W4111 – Introduction to Databases
Section 002, Fall 2025
Lecture 5: Foundation – ER, Relational, SQL (4)

Contents

Notes and Content

- ER Model/Modeling
  - Relationship degree and cardinality
  - Aggregation
- Relational algebra
  - Some complex examples
  - More fun with the RelaX calculator
- SQL
  - Loading Classic Models dataset.
  - Indexes
  - Advanced Windows Functions
  - Recursive queries
  - Procedures, functions and triggers
  - Using a programming language
- Project concepts: (Web) applications, REST, data engineering

ER Model/ER Modeling

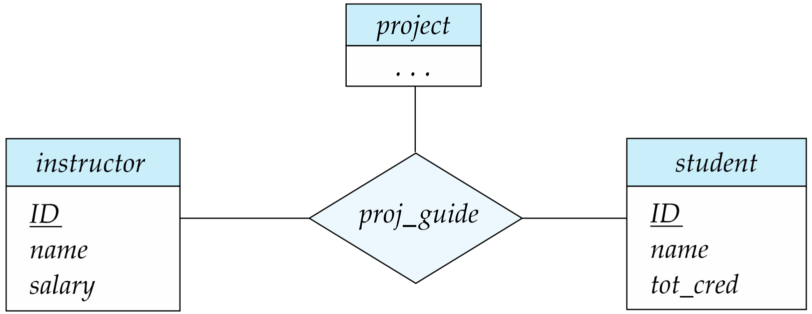
Degree
Cardinality

Degree of a Relationship Set

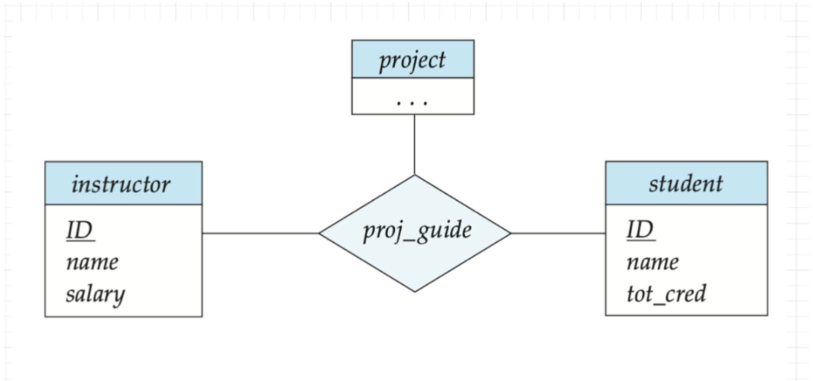
- Binary relationship
  - involve two entity sets (or degree two).
  - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
  - Example: *students* work on research *projects* under the guidance of an *instructor*.
  - relationship *proj_guide* is a ternary relationship between *instructor, student,* and *project*

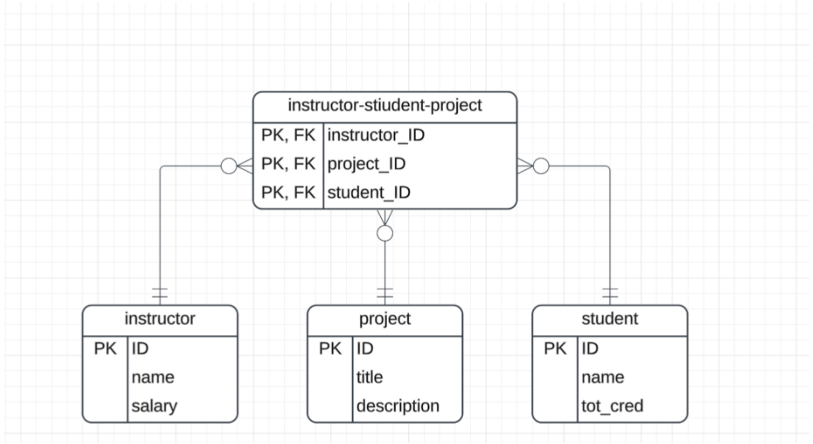
Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship



Crow's Foot





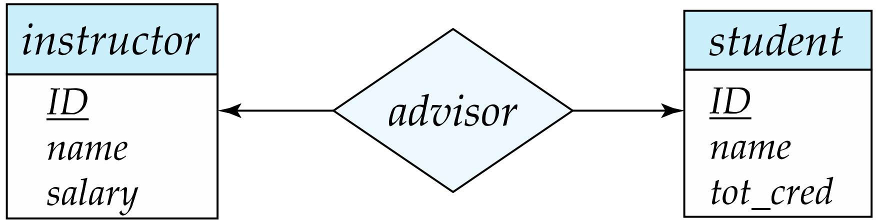
- Similar to many-to-many, represent the relationship with an associative entity.

Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one
  - Many to many

Representing Cardinality Constraints in ER Diagram

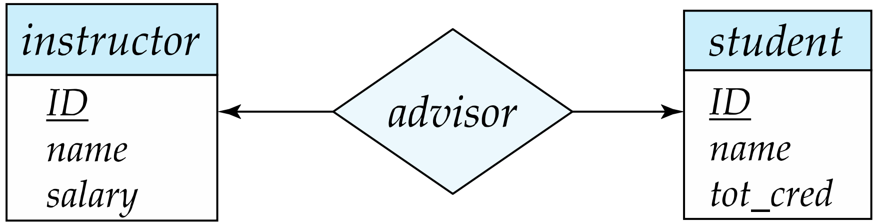
- We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (—), signifying "many," between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student* :
  - A student is associated with at most one *instructor* via the relationship *advisor*
  - A *student* is associated with at most one *department* via *stud_dept*

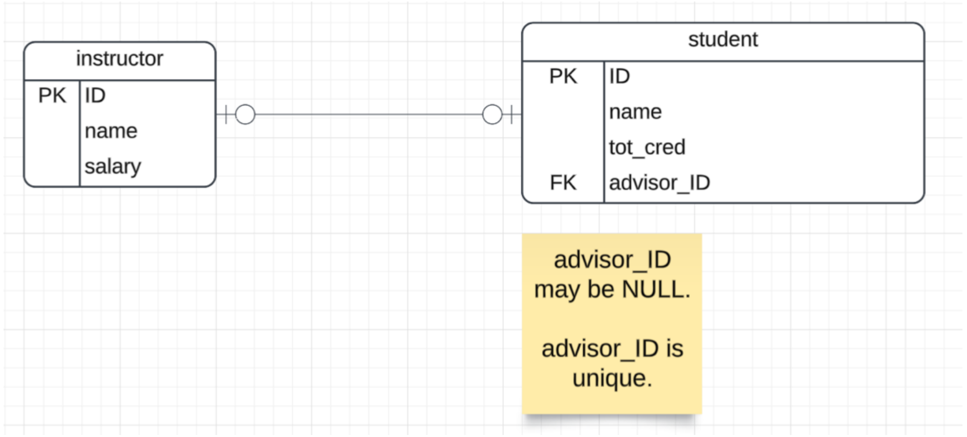


This actually means that

- A student is associated with at most one advisor.
- An instructor advises at most one student.
- *Implies* that the relationship is bi-directional, (instructor, student) come in pairs.

Crow's Foot



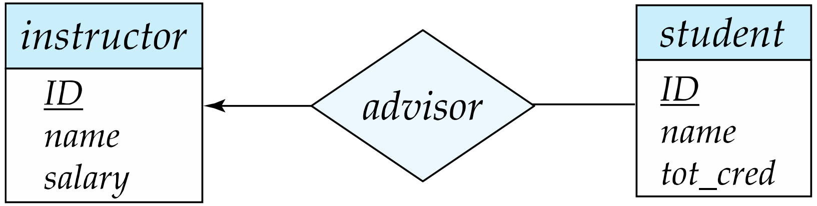


advisor_ID may be NULL.

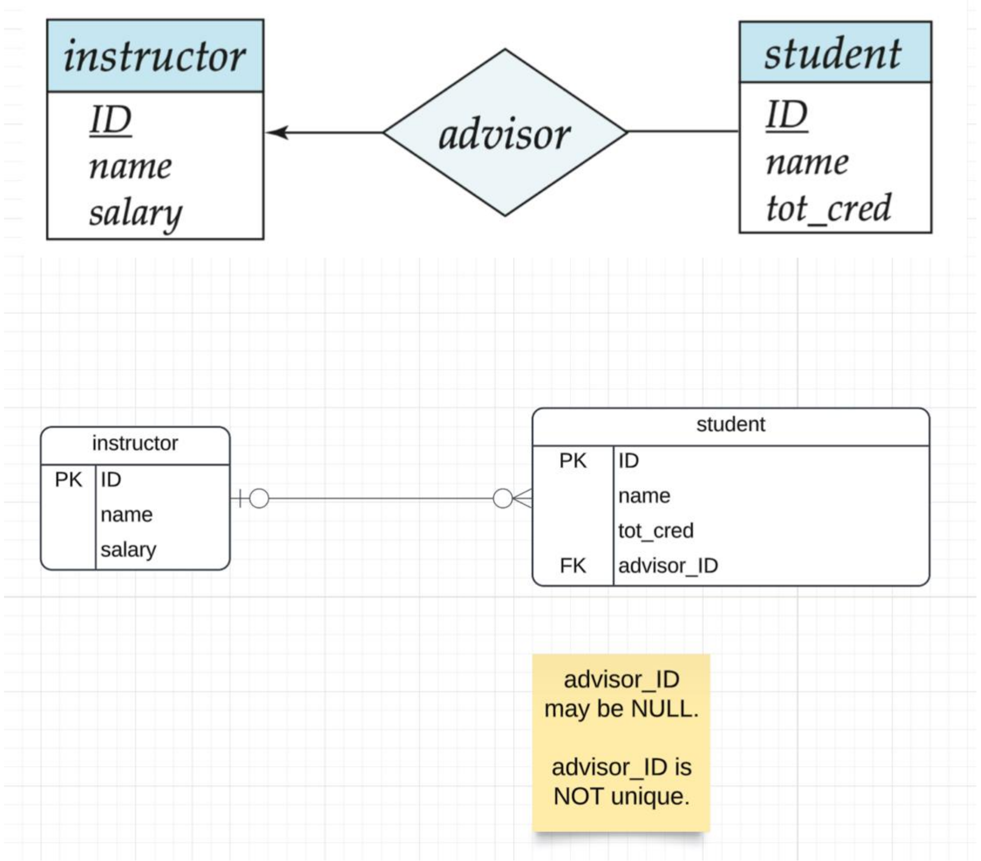
advisor_ID is unique.

One-to-Many Relationship

- one-to-many relationship between an *instructor* and a *student*
  - an instructor is associated with several (including 0) students via *advisor*
  - a student is associated with at most one instructor via advisor,



Crow's Foot

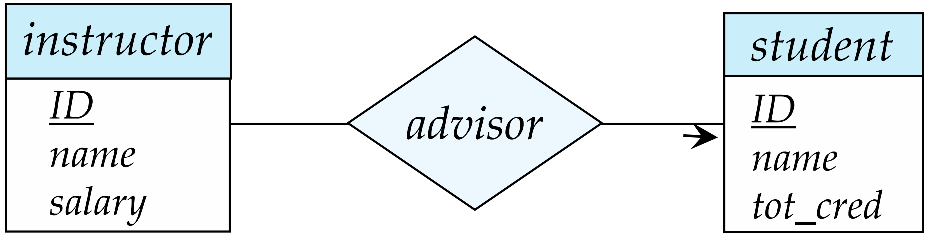


advisor_ID may be NULL.

advisor_ID is NOT unique.

Many-to-One Relationships

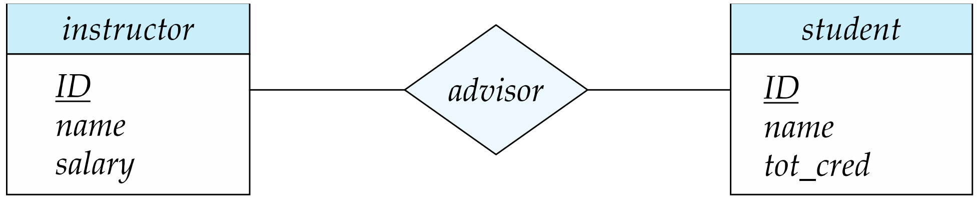
- In a many-to-one relationship between an *instructor* and a *student,*
  - an instructor is associated with at most one student via *advisor*,
  - and a student is associated with several (including 0) instructors via *advisor*



This is basically the same as the previous one, only the FK goes on the other end.

Many-to-Many Relationship

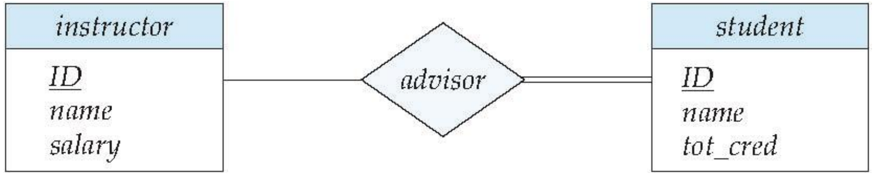
- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*



We handle this with an associative entity.

Total and Partial Participation

- **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set

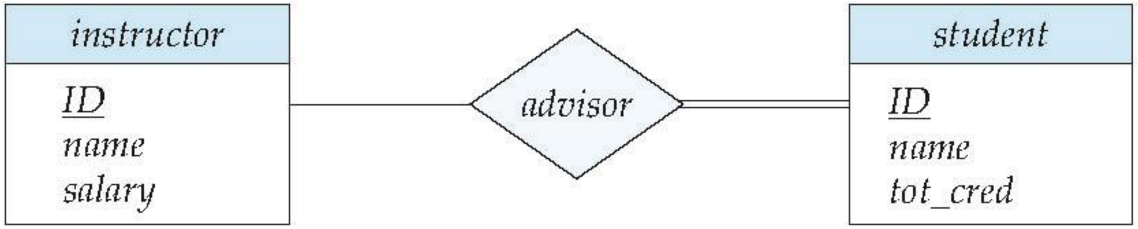


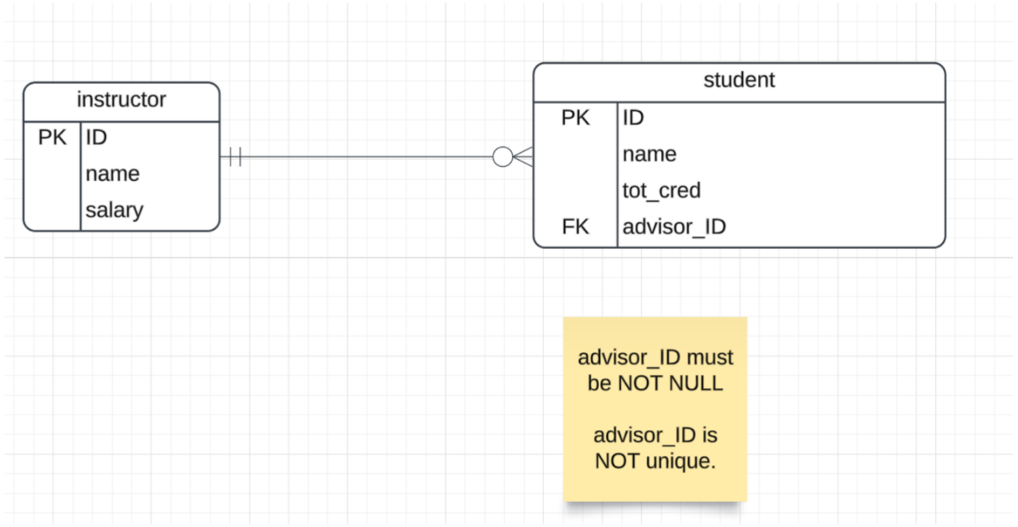
participation of *student* in *advisor r*elation is total

  - every *student* must have an associated instructor

- **Partial participation**: some entities may not participate in any relationship in the relationship set
  - Example: participation of *instructor* in *advisor* is partial

Crow's Foot



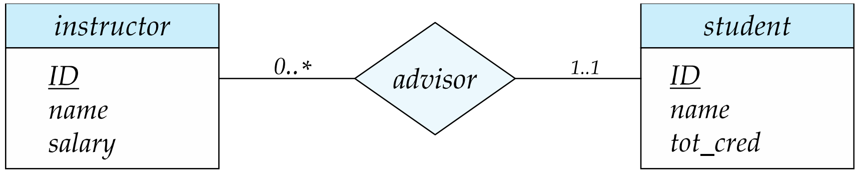


advisor_ID must be NOT NULL

advisor_ID is NOT unique.

Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form *l..h*, where *l* is the minimum and *h* the maximum cardinality
  - A minimum value of 1 indicates total participation.
  - A maximum value of 1 indicates that the entity participates in at most one relationship
  - A maximum value of * indicates no limit.
- Example



  - Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

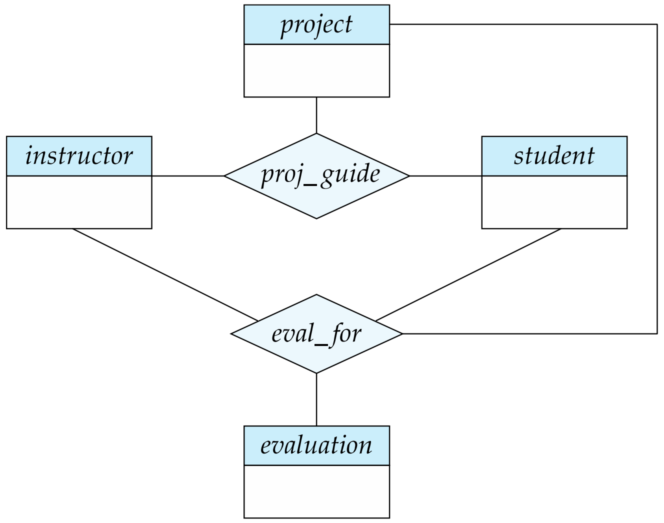
There's no way to do this in Crow's Foot.

Implementing 1..h is tricky in RDBMS and requires and associative entity and/or triggers.

Aggregation

Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project

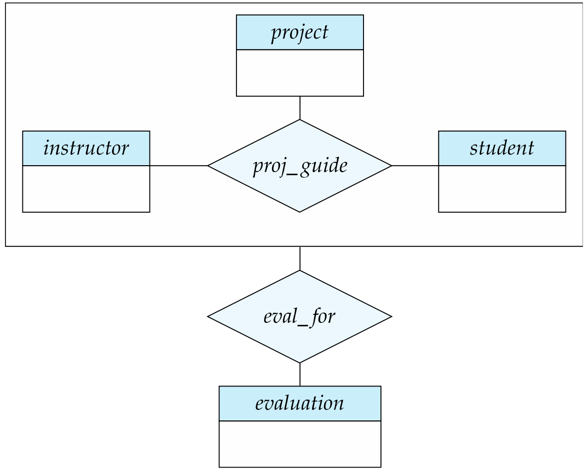


Aggregation (Cont.)

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
  - Every *eval_for* relationship corresponds to a *proj_guide* relationship
  - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
    - So we can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity

Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
  - A student is guided by a particular instructor on a particular project
  - A student, instructor, project combination may have an associated evaluation



The simplest way to handle in relational is an associative entity.

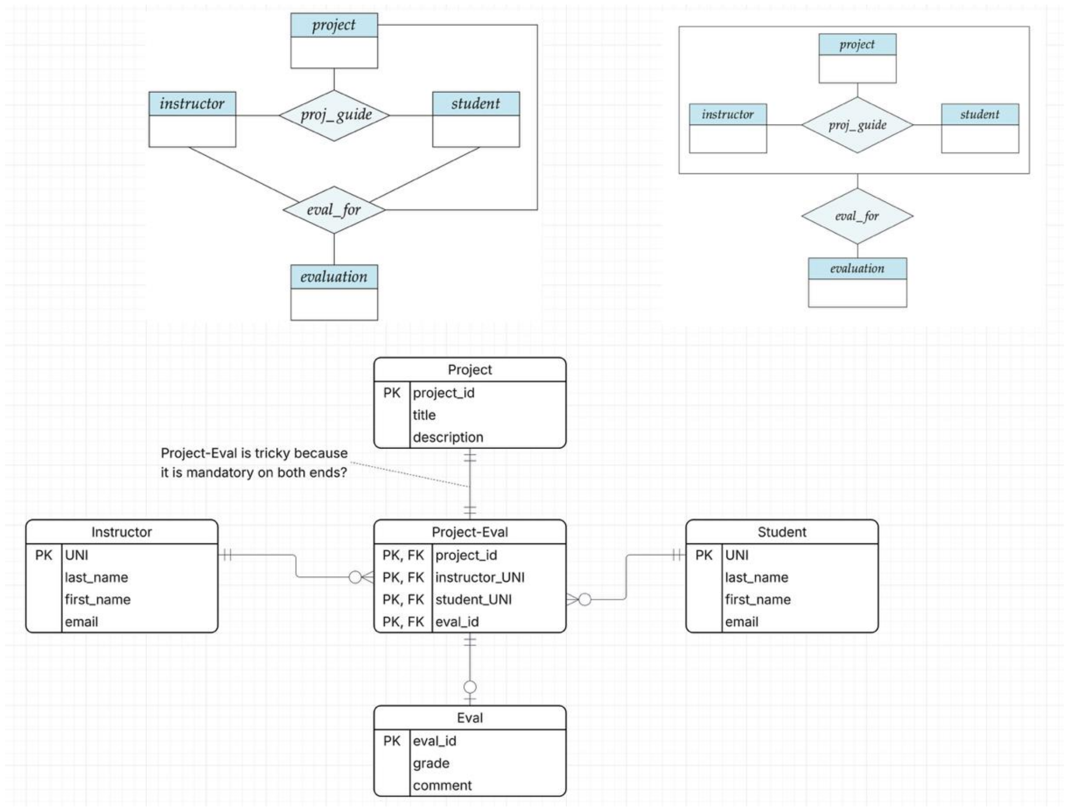
Some thoughts here: https://www.geeksforgeeks.org/aggregate-data-model-in-nosql/

Reduction to Relational Schemas

- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval_for* is:

    eval_for (*s_ID, project_id, i_ID, evaluation_id*)
  - The schema *proj_guide* is redundant.

Let's Do a More Example in Crow's Foot



If I got that right, it is
a miracle.

Normally, you cannot be sure
until you implement
and test it.

Relational Databases/SQL

Load Classic Models

(Show how to load Classic Models in DataGrip)
(Explain how I could also do this from the command line)
(Walk through SQL Script)

Indexes

Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

  create index <name> **on** <relation-name> (attribute);

Index Creation Example

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- The query:

  select *
  from *student*
  where *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

DFF:

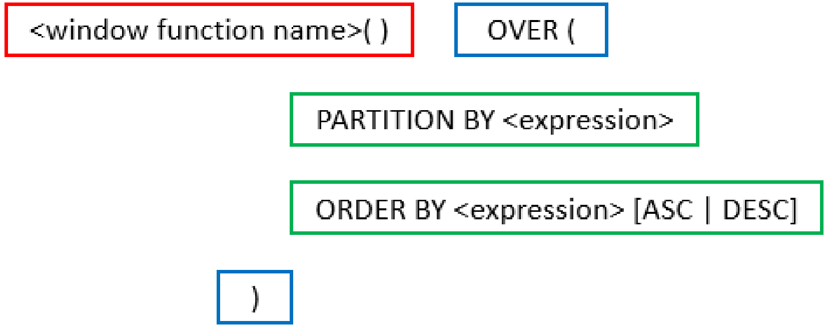
- An index on (column1, column2, column3)
- Is also an index on (column1) and (column1, column2)
- But not (column2), (column3), (column2, column3).
- We will see "why" later in the semester.

Show performance in Notebook

Advanced Windows Functions

Advanced Aggregates and Window Functions

- The aggregation functions in SQL are powerful.
- There are some scenarios that are very difficult. SQL and databases add support for more advanced capabilities:
  - Ranking
  - Windows
  - Pivot, Slice and Dice, but we will cover these with a different technology (OLAP) later in the semester.

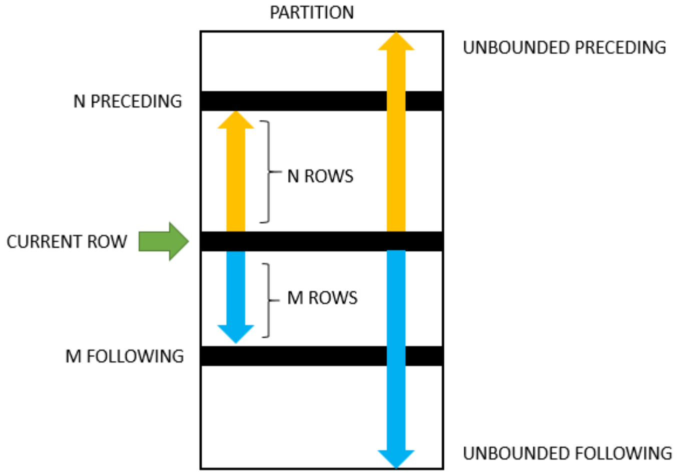


```
<window function name>( )    OVER (

                 PARTITION BY <expression>

                 ORDER BY <expression> [ASC | DESC]

             )
```



  - This is powerful, a little complex and requires trial and error.

Classic Models: Annual Revenue and YoY Growth by Country

```
SELECT
    country,
    YEAR(orderDate) AS year,
    ROUND(SUM(priceEach * quantityOrdered), 2) AS annual_revenue,
    ROUND(
        (SUM(priceEach * quantityOrdered) -
        LAG(SUM(priceEach * quantityOrdered)) OVER (PARTITION BY country ORDER BY YEAR(orderDate))) /
        LAG(SUM(priceEach * quantityOrdered)) OVER (PARTITION BY country ORDER BY YEAR(orderDate)) * 100, 2
    ) AS yoy_growth_percentage
FROM
    customers
    JOIN orders ON customers.customerNumber = orders.customerNumber
    JOIN orderdetails ON orders.orderNumber = orderdetails.orderNumber
GROUP BY
    country, year
ORDER BY
    country, year;
```

Revenue and Growth Quarter to Quarter

```
WITH quarterly_revenue AS (
   SELECT
      YEAR(o.orderDate) AS order_year, QUARTER(o.orderDate) AS order_quarter, SUM(od.quantityOrdered * od.priceEach) AS revenue
   FROM
      orders o JOIN orderdetails od ON o.orderNumber = od.orderNumber WHERE o.status = 'Shipped'
   GROUP BY  YEAR(o.orderDate), QUARTER(o.orderDate)
)

SELECT
   qr.order_year, qr.order_quarter, qr.revenue,
   LAG(qr.revenue) OVER (ORDER BY qr.order_year, qr.order_quarter) AS previous_quarter_revenue,
   ROUND(
      (qr.revenue - LAG(qr.revenue) OVER (ORDER BY qr.order_year, qr.order_quarter))
      / LAG(qr.revenue) OVER (ORDER BY qr.order_year, qr.order_quarter) * 100,
      2
   ) AS qoq_growth_percent
FROM
   quarterly_revenue qr
ORDER BY
   qr.order_year, qr.order_quarter;
```

Summary

- These are powerful and useful capabilities.
- They are part of the SQL 2003 standard.
- Implementation may differ from RDB implementation to RDB implementation.
- I will not ask you questions on exams.
  I will ask questions on homework.
  You can do what I do and ask ChatGPT because I keep forgetting how do to it.

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be **monotonic**. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

This is from the book.
There is a typo in the query.
I cannot figure out what they are trying to accomplish.
The data is not complex enough to justify recursion.
See the following example.

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Iteration Number	*Tuples in c1*
0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done

Classic Models Management Chain

```
WITH RECURSIVE employee_chain AS (
   -- Base case: Start with employees who do not report to anyone (top-level managers)
   SELECT
      employeeNumber AS employee_id, reportsTo AS manager_id, CONCAT(firstName, ' ', lastName) AS employee_name,
         CAST(employeeNumber AS CHAR(100)) AS management_chain
   FROM
      employees
   WHERE
      reportsTo IS NULL
   UNION ALL
   -- Recursive step: Add employees who report to employees already in the chain
   SELECT
      e.employeeNumber AS employee_id, e.reportsTo AS manager_id, CONCAT(e.firstName, ' ', e.lastName) AS employee_name,
         CONCAT(ec.management_chain, ' -> ', e.employeeNumber) AS management_chain
   FROM
      employees e JOIN employee_chain ec
   ON
      e.reportsTo = ec.employee_id
)
SELECT
   employee_id, employee_name, management_chain
FROM
   employee_chain
ORDER BY    management_chain;
```

Recursive Query

Functions, Procedures and Trigger Concepts

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

Note:

- The programming language, runtime and tools for functions, procedures and triggers are not easy to use.
- My view is that calling external functions is an anti-pattern (bad idea).
  - External code degrades the reliability, security and performance of the database.
  - Databases are often mission critical and the heart of environments.

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin** … **end,** {}
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- While and repeat statements:

  -
    ```
    while boolean expression  do
             sequence of statements ;
         end while
    ```

  -
    ```
    repeat
           sequence of statements ;
         until boolean expression
         end repeat
    ```

(Core) Language Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n  integer default 0;
for r  as
      select budget from department
           where dept_name = 'Music'
do
                  set n = n + r.budget
end for
```

Note:

- There are various other looping constructs.

(Core) Language Constructs – if-then-else

- Conditional statements (**if-then-else**)

```
if boolean  expression
        then statement or compound statement
     elseif boolean  expression
        then statement or compound statement
 else statement or compound statement
     end if
```

Note:

- We will not spend a lot of time writing functions, procedures, or triggers.
- The language and development environment are not easy to use.

Functions

Declaring SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
        returns integer
        begin
        declare d_count  integer;
               select count (* ) into d_count
               from instructor
               where instructor.dept_name = dept_name
        return d_count;
end
```

- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))
        returns table  (
          ID varchar(5),
              name varchar(20),
          dept_name varchar(20),
              salary numeric(8,2))
     return table
              (select ID, name, dept_name, salary
               from instructor
               where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
        select *
        from table (instructor_of ('Music'))
```

Procedures

SQL Procedures

- The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20),
                                   out d_count integer)
 begin
     select count(*) into d_count
     from instructor
     where instructor.dept_name = dept_count_proc.dept_name
 end
```

- The keywords **in** and **out** are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

SQL Procedures (Cont.)

- Procedures and functions can be invoked also from dynamic SQL
- SQL allows more than one procedure of the so long as the number of arguments of the procedures with the same name is different.
- The name, along with the number of arguments, is used to identify the procedure.

Triggers

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

X on T

before insert

before update

before delete

after insert

after update

after delete

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as :** for deletes and updates
  - **referencing new row as :** for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
    create trigger setnull_trigger before update of takes
    referencing new row as nrow
    for each row
        when (nrow.grade = ' ')
    begin atomic
            set nrow.grade = null;
end;
```

Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> 'F' **and** *nrow.grade* **is not null**
      and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
  begin atomic
       update *student*
       set *tot_cred*= *tot_cred* +
            (**select** *credits*
             from *course*
             where *course.course_id*= *nrow.course_id*)
       where *student.id* = *nrow.id*;
  end;

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

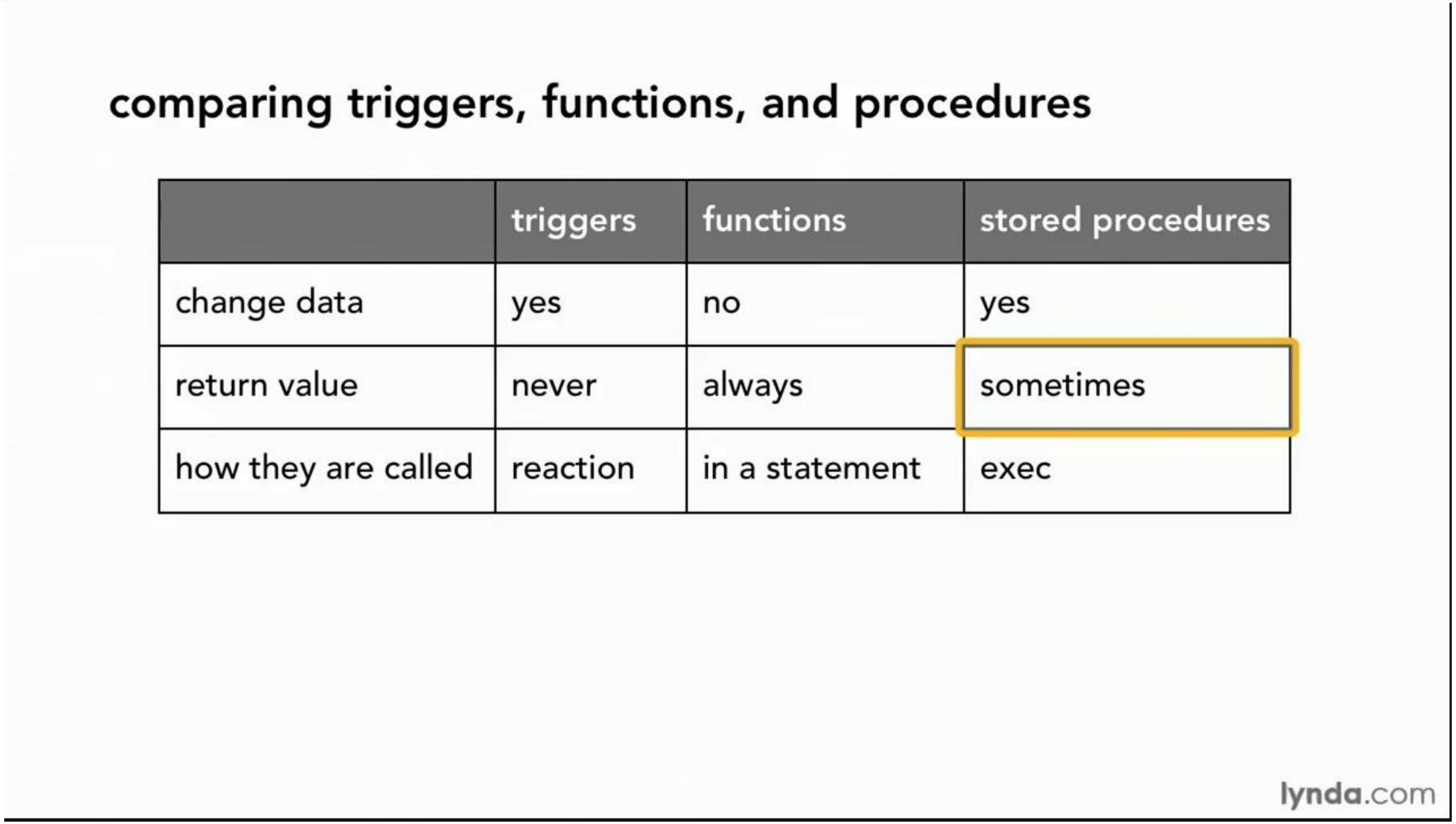
- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Summary

Comparison



comparing triggers, functions, and procedures

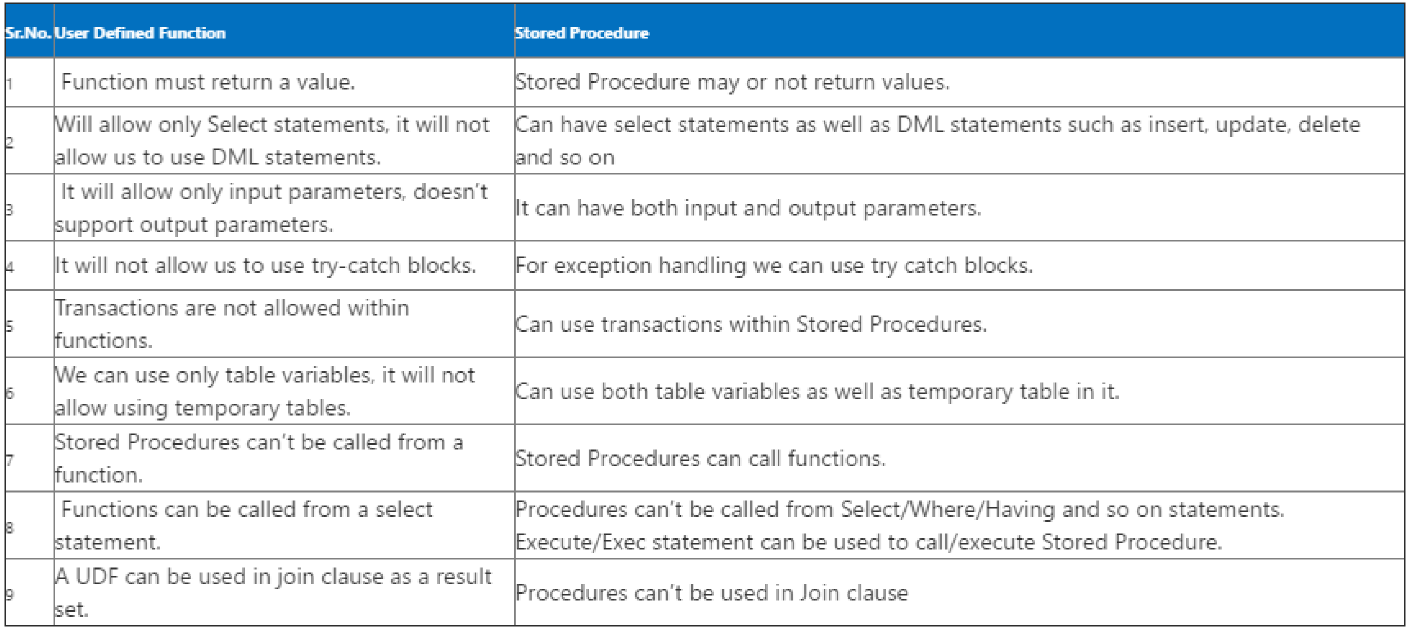
	triggers	functions	stored procedures
change data	yes	no	yes
return value	never	always	sometimes
how they are called	reaction	in a statement	exec

lynda.com

Comparison – Some Details

A *trigger* has capabilities like a procedure, except ...

- You do not call it. The DB engine calls it before or after an INSERT, UPDATE, DELETE.
- The inputs are the list of incoming new, modified rows.
- The outputs are the modified versions of the new or modified rows.



Sr.No.	User Defined Function	Stored Procedure
1	Function must return a value.	Stored Procedure may or not return values.
2	Will allow only Select statements, it will not allow us to use DML statements.	Can have select statements as well as DML statements such as insert, update, delete and so on
3	It will allow only input parameters, doesn't support output parameters.	It can have both input and output parameters.
4	It will not allow us to use try-catch blocks.	For exception handling we can use try catch blocks.
5	Transactions are not allowed within functions.	Can use transactions within Stored Procedures.
6	We can use only table variables, it will not allow using temporary tables.	Can use both table variables as well as temporary table in it.
7	Stored Procedures can't be called from a function.	Stored Procedures can call functions.
8	Functions can be called from a select statement.	Procedures can't be called from Select/Where/Having and so on statements. Execute/Exec statement can be used to call/execute Stored Procedure.
9	A UDF can be used in join clause as a result set.	Procedures can't be used in Join clause

Using a Programming Language

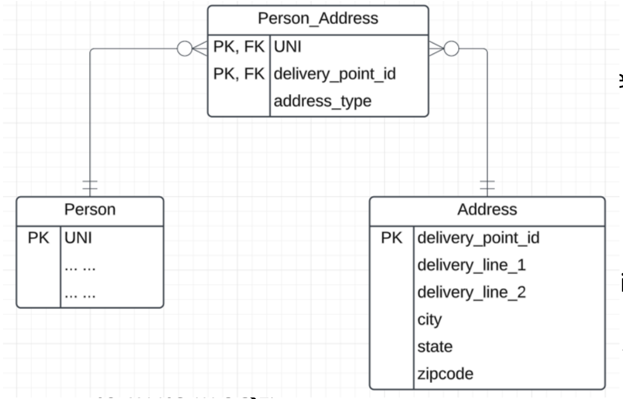
Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language for at least two reasons

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

Entering an address is an example that I use in my cloud computing course.

User Profile

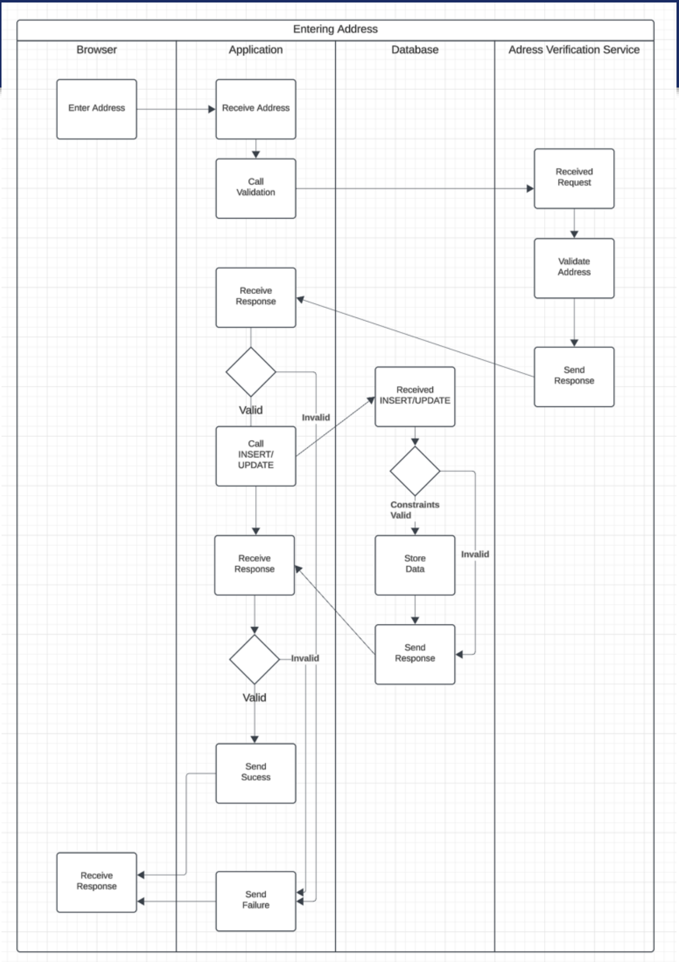


esses:

ion

very hard

- “Logic” in the database should focus on integrity of the data, not general purpose, complex application behavior.



Accessing SQL from a Programming Language (Cont.)

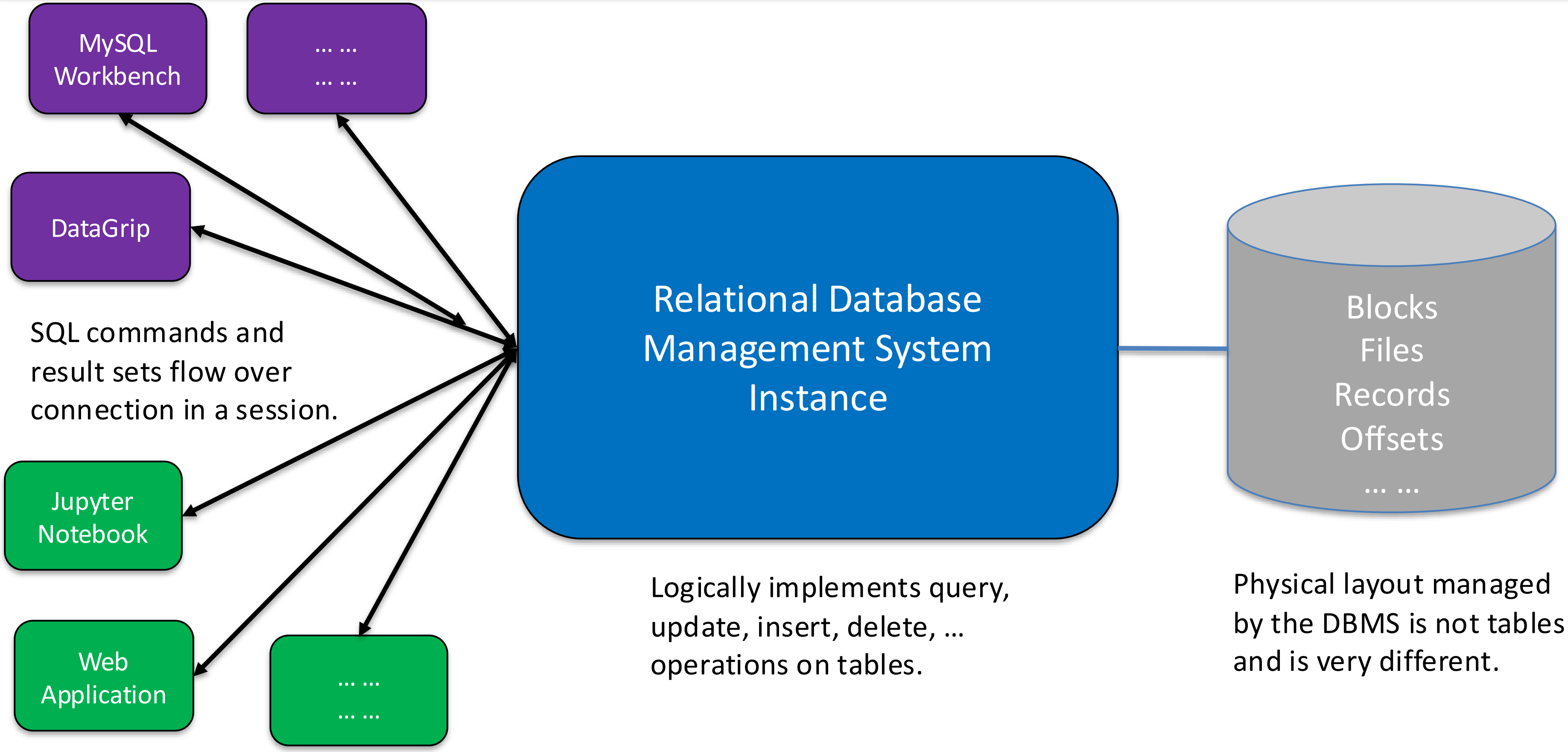
There are two approaches to accessing SQL from a general-purpose programming language

- A general-purpose program -- can connect to and communicate with a database server using a collection of functions
- Embedded SQL -- provides a means by which a program can interact with a database server.
  - The SQL statements are translated at compile time into function calls.
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

Some Terms

- "A database connection is a facility in computer science that allows client software to talk to database server software, whether on the same machine or not. A connection is required to send commands and receive answers, usually in the form of a result set." (https://en.wikipedia.org/wiki/Database_connection)
- Session:
  - "Connection is the relationship between a client and a MySQL database.
    Session is the period of time between a client logging in (connecting to) a MySQL database and the client logging out (exiting) the MySQL database."
    (https://stackoverflow.com/questions/8797724/mysql-concepts-session-vs-connection)
  - "A session is just a result of a successful connection."
- Network protocols are layered. You can think of:
  - Connection as the low-level network connection.
  - Session is the next layer up and has additional information associated with it, e.g. the user.
- Connection libraries like pymysql sometimes blur the distinction.

Concepts

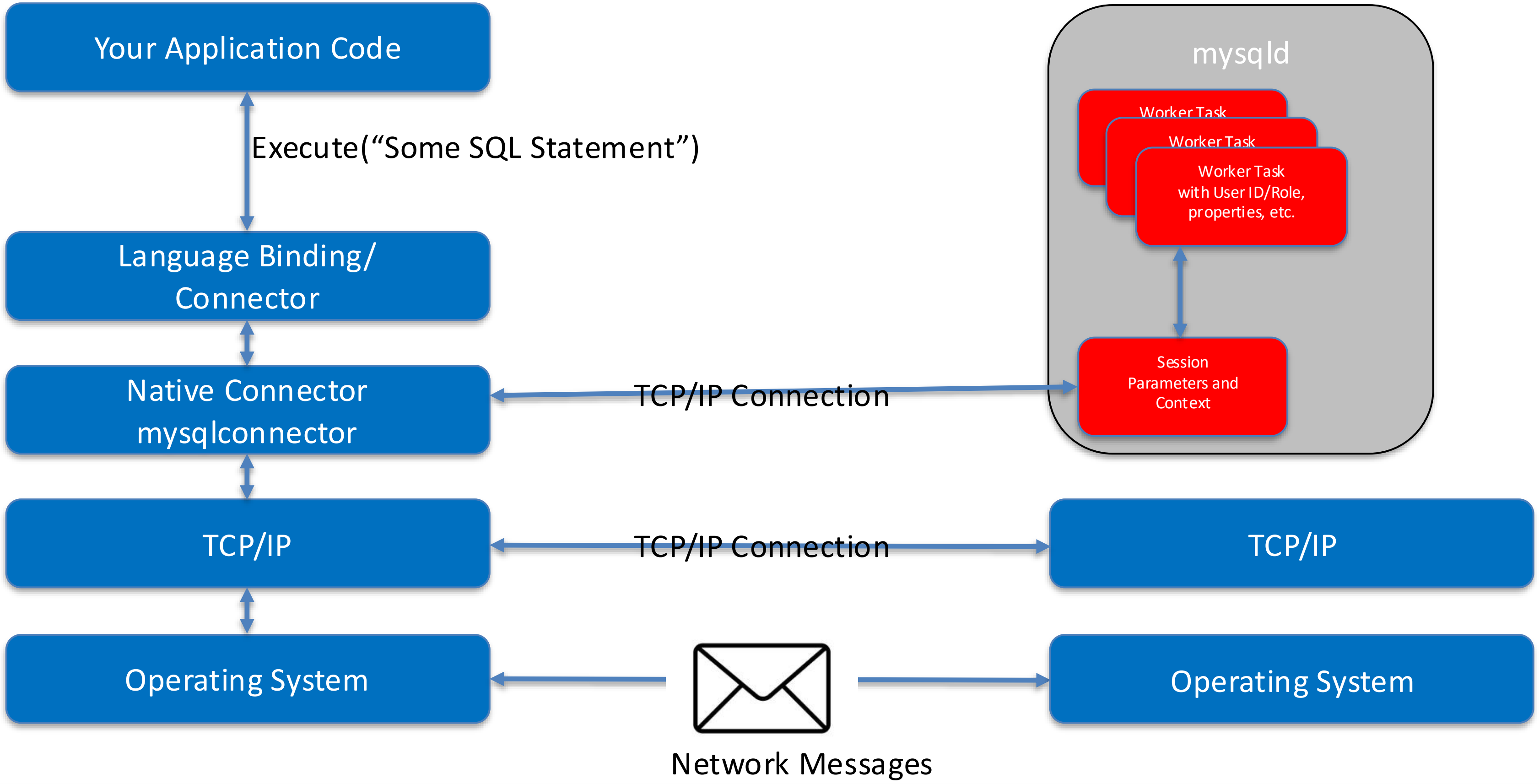


SQL commands and result sets flow over connection in a session.

Logically implements query, update, insert, delete, … operations on tables.

Physical layout managed by the DBMS is not tables and is very different.

Simplistic View


Your Application Code
Execute("Some SQL Statement")
Language Binding/ Connector
Native Connector mysqlconnector
TCP/IP Connection
TCP/IP
TCP/IP Connection
Operating System
Network Messages
mysqld
Worker Task
Worker Task
Worker Task with User ID/Role, properties, etc.
Session Parameters and Context
TCP/IP
Operating System

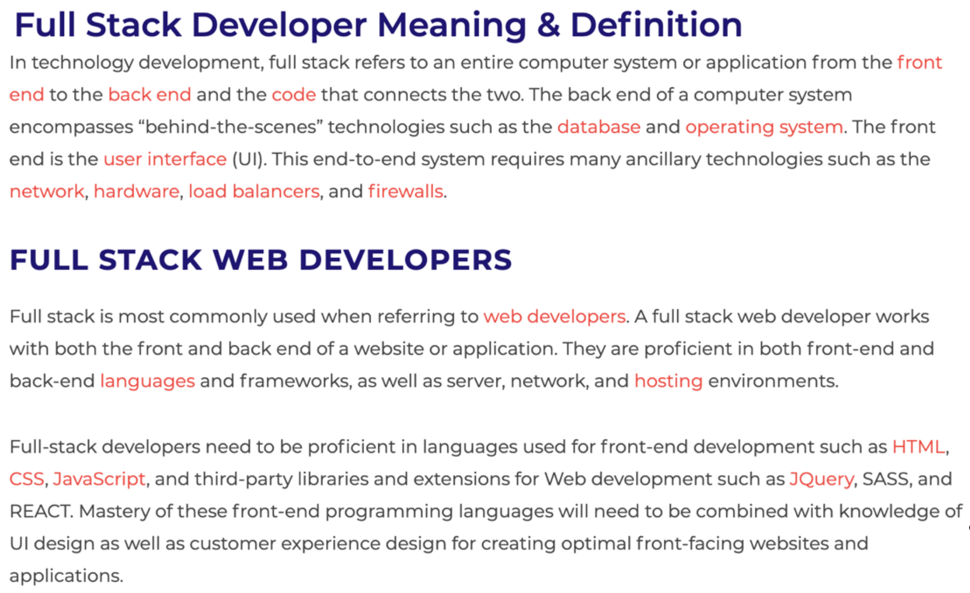
Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a “unit” of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work**. The updates performed by the transaction become permanent in the database.
  - **Rollback work**. All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions

Show connecting
Explain auto-commit

Applications, REST, Data Engineering Project

Full Stack Application



Full Stack Developer Meaning & Definition

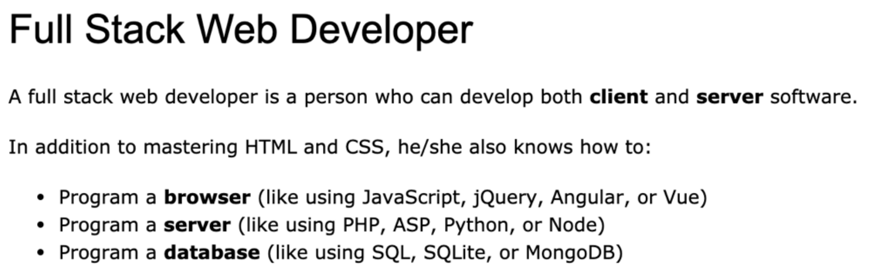
In technology development, full stack refers to an entire computer system or application from the front end to the back end and the code that connects the two. The back end of a computer system encompasses "behind-the-scenes" technologies such as the database and operating system. The front end is the user interface (UI). This end-to-end system requires many ancillary technologies such as the network, hardware, load balancers, and firewalls.

FULL STACK WEB DEVELOPERS

Full stack is most commonly used when referring to web developers. A full stack web developer works with both the front and back end of a website or application. They are proficient in both front-end and back-end languages and frameworks, as well as server, network, and hosting environments.

Full-stack developers need to be proficient in languages used for front-end development such as HTML, CSS, JavaScript, and third-party libraries and extensions for Web development such as JQuery, SASS, and REACT. Mastery of these front-end programming languages will need to be combined with knowledge of UI design as well as customer experience design for creating optimal front-facing websites and applications.

https://www.webopedia.com/definitions/full-stack/



Full Stack Web Developer

A full stack web developer is a person who can develop both **client** and **server** software.

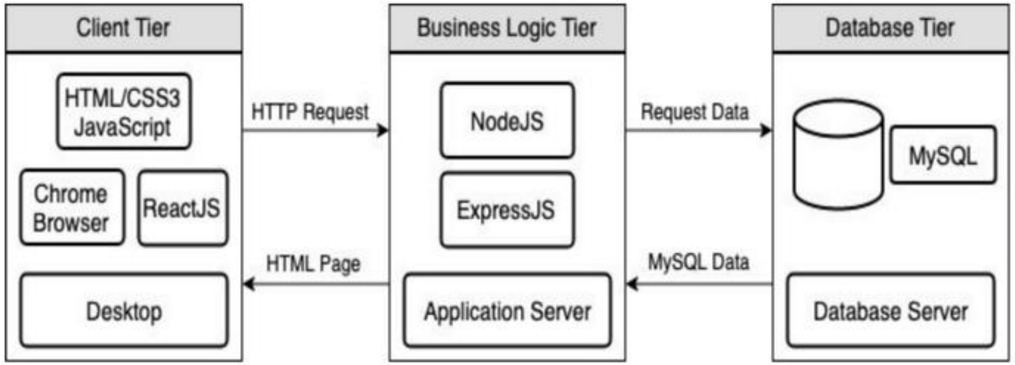
In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (like using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (like using PHP, ASP, Python, or Node)
- Program a **database** (like using SQL, SQLite, or MongoDB)

https://www.w3schools.com/whatis/whatis_fullstack.asp

- There are courses that cover topics:
  - COMS W4153: Advanced Software Engineering
  - COMS W4111: Introduction to Databases
  - COMS W4170 - User Interface Design
- This course will focus on cloud realization, microservices and application patterns, ...
- Also, I am not great at UIs ... ... We will not emphasize or require a lot of UI work.

Full Stack Web Application



M	=	Mongo
E	=	Express
R	=	React
N	=	Node

I start with FastAPI and MySQL, but all the concepts are the same.

https://levelup.gitconnected.com/a-complete-guide-build-a-scalable-3-tier-architecture-with-mern-stack-es6-ca129d7df805

- My preferences are to replace React with Angular, and Node with FastAPI.
- There are three projects to design, develop, test, deploy, ...
  1. Browser UI application.
  2. Microservice.
  3. Database.
- The programming track will implement a simple web application to access IMDB and Game of Thrones data.

Some Terms

- A web application server or web application framework: “A web framework (WF) or web application framework (WAF) is a software framework that is designed to support the development of web applications including web services, web resources, and web APIs. Web frameworks provide a standard way to build and deploy web applications on the World Wide Web. Web frameworks aim to automate the overhead associated with common activities performed in web development. For example, many web frameworks provide libraries for database access, templating frameworks, and session management, and they often promote code reuse.” (https://en.wikipedia.org/wiki/Web_framework)

- REST: “REST (Representational State Transfer) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of a distributed, Internet-scale hypermedia system, such as the Web, should behave. The REST architectural style emphasises uniform interfaces, independent deployment of components, the scalability of interactions between them, and creating a layered architecture to promote caching to reduce user-perceived latency, enforce security, and encapsulate legacy systems.[1]

  REST has been employed throughout the software industry to create stateless, reliable web-based applications.” (https://en.wikipedia.org/wiki/REST)

Some Terms

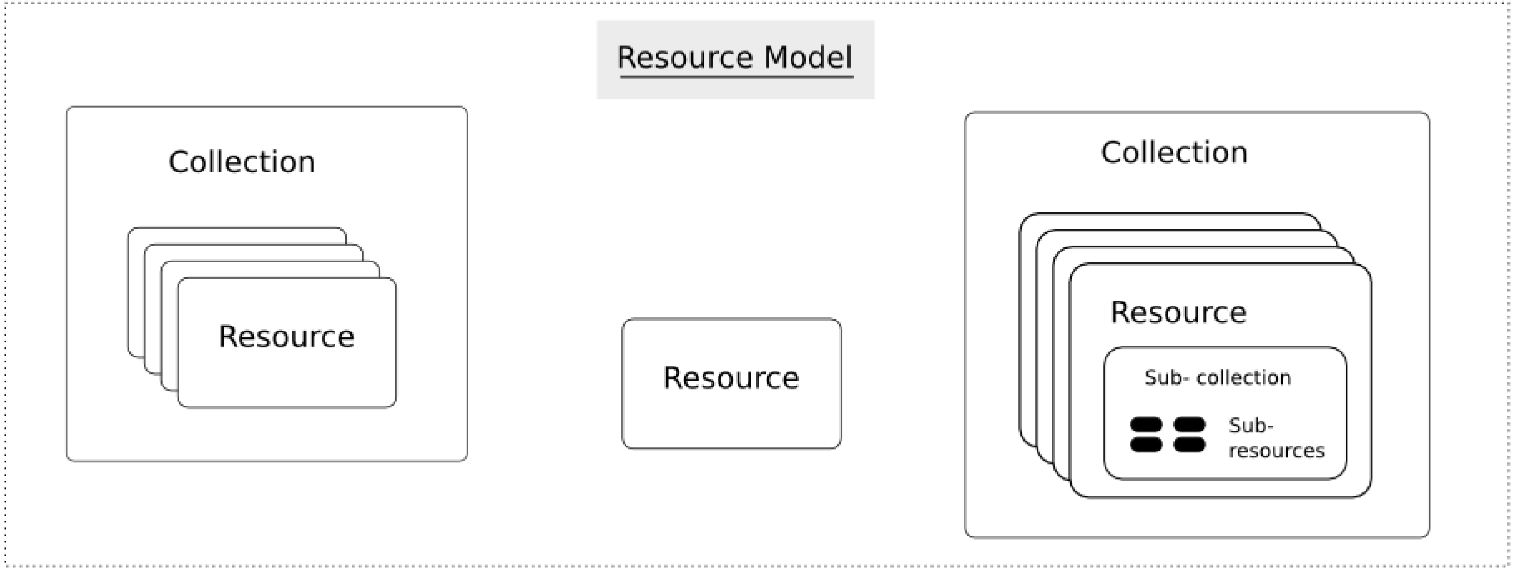
- OpenAPI: “The OpenAPI Specification, previously known as the Swagger Specification, is a specification for a machine-readable interface definition language for describing, producing, consuming and visualizing web services.” (https://en.wikipedia.org/wiki/OpenAPI_Specification)

- Model: “A model represents an entity of our application domain with an associated type.” (https://medium.com/@nicola88/your-first-openapi-document-part-ii-data-model-52ee1d6503e0)

- Routers: “What fastapi docs says about routers: If you are building an application or a web API, it’s rarely the case that you can put everything on a single file. FastAPI provides a convenience tool to structure your application while keeping all the flexibility.” (https://medium.com/@rushikeshnaik779/routers-in-fastapi-tutorial-2-adf3e505fdca)

- Summary:
  - These are general concepts, and we will go into more detail in the semester.
  - FastAPI is a specific technology for Python.
  - There are many other frameworks applicable to Python, NodeJS/TypeScript, Go, C#, Java, … …
  - They all surface similar concepts with slightly different names.

REST and Resources



A Collection with Resources

A Singleton Resource

Sub-collections and Sub-resources

REST (https://restfulapi.net/)

What is REST

- REST is acronym for **RE**presentational **S**tate **T**ransfer. It is architectural style for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000 in his famous dissertation.
- Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below.

Guiding Principles of REST

- **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
- **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
- **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.
- **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

Resources

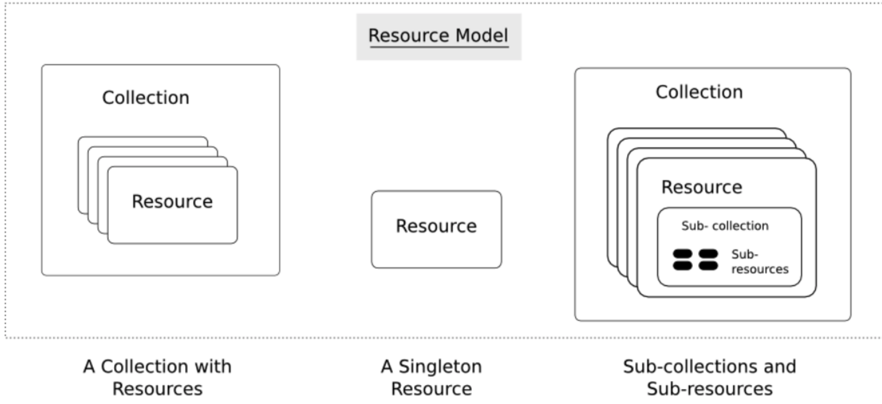
Resources are an abstraction. The application maps to create things and actions.

"A resource-oriented API is generally modeled as a resource hierarchy, where each node is either a *simple resource* or a *collection resource*. For convenience, they are often called a resource and a collection, respectively.

- A collection contains a list of resources of **the same type**. For example, a user has a collection of contacts.
- A resource has some state and zero or more sub-resources. Each sub-resource can be either a simple resource or a collection resource.

For example, Gmail API has a collection of users, each user has a collection of messages, a collection of threads, a collection of labels, a profile resource, and several setting resources.

While there is some conceptual alignment between storage systems and REST APIs, a service with a resource-oriented API is not necessarily a database, and has enormous flexibility in how it interprets resources and methods. For example, creating a calendar event (resource) may create additional events for attendees, send email invitations to attendees, reserve conference rooms, and update video conference schedules. (Emphasis added)
(https://cloud.google.com/apis/design/resources#resources)



https://restful-api-design.readthedocs.io/en/latest/resources.html

REST – Resource Oriented

- When writing applications, we are used to writing functions or methods:
  - openAccount(last_name, first_name, tax_payer_id)
  - account.deposit(deposit_amount)
  - account.close()

  We can create and implement whatever functions we need.
- REST only allows four methods:
  - POST: Create a resource
  - GET: Retrieve a resource
  - PUT: Update a resource
  - DELETE: Delete a resource

  That's it. That's all you get.

"The key characteristic of a resource-oriented API is that it emphasizes resources (data model) over the methods performed on the resources (functionality). A typical resource-oriented API exposes a large number of resources with a small number of methods." (https://cloud.google.com/apis/design/resources)

- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

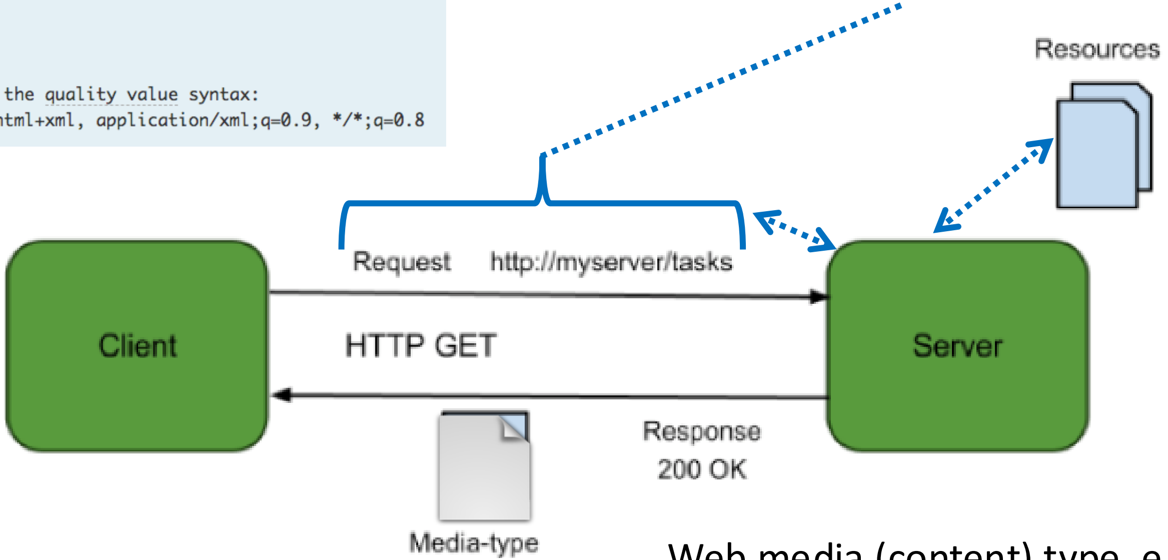
Resources, URLs, Content Types

Accept type in headers.

```
Accept: <MIME_type>/<MIME_subtype>
Accept: <MIME_type>/*
Accept: */*

// Multiple types, weighted with the quality value syntax:
Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8
```

- Relative URL identifies "resource" on the server.
- Server implementation maps abstract resource to tangible "thing," file, DB row, … and any application logic.



Client may be

Browser

Mobile device

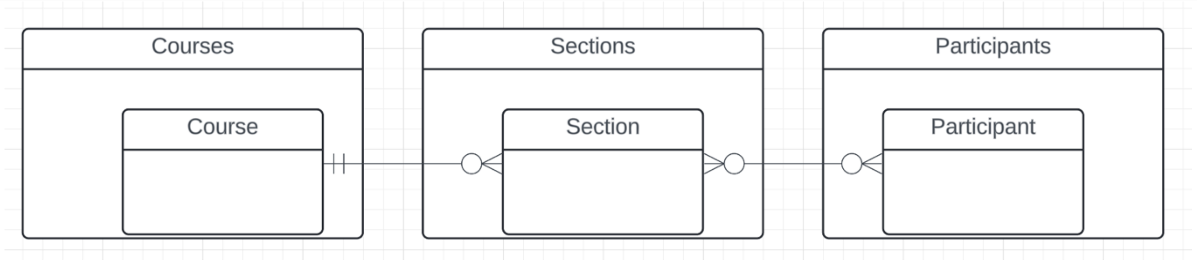
Other REST Service

… …

Web media (content) type, e.g.

- text/html
- application/json

Resources and APIs



- Base resources, paths and methods:
  - /courses: GET, POST
  - /courses/<id>: GET, PUT, DELETE
  - /sections: GET, POST
  - /sections/<id>: GET, PUT, DELETE
  - /participants: GET, POST
  - /participants/<id>: GET, PUT, DELETE
- There are relative, navigation paths:
  - /courses/<id>/sections
  - /participants/<id>/sections
  - etc.
- GET on resources that are collections may also have query parameters.

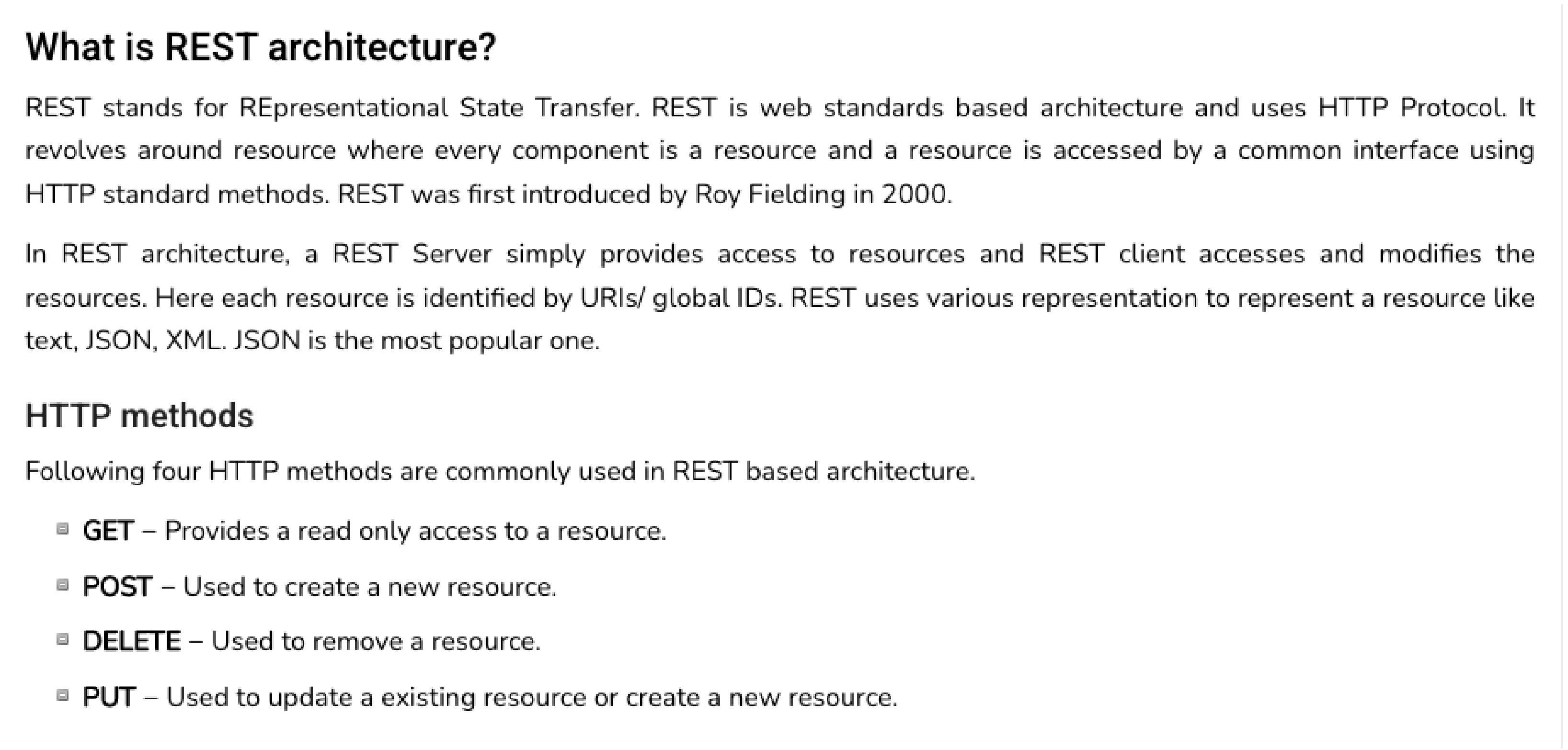
- There are two approaches to defining API
  - Start with OpenAPI document and produce an implementation template.
  - Start with annotated code and generate API document.
- In either approach, I start with *models.*
- Also,
  - I lack the security permission to update CourseWorks.
  - I can choose to not surface the methods or raise and exception.

Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains.***
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...
  - Host/database/table/pk

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)



What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

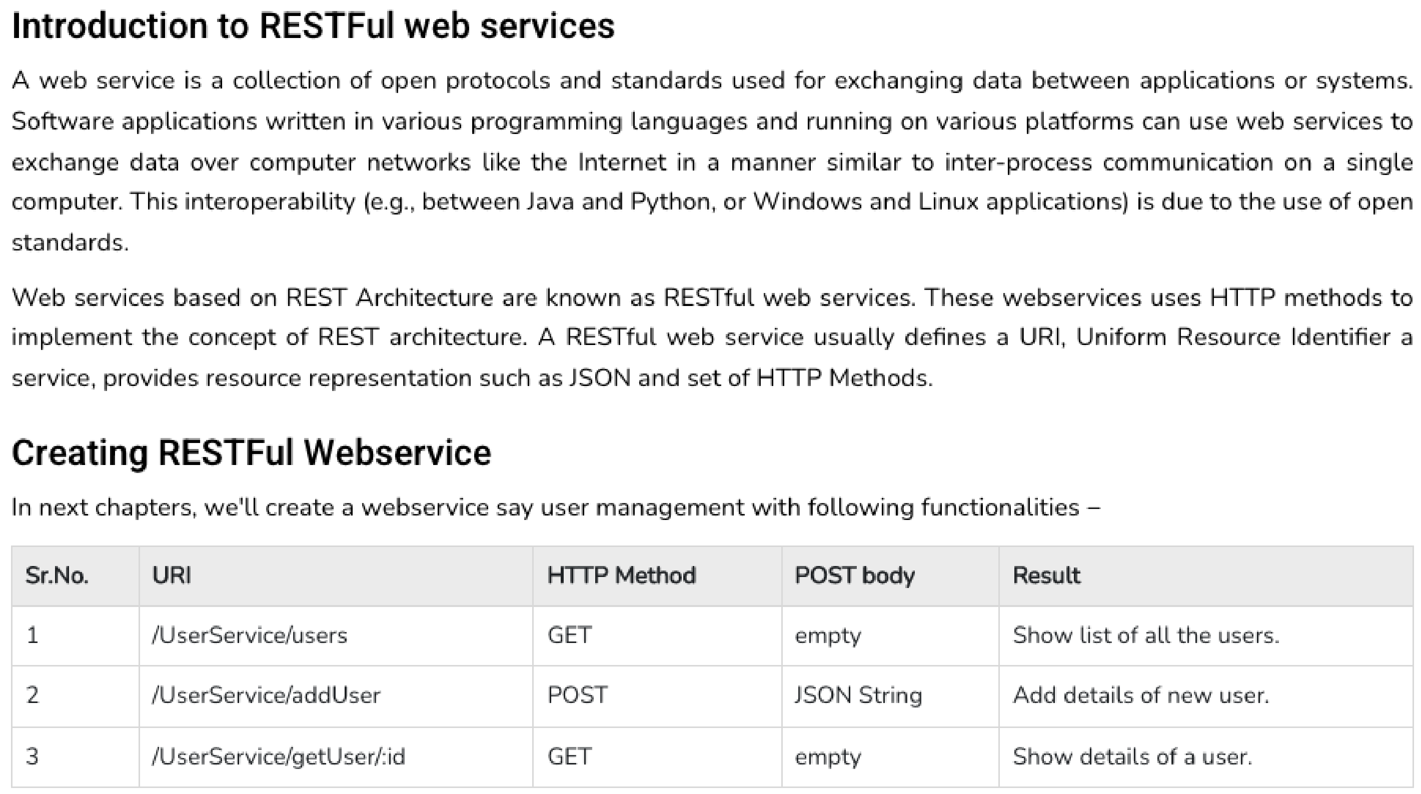
In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)



Introduction to RESTFul web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

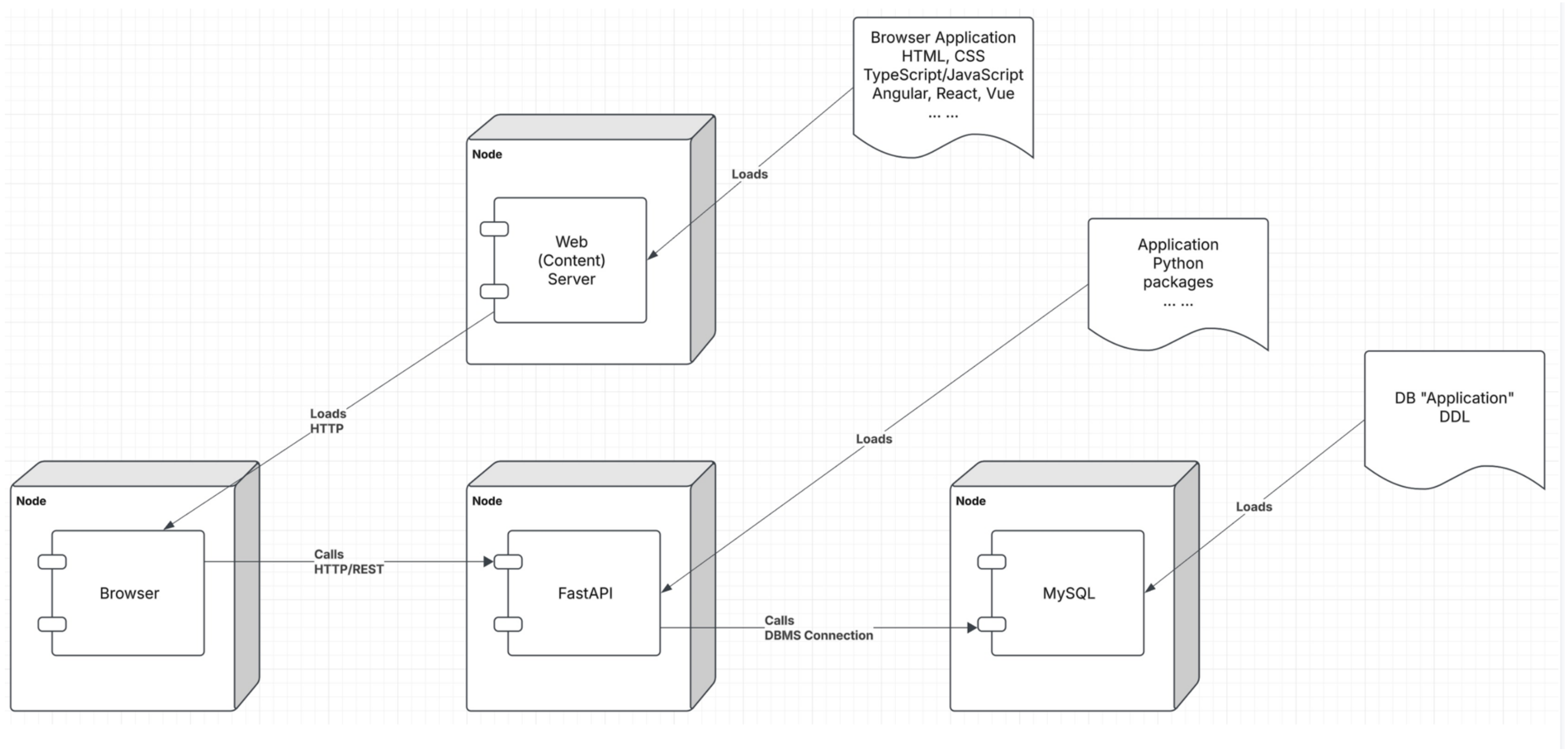
Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

Creating RESTFul Webservice

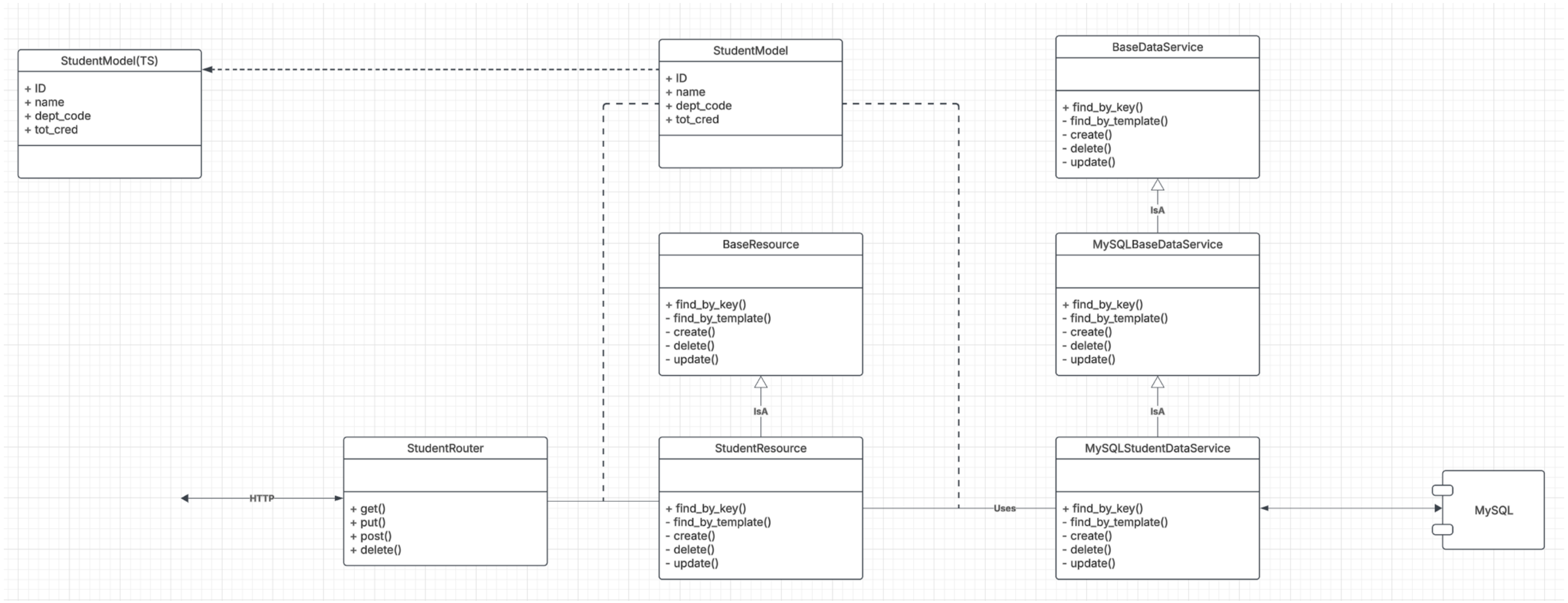
In next chapters, we'll create a webservice say user management with following functionalities –

Sr.No.	URI	HTTP Method	POST body	Result
1	/UserService/users	GET	empty	Show list of all the users.
2	/UserService/addUser	POST	JSON String	Add details of new user.
3	/UserService/getUser/:id	GET	empty	Show details of a user.

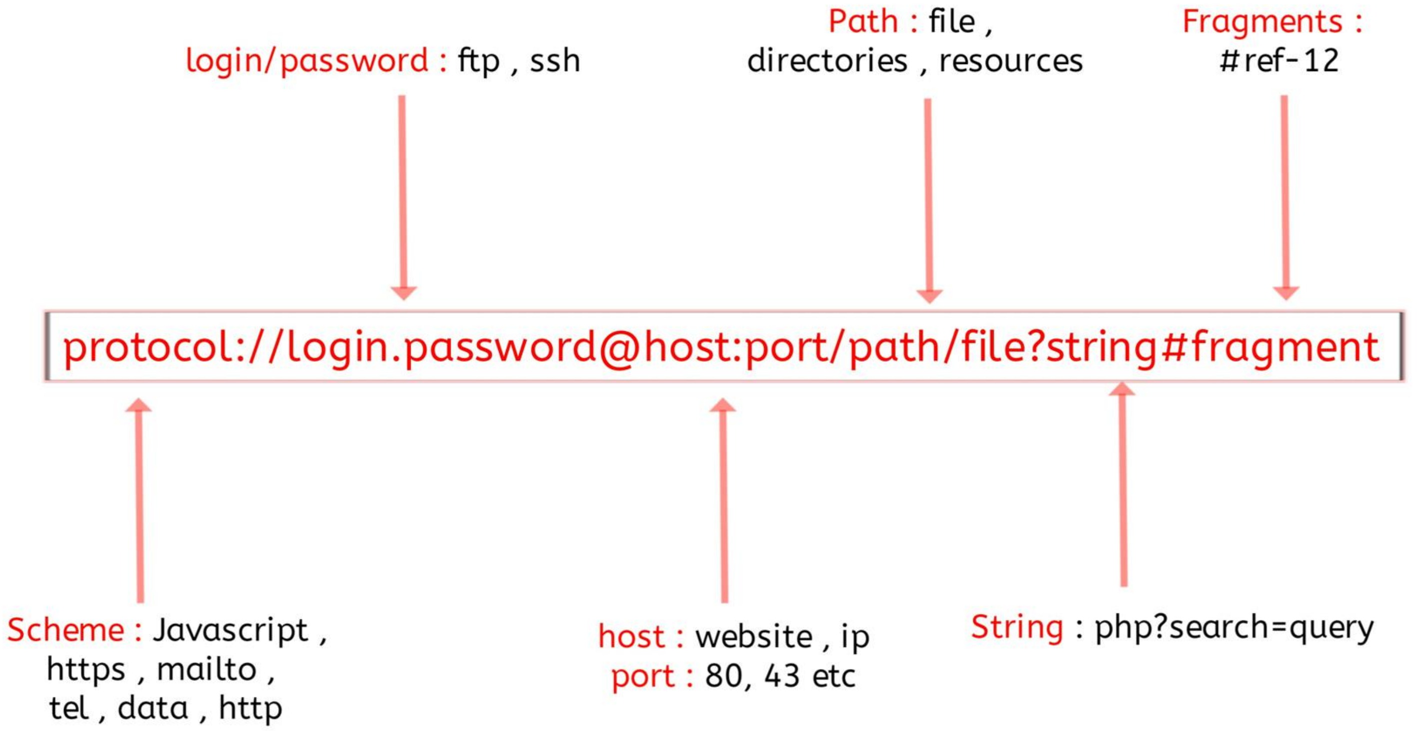
Let’s Look at the Programming Project



Let’s Look at the Programming Project



URLs


login/password : ftp , ssh
Path : file , directories , resources
Fragments : #ref-12
protocol://login.password@host:port/path/file?string#fragment
Scheme : Javascript , https , mailto , tel , data , http
host : website , ip
port : 80, 43 etc
String : php?search=query

Simplistic, Conceptual Mapping (Examples)

REST Method	Resource Path	Relational Operation	DB Resource
DELETE	/people	DROP TABLE	people table
POST	/people	INSERT INTO PEOPLE (...) VALUES(...)	people table people row

GET	/people/21	SHOW KEYS FROM people ...;
		
		SELECT * FROM people WHERE
		playerID= 21
			people row

GET	/people/21/batting	SELECT batting.* FROM
		people JOIN batting USING(playerID)
		WHERE playerID=21	

GET	/people/21/batting/2004_1	SELECT batting.* FROM
		people JOIN batting USING(playerID)
		WHERE playerID=21
		AND yearID=2004 AND stint=1	

Application Architecture

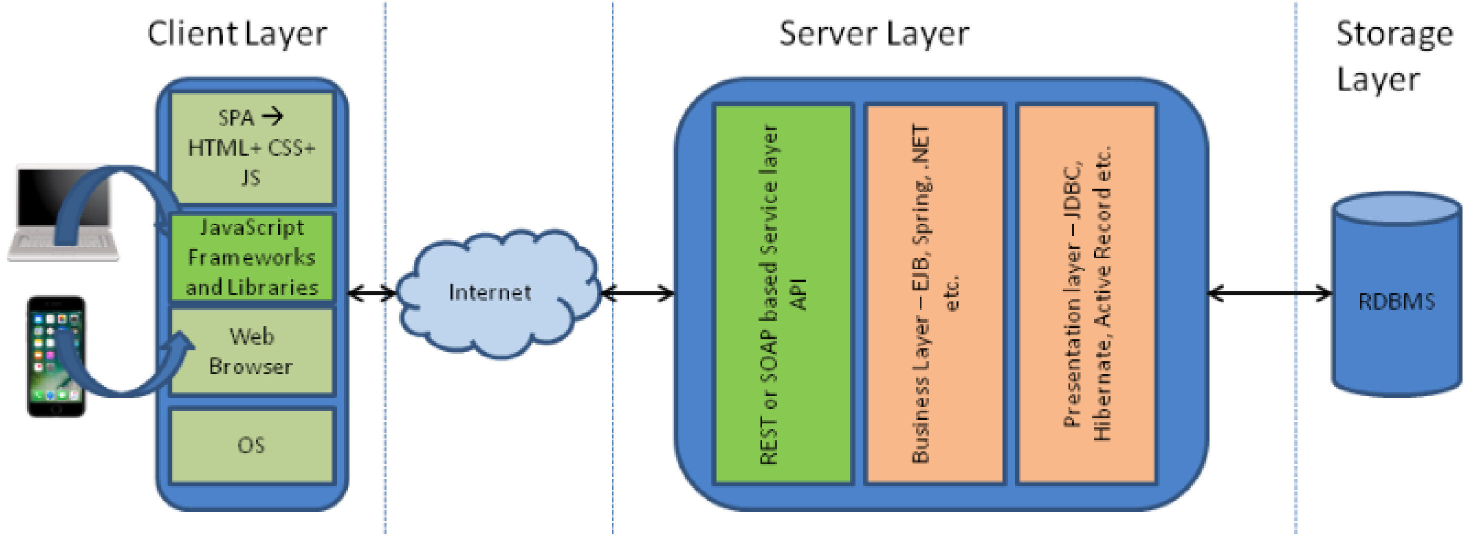

Client Layer
Server Layer
Storage Layer
SPA → HTML+ CSS+ JS
JavaScript Frameworks and Libraries
Web Browser
OS
Internet
REST or SOAP based Service layer API
Business Layer – EJB, Spring, .NET etc.
Presentation layer – JDBC, Hibernate, Active Record etc.
RDBMS

Diagram 2: The moving of the Web Layer from the Server to the Client

Code

- /Users/donald.ferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
- /Users/donald.ferguson/Dropbox/000/000-A-Current-Examples/current-dashboard