CAS CS 411
Lec 6

Data for Web Apps

1. Data and the web
   a. Nearly every web application is backed by a database of some sort
   b. Even static pages might have data requirements
   c. Web-based languages have become optimized for aggregating data into HTML
   d. It isn't just dynamic data that we are interested in…
2. Three kinds of data
   a. Generally speaking we can group data into three buckets
      i. User data: Things that are specific to each user, such as locale, preferences…
      ii. Common data / content: Items that are present for all users
      iii. Infrastructure data: Data required to operate the site
3. Four data activities
   a. We often refer to data operations as implementing CRUD:
      i. Create
      ii. Read
      iii. Update
      iv. Delete
   b. These operations (with the possible exception of Read) need to atomic
4. Atomicity
   a. We almost never build an app that has only one user
   b. We need to account for CRUD operations in which several users might need to access the same data
   c. The problem is that in modern operating systems, processes and threads are constantly switching, often in mid-instruction
   d. Further, a data operation might take several steps…this is called a transaction
   e. How can we guarantee that once a process has started to update data in a table, it will complete before another process starts to update the same data?
5. Locking
   a. Most databases implement some sort of locking
   b. The idea is that the db is locked when one thread or process is working on a dataset, preventing other threads from changing the data midstream
   c. Locks can be either at the table level or row level
   d. It's important to know which it is…it makes a difference in how you write db access code on the app side

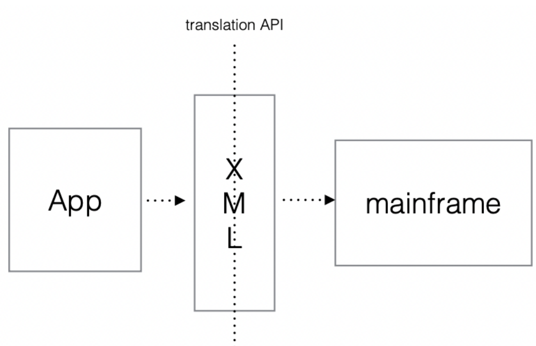
6. ACID
    a. In most cases data encompasses information that we want to keep track of
    b. When we think of transactions in a database, we can implement a set of properties formalized by Jim Gray back in the 70s
    c. We extend the notion of atomicity…
        i. A: Atomicity — all-or-nothing transactions
        ii. C: Consistency — any transaction will result in the database being in a valid state
        iii. I: Isolation — if transactions are done concurrently, the result is the same state that would have been reached had the transactions been done serially
        iv. D: Durability — when a transaction has been committed to the database, it will remain in the dataset until it is updated by another transaction, even if the power is lost
    d. This applies primarily to traditional data sources such as relational databases
    e. We'll see that it isn't always strictly necessary to implement ACID transactions
7. Traditional data sources: mainframes
    a. Mainframes such as IBM's OS/360 series and z/OS machines are optimized for data
    b. Even though we tend to think of mainframes as old-school, there are still a huge number deployed
    c. We typically see these in financial services, insurance, health care, and related industries
    d. The applications that sit on top of these data sources are tightly coupled and custom-built for each client
8. Fixed-width (or fixed-length) data
    a. It's common to find data sources that expect data to be in fixed-width format
    b. This is common in mainframe applications (AS/400 etc)
    c. The format originated with punch cards
    d. In fixed-width, information fits into a certain number of characters
    e. For example, the first 5 characters might indicate the record type, the next 5 characters are a transaction ID, the next 4 a key, and so on
9. Delimited records (EDI)
    a. EDI is Electronic Data Interchange
    b. Many industries move data back and forth in fixed-format message
    c. There are several EDI standards (X12, EDIFACT, and so on)
    d. Healthcare, government, finance, transportation / shipping, supply chain all use EDI
    e. Each field in the record is delimited by one or more special characters

10. EDIFACT sample message

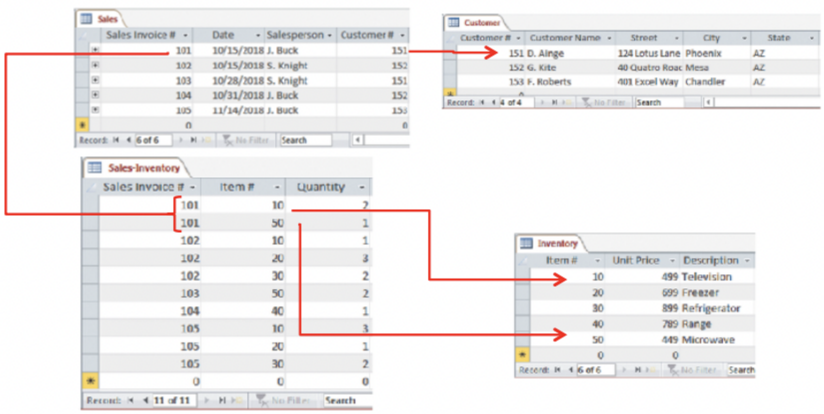
    a.

```
UNA:+.? '
UNB+IATB:1+6XPPC+LHPPC+940101:0950+1'
UNH+1+PAORES:93:1:IA'
MSG+1:45'
IFT+3+XYZCOMPANY AVAILABILITY'
ERC+A7V:1:AMD'
IFT+3+NO MORE FLIGHTS'
ODI'
TVL+240493:1000::1220+FRA+JFK+DL+400+C'
PDI++C:3+Y::3+F::1'
APD+74C:0:::6++++++6X'
TVL+240493:1740::2030+JFK+MIA+DL+081+C'
PDI++C:4'
APD+EM2:0:1630::6+++++++DA'
UNT+13+1'
UNZ+1+1'
```

11. Parsing EDI
    a. Typically a commercial library will be used to parse EDI messages into objects or other data formats
    b. Reading and writing EDI messages conforms to strict error- and formatchecking
    c. It isn't unusual for a batch of EDI messages to be FTPd to a destination rather than be sent in real time
    d. However!
    e. There's increasing demand to provide app- and web- based access to this data
    f. It's not cost effective to simply replace the mainframe…there's a large investment and they actually are extremely good at processing data
    g. The solution is usually middleware that sits between the client application and the mainframe
    h.



    i. From the app's point of view, all data operations are accomplished via XML requests and response
    j. This is a pretty common pattern
    k. It allows us to aggregate mainframe data operations with other data sources…i.e. the app requests user accounts data, and the account information comes from a traditional database but the account balances come from the mainframe

12. Traditional data sources: RDB
    a. An RDB is a relational database such as Oracle, DB2, or mySQL
    b. In RDBs data is stored in tables
    c. Each table contains a minimal amount of information and is related to other tables by a key
    d. For example, an Employee table and a Department table might be related by an EmployeeID
    e. Key values appear in each related table
    f. 

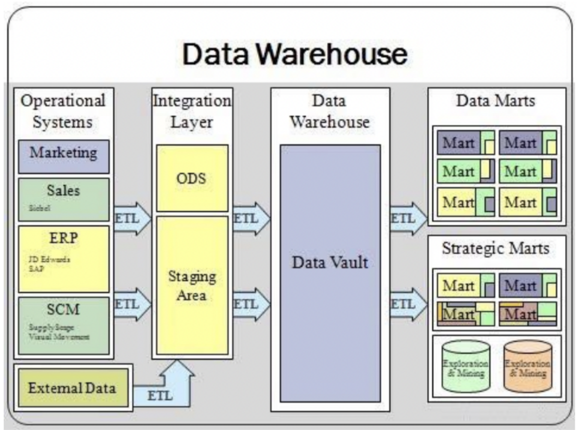
    g. A query language (SQL: Structure Query Language) is used to manipulate data in the RDB
    h. For the db in the example we might write this to find all its ordered by a particular customer:
    i.

```
select customer.id, customer.name, invoice.id,
sales.invoice, sales.item, inventory.item,
inventory.description
FROM customer, invoice, sales, inventory
WHERE sales.invoice = invoice.id AND  sales.item =
inventory.item
```

    j. That earlier diagram is a form of ERD or entity-relationship diagram
    k. The ERD is a graphical way of showing how the tables are related
    l. Related in this case means that the tables share a common key
    m. Most DBs have tools to report ERDs, or you can start with an ERD and build tables from it
13. Hitting a mysql db from an app in PHP
    a. In the app, a connection is made to the db, queries are passed to the db, and results are stored in variables for processing
    b.

```
$db = mysql->connect('localhost:3307', 'accountDB', 'passwd');
$query='select * from accounts WHERE uid =' + uid;
$result = $db->query($query);
while ($row = $result->fetchRow()) {
     echo $row;
}
```

14. Relational db plusses and minuses
    a. RDBs tend to be very good at structured, related data
    b. We usually normalize the data so that each table only contains a very specific piece of information; relationships are used to tie these pieces together
    c. For example, a table of cities and zip codes would be used to pull data into a row that had a user's city…the user row would just have an id into the city table
    d. The minus is that complex relationships can take significant resources (time, memory) to build when needed
    e. We try to mitigate this with as much memory for caches and indexes as we can afford
15. Views
    a. It's common to have a relational set of data that's used across several apps
    b. For example, we might want a user's name, address, city, and list of accounts
    c. These bits might come from several related tables, and it would be time-consuming to have to construct the query in either the app or the DB engine
    d. Instead we create a view, which is essentially a pre-built query that we can use instead of the full complex query
    e. The view ends up behaving like its own table…we do the lookup once and the db stores the result (not the same as caching)
    f. Views sometimes span multiple data sources, but usually they are constrained to a single DB
        i. When we need a view that spans DBs we use middleware instead
    g. The view is an abstraction of the query
    h. In most DBs the views are pre-compiled for optimization
    i. We might also want a series of steps to happen when a query is made…these are called stored procedures or just stored procs
16. Prepared statements
    a. This is a template that is parameterized and precompiled on the db side
    b. For example
       insert into contacts (name, phone) values (?, ?);
    c. This is compiled once and validated
    d. Normal statements are compiled and processed each time
    e. You can see significant performance enhancements
17. The architect vs the DB guru
    a. One point of conflict in design is: Where does related data get assembled?
    b. The DB guru will want to do it in-db because it is faster…the db is optimized for that sort of thing
    c. The architect will want to do it in the app, because it can be optimized there
    d. Who is right?

18. Data warehouses
    a. There's a large family of apps that have heavy read-only requirements for data…reporting is a big example
    b. The data itself might be changing periodically, but for reporting purposes we only need most-recent data
    c. A common pattern for this problem is a data warehouse
    d. The warehouse collects and combines data from several sources into views
    e. The views are pushed into function-specific data marts for consumption by apps
    f. In most cases these views are denormalized…they contain one table per view
    g. 

    h. These data marts / warehouses are often coupled with a reporting language or app
    i. Crystal Reports, for example, is a popular one
    j. The apps allow non-db users to build reports using drag-and-drop
    k. The reports are compiled into what is essentially a stored procedure indb and executed against current data
19. Non-relational DBs
    a. For web-based apps the trend has been toward non-relational DBs such as mongoDB
    b. The DBs are document-based rather than table based
    c. The term noSQL is used for this style…it doesn't necessarily mean 'no SQL' but rather 'non-relational'
    d. noSQL DBs trade simplicity for performance
    e. There are quite a few of them …
    f. The idea behind non-relational DBs is to store all of the data necessary for a record into a single object
    g. Relationships are folded into that single object
    h. In the sales example from earlier, the invoices and items would be folded into a single customer object with arrays for invoices
    i. There can potentially be a lot of duplication
        i. For example, the description of an item, or the address of a customer, might be repeated in hundreds of documents

j. Turns out the answer is ‘not all the time’
k. In typical web-based applications the focus is on a user and his or her interaction with a site
l. For this use case, a fully denormalized data store (ala noSQL) can perform better than a fully relational (highly normalized) DB
m. For the corner cases where we really do need relational data, we take the performance hit

20. ORMs, ORDs, and impedance
    a. In most case an RDB does not map one-to-one onto the objects in the application code
    b. Work is required to move data from app to DB; the pattern we use is an ORM (object-relational mapping), basically a way to take RDB table info and store it in an object
    c. The problem is that the data types in the DB don’t necessarily match the way we want to use them in the app…this is called an impedance mismatch
    d. In document-based DBs (such as mongoDB), there are few or no restrictions on data types in a record, reducing or elimination impedance
    e. However, this means that there are basically no filters on what goes into the mongoDB document store…the risk is GIGO
    f. Libraries such as Mongoose provide ODM (object-document mapping) services on top of mongoDB
    g. ODM is the equivalent of ORM but for non-relational DBs
    h. Bottom line is that ODM on top of document-based DBs provide a cleaner, more performant way to handle data in most applications
21. A note about ODM
    a. It’s certainly possible to throw random, possibly unrelated JSON at a document-based DB like mongoDB, and then go and operate on it
    b. However, mongoDB does essentially no type checking, and the risk is that you’ll be working on undefined data
    c. Using ODM tools such as Mongoose allow you to place schema-based constraints on types and ensures that data in the store is uniform
22. Using noSQL DBs
    a. We’ll use mongoDB and Javascript for our examples
    b. In addition, Mongoose will provide ODM services through schemas
    c. The TL;DL is that we map Javascript objects 1:1 to database documents
    d. Mapping is done through JSON
23. JSON
    a. JSON = Javascript Object Notation
    b. It’s a human-readable way to store and transmit data that maps directly to an object

c. JSON has pretty much replaced XML in web apps, though there are still a huge number of AJAX based apps running on the web
d. NB: You might see references to AJAJ (asynch Javascript and JSON)
e. JSON is the basic format for noSQL databases such as mongoDB
f.


```json
{
 "message": "Successfully retrieved all TODOs.",
 "todos": [
  {
   "_id": "56cf81a60e444bc7faa2ed43",
   "title": "Tonight's TODO!",
   "description": "Some description or not",
   "details": "this is not a detail",
   "__v": 0
  },
  {
   "_id": "56cf836f0e444bc7faa2ed44",
   "title": "new title",
   "description": "sdfsf",
   "details": "hhhh junk",
   "__v": 0
  },
  {
   "_id": "56ce309851068982a09a573e",
   "title": "New todo today!",
   "description": "Some or not",
   "details": "Details now!",
   "__v": 0
  }
 ]
}
```


Mapping between JS model and JSON mongoDB document

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var todoSchema = new Schema({
    title: String,
    description: String,
    details: String,
    owner: String
});
var Todo = mongoose.model('Todo', todoSchema);
```

24. Data directory
    a. By default mongoDB looks for data in /data/db
    b. It'll exit at start if the directory doesn't exist
    c. You also can point to any directory at startup, i.e.
        i. mongod& —dbpath ~/data
        ii. Might have to fiddle with permissions depending on path
25. Starting mongoDB
    a. The database runs as a daemon and listens for connection on port 27017 by default
        i. This can be changed by editing resource file
        ii. You might need to hunt for mongod.conf…it'll end up in the install directory
        iii. Useful to create an alias in /etc to make it easy to find the second time (via ln -s)
    b. Startup
        i. mongod&
        ii. mongod -dbpath ~/data& (if you've moved the data directory to ~)
26. Security
    a. The base installation has authentication TURNED OFF by default
    b. This means anyone with access to the port has access to all databases
    c. (This is why there was a huge mongoDB ransomware attack a few months ago)
    d. Authentication is set up in mongod.conf
        i. Requires definition of users and roles

        ii. Docs are at https://docs.mongodb.com/manual/reference/configuration-options/#security-options
        iii. Lots of info on the web, too, especially after the attacks
    e. In class we'll use defaults, but if you deploy to something other than your local machine you'll want to set up authentication
27. Import sample collections from mongoDB
    a. Grab the sample data
        i. curl https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/primer-dataset.json > mongo-sample-collection.json
    b. Import into running mongoDB instance
        i. mongoimport --db test --collection restaurants -- drop --file ./mongo-sample-collection.json
28. Model and View development
    a. Typical workflow:
        i. Business decides what data should be show on each screen / state
        ii. Designers create HTML frames / mockup (tested FE)
        iii. Data / back-end developers provide views (tested BE)
        iv. Views are wired into FE via model/view
29. Caches
    a. API network calls are extremely expensive (in terms of processing time)
    b. Caching provides a 'close' store for frequently used data
    c. The risk is that the data in cache is older than what is in the database
    d. There are two general ways to approach this (ignoring client-side caches)
30. Caching strategies
    a. A write-through cache stores data in cache and immediately stores it in the DB; confirmation is delayed until the write is complete
    b. A write-back cache stores write data in cache and immediately confirms the write, then updates the DB when it becomes necessary
    c. The big downside of caches is coherency
31. Cache coherency
    a. When you need a piece of data and it is in cache (a "cache hit", how do you know it is the most current value? Especially if there's more than on user/ thread updating the DB
    b. One strategy is to do a fast metadata query on the backend to check if the data has been updated
    c. Another strategy is to turn the radio up louder and hope for the best (a surprisingly common approach)

   d. Sometimes it really doesn't matter…if you have data that doesn't change much, just load it up into memory when the app starts (things like state abbrevs/names and so on)
   e. Most DBs will do a fair amount of caching on their own, but they tend not to be as aggressive as an app-side cache

32. HTTP is stateless
   a. Infrastructure data includes session-related items that help us give the user an end-to-end experience
   b. HTTP is not naturally state-aware…each click (and in fact each element on a web page) is treated as a completely new transaction
   c. We use data both on the back end and stored in local cookies to give the user the illusion of a session
   d. Each time a user clicks a site link, several data transactions might be necessary to set up the resulting page
   e. A common approach is to generate and use a session ID which is transmitted to the web server on each click, and is used to store data in a session table

33. Session IDs
   a. Most frameworks provide a simple way to generate, store, and transmit session tokens
      i. In cookies, transmitted as part of the HTTP request
      ii. As a parameter in the query string (either a POST or a GET), generated and embedded in the link
   b. Cookies are the most common, though there's always some subset of users who refuse to enable them

34. PHP
   a. Stands for PHP Hypertext Processor
   b. Cross-platform
   c. Server-side language
      i. PHP is processed by web server plugin which produces HTML
      ii. Requires overhead to process
   d. PHP is an embedded language…it is mixed in with HTML
   e. The tokens surround PHP code
   f. The language itself is similar to PERL and C++
   g. It's optimized for web pages…deep database features, HTML code generation and manipulation
   h. Many popular platforms, such as Drupal, are based on PHP

35. Getting data using PHP

a.
```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// Create connection
$conn = new mysqli($servername, $username, $password, $dbname);
// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$sql = "SELECT id, firstname, lastname FROM MyGuests";
$result = $conn->query($sql);

if ($result->num_rows > 0) {
    // output data of each row
    while($row = $result->fetch_assoc()) {
        echo "id: " . $row["id"]. " - Name: " . $row["firstname"]. " " . $row["lastname"]. "<br>";
    }
} else {
    echo "0 results";
}
$conn->close();

//Source: W3Schools
?>
```

36. Sessions
    a. Session tables can help provide the appearance of a session
    b. Use session IDs to store things like last page visited, shopping cart info, preferences, and so on
    c. Normally there is one session row per logged in user, and the session data itself can either be stored in that row in a denormalized way, or in a relational way
37. Session variables (PHP)

a.
```
PAGE 1
<?php
// Start the session
session_start();
?>
<!DOCTYPE html>
<html>
<body>

<?php
// Set session variables
$_SESSION["favcolor"] = "green";
$_SESSION["favanimal"] = "cat";
echo "Session variables are set.";
?>

</body>
</html>
```

```
PAGE2
<?php
session_start();
?>
<!DOCTYPE html>
<html>
<body>

<?php
// Echo session variables that were set on previous page
echo "Favorite color is " . $_SESSION["favcolor"] . ".<br>";
echo "Favorite animal is " . $_SESSION["favanimal"] . ".";
?>

</body>
</html>
```