SOEN 387: Web-Based Enterprise Application Design

Introduction to Enterprise Applications

- Enterprise applications often have complex data to work on, together with business rules.
- Enterprise applications include payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading.
- Enterprise applications do not include word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, and games.
- Usually involve persistent data. The data is persistent because it needs to be around between multiple runs of the program—indeed, it usually needs to persist for several years.
- Persistent data will often outlast the hardware that originally created much of it, and outlast operating systems and compilers. During that time there'll be many changes to the structure of the data in order to store new pieces of information without disturbing the old pieces.
- Usually many people access data concurrently.
- With so much data, there's usually a lot of user interface screens to handle it. It's not unusual to have hundreds of distinct screens. Users of enterprise applications vary from occasional to regular, and normally they will have little technical expertise. Thus, the data has to be presented lots of different ways for different purposes.
- Usually enterprise applications need to integrate with other enterprise applications .

Kinds of Enterprise Application

Example 1: Consider a B2C (business to customer) online retailer: People browse and buy. For such a system we need to be able to handle a very high volume of users, so our solution needs to be not only reasonably efficient in terms of resources used but also scalable so that you can increase the load by adding more hardware.

- The domain logic for such an application can be pretty straightforward: order capturing, some relatively simple pricing and shipping calculations, and shipment notification. We want anyone to be able access the system easily, so that implies a pretty generic Web presentation that can be used with the widest possible range of browsers. Data source includes a database for holding orders and perhaps some communication with an inventory system to help with availability and delivery information.

Example 2: a system that automates the processing of leasing agreements. In some ways this is a much simpler system than the B2C retailer's because there are many fewer users. Where it's more complicated is in the business logic. Calculating monthly bills on a lease, handling events such as early returns and late payments, and validating data as a lease is booked are all complicated tasks. A complex business domain such as this is challenging because the rules are so arbitrary.

- Such a system also has more complexity in the user interface (UI).
- Much more involved HTML interface with more, and more complex, screens. Often these systems have UI demands that lead users to want a more sophisticated presentation than a HTML front end allows.
- A more complex user interaction also leads to more complicated transaction behavior: Booking a lease may take an hour or two, during which time the user is in a logical transaction.
- Complex database schema with perhaps two hundred tables and connections to packages for asset valuation and pricing

Example 3: A simple expense-tracking system for a small company.

- Such a system has few users and simple logic and can easily be made accessible across the company with an HTML presentation.
- The only data source is a few tables in a database. You have to build it very quickly and you have to bear in mind that it may grow as people want to calculate reimbursement checks, feed them into the payroll system, understand tax implications, provide reports, tie into airline reservation Web services, and so on.

Each of these three enterprise application examples has difficulties, and they are different difficulties. As a result you can't come up with a single architecture that will be right for all three. Choosing an architecture means that you have to understand the particular problems of your system and choose an appropriate design based on that understanding.

Thinking About Performance

- Many architectural decisions are about performance.
- However, some architectural decisions affect performance in a way that's difficult to fix with later optimization.
- Also a significant change in configuration may invalidate any facts about performance. If you upgrade to a new version of your virtual machine, hardware, database, or almost anything else, you must redo your performance optimizations and make sure they're still helping.

Performance measures and terminology

- **Response time**: is the amount of time it takes for the system to process a request from the outside. This may be a UI action, such as pressing a button, or a server API call.
- **Responsiveness**: is about how quickly the system acknowledges a request as opposed to processing it. This is important in many systems because users may become frustrated if a system has low responsiveness, even if its response time is good. If your system waits during the whole request, then your responsiveness and response time are the same. Providing a progress bar during a file copy improves the responsiveness of your user interface, even though it doesn't improve response time.
- **Latency**: is the minimum time required to get any form of response, even if the work to be done is nonexistent. It's usually the big issue in remote systems. If I ask a program to do nothing, but to tell me when it's done doing nothing, then I should get an almost instantaneous response if the program runs on my laptop. However, if the program runs on a remote computer, I may get a few seconds just because of the time taken for the request and response to make their way across the wire. As an application developer, I can usually do nothing to improve latency. Latency is also the reason why you should minimize remote calls
- **Throughput**: is how much stuff you can do in a given amount of time. If you're timing the copying of a file, throughput might be measured in bytes per second. For enterprise applications a typical measure is transactions per second (tps).

- **Load**: is a statement of how much stress a system is under, which might be measured in how many users are currently connected to it. The load is usually a context for some other measurement, such as a response time. Thus, you may say that the response time for some request is 0.5 seconds with 10 users and 2 seconds with 20 users.
- **Load sensitivity:** is an expression of how the response time varies with the load. Let's say that system A has a response time of 0.5 seconds for 10 through 20 users and system B has a response time of 0.2 seconds for 10 users that rises to 2 seconds for 20 users. In this case system A has a lower load sensitivity than system B. We might also use the term degradation to say that system B degrades more than system A.
- **Efficiency**: is performance divided by resources. A system that gets 30 tps on two CPUs is more efficient than a system that gets 40 tps on four identical CPUs.
- **Capacity** : is an indication of maximum effective throughput or load. This might be an absolute maximum or a point at which the performance dips below an acceptable threshold.
- **Scalability**: is a measure of how adding resources (usually hardware) affects performance. A scalable system is one that allows you to add hardware and get a commensurate performance improvement, such as doubling how many servers you have to double your throughput. Vertical scalability, or scaling up, means adding more power to a single server, such as more memory. Horizontal scalability, or scaling out, means adding more servers.

Chapter 1. Layering

- Layering is one of the most common techniques that software designers use to break apart a complicated software system. You see it in machine architectures, where layers descend from a programming language with operating system calls into device drivers and CPU instruction sets, and into logic gates inside chips. Networking has FTP layered on top of TCP, which is on top of IP, which is on top of ethernet.
- When thinking of a system in terms of layers. The higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer.
- Furthermore, each layer usually hides its lower layers from the layers above, so layer 4 uses the services of layer 3, which uses the services of layer 2, but layer 4 is unaware of layer 2.

Breaking down a system into layers has a number of important benefits.

- You can understand a single layer as a coherent whole without knowing much about the other layers.
- You can substitute layers with alternative implementations of the same basic services.
- You minimize dependencies between layers.
- Layers make good places for standardization. TCP and IP are standards because they define how their layers should operate.
- Once you have a layer built, you can use it for many higher-level services. Thus, TCP/IP is used by FTP, telnet, SSH, and HTTP. Otherwise, all of these higher-level protocols would have to write their own lower-level protocols.

Layering is an important technique, but there are downsides.

- Layers encapsulate some, but not all, things well. As a result you sometimes get cascading changes. The classic example of this in a layered enterprise application is adding a field that needs to display on the UI, must be in the database, and thus must be added to every layer in between.

- Extra layers can harm performance. At every layer things typically need to be transformed from one representation to another.

The Three Principal Layers

- **Presentation logic** is about how to handle the interaction between the user and the software. This can be as simple as a command-line or text-based menu system, but these days it's more likely to be a rich-client graphics UI or an HTML-based browser UI. The primary responsibilities of the presentation layer are to display information to the user and to interpret commands from the user into actions upon the domain and data source.
- **Data source logic** is about communicating with other systems that carry out tasks on behalf of the application. These can be transaction monitors, other applications, messaging systems, and so forth. For most enterprise applications the biggest piece of data source logic is a database that is primarily responsible for storing persistent data.
- The remaining piece is the **domain logic**, also referred to as **business logic**. This is the work that this application needs to do for the domain you're working with. It involves calculations based on inputs and stored data, validation of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch, depending on commands received from the presentation.