For our architecture design, we have chosen a layered architecture with 4 layers (on the client side is the Presentation and Interaction Logic layers and on the server side is the Business Logic and Data layers). There are many advantages to using this layered architecture. One huge advantage is that by using layers, we reduce coupling between the different layers. This makes it easier to swap out individual layers in the future as necessary. For example, we are planning on using PHP for our Business Logic Layer. If our customer down the line wanted to instead use Java for the server, rather than having to swap out everything and start from scratch, the only thing that would need to change is the specific call in the Interaction Logic layer to call the appropriate Java method instead of PHP page. Another advantage of the layered architecture is that each layer has a predefined role. This helps to increase cohesion - each layer has its specific role and focus on doing just that. In addition, a good chunk of modern web frameworks are based around this general layered architecture. This gives us many different options of tools to use for our different layers. For example, for our Presentation layer, we can use frameworks like jQueryMobile, ExtJs, or Dojo. The biggest disadvantage of this layered architecture for us is that it requires knowing different frameworks/ languages for each layer involved (for us, it requires knowing jQueryMobile/Javascript in general, PHP/HTML, and mySQL), but since most of our team had experience with the same languages, this was deemed not a big enough issue to stop us from using a layered design.

We considered some other architecture designs as well. One that we looked closely at was Model-View-Controller. MVC, like layered, has the advantage of being very common in web design so many frameworks like GWT and Ruby on Rails are available for MVC. However, we felt that MVC and its frameworks simply take too much time to figure out. In addition, MVC applications tend to have a lot of overhead they need to set up before the actual application can be worked on. Since we are making a much smaller application, the time it takes to set up MVC is significantly more than the time it would take to simply program the entire thing. For our smaller application, it simply wasn't worth using MVC.

We also considered an Event-Driven architecture. The biggest advantage of this architecture is that there is virtually no coupling- since you are calling an event and not a function, the only thing that needs to change is the code listening for the event, not the code calling it. This would result in even less coupling than our layered design. On the other hand, Event-Driven architectures are much harder to follow just by reading the code. Since they use some kind of event bus framework to tell different parts of the code what to do, you can't follow method calls nearly as easily as in other architectures. Instead you have to look for hints in the code manually. We likely won't have the time necessary to devote to debugging obscure event issues. In addition, you can very easily run into event timing issues (like you can run into timing issues in parallel programming). Overall, an Event-Driven Architecture is just too likely to have many obscure and difficult to detect bugs that we won't have the time to finish.