Josh Dinges

May 10, 2012

Web Applications: Final Exam

1. **Statelessness, Statefulness**

Because of the very nature of the HTTP, requests and response are stateless, meaning that it doesn’t distinguish one request from another. In order to get around this, Rails uses cookies to keep track of extra data that is necessary in order for an application to achieve this idea of statefulness. The key piece of information found in the cookie that ensures this statefulness is the session ID. A session ID is, as its title would suggest, a way of uniquely identifying the user requesting data from the server. The session ID is stored in the cookie as part of the payload in a HTTP request.

Structuring statefulness in this fashion allows the server to easily recognize that the end user is still in the same session, whether that’s logged in or not or logged in as some specific user with a given set of privileges. A session is typically defined as something like a state of being given access to a site’s data from a single browser and window.

1. **Authentication, Authorization, Security**

One of the benefits of “rolling your own” authentication and security mechanisms is that you are directly in control of what exactly your security measures are and how they are used in the application. More specifically, “rolling your own” allows you, the developer, to have direct control over just how much security is present in your app as well as how it behaves and integrates with the rest of your app. More importantly, perhaps, is that it can be as large and robust or as small and simple as you and your application desire/require.

However, “rolling your own” authentication is not without its problems. Perhaps the strongest reason for “rolling your own” is also its greatest weakness – it’s all up to the developer. Several problems can arise from this scenario, the first being bugs. Bugs are common and unless the app has a strong unit test suite, it is susceptible to not only internal errors, but also external attacks. Another weakness is the developer’s knowledge of common security techniques such as password salting, hashing, and input sanitization. All of these are paramount for building a secure service and any mishap in anyone of these areas leave the app vulnerable to attack. Lastly, third party tools allow for community input, maintenance, and evolution.

1. **Database, Schema Version Control**

Ruby on Rails uses ActiveRecord and migrations to handle schema version control. A migration is simply a way to modify the database’s schema in a quantifiable step. Migrations are not limited to single changes to the schema, such as only changing the type of a column, or adding a column, or altering the name of a table, or dropping a table. Many alterations can be accomplished in a single migration. Migrations are, at their heart, just a modification of the database so that it can correctly interact with the rest of the rails application. Another way of putting it, migrations modify the schema in a controlled way, but they don’t define it. When the application is deployed, it doesn’t run all of the migrations. Instead, Rails keeps a separate file – schema.rb – that describes the databases schema as a culmination of all the migrations.

I believe that this is an effective way of maintaining a database. Migrations offer on-the-fly fixes to your database without requiring the developer to mess with the SQL their self. It also ensures that the database is described in an ORM fashion to better mesh with the structure of OO programming. The benefits of this approach, as stated, can be seen in on-the-fly fixes to a database, it’s ORM structure, and its changes being in quantums that are introduced as needed. The downside is that each migration requires that the database be dumped and then reloaded with the new schema. Another, potentially large problem, lies in multiple developers working on a single project. It’s entirely possible for several developers to create two migrations that accomplish the same or similar things in entirely different ways. The problem is compounded if even more developers continue work after using different migrations. In essence, the problem can be summed up as, if anyone is allowed to modify the database, how do you ensure that everyone has the most current migration? While there may not be a direct solution to this problem, one could argue that this issue can be minimized by better communication between the team.

1. **Everything RESTful**

I think that it’s useful to keep thinking that only CRUD operations can be performed on data at the this level. This keeps the communication between the client and the server simple and puts the burden of translating client and server-side data on the application or the browser itself. In short, keeping actual communication between the client and the server as simple and close to the intent of the Internet as possible is for the best.

SOAP stands for Simple Object Access Protocol and is the primary alternative to REST. It was designed to be language, transport and platform independent (REST is not transport independent). Its purpose is to communicate XML messages. WSDL is essentially a contract describing what features a web service will or will not provide. It has been described as isometric to a programming language’s method signature. It tells the machine of the service that’s going to be called, its parameters and what it returns [Source: Wikipedia]. WSDL used to only support SOAP, but has since been updated that it now supports RESTful web services. SOAP differs from REST in that SOAP sends XML, REST only uses HTTP GET requests that often contain XML. Another difference is that SOAP uses HTTP POST on an endpoint, REST only uses HTTP GET. SOAP has better support of other web standards like WSDL and WS-\*. WS-\* are used for essentially reliability suites that ensure the safety of sensitive data like bank statements or personal information (SSN). The big difference between the two is that REST is directly coupled with HTTP, it’s concise and easy to learn but doesn’t support the distributed computing model and has less standards supporting it. SOAP, on the other hand, has the above web services on its side for additional data security and integrity beyond that offered by HTTP and it handles errors, but it’s really freaking hard to understand, a lot of the services that communicate with it are even harder to understand and nobody really likes XML in the first place. Also, if you’re going to develop using SOAP you need a shit ton of tools.

*-- As an aside, I referenced this article heavily in answering this question. http://www.ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest*

1. **Compare & Contrast Different Web Architectures**

In this section, I’ll be discussing the benefits and drawbacks of five web architectures. Each section will not only discuss the pros and cons of that particular architecture, but also how it relates to the other architectures being described. The architectures to be discussed are: the traditional web application service, the in-browser application, multiple apps fitted for HTTP requests, no-app-tier app, and finally traditional client-server.

The traditional web application runs on a stack that, from the bottom-up, goes through the browser, over the wire, to the web server, processed by the app server, and finally communicated with the database server. This is model has the interface transmitted over the wire each time the browser talks to the web server. One advantage is that this puts little strain on the browser or client side of things, as it is simply responsible for rendering what’s given from the web server. However, this is not without its disadvantages; despite putting little strain on the client, it does end up requiring the web server to transmit quite a lot of data and meta data (markup, assets). After all, if what we’re really after is the data, why do we constantly need to be sent the same set of interfaces over and over again? This question is what lead developers to the creation of the second architecture style - the in-browser application.

The in-browser application exists solely as a response to the traditional web application service. It recognized the inefficiencies in sending the markup, assets, the interface, and the data over the wire, when what the user is really after is just the data itself. But how does it work without the interface? Well, this architecture doesn’t entirely reject the interface-over-the-wire approach; it just isolates it to the first connection with the web server. Subsequent requests from the web server receive only the data and NOT the interface. The advantage to this approach, as stated, is that minimizes the redundancy of the traditional web application. However, this begs the question of why are we limiting ourselves to just one web service? The whole Internet is out there waiting queried, why should one web service only communicate with one data store? Why can’t I query other web services as well and receive the information and compile into a one-stop-shop app? This resulted in the creation of the multiple web services communicating with one browser application at once.

As seen in the above, progress seemed to be in scaling outward from the in-browser application. This resulted in the multiple apps fitted for HTTP requests. The idea with this architecture is that, why should a browser be the only application limited to communication with the outside world? This, of course, results in not only multiple services communicating with one app, but also multiple services communicating with multiple client apps. There’s the browser itself, then there’s the desktop app that queries some small tidbit of information from web service and reports back, and there’s the mobile application (an area of ever increasing popularity). This service is great because it lets multiple apps communicate with multiple services, but the problem is that the services are still structured in the same web-app-db stack. Since the client-side app is really only interested in data, as per in-browser app, aren’t we really just seeking interaction with the NoSQL database? This would be nice, but without the app serve-side, all of that necessary structure is lost. After all, NoSQL’s most popular characteristic is that it throws the rigidity of the traditional RDBMS out the window, but doing so requires another service or application to handle, interpret and organize the data appropriately. It turns out, that perhaps the future is lies in the past.

If the client-side app is really only after data, why bother with app server at all? Couldn’t we just query a RDBMS? This is certainly an option, but is kind of a throw back to the old way of doing things. This architecture isn’t certain, but it seems like a possible evolutionary step forward.

Personally, I feel that the future lies in multiple apps communicating with multiple services. This seems a bit backwards from where we may be going with multiple apps communicating with multiple RDBMS, but consider the struggle between RDBMS and NoSQL like databases – it’s a struggle for how should data be stored and organized. On one hand there’s the RDBMS struggling for rigid order. Data will be stored and structured based on a very specific set of meta-data. NoSQL struggles to release itself from the chains of RDBMS in order to scale-out to meet the ever-increasing demands of a unified data store. Each of these models of data storage are applicable in different situations, but I don’t think it’s for the future of web applications to decide. Data storage is an issue that should be abstracted away from web applications. Web applications, moving forward, should simply rely on the notion that from server, their data queries will be responded to with a organized quantum of data. When we abstract away the DB Server and the App Server that organizes and communicates with the DB, we’re left with multiple web servers communicating with multiple client apps. The future lies in constantly being connected. Ensuring that applications have HTTP communication capabilities, is how this is going to happen.