Design and Engineering of Computer Systems

Lecture 28:
Multi-tier application design

Mythili Vutukuru
IIT Bombay
Multi-tier applications

• Real-world computer systems are built as **multi-tier applications**
  • Multiple components/tiers distributed across several machines

• High-level architecture of a multi-tier application
  • Clients access applications hosted in organizations or public clouds
  • **Front-end** components (e.g., web servers) receive user requests, reply to user with responses, consult various **application servers** to build responses
  • App servers contain **business logic** to process different types of user requests
  • Application data is stored in several **database servers in the backend**

• Example: e-commerce application has front-end web server, multiple application servers to handle different functions (e.g., product search, shopping cart, purchases), and multiple databases (e.g., product catalogue, user profile, order history)
Decomposing applications into components

• Why to build applications in modular (not monolithic) design?
  • Easier to design, develop, optimize smaller components
  • Easier to replace/upgrade components without bringing down entire system

• General guidelines to modularize applications (not hard rules)
  • Identify what functionalities the system should provide
  • Identify what application data to be stored to satisfy functionality
  • Encapsulate one type of data and functions on data into one logical component
    (e.g., one component for product catalogue and related functions)
  • Each component / application server can be further decomposed into multiple
    micro-services (processes/threads) for separate functions/features
  • Components interact with each other via well-defined interfaces or APIs, no
    need to know the internal implementation details
Example: functional requirements

- Functional requirements of a simplified e-commerce system
  - Maintain user authentication and profile information, billing details
  - Add products to catalogue, search with keywords
  - Add items to shopping cart, view shopping cart
  - Checkout, billing, shipping of items
  - Keep order history, support cancellations and returns
  - Recommend future purchases based on past history
- Group together (app data + functions on data) into one component
  - Product catalogue, add/search/buy/return products
  - User profile database, add/delete/authenticate/modify user data
Example: modular architecture of e-commerce application

<table>
<thead>
<tr>
<th>Component</th>
<th>Data maintained</th>
<th>Functions / microservices</th>
</tr>
</thead>
<tbody>
<tr>
<td>User profile</td>
<td>User information, password, billing info, shipping address</td>
<td>Add new user, authenticate login, update info, delete user</td>
</tr>
<tr>
<td>Products</td>
<td>Catalogue of products available</td>
<td>Add products (used by supplier), search by keywords, buy, return</td>
</tr>
<tr>
<td>Shopping cart</td>
<td>Shopping cart for each user</td>
<td>Add item, delete item, view cart</td>
</tr>
<tr>
<td>Orders</td>
<td>For each order placed, details of items purchased in that order, billing and shipping information</td>
<td>Create new order (from shopping cart), billing for order, shipping and tracking of order, retrieve order history of user, cancellations and returns</td>
</tr>
<tr>
<td>Recommendations</td>
<td>What products to recommend for each user based on past history of purchases</td>
<td>Retrieve recommendations</td>
</tr>
</tbody>
</table>
Application data storage options

- Relational database management systems (RDBMS)
  - Store structured data in the form of relational database tables with strict schema
  - Provide strong guarantees (ACID – Atomicity, Consistency, Isolation, Durability)
  - Support for transactions (complex operations spanning multiple tables)
- NoSQL data stores: for unstructured data (e.g., key-value stores) or semi-structured data (e.g., document stores) or specialized data (e.g., graphs)
  - Dynamic or flexible schema, no strict consistency guarantees or transaction support
  - Easier to scale, better performance than RDBMS
  - In-memory only for transient data, disk storage option if persistence needed
- Many data stores available, choice depends on type of data in app server
  - User profile data stored in RDBMS, shopping cart stored on NoSQL key-value store
- Data moves from one data store to another as it is processed
  - User clicks on videos stored temporarily in NoSQL, aggregated and stored in RDBMS
API design: REST

• Front-end, app servers, backend components interact over well-defined interfaces or APIs: how to design these?

• REST (Representational State Transfer): popular way to design APIs
  • Reuse HTTP client-server mechanism for data beyond web pages
  • Data stored in a component represented as URLs (e.g., /user/foo/profile/address)
  • Data can be created, updated, read, or deleted (CRUD) via different types of HTTP requests (GET, POST, ..)
  • App data exchanged via standard serialization formats (e.g., JSON) over HTTP
  • Easily implemented by existing HTTP frameworks
  • Responses can be cached like web content fetched over HTTP

• What do clients use to communicate with front-end servers?
  • Standard application layer protocols, e.g., HTTP, SMTP
  • REST-based APIs, e.g., accessing mapping service from mobile app
API design: RPC

• REST-based APIs are popular but may not be suitable in all cases
  • Not easy to represent all data in a component as URLs
  • Cannot do all actions using fixed set of HTTP verbs (GET, POST, ..)
  • REST (HTTP) is stateless, no dependence on previous state of data allowed
  • Example: “cancel” order API not easy in REST, need to do action depending on state (whether confirmed or not) of order, HTTP DELETE verb is not suitable

• Alternate way of designing APIs: use RPC frameworks
  • Components interact and exchange data using remote procedure calls
  • Can customize interface: messages exchanged in requests and responses between client and server, function/services provided by server
  • Needs close coordination between client and server, not suitable for external facing interfaces, more useful for inter-component interactions within system
API design: publish-subscribe

• RPC and REST are client-server model: one component (client) sends a request and waits for response from server to proceed

• Some interfaces need more asynchronous interaction, for example:
  • Server making purchases pushes order info to recommendation server, no response is expected. Recommendation server runs algorithms asynchronously on orders to come up with recommendations for user.
  • User uploads video to video upload server, which pushes video to another server that converts video into various resolutions later on asynchronously

• Such interactions between components are called publish-subscribe model, happen via frameworks called message brokers or task queues
  • Some components publish information to a task queue, other components subscribe for this information and process it
  • Subscribers can subscribe to specific topics selectively
  • Message brokers provide temporary storage of messages, high performance reliable message delivery using network protocols
Summary

• In this lecture:
  • How to design multi-tier applications
  • Design choices for data storage, APIs

• Work out an end-to-end design of any computer system you use in your day-to-day life. Think through what are the functional requirements, how to modularize, what kind of data stores you will need, and what kind of APIs will be appropriate.