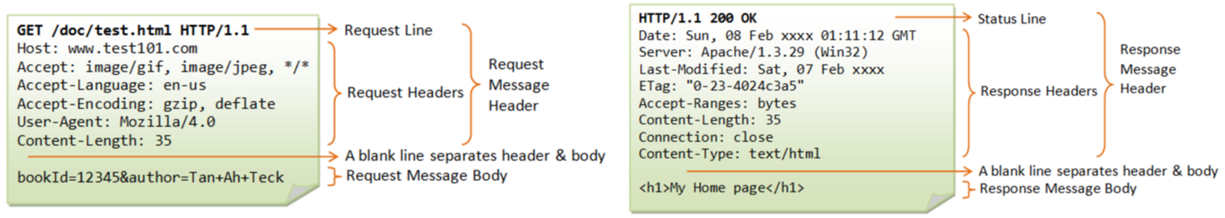
Juputer Introduction

Tomcat

1.Apache Tomcat, often referred to as Tomcat Server, is an open-source Java Servlet Container developed by the Apache Software Foundation (ASF). Tomcat implements several Java EE specifications including Java Servlet, JavaServer Pages (JSP), Java EL, and WebSocket, and provides a "pure Java" HTTP web server environment in which Java code can run.
2.RPC(Remote Procedure Call): a function call to a remote server.
3.Java Servlet: Java class to handle RPC on server side.
4.Tomcat is an environment to run your web service, it provides low level support such as making TCP connection, receiving request from client, finding the correct service to handle that request, and sending response back.
5.If you want to create a web service based on Tomcat Server, all you need to do is implement the logic to handle certain HTTP request.

HTTP request and response



```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```

HTTP supports all CRUD (Create/Read/Update/Delete) operations. Two commonly used methods are:
GET - Request data from server.
POST - Update data on server.
A URL is used to uniquely identify a resource over the web. URL has the following syntax:
protocol://hostname:port/endpoint?query

A message body is the one which carries the actual HTTP request data (including form data and uploaded etc.) and HTTP response data from the server (including files, images etc). Normally we don't return static HTML code to frontend directly because it should be created by frontend developer. We just need to return correct data that should be displayed by frontend. In our project, we'll use JSON as for body format.

Several ideas we've already mentioned earlier
1.Using HTTP methods to indicate what kind of operation a client wants to take.
2.Using HTTP url to indicate which service and data a client want to use and what kind of data they request.
3.Every request is separated, there is no support for doing one post request in several post requests, or doing a delete in a pair of get and post requests.

Why we want to do that?
1.Operations are directly based on HTTP methods, so that server don't need to parse extra thing
2.URL clearly indicates which resource a client want, easy for client side users to understand.
3.Server is running in stateless mode, improve scalability.

TicketMaster API

What's TicketMaster API? A web based API provided by TicketMaster so that clients can get real events data from TicketMaster server. You cannot see the source code of it, but you can refer to the documentation to figure out how to use the interface to make connection by sending request to it's backend.

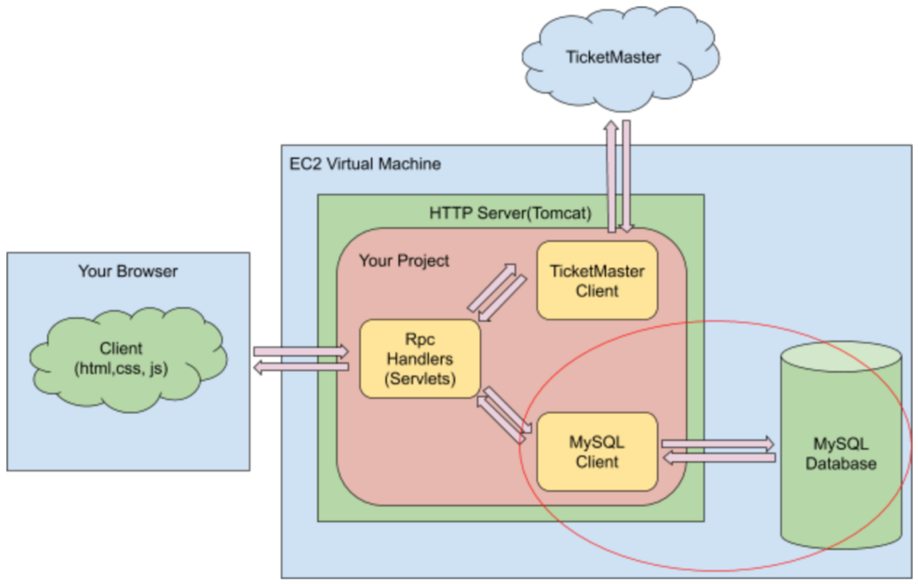
HTTP method (GET, POST, DELETE, PUT, ...) : GET
URL of discover API (**protocol://hostname:port/endpoint?query**) :
protocol: https
hostname: app.ticketmaster.com
endpoint: /discovery/v2/events.json

query:
1.apikey: it’s required by TicketMaster API for authn/authz
2.lat/long: lat/long since our search is based on client location
3.radius: radius of search area
4.keyword: search a specific kind of events
So in the end, the url should be like:
https://app.ticketmaster.com/discovery/v2/events.json?apikey=12345&latlong=37,120&keyword=music&radius=50

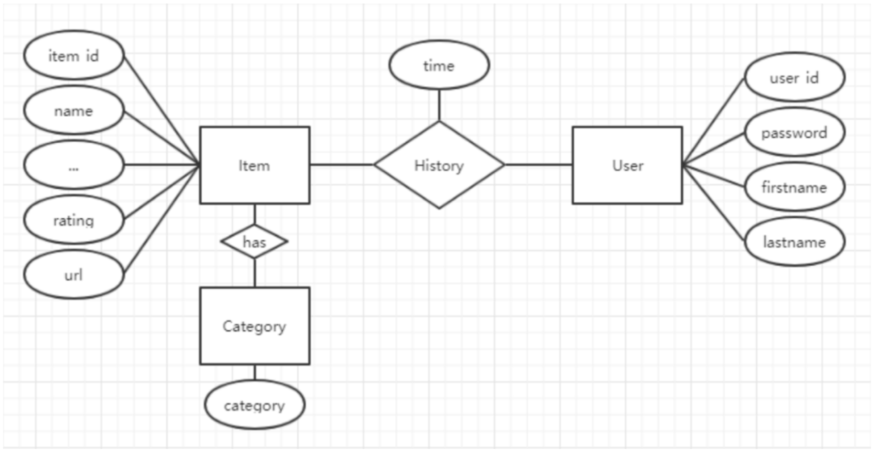
folder **src** -> package **entity** -> class **Item.java** -> **Builder Pattern**
Q: But think about this question before adding new constructors: could you guarantee that TicketMaster can return all data fields to us every time? If it returns null for some data field, how could your constructor deal with that? Or do we have a better solution to handle this problem?
A: Builder pattern builds a complex object using simple objects and using a step by step approach. It separates the construction of a complex object from its representation so that the same construction process can create different representations. We can also make the object to build immutable.

folder **src** -> package **external** -> class **TicketMasterAPI.java** ->
_embedded(JSON Object) -> events(JSON Array) -> item object(JSON Object)
package **rpc** -> new **servlet** -> class **SearchItem**
url mapping to “/search” -> localhost:8080/Jupiter/search
package **rpc** -> new **servlet** -> class **RecommendItem**
url mapping to “/search” -> localhost:8080/Jupiter/recommend

MySQL

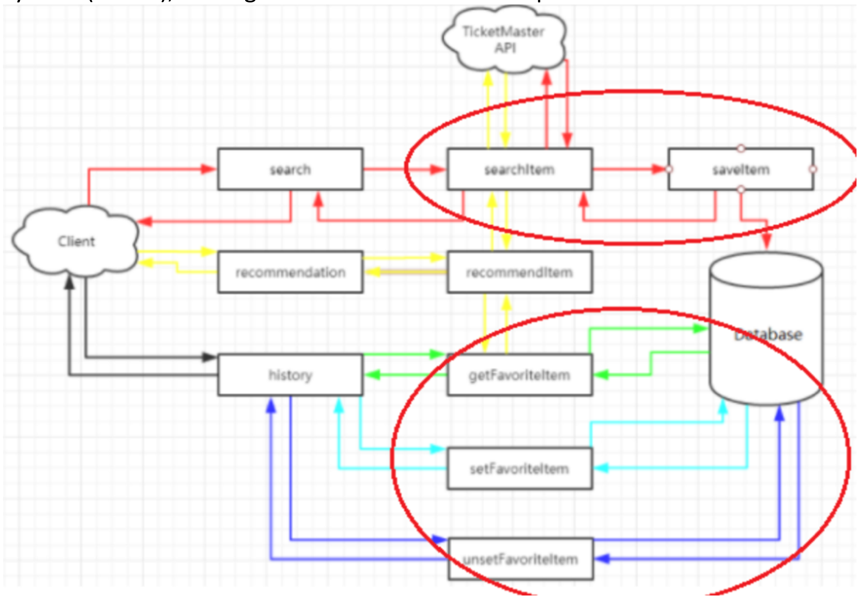


A few more concept:
Unique key: a key in a relational database that is unique for each record.
Primary key: a key that is unique for each record. Cannot be NULL and used as a unique identifier.
Foreign key: a key used to link two tables together. A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.
Index: improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. MySQL will create index on column which is declared as key.



users - store user information. User_id, password, first_name, last_name
items - store item information. Item_id, name, …, rating, url
category store item-category relationship. item_id, category
It's an implementation detail, we could save category in item table, but there will be more string join/split manipulations in our code, so let's save them in a separate table.
Primary key = item_id + category, Foreign key = item_id => items(item_id)
history - store user favorite history. user_id, item_id, time
Primary key = item_id + user_id, Foreign key = user_id => users(user_id), Foreign key = item_id => items(item_id)

JDBC provides interfaces and classes for writing database operations. Technically speaking, JDBC (Java Database Connectivity) is a standard API that defines how Java programs access database management systems. Since JDBC is a standard specification, one Java program that uses the JDBC API can connect to any database management system (DBMS), as long as a driver exists for that particular DBMS.



Syntax for **DROP**: DROP TABLE IF EXISTS table_name;
Syntax for **CREATE**: CREATE TABLE table_name (column1 datatype, column2 datatype, column3 datatype, ...);
Syntax for **INSERT**: INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);
drag mysql-connector-java-8.0.14.jar file to WebContent/WEB-INF/lib

package **db** -> interface **DBConnection**
package **db** -> class **DBConnectionFactory**

package **db.mysql** -> class **MYSQLDBUtil**
package **db.mysql** -> class **MySQLTableCreation.java**
Package **db.mysql** -> class **MySQLConnection.java**
src/rpc/SearchItem.java
/search -> doGet -> call connection.searchItem -> call ticketMasterAPI.search(lat, lon, term) return List<Item>, connection.saveItem, return JsonArray
src/rpc/RpcHelper.java
src/rpc/ItemHistory.java/doPost()
/history -> doPost() -> read userId 和 itemIds -> insert itemIds into table history for userId
src/rpc/ItemHistory.java/doDelete()
/history -> doDelete() -> read userId 和 itemIds -> delete itemIds from table history for userId
src/rpc/ItemHistory.java/doGet()
/history -> doGet() -> read userId -> take items from table history for userId

Why do we need **authentication**?
Access control: user can only access data that are authorized to that user.
Logging: record user specific activity for book keeping, statistics, etc.

User Journey:
1.For an application, some resources can only be accessed by authenticated user.
2.Once a user is authenticated, the server uses a **session** to maintain his/her status. The session object is stored on server side, only session ID is returned back to the client side.
3.User needs to provide **session ID** to access resource that requires authentication.
4.When user logs out, the session is destroyed. Next time a user comes, he/she has to authenticate again to get a new session.

Client – Home Page
GET index.html (page with search, favorite, recommendation)
No valid session, redirect to login.html (page with username and password)
Client – Login Page
GET login.html (page with username and password)
Return login.html (page with username and password)
Client – Login Handler
POST login.html (user_id = 1111, password = 2222)
Login successfully, redirect to index.html (session_id = abcd)
Client – Home Page
GET index.html (session_id = abcd)
Return index.html
Client – Search Handler
GET /search?lat=37&lon=-122 (session_id = abcd)
Return search result [{item1:1111}, {item2:2222}, …]
Client – Logout Handler
GET logout (session_id = abcd)
Session destroyed, redirect to index.html

Package **db.mysql** -> class **MySQLConnection**
src/rpc/login/Login.java
/login -> doGet()
Get session from server， get user_id from session, return user_id and name
Before go to index.html， check if session is valid, if so, we show index.html, otherwise return login.html
/login -> doPost()

Get user_id and password from request, build new session in server, add new user_id and valid period=10min in this session, return user_id and name.
src/rpc/logout/Logout.java
/login -> doGet() get session from server, delete session, redirect to index.html.
src/rpc/register/Register.java
/login -> doPost() get parameters from request, save them in user table.

Recommendation System
1.Content-based Recommendation
Key point: You will like people or things of similar characteristics.
Given item profiles (category, price, etc.) of your favorite, recommend items that are similar to what you liked before. This is what we'll use in our application.
2.Item-based method
Filter based on the similarity of Items.
Item A is liked by User A, User B, User C
Item B is liked by User B.
Item C is liked by User A, User B
=> Item A and Item C are alike
Item C is liked by users who like Item A, so recommend it to user C (another user who likes Item A).
3.User-based method
Filter based on the similarity of Users.
User A likes Item A, Item C.
User B likes Item B.
User C likes Item A, Item C, Item D.
=> User A shares similar preference as User C compared to User B.
User C also likes Item D => User A may like Item D.

Precision and Recall

	Human: Like	Human: Unlike
Algorithm: Recommend	A	B
Algorithm: Not Recommend	C	D

Precision = A / (A + B), Recall = A / (A + C)

Engineering Design
Step 1, given a user, get all the events (ids) this user has visited.
history: history_id, user_id, item_id, last_favor_time.
Set<String> itemIds = connection.getFavoriteItemIds(userId);
Step 2, given all these events, get their categories and sort by count.
categories: item_id, category.
Set<String> categories = connection.getCategories(itemId);
Step 3, given these categories, use TicketMaster API with category as keyword, then filter out user favorited events.
external API, List<Item> items = connection.searchItems(userId, lat, lon, category);

src/recommendation/ GeoRecommendation.java
Recommendation based on geo distance and similar categories.
src/rpc/RecommendItem.java
/recommend -> doGet()