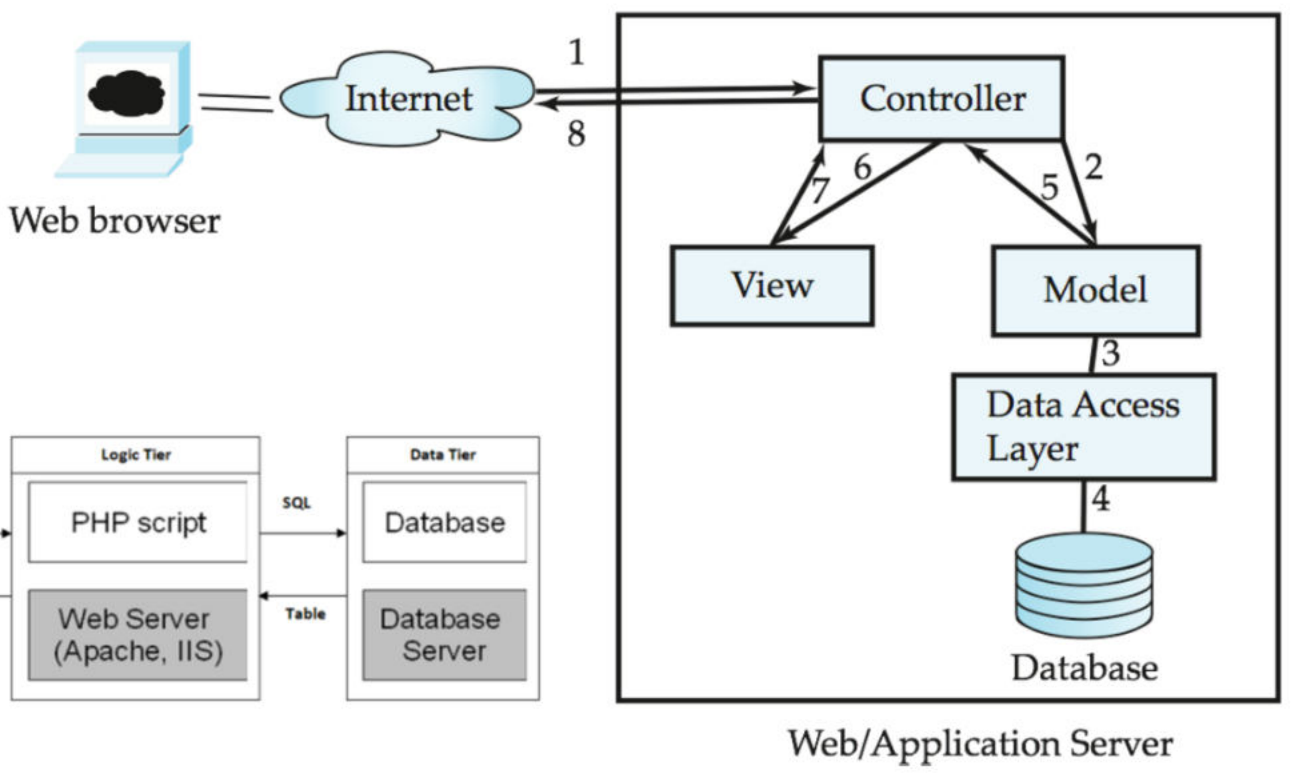

Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
Database
Storage

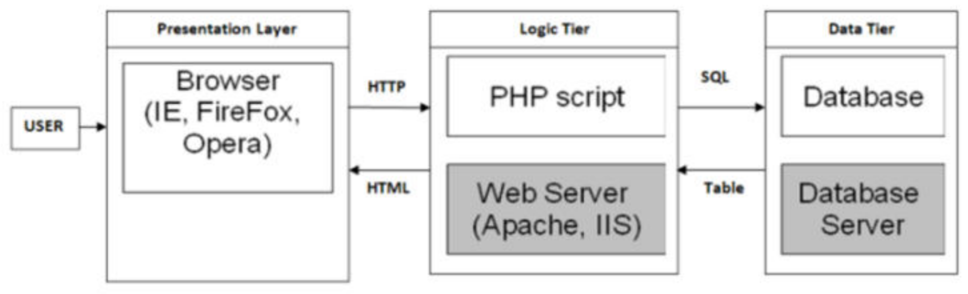
Source: https://en.wikipedia.org/wiki/Multitier_architecture

Application Architectures: Layers

- **Presentation Layer/Tier**
    - **Model-View-Controller (MVC)** architecture
        - **Model** → Business logic
        - **View** → Presentation of data, depends on the display service
        - **Controller** → Receive events, execute actions, and return a view to the user
- **Business Layer/Tier**
    - Provides the high-level view of data and actions on the data
        - Often using an object data model
    - Hides the details of data storage schema
- **Data Access Layer/Tier**
    - Interfaces between business logic layer and the underlying DB
    - Provides mapping from object model of business layer to the relational model of the DB

Application Architecture: MVC





Application Architecture: User Interface

- Web browsers have become the de-facto standard user interface to the DBs
  - Enable large number of users to access DBs from anywhere
  - Avoid the need for downloading/installing specialized code, while providing a good Graphical User Interface
    - JavaScript, Flash (dead apparently) and other scripting language runs in the browser, but are downloaded transparently
  - **Examples** → Banks, Airlines and Rental Car reservations, university course registration and grading and so on
- Use in Mobile Devices are getting popular
  - Mobile apps or Browser in Mobile
  - These are similar in architecture and workflow with the web, but have significant differences with their smaller (but wide range of) form factor, and extremely low resources

Application Architecture: Business Logic Layer

- Provides abstractions of entities
  - For example → students, instructors, courses, etc
- Enforces **business rules** for carrying out actions
  - For example → student can enroll in a class only if she has completed all the prerequisites, and has paid her tuition fee
- Support **workflows** which define how a task involving multiple participants is to be carried out
  - For example → How to process an application by a student applying to a University
  - Sequence of steps to carry out the task
  - Error handling
    - For example → What to do if the recommendation letters not received on time

Application Architecture: Object-Relational Mapping

- Allows application code to be written on top of object-oriented data model, while storing data in a traditional relational DB
  - Alternative → implement object-oriented or object-relational DB to store object model
    - It has not been commercially successful
- Schema designer has to provide a mapping between object data and relational schema
  - For example → Java class `Student` mapped to a relation *student*, with corresponding mapping of attributes
  - An object can map to multiple tuples in multiple relations
- Application opens a session, which connects to the DB

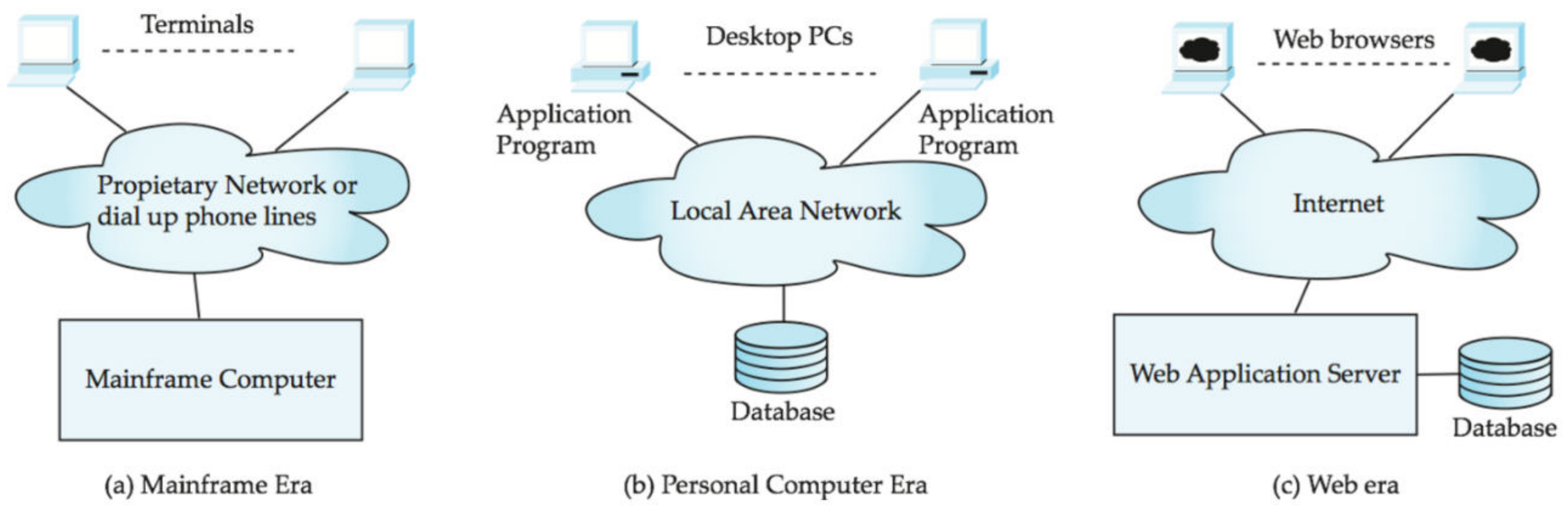
- Objects can be created and saved to the DB using `session.save(object)`
    - Mapping used to create appropriate tuples in the DB
- Query can be run to retrieve objects satisfying specified predicates

Architecture Classification

- DB architecture uses programming languages to design a particular type of software for business or organizations
- DB architecture focuses on the design, development, implementation and maintenance of computer programs that store and organize information for businesses, agencies and institutions
- A DB architect develops and implements software to meet the needs of users
- The design of a DBMS depends on its architecture
    - It can be
        - Centralized
        - Decentralized
        - Hierarchical
- The architecture of a DBMS can be seen as either single tier or multi tier:
    - 1-tier architecture
    - 2-tier architecture
    - 3-tier architecture
    - n-tier architecture

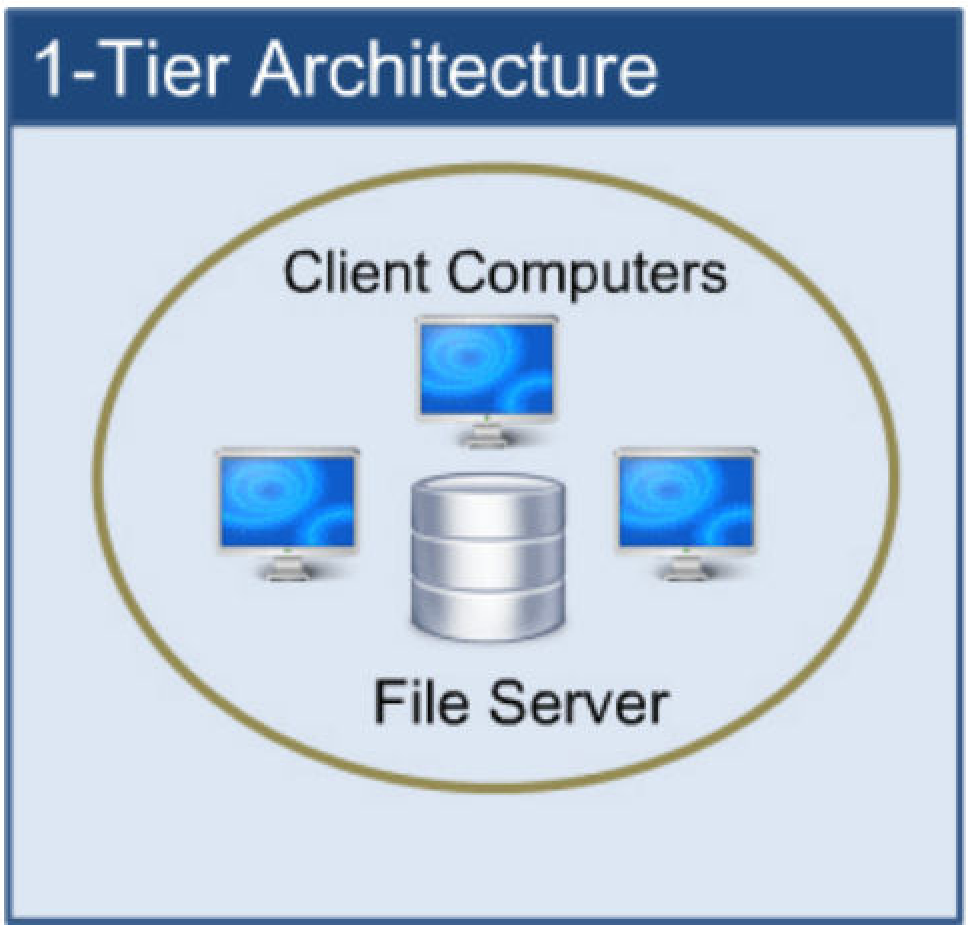
Architecture Evolution

- Three distinct eras of application architecture
    - Mainframe (1960s and 1970s)
    - Personal Computer era (1980s)
    - Web/Mobile era (1990s onwards)



1-tier Architecture

- One-tier architecture involves putting all of the required components for a software application or technology on a single server or platform

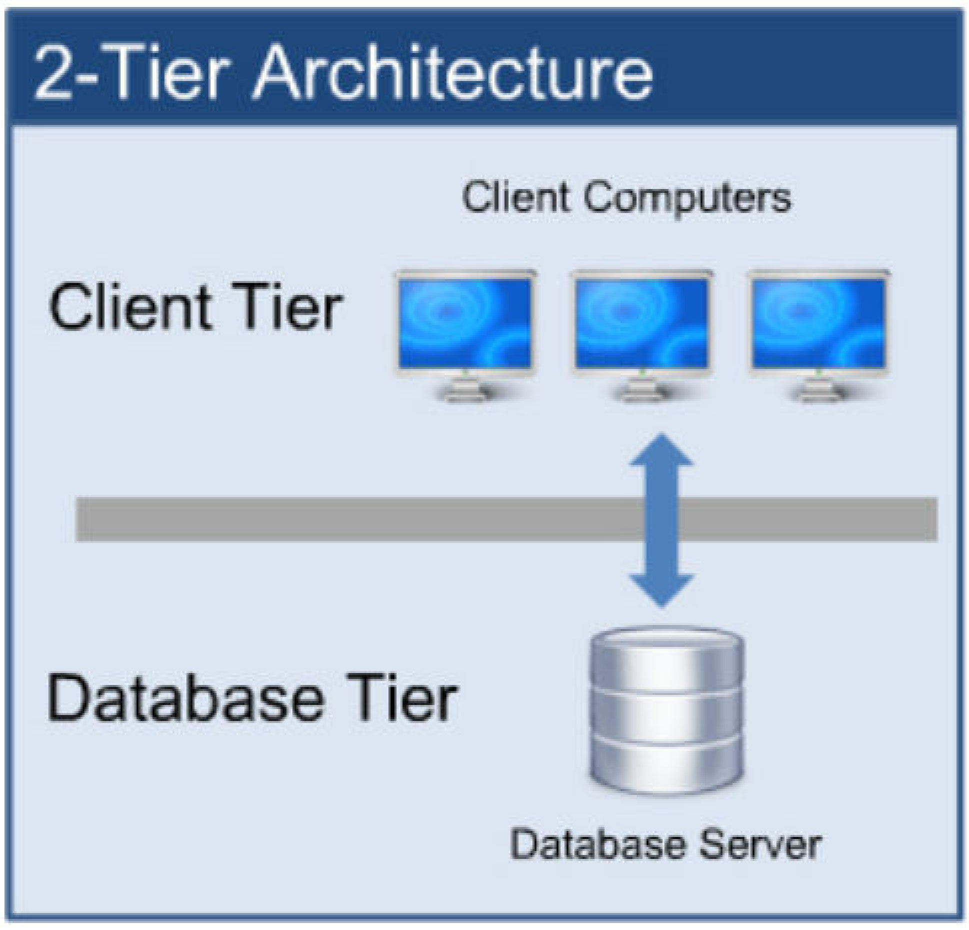


- Basically, a one-tier architecture keeps all of the elements of an application, including the interface, Middleware and back-end data, in one place
- Developers see these types of systems as the simplest and most direct way

Source: https://medium.com/oceanize-geeks/concepts-of-database-architecture-dfdc558a93e4

2-tier Architecture

- The two-tier architecture is based on Client Server architecture
- It is like client server application



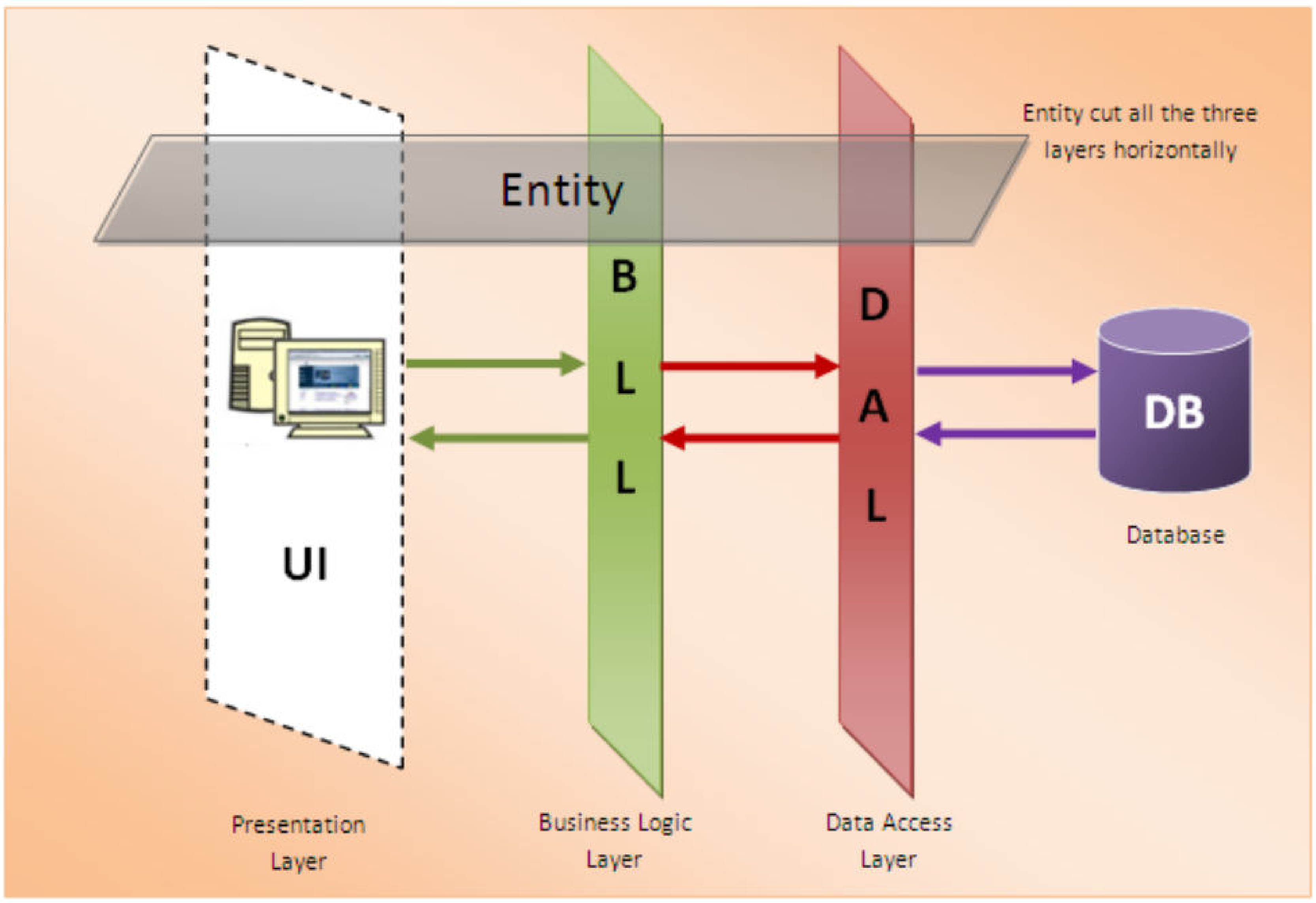
- The direct communication takes place between client and server
- There is no intermediate between the client and the server

Source: https://medium.com/oceanize-geeks/concepts-of-database-architecture-dfdc558a93e4

3-tier Architecture

- A 3-tier architecture separates it tiers - ***Presentation***, ***Logic*** and ***Data Access*** from each other based on the complexity of the users and how they use the data present in the DB

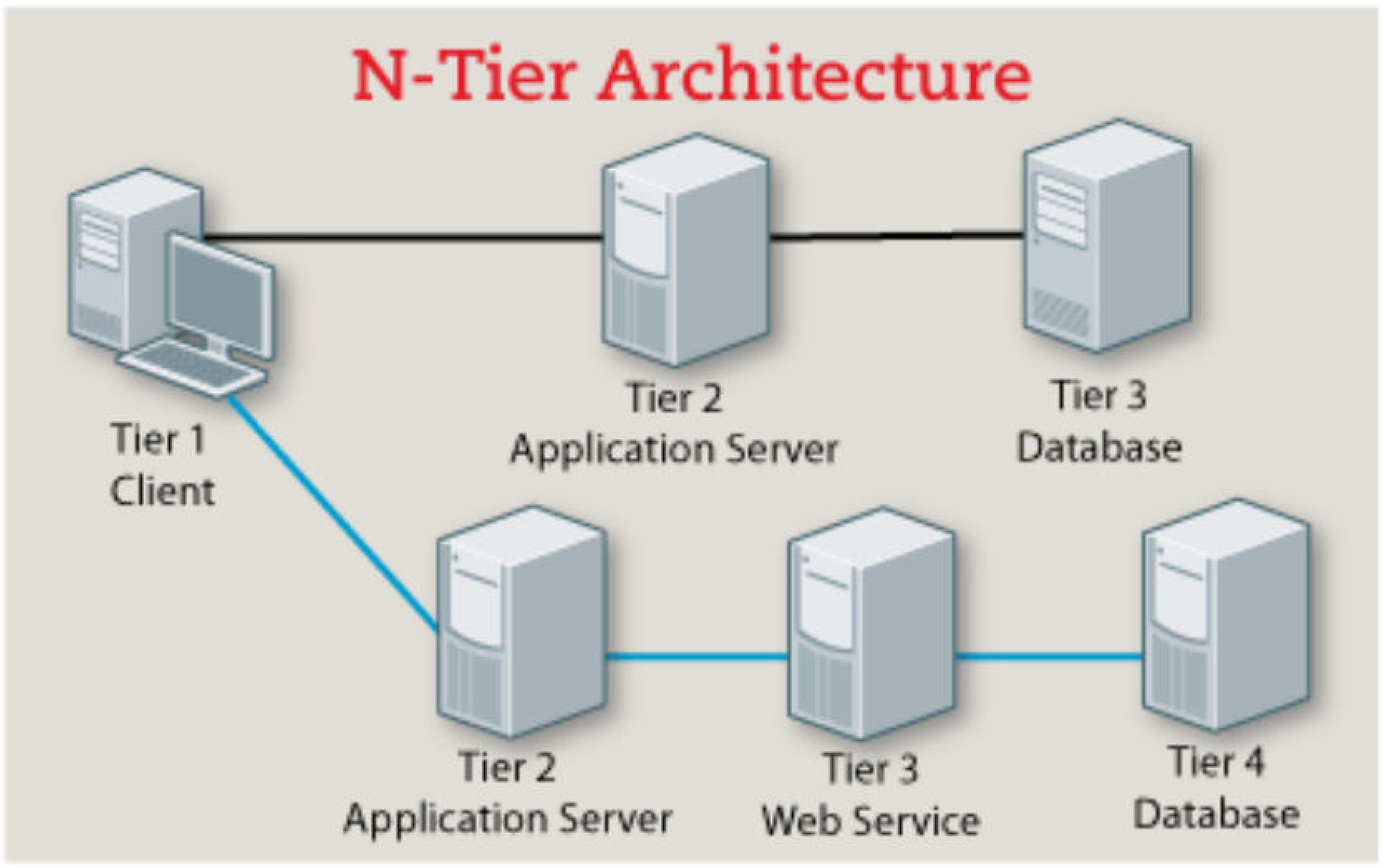
- It is the most widely used architecture to design a DBMS



Source: https://medium.com/oceanize-geeks/concepts-of-database-architecture-dfdc558a93e4

n-tier Architecture

- An n-tier architecture distributes different components of the 3-tiers between different servers and adds interface tiers for interactions and workload balancing



Source: https://medium.com/oceanize-geeks/concepts-of-database-architecture-dfdc558a93e4

Sample Applications in Multiple Tiers

Application	Presentation	Logic	Data	Functionality
Web Mail	• Login • Mail List View • Inbox • Sent Items • Outbox • Trash • Mail Composer • Filters	• User Authentication • Connection to Mail Server (SMTP, POP, IMAP) • Encryption/Decryption	• Mail Users • Address Book • Mail items	• Send/Receive Mails • Manage Address Book
Net Banking	• Login • Account View • Add/Delete Account • Add/Delete Beneficiary • Fund Transfer	• User Authentication • Beneficiary Authentication • Transaction Validation • Connection to Banks/Gateways • Encryption/Decryption	• Account Holders • Beneficiaries • Accounts • Debit/Credit Transactions	• Check Balance and Transactions • Transfer Funds
Timetable	• Login • Add/Delete Courses, Teachers, Rooms, Slots • Assignments: • Teachers → Course • Allocations • Course → Room, Slots • Views	• User Authentication • Timetable Assignment Logic • Encryption/Decryption	• Courses • Teachers • Rooms • Slots • Assignments • Allocations	• Manage timetable for multiple courses taken by multiple teachers

Week 7 Lecture 2

Class	BSCCS2001
Created	@October 19, 2021 5:12 PM
Materials	
Module #	32
Type	Lecture
Week #	7

Application Design & Development: Web Applications

Web Fundamentals

The World Wide Web (WWW)

- The web is a distributed information system based on HyperText
- Most web documents are HyperText documents formatted via the HyperText Markup Language (HTML)
- HTML documents contain
    - Text along with font specifications, and other formatting instructions
    - HyperText links to other documents, which can be associated with regions of the text
    - Forms, enabling users to enter data which can then be sent back to the Web server

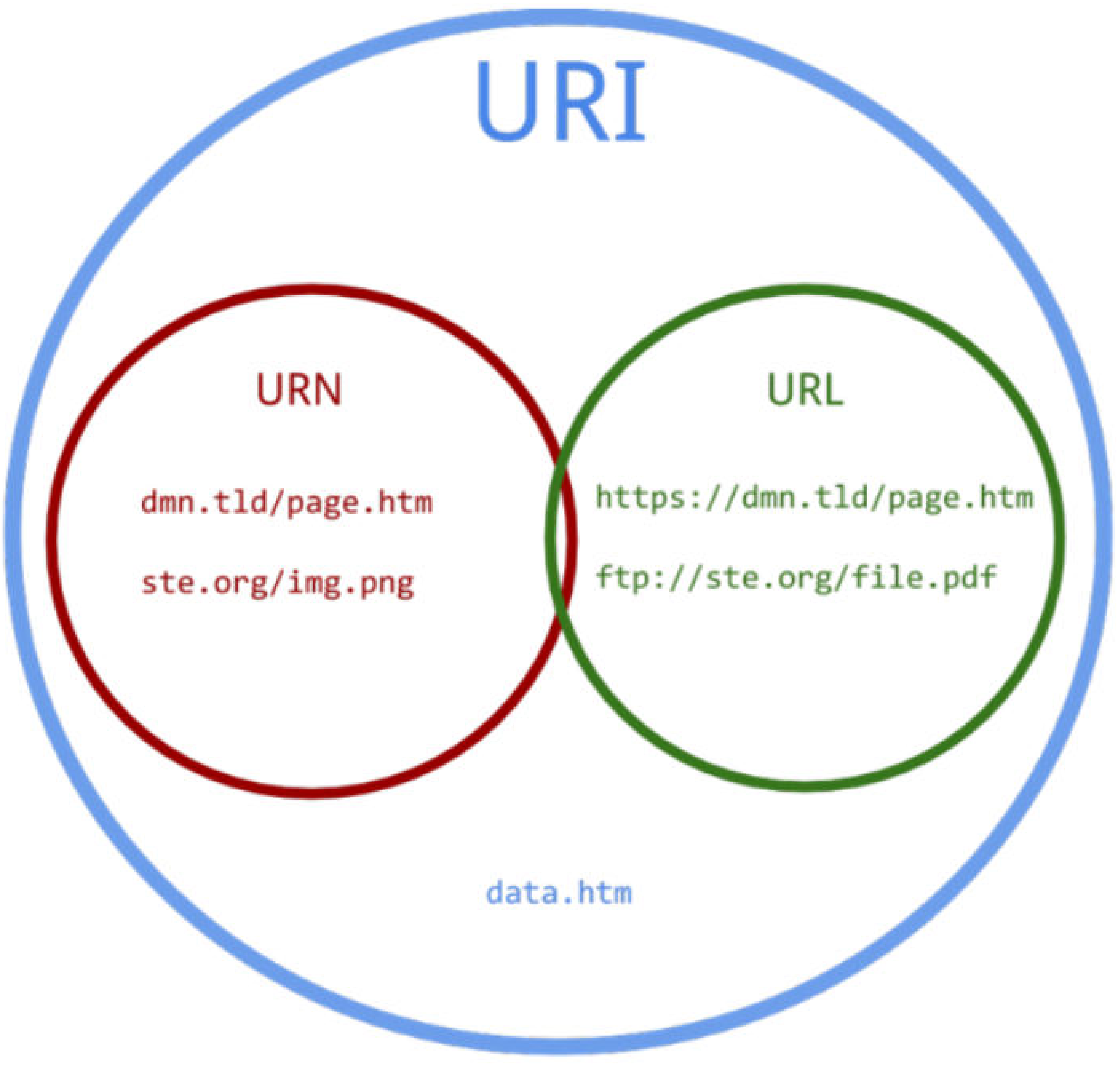
Uniform Resource Locators

- On the Web, functionality of pointers is provided by Uniform Resource Locators (URLs)
- URL example: https://www.acm.org/sigmod
    - The first part indicates how the document is to be accessed (protocol)
        - "http" indicates that the document is to be accessed using the HyperText Transfer Protocol
    - The second part gives the unique name of a machine on the Internet
    - The rest of the URL identifies the document within the machine
- The local identification can be:

- The path name of a file on the machine:
    - A file at `C:\WINDOWS\media\Alarm01.wav` of the local machine can be accessed as:
        - `file:///C:/WINDOWS/media/Alarm01.wav`
        - `file:///localhost/c:/WINDOWS/media/Alarm01.wav`
- An identifier (path name) of a program, plus arguments to be passed to the program:
    - Searching google.com with 'silberschatz' has the uri:
        - `http://www.google.com/search?q=silberschatz`

URI, URL and URN

- Uniform Resource Identifier (URI)
- Uniform Resource Locator (URL)
- Uniform Resource Name (URN)
- Relationships
    - URIs can be classified as locators (URLs), or as names (URNs), or as both
    - URN functions like a person's name
    - URL resembles that person's street address
    - URN defines an item's identity, while the URL provides a method for finding it



HTML and HTTP

- HTML provides formatting, hypertext link, and image display features
    - including tables, stylesheets (to alter default formatting), etc
- HTML also provides input features

    - Select from a set of options
        - Pop-up menus, radio buttons, check lists
    - Enter values
        - Text boxes
    - Filled in input sent back to the server, to be acted upon by an executable at the server
- HyperText Transfer Protocol (HTTP) used to communication with the Web Server

HTTP and Sessions

- The HTTP protocol is **connectionless**
    - That is, once the server replies to a request, the server closes the connection with the client, and forgets all about the request
    - In contrast, Unix logins, and JDBC/ODBC connections stay connected until the client disconnects
        - Retaining user authentication and other information
    - **Motivation** → Reduces the load on the server
        - Operating Systems have tight limits on the number of open connections on a machine
- Information services need session information
    - For example, user authentication should be done only once per session
- Solution → use a **cookie**

Sessions and Cookies

- A ***cookie*** is a small piece of text containing identifying information
    - Sent by the server to the browser
        - Sent on first interaction to identify the session
    - Sent by the browser to the server that created the cookie on further interactions
        - part of HTTP
    - Server saves the information about cookies it issued, and can use it when serving a request
        - For example, authentication information and user preferences
- Cookies can be stored permanently or for a limited time

Web Browser

- A web browser is an application software for accessing the World Wide Web (WWW)
- A web browser's job is to fetch content from the Web and display it on the user's device
- This process begins when the user inputs the URL into the browser address bar, starting with either `http://` or `https://`
- Once a web page has been retrieved, the *rendering engine* displays it on the user's device
    - A browser or the rendering engine is a core software component for a web browser
    - The primary job of a browser engine is to transform HTML documents and other resources of a web page into an interactive visual representation on a user's device
    - This includes image and video formats supported by the browser
- Web pages usually contains hyperlinks to other pages and resources
    - Each link contains a URL, and when it is clicked or tapped, the browser navigates to the new resource
- Web browsers are used on a range of devices, including desktops, laptops, tablets and smartphones
    - In 2020, an estimated 4.9B people used a browser
    - The most used browser is Google Chrome, with 64% global market share on all devices, followed by Safari with 19%

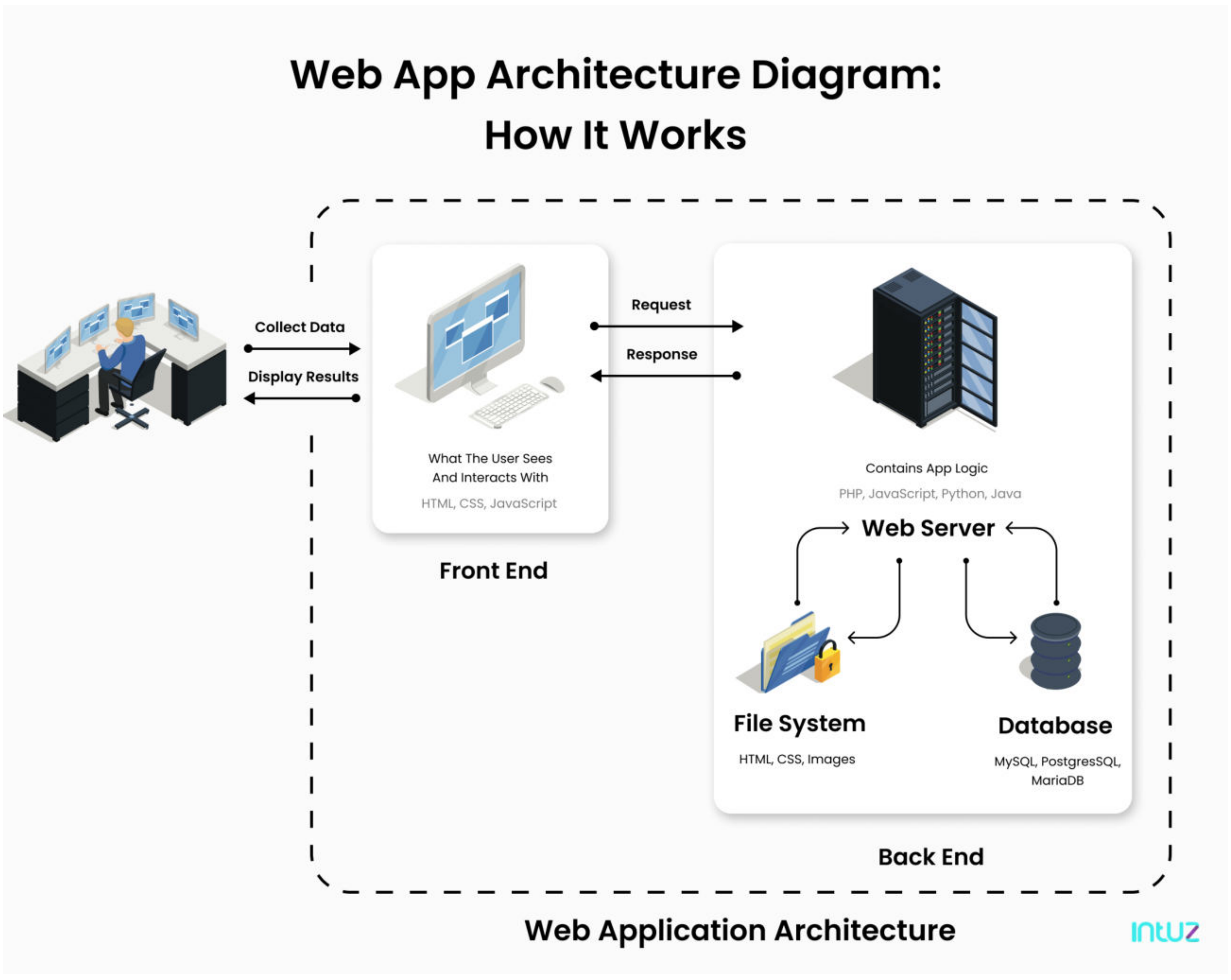
Web Servers

- A web server is a software and underlying hardware that accepts requests via HTTP or its secure variant HTTPS
- A web browser or crawler, requests for a specific resource using HTTP, and the server responds with the content of that resource or an error message
- The server can also accept and store resources sent from the user agent
- The document name in a URL may identify an executable program, that when run, generated the HTML document
    - When an HTTP server receives a request for such a document, it executes the program and sends back the HTML document that is generated
    - The Web client can pass extra arguments with the name of the document
- To install a new service on the Web, one simply needs to create and install an executable that provides the said service
    - The Web browser provides a GUI to the information service
- Common Gateway Interface (CGI) → a standard interface between web and application server

Web Services

- Allow data on Web to be accessed using remote procedure call mechanism
- Two approaches are widely used
    - Representational State Transfer (REST) → allows the use of standard HTTP request to a URL to execute a request and return data
        - Returned data is encoded either in XML or in JSON (JavaScript Object Notation)
    - Big Web Services
        - uses XML representation for sending request data, as well as for returning results
        - Standard protocol layer built on top of HTTP

Web Architecture



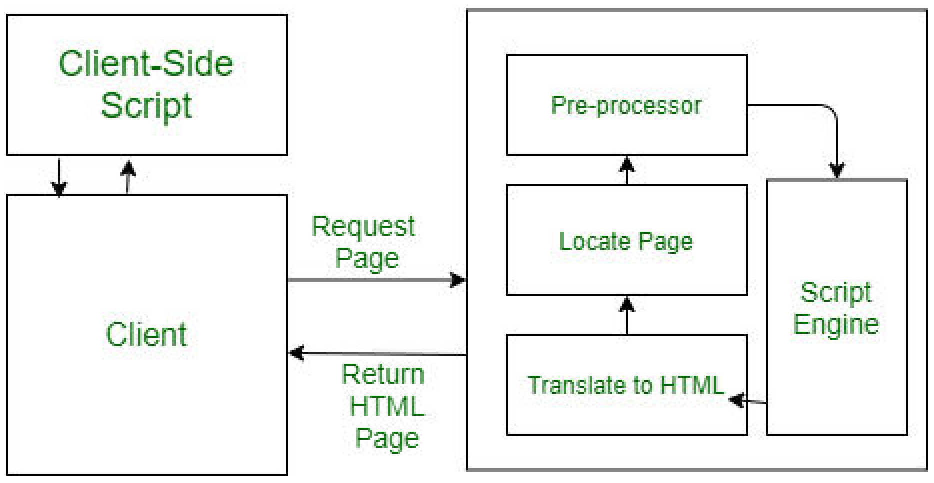
Source: https://www.intuz.com/guide-on-web-app-architecture

Scripting for Web Applications

- A script is a list of (text) commands that are enabled in a web-page or in the server
- They are *interpreted and executed* by a certain program or scripting engine
- Scripts may be written for a variety of purposes such as for automating process on a local-computer or to generate web pages
- The programming languages in which scripts are written are called **scripting language**
- Common scripting languages are VBScript, JavaScript, ASP, PHP, PERL, JSP, etc.

Scripting of 2 types:

- **Client Side** → Client-side scripting is responsible for interaction within a web page
  - The client-side scripts are firstly downloaded at the client-end and then interpreted and executed by the browser
- **Server Side** → Server-side scripting is responsible for the completion or carrying out a task at the server-end and then sending the result to the client-end



Source: https://www.geeksforgeeks.org/web-scripting-and-its-types/

Client Side Scripting

- Browsers can fetch certain scripts (client-side scripts) or programs along with documents, and execute them in "safe mode" at the client side
  - JavaScript
  - Macromedia Flash and Shockwave for animation/games (isn't Flash dead?)
  - VRML
  - Applets (I guess these are dead too?)
- Client-side scripts/programs allow documents to be active
  - For example, animation by executing programs at the local site
  - For example, ensure that values entered by the users satisfy some correctness checks
  - Permit flexible interaction with the user
    - Executing programs at the client site speeds up interaction by avoiding may round trips to the server

Client Side Scripting: Security

- Security mechanisms needed to ensure that malicious scripts do not cause damage to the client machine
  - Easy for limited capability scripting languages, harder for general purpose programming languages like Java
- For example, Java's security system ensures that the Java applet code does not make any system calls directly
  - Disallows dangerous actions such as file writes
  - Notifies the user about potentially dangerous actions and allows the option to abort the program or to continue execution

JavaScript

- JavaScript very widely used
    - Forms basis of new generation of Web applications (called Web 2.0 applications) offering rich user interfaces
- JavaScript functions can
    - Check input for validity
    - Modify the displayed web page, by altering the underlying ***Document Object Model (DOM)*** tree representation of the displayed HTML text
    - Communicate with a web server to fetch data and modify the current page using fetched data, without needing to reload/refresh the page
        - Forms basis of AJAX technology used widely in Web 2.0 applications
        - For example, on selecting a country in the drop-down menu, the list of states in the country is automatically populated in a linked drop-down menu

JavaScript: Example

```
<html>
<head>
  <script type="text/javascript">
  function validate() {
    var credits = document.getElementById("credits").value;
    if (isNaN(credits) || credits <= 0 || credits >= 16) {
      alert("Credits must be a number greater than 0 and less than 16");
      return false;
    }
  }
</script>
</head>
<body>
  <form action="createCourse" onsubmit="return validate()">
    Title: <input type="text" id="title" size="20"><br />
    Credits: <input type="text" id="credits" size="2"><br />
    <Input type="submit" value="Submit">
  </form>
</body>
</html>
```

Server Side scripting

- Server-side scripting simplifies the task of connecting a DB to the Web
    - Define an HTML document with embedded executable code/SQL queries
    - Input values from HTML forms can be used directly in the embedded code/SQL queries
    - When the document is requested, the Web server executes the embedded code/SQL queries to generate the actual HTML document
- Numerous server-side scripting languages
    - JSP, PHP
    - General purpose scripting languages like VBScript, Perl, Python

Servlets

- Java Servlet specification defines an API for communication between the Web/application server and application program running in the server
    - For example, methods to get parameter values from Web forms, and to send HTML text back to the client
- Application program (also called a servlet) is loaded into the server
    - Each request spawns a new thread in the server
        - Thread is closed once the request is serviced

Servlets: Example

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class PersonQueryServlet extends HttpServlet {
  public void doGet(HttpServlet request, HttpServletResponse response)
                throws ServletException IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<head><title>Query Result</title></head>");
    out.println("<body>");

    String persontype = request.getParameter(”persontype”);
    String number = request.getParameter(”name”);

    if (persontype.equals(”student”)) {
      · · · code to find students with the specified name · · ·
      · · · using JDBC to communicate with the database · · ·

      out.println(”<table BORDER COLS=3>”);
      out.println(”<tr> <td>ID</td><td>Name:
      </td>” + ”<td> Department </td> </tr>”);

      for(· · · each result · · · ) {
      · · · retrieve ID, name and dept name
      · · · into variables ID, name and deptname

      · · · out.println(”<tr> <td>” + ID + ”</td>” + ”<td>” + name
      + ”</td>” + ”<td>” + deptname + ”</td></tr>”);
      };

      out.println(”</table>”);
    }
    else {
    · · · as above, but for instructors · · ·
    }
    out.println("</body>");
    out.close();
  }
}
```

Servlet: Sessions

- Servlet API supports handling of sessions
    - Sets a cookie on first interaction with browser, and uses it to identify session on further interactions
- To check if session is already active:
    - `if (request.getSession(false) == true)`
        - .. then existing session
        - else .. redirect to authentication page
    - authentication page
        - check login/password
        - `request.getSession(true)`: creates new session
- Store/retrieve attribute value pairs for a particular session
    - `session.getAttribute("userid", userid)`
    - `session.getAttribute("userid")`

Servlet: Support

- Servlets run inside application server such as
    - Apache Tomcat, Glassfish, JBoss
    - BEA Weblogic, IBM WebSphere and Oracle Application Servers
- Application servers support
    - deployment and monitoring of servlets
    - Java 2 Enterprise Edition (J2EE) platform supporting objects, parallel processing across multiple application servers, etc

Java Server Pages (JSP)

- A JSP page with embedded Java code

```
<html>
<head>
  <title>Hello</title>
</head>
<body>
  <% if (request.getParameter("name") == null)
  { out.println("Hello World"); }
  else { out.println("Hello, " + request.getParameter("name")); }
  %>
</body>
</html>
```

- JSP is compiled into Java + Servlets
- JSP allows new tags to be defined, in tag libraries
    - Such tags are likely library functions, can be used to build rich user interfaces such as paginated display of large datasets

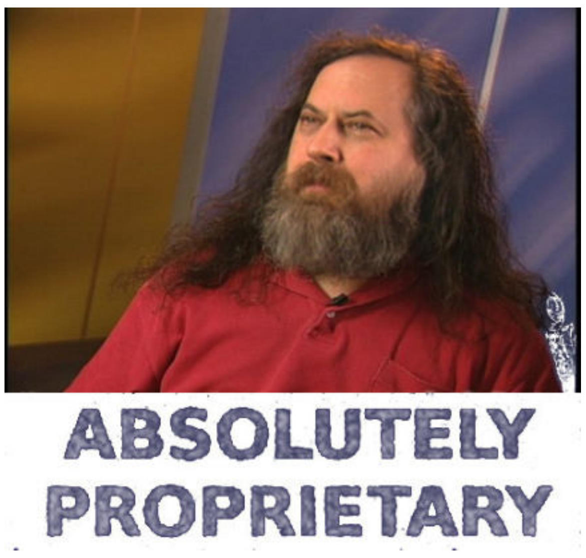
PHP (Oh Lord, not this)

- PHP is widely used for Web server scripting
- Extensive libraries including for DB access using ODBC

```
<html>
<head>
  <title>Hello</title>
</head>
<body>
  <? php if (!isset($_REQUEST['name']))
  { echo "Hello World"; }
  else { echo "Hello, " + $_REQUEST['name']; }
  ?>
</body>
</html>
```

USP (Unique Selling Proposition) of JSP

- **JSP vs Active Server Pages (ASP)**
    - ASP is a similar technology from Microsoft and is proprietary (uses VB)



    - JSP is platform independent and portable
- **JSP vs Pure Servlets**
    - JSP is a servlet but it is more convenient to write and to modify regular HTML than to have a million `println` statements that generate the HTML
    - The Web page design experts can build the HTML, leaving places for the servlet programmers to insert the dynamic content
- **JSP vs JavaScript**
    - JavaScript can generate HTML dynamically on the client
    - "Client side" → JavaScript code is executed by the browser after the web server sends the HTTP response
        - With the exception of cookies, HTTP and form submission data is not available to JavaScript

    - "Server side" → Java Server Pages are executed by the web server before the web server sends the HTTP response
        - It can access server-side resources like DBs, catalogs
- **JSP vs Static HTML**
    - Regular HTML cannot contain dynamic information

Week 7 Lecture 3

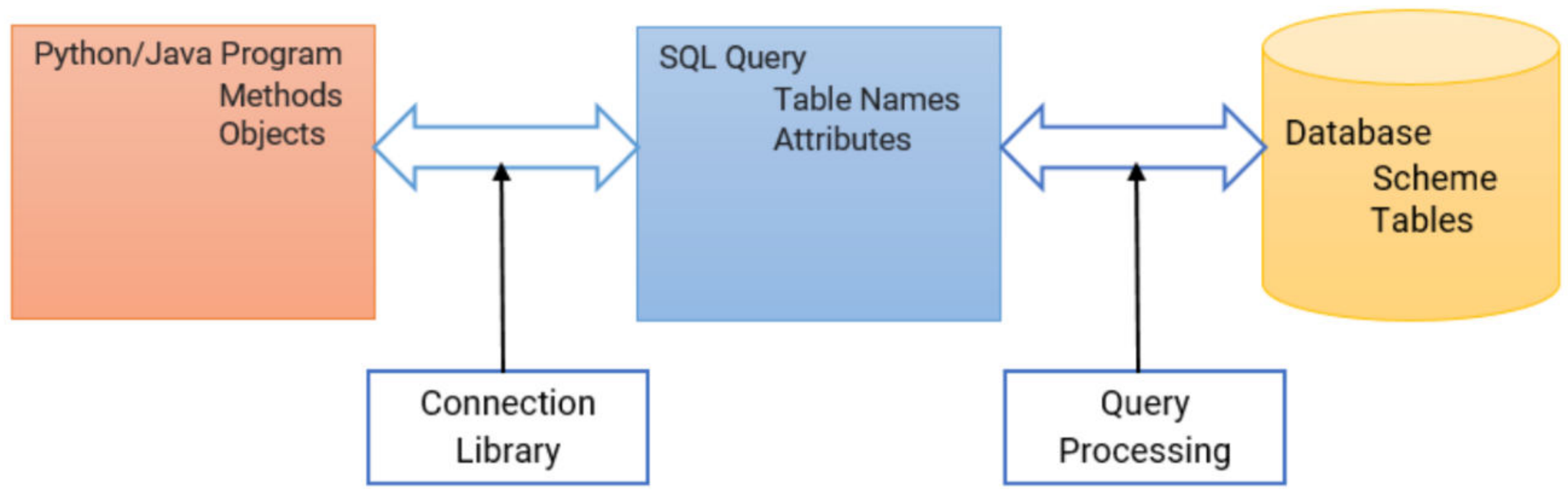
Class	BSCCS2001
Created	@October 20, 2021 1:32 AM
Materials	
Module #	33
Type	Lecture
Week #	7

Application Design & Development: SQL and Native Language

Working with SQL and Native Language

- Applications use ***Application Programming Interface (API)*** to interact with the DB server
- Applications make calls to
  - Connect with the DB server
  - Send SQL commands to the DB server
  - Fetch tuples of result one-by-one into the program variables
- Frameworks
  - **Connectionist**
    - **Open DB Connectivity (ODBC)** → works with C, C++, C#, Visual Basic and Python
      - Other data APIs include
        - OLEDB
        - ADO.NET
      - **Java DB Connectivity (JDBC)** → works with Java
  - **Embedding**
    - Embedded SQL works with C, C++, C#, Java, COBOL, FORTRAN and Pascal

Native Language $\Longleftarrow\Longrightarrow$ Query Language: Connectionist



ODBC

- **Open DB Connectivity (ODBC)** is a standard API for accessing DBMS
- It aimed to be independent of DB systems and OS
- An application written using ODBC can be ported to other platforms, both on the client and the server side, with few changes to the data access code
- ODBC is
  - A standard for application program to communicate with a DB server
  - An API to
    - Open a connection with a DB
    - Send queries and updates
    - Get back the results
- Applications such as GUI, Spreadsheets, etc. can use ODBC
- ODBC was originally developed by Microsoft and Simba Technologies during the early 1990s, and became the basis for the Call Level Interface (CLI) standardized by SQL Access Group in the Unix and mainframe field

ODBC: Python Example

- The code uses a data source names "SQLS" from the `odbc.ini` file to connect and issue a query
- It creates a table, inserts data using literal and parameterized statements and fetches the data

```
import pyodbc

conn = pyodbc.connect('DSN=SQLS;UID=test01;PWD=test01')
cursor = conn.cursor()

cursor.execute("create table rvtest (col1 int, col2 float, col3 varchar(10))")
cursor.execute("insert into rvtest values (1, 10.0, \"ABC\")")
cursor.execute("select * from rvtest")

while True:
  row = cursor.fetchone()

  if not row:
    break

  print(row)

cursor.execute("delete from rvtest")
cursor.execute("insert into rvtest values (?, ?, ?)", 2, 20.0, 'XYZ')
cursor.execute("select * from rvtest")

while True:
  row = cursor.fetchone()

  if not row:
    break

  print(row)
```

Source: https://dzone.com/articles/tutorial-connecting-to-odbc-data-sources-with-pyth

JDBC

- **Java DB Connectivity (JDBC)** is an API for the programming language Java, which defines how a client may access a DB
- It is a Java-based data access technology used for Java DB connectivity
- JDBC supports a variety of features for querying and updating data, and for retrieving query results; metadata retrieval, such as querying about relations present in the DB and the names and the types of relation attributes
- Model for communicating with the DB:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors
- JDBC, original released by Sun Microsystems released as part of Java Development Kit (JDK) 1.1 on in 1997, is part of Java Standard Edition platform, from Oracle corporation

JDBC: Example

- We show a simple example here to connect to SQL server from Java using JDBC to execute DB commands
- In the example, the sample code makes a connection to the sample DB
- Then, using an SQL statement with the `SQLServerStatement` object, it runs the SQL statement and places the data that it returns into a `SQLServerResultSet` object
- Next, the sample code calls the custom `displayRow` method to iterate through the rows of data that are in the result set, and uses the `getString` method to display some of the data
- Complete example can be found at: *https://docs.microsoft.com/en-us/sql/connect/jdbc/retrieving-result-set-data-sample?view=sql-server-ver15*

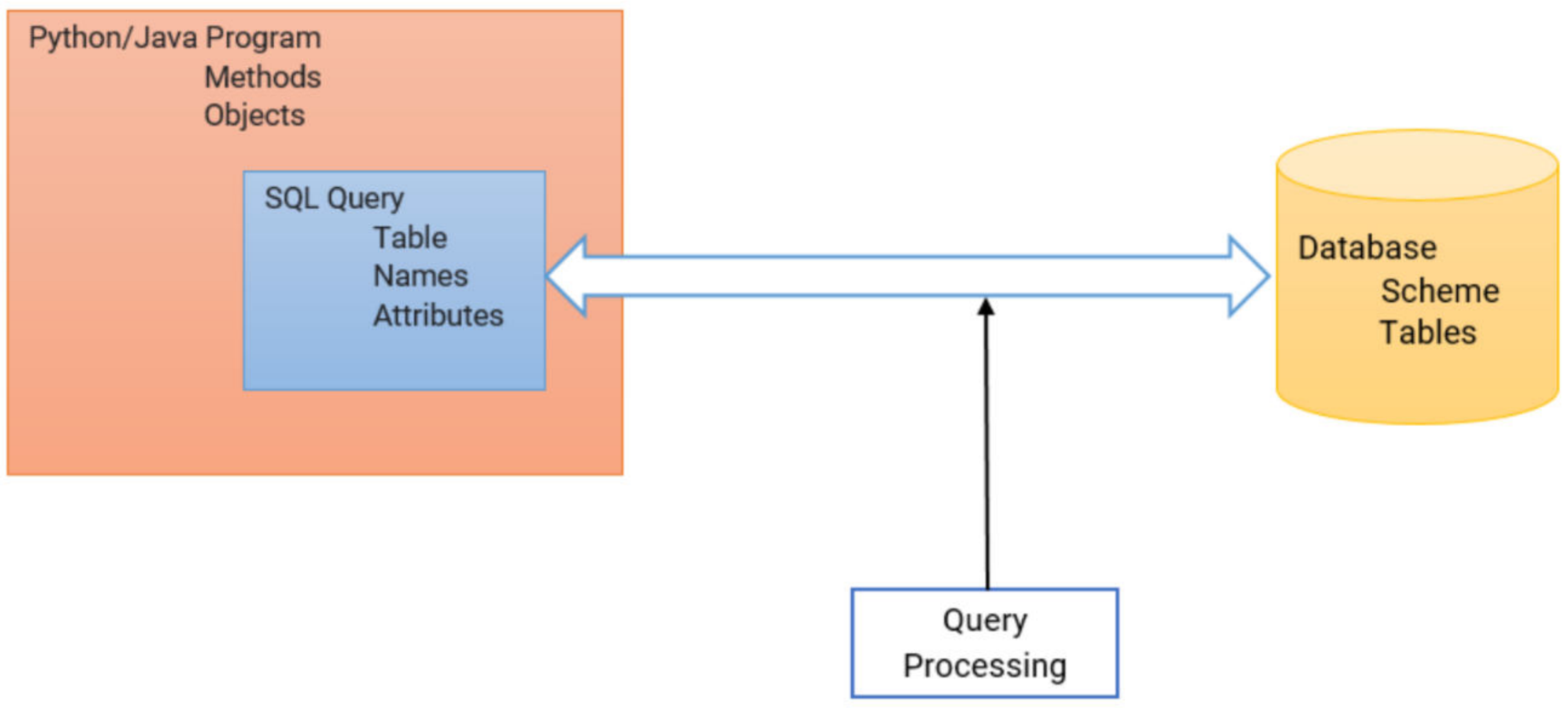
```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class RetrieveResultSet {
  public static void main(String[] args) {
    // Create a variable for the connection string.
    String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;";
    connectionUrl += "user=<user>; password=<password>";

    try (Connection con = DriverManager.getConnection(connectionUrl);
    Statement stmt = con.createStatement();) {
      createTable(stmt);
      String SQL = "SELECT * FROM Production.Product;";
      ResultSet rs = stmt.executeQuery(SQL);
      displayRow("PRODUCTS", rs);
    }

    // Handle any errors that may have occurred.
    catch (SQLException e) {
      e.printStackTrace();
    }
  }

  private static void displayRow(String title, ResultSet rs) throws SQLException {
    System.out.println(title);

    while (rs.next()) {
      // Iterate on Table("ProductID", "Name")
      System.out.println(rs.getString("ProductID") + " : " + rs.getString("Name"));
    }
  }

  private static void createTable(Statement stmt) throws SQLException {
    stmt.execute("if exists (select * from sys.objects where name = ’Product_JDBC_Sample’)"
    + "drop table Product_JDBC_Sample");

    String sql = "CREATE TABLE [Product_JDBC_Sample](" // Table Name
    + "[ProductID] [int] IDENTITY(1,1) NOT NULL," // Attribute 1
    + "[Name] [varchar](30) NOT NULL,)"; // Attribute 2

    stmt.execute(sql);
    sql = "INSERT Product_JDBC_Sample VALUES (’Adjustable Time’,’AR-5381’)"; // Add Product 1
    stmt.execute(sql);
    sql = "INSERT Product_JDBC_Sample VALUES (’ML Bottom Bracket’,’BB-8107’)"; // Add Product 2
    stmt.execute(sql);
    sql = "INSERT Product_JDBC_Sample VALUES ('Mountain-500 Black','BK-M18B-44')"; // Add Product 3
    stmt.execute(sql);
  }
}
```

Connectionist Bridge Configurations

A **bridge** is a special kind of driver that uses another driver-based technology

- This driver translates ***source function-calls*** into ***target function-calls***
- Programmers usually use such a bridge when they lack a ***source driver*** for some DB but have access to a ***target driver***
- Common bridges are:
    - **ODBC-to-JDBC (ODBC-JDBC) bridges** → An ODBC-JDBC bridge consists of an ODBC driver which uses the services of a JDBC driver to connect to a database
        - **Example** → OpenLink ODBC-JDBC Bridge, SequeLink ODBC-JDBC Bridge
    - **JDBC-to-ODBC (JDBC-ODBC) bridges** → A JDBC-ODBC bridge consists of a JDBC driver which employs an ODBC driver to connect to a target DB
        - Example → OpenLink JDBC-ODBC Bridge, SequeLink JDBC-ODBC Bridge
    - **OLE DB-to-ODBC bridges** → An OLE DB-ODBC bridge consists of an OLE DB Provider which uses the services of an ODBC function calls
        - **Example** → OpenLink OLEDB-ODBC Bridge, SequeLink OLEDB-ODBC Bridge
    - **ADO.NET-to-ODBC bridges** → An ADO.NET-ODBC bridge consists of an ADO.NET Provider which uses the services of an ODBC driver to connect to a target DB
        - **Example** → OpenLink ADO.NET-ODBC Bridge, SequeLink ADO.NET-ODBC Bridge

Native Language $\Longleftrightarrow$ Query Language: Embedded SQL



Embedded SQL

- The SQL standard defines embedding of SQL in a variety of programming languages such as C, C++, Java, FORTRAN, and PL/1
- A language to which SQL queries are embedded is referred to as a ***host language***, and the SQL structures permitted in the host language comprise ***embedded SQL***
- The basic form of these languages follow that of the System R embedding of SQL into PL/1
- `EXEC SQL` (or similar alternate like `#sql`) statement is used to identify embedded SQL request to the pre-processor

  `EXEC SQL <embedded SQL statement>;`

  NOTE → this varies by language:

- In some languages, like COBOL, the semi-colon is replaced with `END-EXEC`
- In Java embedding uses `# SQL {...};`

- Before executing any SQL statements, the program must first connect to the DB

  This is done using:

  EXEC-SQL **connect to** *server* **user** *user-name* **using** *password;*

  Here, *server* identifies the server to which a connection is to be established
- Variables of the host language can be used within embedded SQL statements
- They are preceded by a colon (:) to distinguish from SQL variables (for example, *:credit_amount*)
- Variables used as above must be declared within DECLARE section, as illustrated below

  The syntax for declaring the variables, however, follows the usual host language syntax

  EXEC-SQL BEGIN DECLARE SECTION

  int *credit-amount;*

  EXEC-SQL END DECLARE SECTION;

- To write an embedded SQL query, we use the

  declare *c* ***cursor*** **for <SQL query>**
- Example
  - From within a host language, find the ID and name of the students who have completed more than the number of credits stored in variable `credit_amount` in the host language
  - Specify the query in SQL as follows:

    EXEC SQL

    declare *c* **cursor for**

    select *ID, name*

    from *student*

    where tot_cred > :credit_amount

    END_EXEC
- The variable *c* (used in the cursor definition) is used to identify the query

---

- The open statement for our examples is as follows:

  EXEC SQL **open** c;

  This statement causes the DB system to execute the query and to save the results within a temporary relation

  The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed
- The fetch statement causes the values of one tuple in the query result to be placed on host language variables

  EXEC SQL **fetch** *c* **into** :si, :sn END_EXEC

  Repeated calls to fetch get successive tuples in the query result

---

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the DB system to delete the temporary relation that holds the result of the query

  EXEC SQL **close** c;

  NOTE: Above details vary with language

  For example, the Java embedding defines Java iterators to step through result tuples

---

- Embedded SQL expressions for DB modification (**update, insert** and **delete**)
- Can update tuples fetched by the cursor by declaring that the cursor is for update

  EXEC SQL

  declare *c* **cursor for**

select *

from *instructor*

where *dept_name* = *'Music'*

for update

- We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier) and after fetching each tuple we execute the following code

  update *instructor*

  set *salary* = *salary* + *1000*

  where current of *c*

Embedded SQL: C Example

- Here is an example embedded SQL C program from ***DB2: Embedded SQL for C and C++*** (by P. Godfrey, Nov. 2002)
- It does not do much, but is instructive
- The app queries a table `sailor` in schema one
- User one has granted select privileges to all on table `sailor`, so the bind step will be legal
- The app takes one argument on the command line, a sailor's SID
  - It then finds the sailor SID's age out of the table ONE.SAILOR and reports it

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlcodes.h>
#include <sys/time.h>

#define EXIT 0
#define NOEXIT 1

EXEC SQL INCLUDE SQLCA; // Include DB2’s SQL error reporting facility.
EXEC SQL BEGIN DECLARE SECTION; // Declare the SQL interface variables.

short sage, sid; char sname[16];
EXEC SQL END DECLARE SECTION;

// Declare variables to be used in the following C program
char msg[1025]; int rc, errcount;

// This macro prints the message in the SQLCA if the return code is 0 and the SQLCODE is not 0
#define PRINT_MESSAGE() { \

if (rc == 0 && sqlca.sqlcode != 0) {
  sqlaintp(msg, 1024, 0, &sqlca);
  printf("%s\n",msg);
}

// The macro prints out all fields in the SQLCA
#define DUMP_SQLCA() {
  printf("********** DUMP OF SQLCA *********************\n");
  printf("SQLCAID: %s\n", sqlca.sqlcaid); printf("SQLCABC: %d\n", sqlca.sqlcabc);
  printf("SQLCODE: %d\n", sqlca.sqlcode); printf("SQLERRML: %d\n", sqlca.sqlerrml);
  printf("SQLERRD[0]: %d\n", sqlca.sqlerrd[0]); printf("SQLERRD[1]: %d\n", sqlca.sqlerrd[1]);
  printf("SQLERRD[2]: %d\n", sqlca.sqlerrd[2]); printf("SQLERRD[3]: %d\n", sqlca.sqlerrd[3]);
  printf("SQLERRD[4]: %d\n", sqlca.sqlerrd[4]); printf("SQLERRD[5]: %d\n", sqlca.sqlerrd[5]);
  printf("SQLWARN: %s\n", sqlca.sqlwarn); printf("SQLSTATE: %s\n", sqlca.sqlstate);
  printf("********** END OF SQLCA DUMP *******************\n");
}

// This macro prints the message in the SQLCA if one exists
// If the return code is not 0 or the SQLCODE is not expected, an error occurred and must be recorded.
#define CHECK_SQL(code,text_string,eExit) {
  PRINT_MESSAGE();

  if (rc != 0 || sqlca.sqlcode != code) {
    printf("%s\n",text_string); printf("Expected code = %d\n",code);

    if (rc == 0) DUMP_SQLCA();
    else printf("RC: %d\n",rc);

    errcount += 1;

    if (eExit == EXIT) goto errorexit;
  }
}

main (int argc, char *argv[]) { // The PROGRAM
  // Grab the first command argument. This is the SID
  if (argc > 1) {
    sid = atoi(argv[1]);
    printf("SID requested is %d.\n", sid); // If there is no argument, bail
  } else {
    printf("Which SID?\n");
    exit(0);
  }

  EXEC SQL CONNECT TO C3421M;
  CHECK_SQL(0, "Connect failed", EXIT);

  // Find the name and age of sailor SID
  EXEC SQL SELECT SNAME, AGE into :sname, :sage
  FROM ONE.SAILOR
  WHERE sid = :sid;
  CHECK_SQL(0, "The SELECT query failed.", EXIT);

  // Report the age
  printf("Sailor %s’s age is %d.\nExecuted Successfully\nBye\n", sname, sage);
  errorexit:
  EXEC SQL CONNECT RESET;
}
```

- The instance of the table `sailor`:

SNAME	SID	RATING	AGE
yuppy	22	1	20
lubber	31	1	25
guppy	44	2	31
rusty	58	3	47

- If the name of the executable is `sage`, and if you ask:

  `% sage 44`

- The output should be

  `SID requested is 44`

  `Sailor guppy's age is 31`

  `Executed Successfully`

  `Bye`

Embedded SQL: C Example

- The program prompts the user for an order number, retrieves the customer number, salesperson and status of the order, and displays the retrieved information on the screen

```
int main() {
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      int OrderID;         /* Employee ID (from user)         */
      int CustID;            /* Retrieved customer ID         */
      char SalesPerson[10]   /* Retrieved salesperson name      */
      char Status[6]         /* Retrieved order status        */
   EXEC SQL END DECLARE SECTION;

   /* Set up error processing */
   EXEC SQL WHENEVER SQLERROR GOTO query_error;
   EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

   /* Prompt the user for order number */
   printf ("Enter order number: ");
   scanf_s("%d", &OrderID);

   /* Execute the SQL query */
   EXEC SQL SELECT CustID, SalesPerson, Status
      FROM Orders
      WHERE OrderID = :OrderID
      INTO :CustID, :SalesPerson, :Status;

   /* Display the results */
   printf ("Customer number:  %d\n", CustID);
   printf ("Salesperson: %s\n", SalesPerson);
   printf ("Status: %s\n", Status);
   exit();

query_error:
   printf ("SQL error: %ld\n", sqlca->sqlcode);
   exit();

bad_number:
   printf ("Invalid order number.\n");
   exit();
}
```

Source: https://docs.microsoft.com/en-us/sql/odbc/reference/embedded-sql-example?view=sql-server-ver15

- The statement used to return the data is a singleton `SELECT` statement
  - That is, it returns only a single row of data
  - So, the code example does not declare or use cursors

Embedded SQL: Java Example

- The following example SQLJ Application, `App.sqlj`, uses static SQL to retrieve and update data from the `EMPLOYEE` table of the sample DB
- Complete example can be found at *https://www.ibm.com/docs/en/i/7.1?topic=essiyja-example-embedding-sql-statements-in-your-java-application*

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; // 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

 public static void main(String argv[])
 {
   try
   {
     App_Cursor1 cursor1;
     App_Cursor2 cursor2;

     String str1 = null;
     String str2 = null;
     long   count1;

     // URL is jdbc:db2:dbname
     String url = "jdbc:db2:sample";

     DefaultContext ctx = DefaultContext.getDefaultContext();
     if (ctx == null)
     {
       try
       {
         // connect with default id/password
         Connection con = DriverManager.getConnection(url);
         con.setAutoCommit(false);
         ctx = new DefaultContext(con);
       }
       catch (SQLException e)
       {
         System.out.println("Error: could not get a default context");
         System.err.println(e) ;
```

```
          System.exit(1);
        }
        DefaultContext.setDefaultContext(ctx);
      }

      // retrieve data from the database
      System.out.println("Retrieve some data from the database.");
      #sql cursor1 = {SELECT empno, firstnme FROM employee}; // 2

      // display the result set
      // cursor1.next() returns false when there are no more rows
      System.out.println("Received results:");
      while (cursor1.next()) // 3
      {
        str1 = cursor1.empno(); // 4
        str2 = cursor1.firstnme();

        System.out.print (" empno= " + str1);
        System.out.print (" firstname= " + str2);
        System.out.println("");
      }
      cursor1.close(); // 9

      // retrieve number of employee from the database
      #sql { SELECT count(*) into :count1 FROM employee }; // 5
      if (1 == count1)
        System.out.println ("There is 1 row in employee table");
      else
        System.out.println ("There are " + count1
                           + " rows in employee table");

      // update the database
      System.out.println("Update the database.");
      #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

      // retrieve the updated data from the database
      System.out.println("Retrieve the updated data from the database.");
      str1 = "000010";
      #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; // 6

      // display the result set
      // cursor2.next() returns false when there are no more rows
      System.out.println("Received results:");
      while (true)
      {
        #sql { FETCH :cursor2 INTO :str2 }; // 7
        if (cursor2.endFetch()) break; // 8

        System.out.print (" empno= " + str1);
        System.out.print (" firstname= " + str2);
        System.out.println("");
      }
      cursor2.close(); // 9

      // rollback the update
      System.out.println("Rollback the update.");
      #sql { ROLLBACK work };
      System.out.println("Rollback done.");
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }
}
```

Embedded SQL: Java Example → Notes

- **Declare iterators** → This section declares two types of iterators
  - `App_Cursor1` → Declares column data types and names, and returns the values of the columns according to column name (Named binding to columns)
  - `App_Cursor2` → Declares column data types, and returns the values of the columns by column position (Positional binding to columns)
- **Initialize the iterator** → The iterator object `cursor1` is initialized using the result of a query
  - The query stores the result in `cursor1`
- **Advance the iterator to the new row** → The `cursor1.next()` method returns a Boolean false if there are no more rows to retrieve
- **Move the data** → The named accessor method `empno()` returns the value of the column named `empno` on the current row

    - The named accessor method `firstnme()` returns the value of the column named `firstnme` on the current row
- **SELECT data into a host variable** → The SELECT statement passes the number of rows in the table into the host variable `count1`
- **Close the iterators** → The close() method releases any resources held by the iterators
    - You should explicitly close iterators to ensure that system resources are released in a timely fashion

---

- **Initialize the iterator** → The iterator object `cursor2` is initialized using the result of a query
    - The query stores the result in `cursor2`
- **Retrieve the data** → The FETCH statement returns the current value of the first column declared in the `ByPos` cursor from the result table into the host variable `str2`
- **Check the success of a FETCH.INTO statement** → The `endFetch()` method returns a Boolean true if the iterator is not positioned on a row, that is, if the last attempt to fetch a row failed
    - The `endFetch()` method returns False if the last attempt to fetch a row was successful
    - DB2 attempts to fetch a row when the `next()` method is called
    - A FETCH..INTO statement implicitly calls the `next()` method
- **Close the iterators** → The close() method releases any resources held by the iterators
    - You should explicitly close iterators to ensure that system resources are released in a timely fashion

Week 7 Lecture 4

Class	BSCCS2001
Created	@October 20, 2021 3:36 PM
Materials	
Module #	34
Type	Lecture
Week #	7

Application Design & Development: Python & PostgreSQL

Working with PostgreSQL and Python

Python modules for PostgreSQL

Following Python modules that can be used to work with a PostgreSQL DB server:

- psycopg2
- pg8000
- py-postgresql
- PyGreSQL
- ocpgdb
- bpgsql
- SQLAlchemy

Source: https://pynative.com/python-postgresql-tutorial/

Package: `psycopg2`

Advantages of psycopg2

- Most popular and stable module to work with PostgreSQL
- Used in most of the Python and Postgres frameworks
- An actively maintained package and supports Python 2.x and 3.x
- Thread-safe and designed for heavily multi-threaded applications

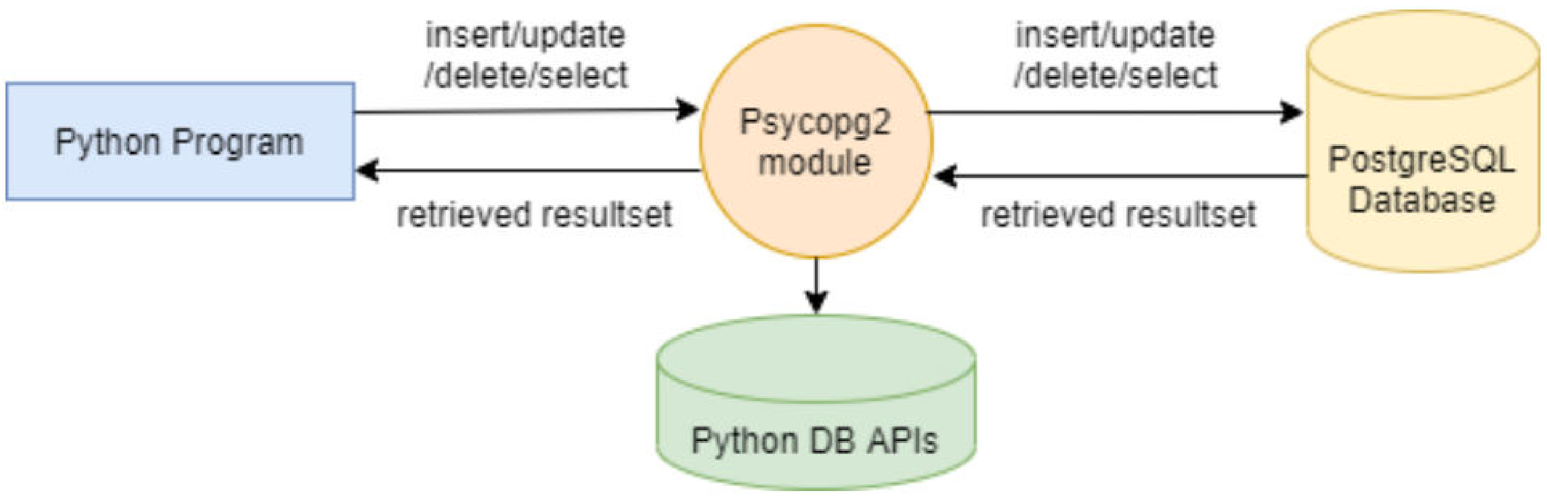
Installing Psycopg2 using the `pip` command

- The following `pip` command installs psycopg2 on different OSs including Windows, Mac OS, Linux and Unix
  `pip install psycopg2`
- For installing a specific version, the following command can be used
  `pip install psycopg2=2.8.6`

Steps to access PostgreSQL from Python (using psycopg2)

- Create a connection
- Create a cursor
- Execute the query
- Commit/rollback

- Close the cursor
- Close the connection



Python psycopg2 Module APIs: Connection objects

- `psycopg2.connect(database="mydb", user="myuser", password="mypass", host="127.0.0.1", port="5432")`

  This API opens a connection to the PostgreSQL DB

  If the DB is opened successfully, it returns the connection object

- `connection.close()`

  This method closes the DB connection

Important **psycopg2** module routines for managing cursor object:

- `connection.cursor()`

  This routine creates a cursor which will be used throughout the program

- `cursor.close()`

  This method closes the cursor

Python psycopg2 Module APIs: insert, delete, update & stored procedures

- `cursor.execute(sql [, optional parameters])`

  This routine executes an SQL statement

  The SQL statement may be parameterized (i.e. placeholders instead of SQL literals)

  The **psycopg2** module supports placeholder using `%s` sign

  For example, `cursor.execute("insert into people values (%s, %s)", (who, age))`

- `cursor.executemany(sql, seq_of_parameters)`

  This routine executes an SQL command against all parameters sequences or mappings found in the sequence SQL

- `cursor.callproc(procname[, parameters])`

  This routine executes a stored database procedure with the given name

  The sequence of parameters must contain one entry for each argument that the procedure expects

- `cursor.rowcount()`

  This is a read-only attribute which returns the total number of DB rows that have been modified, inserted or deleted by the last `execute()`

Python psycopg2 Module APIs: select

- `cursor.fetchone()`

  This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available

- `cursor.fetchmany([size=cursor.arraysize])`

  This routine fetches the next set of rows of a query result, returning a list

  An empty list is returned when no more rows are available

  The method tries to fetch as many rows as indicted by the size parameter

- `cursor.fetchall()`

  This routine fetches all (remaining) rows of a query result, returning a list

  An empty list is returned when no rows are available

Python psycopg2 Module APIs: commit & rollback

- `connection.commit()`

  This method commits the current transaction

  If you do not call this method, anything you did since the last call to `commit()` will not be visible to other DB connections

- `connection.rollback()`

  This method rolls back any changes to the DB since the last call to `commit()`

Source: https://www.tutorialspoint.com/postgresql/postgresql_python.htm

Connect to a PostgreSQL DB server

```
import psycopg2

def connectDb(dbname, usrname, pwd, address, portnum):
  conn = None

  try:
    # Connect to the PostgreSQL DB
    conn = psycopg2.connect(database = dbname, user = usrname, password = pwd, host = address, port = portnum)
    print("Database connected successfully")
  except(Exception, psycopg2.DatabaseError) as error:
    print(error)
  finally:
    # Close the connection
    conn.close()

connectDb("mydb", "myuser", "mypass", "127.0.0.1", "5432")
```

Output

`Database connected successfully`

`psycopg2.DatabaseError` → Exception raised for errors that are related to the PostgreSQL DB

We assume the following for all the programs in this module

- Database Name → *mydb*
- Username → *myuser*
- Password → *mypass*
- Host name → localhost or 127.0.0.1

Steps to execute SQL commands

- Use the `psycopg2.connect()` method with the required arguments to connect PostgreSQL
    - It would return a Connection object if the connection is established successfully
- Create a cursor object using the `cursor()` method of the connection object
- The `execute()` methods run the SQL commands and return the result
- Use `cursor.fetchall()` or `fetchone()` or `fetchmany()` to read query result
- Use `commit()` to make the changes in the DB persistent, or use `rollback()` to revert the DB changes
- Use `cursor.close()` and `connection.close()` method to close the cursor and the PostgreSQL connection

CREATE new PostgreSQL tables

```
import psycopg2

def createTable():
  conn = None

  try:
    conn = psycopg2.connect(database = "mydb", user = "myuser", password = "mypass", host = "127.0.0.1", port = "5432")

    cur = conn.cursor() # Create a new cursor
    cur.execute('''CREATE TABLE EMPLOYEE \
    (emp_num INT PRIMARY KEY NOT NULL, \
    emp_name VARCHAR(40) NOT NULL, \
    department VARCHAR(40) NOT NULL)''') # Execute the CREATE TABLE statement

    conn.commit() # Commit the changes to the DB
    print("Table created successfully")
    cur.close() # close the cursor
  except (Exception, psycopg2.DatabaseError) as error:
    print(error)
  finally:
    if conn is not None:
      conn.close()

createTable()
```

Output (if the table EMPLOYEE does not exist) → `Table created successfully`

Output (if the table EMPLOYEE already exists) → `relation "employee" already exists`

Executing INSERT statement from Python

```
import psycopg2

def insertRecord(num, name, dept):
  conn = None

  try:
    # Connect to the PostgreSQL DB
    conn = psycopg2.connect(database = "mydb", user = "myuser", password = "mypass", host = "127.0.0.1", port = "5432")
    cur = conn.cursor() # Create a new cursor

    # Execute the INSERT statement
    cur.execute("INSERT INTO EMPLOYEE (emp_num, emp_name, department) \
    VALUES (%s, %s, %s)", (num, name, dept))
    conn.commit() # Commit the changes to the DB
    print ("Total number of rows inserted :", cur.rowcount);
    cur.close() # close the cursor
  except (Exception, psycopg2.DatabaseError) as error:
    print(error)
  finally:
    if conn is not None:
      conn.close()

insertRecord(110, ‘Bhaskar’, ’HR’)
```

Output → `Total number of rows inserted: 1`

Output (if a row already exists with the emp_num = 110) → `duplicate key value violates unique constraint "employee_pkey" DETAIL: Key (emp_num)=(110) already exists.`

Executing DELETE statement from Python

```
import psycopg2

def deleteRecord(num):
  conn = None

  try:
    # Connect to the PostgreSQL DB
    conn = psycopg2.connect(database = "mydb", user = "myuser", password = "mypass", host = "127.0.0.1", port = "5432")
    cur = conn.cursor() # Create a new cursor

    # Execute the DELETE statement
    cur.execute("DELETE FROM EMPLOYEE WHERE emp_num = %s", (num,))
    conn.commit() # Commit the changes to the DB
    print ("Total number of rows deleted :", cur.rowcount)
    cur.close() # Close the cursor
  except (Exception, psycopg2.DatabaseError) as error:
    print(error)
  finally:
    conn.close() # Close the connection

deleteRecord(110)
```

Output → `Total number of rows deleted: 1`

Output (If the row does not exist) → `Total number of rows deleted: 0`

Executing UPDATE statement from Python

```
import psycopg2

def updateRecord(num, dept):
  conn = None

  try:
    # Connect to the PostgreSQL DB
    conn = psycopg2.connect(database = "mydb", user = "myuser", password = "mypass", host = "127.0.0.1", port = "5432")
    cur = conn.cursor() # Create a new cursor

    # Execute the UPDATE statement
    cur.execute("UPDATE EMPLOYEE set department = %s where emp_num = %s", (dept, num))
    conn.commit() # Commit the changes to the DB
    print ("Total number of rows updated :", cur.rowcount)
    cur.close() # Close the cursor
  except (Exception, psycopg2.DatabaseError) as error:
    print(error)
  finally:
    conn.close() # Close the connection

updateRecord(110, "Finance")
```

Output → `Total number of rows updated: 1`

Output (If the row does not exist) → `Total number of rows updated: 0`

Executing SELECT statement from Python

```
import psycopg2

def selectAll():
  conn = None

  try:
    # Connect to the PostgreSQL DB
    conn = psycopg2.connect(database = "mydb", user = "myuser", password = "mypass", host = "127.0.0.1", port = "5432")
    cur = conn.cursor() # Create a new cursor

    # Execute the SELECT statement
    cur.execute("SELECT emp_num, emp_name, department FROM EMPLOYEE")
    rows = cur.fetchall() # Fetches all rows of the query result set

    for row in rows:
      print (print ("Employee ID = ", row[0], ", NAME = ", row[1], ", DEPARTMENT = ", row[2]))

    cur.close() # Close the cursor
  except (Exception, psycopg2.DatabaseError) as error:
    print(error)
  finally:
    conn.close() # Close the connection

selectAll()
```

Output →

`Employee ID = 110, NAME = Bhaskar, DEPARTMENT = HR`
`Employee ID = 111, NAME = Ishaan, DEPARTMENT = FINANCE`
`Employee ID = 112, NAME = Jairaj, DEPARTMENT = TECHNOLOGY`
`Employee ID = 113, NAME = Ananya, DEPARTMENT = TECHNOLOGY`

Python frameworks for PostgreSQL

Web and Internet development using Python

Python offers several frameworks such as **bottle.py, Flask, CherryPy, Pyramid, Django** and **web2py** for web development

- Python offers many choices for web development
  - Frameworks such as **Django** and **Pyramid**
  - Micro-frameworks such as **Flask** and **Bottle**
  - Advanced content management systems such as **Plone** and **Django CMS**
- Python's standard library supports many internet protocols
  - HTML and XML
  - JSON
  - E-mail processing
  - Support for FTP, IMAP and other Internet protocols
  - Easy-to-use socket interface
- The package Index has more libraries
  - **Requests** → a powerful HTTP client library
  - **Beautiful Soup** → an HTML parser that can handle all sorts of HTML
  - **Feedparser** → for parsing RSS/Atom feeds
  - **Paramiko** → implementing the SSH2 protocol
  - **Twisted Python** → a framework for asynchronous network programming

  Source: https://www.python.org/about/apps/

Flask Web Application Framework

- Flask is a lightweight ***WSGI (Web Server Gateway Interface)*** web application framework
- It is desired to make getting started quick and easy, with the ability to scale up to complex applications
- It began as a simple wrapper around **Werkzeug (Werkzeug WSGI toolkit)** and Jinja (Jinja templating engine) and has since then become one of the most popular Python web application frameworks
- Flask offers suggestions, but does not enforce any dependencies or project layouts
  - It is up to the developer to choose the tools and libraries they want to use
- There are many extensions provided by the community that make adding new functionality easy

Installing Flask using the `pip` command

`pip install -U Flask`

Source: https://pypi.org/project/Flask/

A simple example

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
  return "Hello, World!"

if __name__ == '__main__':
  app.run()
```

- Importing `flask` module in the project is mandatory
  - Our WSGI application is an object of Flask class
- Flask constructor takes the name of the current module `(__name__)` as argument
- The `route()` function of the Flask class is a decorator, which tells the application which URL should call the associated function
  `app.route(rule, options)`
  - The `rule` parameter represents URL binding with the function
  - The `options` is a list of parameters to be forwarded to the underlying Rule object
- In the above example '/' URL is bound with `hello_world()` function
  - Hence, when the home page of web server is opened in the browser, the output of this function will be rendered
- Finally, the `run()` method of Flask class runs the application on the local development server

Source: https://www.tutorialspoint.com/flask/flask_application.htm

---

`app.run(host, port, debug, options)`

- `host` → Hostname to listen on
  - Defaults to `127.0.0.1 (localhost)`
  - Set to `0.0.0.0` to have the server available externally
- `port` → Defaults to 5000
- `debug` → Defaults to `false`
  - If set to `true`, provides a debug information
- `options` → To be forwarded to the underlying Werkzeug server

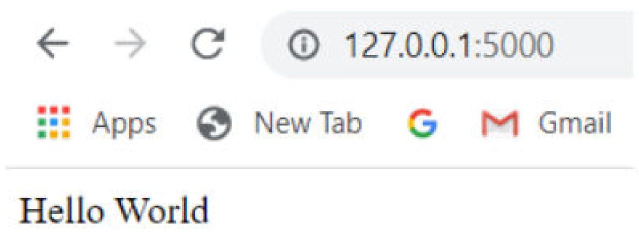
Executing the code

- `python hello.py`

Output

- `* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)`

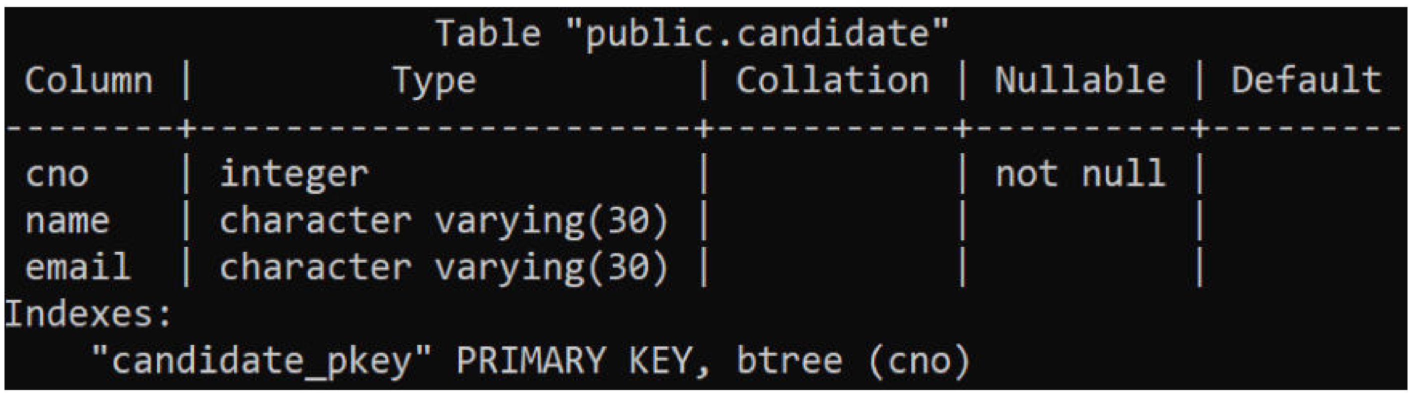
Open the above URL (127.0.0.1:5000) in the browser



Hello World

Python: Flask

- Consider the table **Candidate** (in PostgreSQL) as shown below:



```
                    Table "public.candidate"
 Column |          Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 cno    | integer               |           | not null |
 name   | character varying(30) |           |          |
 email  | character varying(30) |           |          |
Indexes:
    "candidate_pkey" PRIMARY KEY, btree (cno)
```

- Code segment in Python:

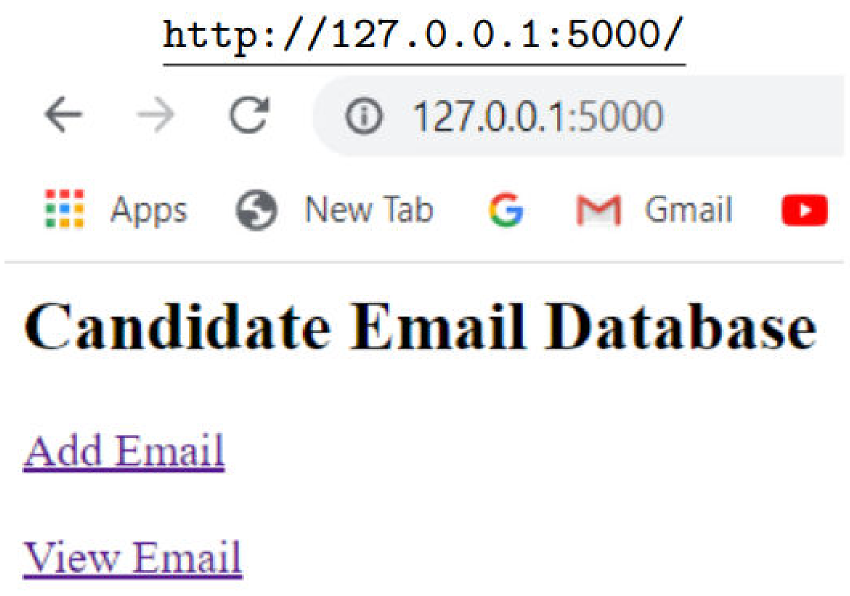
```
from flask import Flask, render_template, request
import psycopg2

app = Flask(__name__, template_folder='templates')

if __name__ == '__main__':
  app.run(host='127.0.0.1', debug=True, port=5000)
```

- Source code for `index.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>Candidate Email DB</title>
</head>
<body>
  <h2>Candidate Email DB</h2>
  <a href="/add">Add Email</a><br><br>
  <a href="/viewall">View Email</a>
</body>
</html>
```

- Source code for rendering `index.html` and `add.html` pages

```
@app.route("/")
def index():
  return render_template("index.html")

@app.route("/add")
def add():
  return render_template("add.html")
```


http://127.0.0.1:5000/

127.0.0.1:5000

Candidate Email Database

Add Email

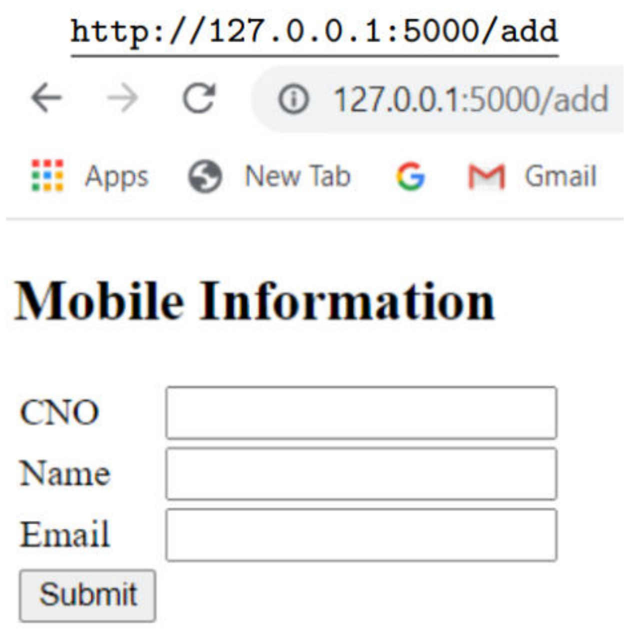
View Email

- Source code for `add.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>Add Email</title>
</head>
<body>
  <h2>Email Information</h2>
  <form action = "/savedetails" method="post">
  <table>
    <tr><td>CNO</td><td><input type="text" name="cno" required></td></tr>
    <tr><td>Name</td><td><input type="text" name="name" required></td></tr>
    <tr><td>Email</td><td><input type="text" name="email" required></td></tr>
    <tr><td><input type="submit" value="Submit"></td></tr>
  </table>
  </form>
</body>
</html>
```

- `saveDetails()` function for `add.html`

```
@app.route("/savedetails",methods = ["POST"])
def saveDetails():
  cno = request.form["cno"]
  name = request.form["name"]
  email = request.form["email"]
  conn = None

  try:
    conn = psycopg2.connect(database = "mydb", user = "myuser", password = "mypass", host = "127.0.0.1", port = "5432") # connect to the PostgreSQL database
    cur = conn.cursor() # Create a new cursor

    cur.execute("INSERT INTO Candidate (cno, name, email) VALUES (%s, %s, %s)", (cno, name, email)) # Execute the INSERT statement
    conn.commit() # Commit the changes to the DB
    cur.close() # Close the cursor
  except (Exception, psycopg2.DatabaseError) as error:
    render_template("fail.html")
  finally:
    if conn is not None:
      conn.close() # Close the connection

return render_template("success.html")
```

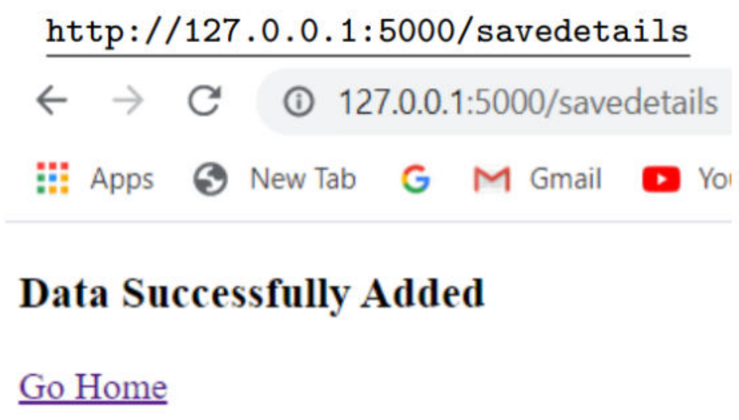

http://127.0.0.1:5000/add

127.0.0.1:5000/add

Mobile Information

CNO	
Name	
Email	

Submit


http://127.0.0.1:5000/savedetails

127.0.0.1:5000/savedetails

Data Successfully Added

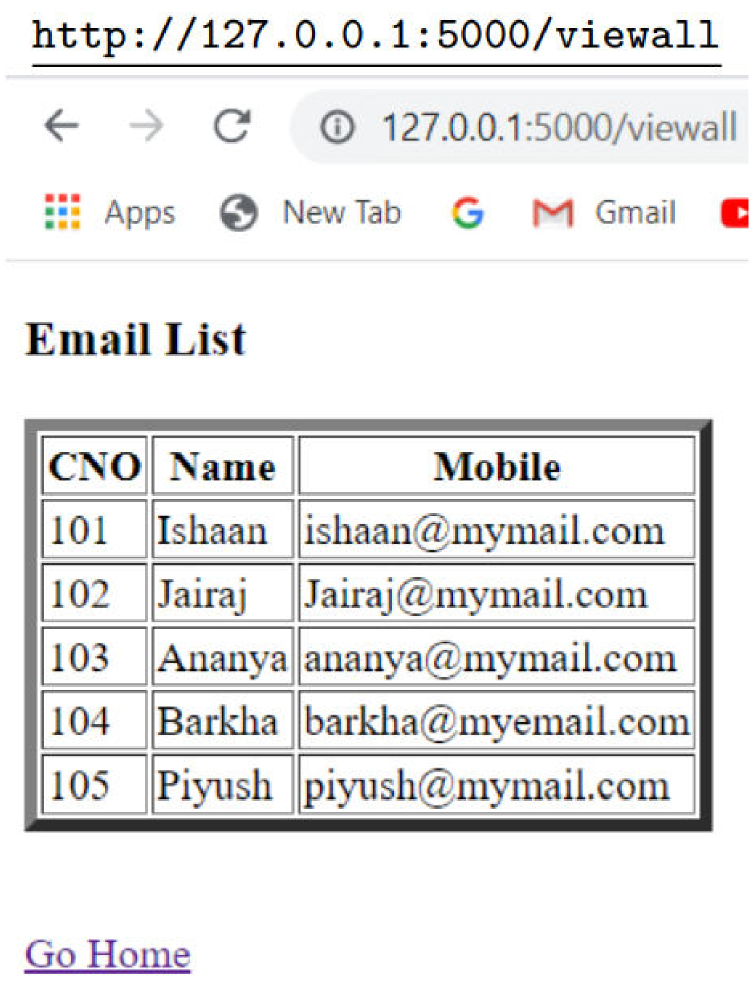
Go Home

- `viewAll()` function for `viewall.html`

```
@app.route("/viewall")
def viewAll():
  conn = None

  try:
    # Connect to the PostgreSQL DB
    conn = psycopg2.connect(database = "mydb", user = "myuser", password = "mypass", host = "127.0.0.1", port = "5432")
    cur = conn.cursor() # Create a new cursor

    # Execute the SELECT statement
    cur.execute("SELECT cno, name, email FROM Candidate")
    results = cur.fetchall() # Fetches all rows of the query result set
    cur.close() # Close the cursor
  except (Exception, psycopg2.DatabaseError) as error:
    render_template("fail.html")
  finally:
    conn.close() # Close the connection

return render_template("viewall.html",rows = results)
```

- Source code for `viewall.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>Email List</title>
</head>
<body>
  <h3>Email List</h3>
  <table border=5>
  <tr>
    <th>CNO</td><th>Name</td><th>Email</td>
  </tr>
  {% for row in rows %}
  <tr>
    <td>{{row[0]}}</td> <td>{{row[1]}}</td> <td>{{row[2]}}</td>
  </tr>
  {% endfor %}
  </table>
  <br><br>
  <a href="/">Go Home</a>
</body>
</html>
```



Email List

CNO	Name	Mobile
101	Ishaan	ishaan@mymail.com
102	Jairaj	Jairaj@mymail.com
103	Ananya	ananya@mymail.com
104	Barkha	barkha@myemail.com
105	Piyush	piyush@mymail.com

Go Home

Week 7 Lecture 5

Class	BSCCS2001
Created	@October 20, 2021 6:56 PM
Materials	
Module #	35
Type	Lecture
Week #	7

Application Design & Development: Application Development and Mobile

Rapid Application Development (RAD)

- A lot of effort is required to develop Web applications interfaces, especially the rich interaction functionality associated with Web 2.0 applications
- Several approaches to speed up application development
    - Function library to generate user-interface elements
    - Drag-and-drop features in an IDE to create user-interface elements
    - Automatically generate code for the user interface from a declarative specification
- Used as part of **Rapid Application Development (RAD)** tools even before Web
- RAD software is an agile model that focuses on fast prototyping and quick feedback in app development to ensure speedier delivery and an efficient result
    - App development has 4 phases → business modeling, data modeling, process modeling and testing & turnover
        - Defining the requirements, Prototyping, Receiving feedback and Finalizing the software
    - With RAD, the time between prototypes and iterations is short, and integration occurs since inception

---

- Web application development frameworks
    - **Java Server Faces (JSF)**
        - A set of APIs for representing UI components and managing the state, handling the events and input validation, defining page navigation and supporting internationalization and accessibility

        - JSP custom tag library for expressing a JSF interface within a JSP page
    - **Ruby on Rails**
        - Allows easy creation of simple **CRUD (Create, Read, Update, Delete)** interfaces by code generation from DB schema or object model
- RAD platforms and tools
    - G Suite
    - Google App Engine
    - Microsoft Azure
    - Amazon Elastic Compute Cloud (EC2)
    - AWS Elastic Beanstalk

ASP.NET and Visual Studio

- ASP.NET provides a variety of controls that are interpreted at server, and generate HTML code
- Visual Studio provides a drag-and-drop development using these controls
    - For example, menus and list boxes can be associated with DataSet object
    - Validator controls (constraints) can be added to form input fields
        - JavaScript to enforce constraints at client, and separately enforced at the server
    - User actions such as selecting a value from a menu can be associated with actions at server
    - DataGrid provides convenient way of displaying SQL query results in a tabular format

Application Performance and Security

Application Performance

- Performance is an issue for popular Web sites
    - May be accessed by millions of users every day, thousands of requests per second at peak time
- Caching techniques used to reduce cost of serving pages by exploiting commonalities between requests
    - At the server side
        - Caching of JDBC connections between servlets requests
            - aka connection pooling
        - Caching results of DB queries
            - Cached results must be updated if underlying DB changes
        - Caching of generated HTML
    - At the client side
        - Caching of pages by Web proxy

Application Security: SQL Injection

- Suppose query is constructed using
    - `"select * from instructor where name = '" + name + "'"`
- Suppose the user, instead of entering a name, enters:
    - `X' or 'Y' = 'Y`
- The the resulting statement becomes
    - `"select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"`
    - Which is ...
        - select * from instructor where name = 'X' or 'Y' = 'Y'
    - User could have even used

        - `X'; update instructor set salary = salary + 10000; - -`
- Prepared statement internally uses:
  `"select * from instructor where name = 'X \' or \'Y\' = \'Y'`
- **Always use prepared statements, with user inputs as parameters**

Application Security: Password Leakage

- Never store password, such as DB passwords, in clear text in scripts that may be accessible to users
    - For example, in files in a directory accessible to a web server
        - Normally, web server will execute, but not provide source of scripts files such as `file.jsp` or `file.php`, but source of editor backup files such as `file.jsp~`, or `.file.jsp.swp` may be server
- Restrict access to DB server from IPs of machine running the application servers
    - Most DBs allow restriction of access by source IP address

Application Security: Authentication

- Single factor authentication such as passwords are too risky for critical applications
    - Guessing of passwords, sniffing of packets if passwords are not encrypted
    - Passwords re-used by user across sites
    - Spyware which captures password
- Two-factor authentication
    - For example, password plus one-time password sent by SMS
    - For example, password plus one-time password devices
        - Device generates a new pseudo-random number every minute and displays to users
        - User enters the current number as password
        - Application server generates same sequence of pseudo-random number to check that the number is correct

Application Security: Application-Level Authorization

- Current SQL standard does not allow fine-grained authorization such as "students can see their own grades, but not other's grades"
    - Problem 1 → DB has no idea who are application users
    - Problem 2 → SQL authorization is at the level of tables, or columns of tables, but not to specific rows of a table
- One workaround → use views such as

  CREATE VIEW *studentTakes* **AS**

  SELECT *

  FROM *takes*

  WHERE *takes.ID* = *syscontext.user_id()*
    - Where `syscontext.user_id()` provides end user identity
        - End user identity must be provided to the DB by the application
    - Having multiple such views is cumbersome
- Currently authorization is done entirely in application
- Entire application code has access to the entire DB
    - Larger surface area, making protection harder
- Alternative → **find-grained (row-level) authorization** schemes
    - Extensions to SQL authorization proposed but not currently implemented
    - Oracle Virtual Private DB (VPD) allows predicates to be added transparently to all the SQL queries, to enforce find-grained authorization
        - For example, add `ID = sys_context.user_id()` to all queries on student relation if user is a student

Application Security: 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 a security breach
    - Trace who carried out the breach
- Audit trails needed at
    - DB level, and at
    - Application level

Challenges in Web Application Development

- User Interface and User Experience
- Scalability
- Performance
- Knowledge of Framework and Platforms
- Security

What is a Mobile App?

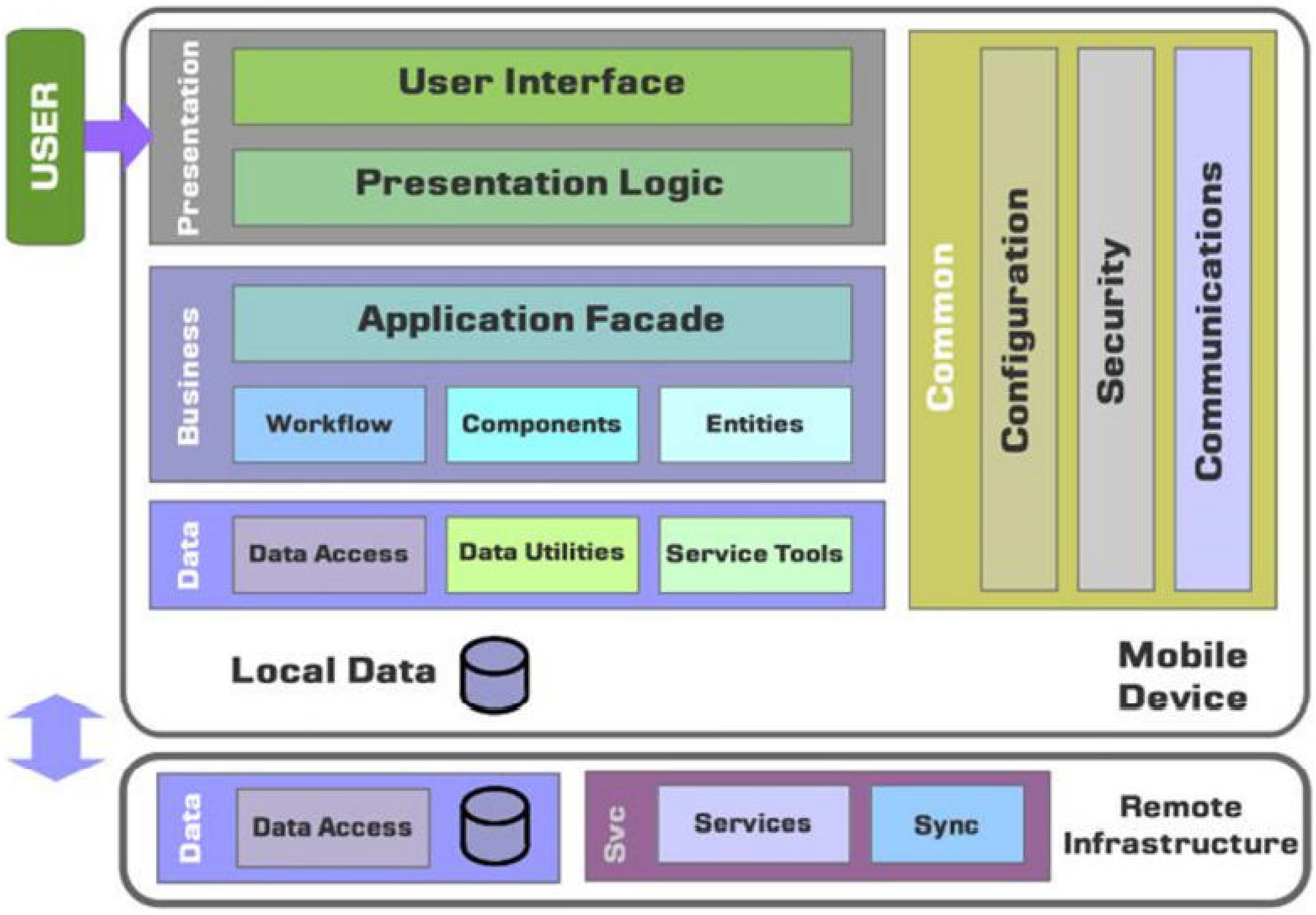
- A type of application software designed to run on a mobile device, such as a smartphone or a tablet computer
- Developed specifically for use on small, wireless computing devices
- Designed with consideration for the demands and constraints of the devices and also to take advantage of any specialized capabilities
    - Cons
        - Form factor — influences display and navigation
        - Limited memory
        - Limited computing power
        - Limited power
        - Limited bandwidth
    - Pros
        - Availability of sensors like accelerometer
        - Availability of touchscreen — Gesture-based navigation

Mobile Websites vis-`a-vis Mobile App

- **Mobile website**
    - Similar to any other website in that it consists of browser-based HTML pages
    - Can display text content, data, images and video
    - Typically accessed over WiFi or 3G or 4G networks
    - Designed for smaller handheld display and touch-screen interface
    - Can also access mobile-specific features such as click-to-call (to dial a number) or location-based mapping
- **Mobile apps**
    - Actual applications that are downloaded and installed on a mobile device
    - Users download apps from device-specific portals such as App Store, Google Play Store
    - The app may
        - Pull content and data from the internet, in a similar fashion to a website or
        - Download the content so that it can be accessed without an internet connection

Architecture of Mobile App

- Typically 3 tier
  - Presentation
  - Business
  - Data
- Data layer is often split between:
  - Local data
  - Remote data
- Needs customization for platform
  - Android
  - iOS
  - Windows



Types of Mobile Apps

- **Native Apps** → Completely written in the language of the platform
  - iOS → Objective-C
  - Android → Java or C/C++
  - Platform specific (Heavily dependent on the OS)
- **Web Apps** → Run completely inside of a web browser
  - Features interfaces built with HTML or CSS
  - Powered via Web programming languages → Ruby on Rails, JavaScript, PHP or Python
  - Portable across any phone, tablet or computer
- **Hybrid Apps** → Combines attributes of both native and Web Apps
  - Attempts to use redundant, common code that can be used across platforms
  - Tailors required attributes to the native system

Design Issues

- Determine Device
- Note Device Resources — memory, power, speed
- Consider bandwidth
- Decide on Architecture Layers
- Select Technology
- Define User Interface
- Select Navigation
- Maintain Flow