SE577: Software Architecture Assignment 4

Joe Durko (jsd94@drexel.edu) *Kaushik Mukherjee* (km3762@drexel.edu)
Taiwo,Owolabi (ot22@drexel.edu) *Wendy Prayer* (wp86@drexel.edu)

Impressions

The term project was a great opportunity to try out the design patterns, architecture styles and designs taught during the lecture. The building of the project was fairly smooth. We started with a proposal early in the course that evolved into a document describing architecture and then were given the opportunity to give feedback and reflect on our own. The freedom on how we designed and implemented the project itself felt refreshing given some other classes may be specific to a certain language - it seems that's the point of the architecture class as a whole is that these concepts are above the notion of programming languages and syntax.

The only issue we see is that there may have needed to be some implementation homework assignments to make sure we understand the concepts, homework during class would help buttress concepts in the class before jumping into the final project. Most of the class assignments and lectures involved writing and documentation while the final project involves a decent bit of coding (depending on what the team proposed), so it would have been nice to try making concrete use of these design patterns instead of only learning about them at a high level.

Team Member Contributions

Joe Durko (Engineer) - Developed server side and data model; helped with front-end
Taiwo Owolabi (Engineer) - Developed front-end
Wendy Prayer (Architect) - Helped with documentation, design and testing
Kaushik Mukherjee (Architect) - Helped with documentation, design and testing

Lessons Learned

The most obvious one is to remember to document layered architectures with the user-facing side at the top. This was a pretty big critique on our first paper along with a poor description of the data flow through our system. We also need to make sure to specifically address security in our architecture if we are going to try it at all as it was pointed out a few times that the structure of the data did not help for keeping passwords secure.

The most important lesson learned is probably to not solidify your data model until you structure it in code. When using a framework like Spring Boot, you'll see that the way data is structured in your documents becomes a bit messy the more you try to morph your code around it - this was obvious when we had properties that looked exactly the same between Patient and Staff Members but were still duplicated on both. During testing, it also became clear that we had a many-to-many relationship between staff members and appointments but in our original documents, this was maintained as a one-to-many relationship which broke down during runtime.

Another lesson learned was that using preconstructed Frameworks can be a bit finicky at times, they can help one do a lot of the heavy lifting but sometimes lead to side effects that can take considerable time to resolve. For example, when integrating Angular and Spring we had issues with hyphenated names in Angular, the framework kept returning NaNs and that necessitated a change to JSON payload after considerable effort to debug the issue.

We also learned that there are many different frameworks for a myriad of languages to support the designs mentioned in class. Spring Boot and Angular seem to be a pretty common combination of implementing server-client architecture which seemed to fit perfectly with our documentation. Furthermore, frameworks when mastered rapidly speed up development and provide a separation of functionality that improves the underlying architecture of the application leveraging functionality and framework patterns that have been tested and proven thus making up for any minor quirks that arise when leveraging these frameworks.

One final lesson is that markdown on GitHub is very finicky. Still not sure why using a single line break needs to be preceded by two spaces to get it to appear, or why line breaks don't appear when using single tick (`) code marking, but there was a lot of hasty merges trying to fix this.

Patterns Used and Vulnerabilities Discovered

The overall architecture of the JKTW EMR Server is an **MVC pattern** where our front-end Angular is the View, the Spring Boot REST API is the Controller and the Model is the actual

Java code (and how it interacts with the MySQL database). At any point, we could swap out the Angular front-end for something else as long as it can send HTTP requests to the server. This can also be seen as a **layered architecture** as the Angular code only talks to the REST API and the REST API only talks to the services while the services handles the database and giving responses back to the REST API to respond. This can also be seen as **multi-tiered** since the client tier is its own computationally-independent execution structure while the server can run completely independently as well. That said, the front-end would need to mock out its own data (which was done during testing) if it would like to truly be independent. There is an obvious case of the **client-server pattern** as well since our Angular front-end serves as a client to the Spring Boot server that serves out information. Finally, since there could be multiple front-ends that ultimately query the same persistent database, we *could* have an instance of a **Shared-Data Pattern** if we ever decide to supply different components to talk to our backend.

The most prevalent pattern used is the **Object Oriented Design** that is used to create the model and to handle components on the front-end. **Abstract Oriented Design** is also used in the model for the relationship between Person and Patients/Staff Members. The Spring Boot Framework heavily favors composition over inheritance as it works on providing interfaces that are injected into the controller(s) so that controllers can be decoupled from the service (and as a matter-of-fact they auto-generate a concrete implementation that does the basic CRUD operations and other queries based on the wording of the method). This also shows off a **service oriented design** as the four entities (Patients, Staff, Lab Records, Appointments) each have their own service with their own controller. This helps them mostly stay in their own silo with the exception of the Appointment and Lab Record service asking for help from the Patient service for queries that return patients based on appointments or lab records respectively. The services themselves implement a **factory pattern** as they build the objects upon request; for example, to get a new patient, we don't need to actually use the keyword "new", we instead call "patientService.findById(id)" which new's one up using information persisted in the database. And because these services are again only interfaces, this demonstrates an **abstract factory pattern** since the services themselves can be swapped out at any time (although Spring Boot automatically injects a default implementation for production).

Some downsides to using these patterns is that the **layers and tiers** involved with this cause a performance overhead - we have to talk over the wire to get information and the fact that the codebase is split into two different areas adds some complexity to understanding it. If the tiers are removed, the **MVC pattern** may be some added complexity that is not necessary as we don't give much in the way of customizing our view. The **client-server pattern** also adds a bottleneck as the server would need to serve out to any number of clients on the network - it also made things complex for how to divvy up client and server responsibilities. The **shared-data pattern** also made it so there's a single-point of failure - if the database fails, there's nothing much that this application can do.

Some omitted patterns that could help with functionality include the command pattern. If we were able to have the create/delete/edit commands be actual objects, we could implement an

undo function would be helpful for the client to undo any mistakes. However, because we are using a shared-data pattern, this would be tricky as the data may already be manipulated by someone else before the undo is done.

There are some obvious vulnerabilities still present in the project that could be addressed at a later time. First is the absence of a concept of user logins which means that authorization is completely absent - anyone who can talk to the server has permission to access the data. We hope that anyone using the hospital staff's internal network (where the server should sit) is an employee of the hospital, but because of our second vulnerability, the lack of authentication, we can't quite verify the assumption that the user is a hospital staff member. Finally, we also are using HTTP without using Transport Level Security or a Secure Sockets Layer. This means anyone who is sniffing the wires of the hospital can see POST and GET requests with SSN's and other information of the patients. These obvious vulnerabilities would be addressed in a real world scenario due to these very visible security concerns but of course that involves considerable more development time to thoroughly address all vulnerabilities.