GitPals Review by Daniil Durnev

GitPals is a web app built on Java, MongoDB. I used Spring Framework to implement security, authorization (there is an oauth2 with GitHub) and manipulation with database. I started working on GitPals on March 8, 2018 when I realized that it is pretty hard to team up with other developers, so GitPals is a web app where you can publish your ideas, prototype or almost finished project and collaborate with other people to have your masterpiece developed.

I faced multiple challenges starting from the very beginning. The first challenge was trying to implement authentication with GitHub, because account on GitHub is mandatory if you want to use GitPals. I could create a Spring Security Configuration that was triggering GitHub auth if you redirected on */login* page. A response was written to *java.security's Principal* class, so it was a really good way to store your user object (that you got as a response after successful authentication). I created a custom *User* class that stores *username, GitHub profile link, skills list, messages, etc.* It was pretty easy to access database user object because there is a function in the database interface class that *returns* you a *User* object by username, that can be fetched from a *Principal* name - your authorization response.

The main idea is that you fill your profile with skills you know and others can invite you or you can apply to projects that match your skills.

So if you look at the project you can see it uses Java as a backend, but for frontend it uses just static html pages, that are stylized using CSS. The only thing that makes it all work (I mean thing that connects backend with frontend) is *Thymeleaf* - a template renderer that is something similar to *Angular*. When you send a request from an html form, backend creates a *Model, which is actually a hashmap*. You store all the required data there (for example you send a request to get a list of user's messages), you add data to a Model and then you display them on html page (like *<h1 th:inline="text">**[[${data}]]**</h1>*). So Thymeleaf renders all the backend responses on static html pages.

What is the *disadvantage* of Thymeleaf + HTML? So the main problem in my opinion is that it is all **static.** I am talking about responses and requests. So each request triggers page reload, where you get a new page with new data from the server. Unlike *REST API* I developed on *Angular*, where you can send an HTTP request to get the required data and then immediately assign that response data to some page variable so it immediately appears on your screen within just seconds, you can't do the same with static html pages. You send a request **->** data is added to hashmap (Model) **->** page is reloaded **->** new data is on your page. That is really sad, especially if you want to make a real

time component (I wanted to create a realtime chat system for GitPals, for now there is a system so when you want to send a message to another user who is registered on GitPals, same process I described above happens. Everything is followed by a page reload. By the way, same happened in my other Spring project that also uses Thymeleaf for frontend rendering. Actually, there is a possible solution I will try to implement later. The solution is *WebSockets*, that will allow me to send data and display it on screen within seconds without reloads and updates, so it will be realtime. It is implemented using JavaScript.

For months I have been improving different bugs that were causing errors and acting as a potential vulnerabilities. Moreover, sometimes I could just look through the code and find such little mistakes as object creation within a loop, which is absolutely wrong because you should create an object only once and then do whatever you want with that (like creating a message with the same text and sending it to 1,000+ users. Message should be created once and sent to users via *for-loop*).

GitPals development has been inactive for about 9 months (I finished development in April and it lasted till December). In Winter 2019 I started implementing new features, such as Java 8 techniques - *Stream API* for example, which allowed me to get rid of huge for-loops (like search feature, where you could find user or project by their name). And many other little mistakes

and bugs were fixed. I published my project to a Heroku hosting so other users could have a look and try the app for themselves, because I can tell you it feels good when you realize that stuff you have been working on for months finally gets on someone's PC. Thanks to Reddit users, I fixed even more bugs. I even thought that bugs will never finish, so I went surfing through files and trying to get them all. I also created forums where you can create your posts and discuss different problems (like Reddit).

Right now (November 2019) I decided to go even further. I realized that today lots of companies and individual developers to create an API for their services. API can be developed using *Java (Spring)*, *JavaScript (Node.js, Express framework)*, *C#* as far as I know and *Python (Flask)*. I wouldn't be able to describe what API is, but all I can say it is a set of tools that have different responses to different requests that can be sent from anywhere. The first example that comes to my mind is *Google's Translate*. You can easily access translate via their page, where you manually choose languages and type text. Also there is a Google Translate API, which can be implemented in your customer service. Moreover, Google Sheets also use Translate API if you implement translate formula in some of your fields. So it is indeed a magnificent tool.

Moreover, *Telegram* created it's own API to allow users to create their own unofficial apps of Telegram. So that's why we can see

many fan-apps of mobile and desktop versions. Frontend can be whatever, backend is done.

Concurrently I am working on GitPals mobile version using *Ionic Framework* (Angular). I use *Axios* for sending HTTP requests and then I display them on the app's screen. It is really convenient to use that API, as it is connected to the same database, so any changes made in mobile app also affect browser version of the app.

But unlike browser version, I am experiencing some challenges. So for authorization in mobile app I use simple GitHub's API to get authenticated user by username and password. The response is stored in ***localStorage***, so it is pretty bad way of doing stuff. Unlike Spring, there is no Principal class analog in Angular, which can reduce the pain for you, so I have localStorage for now. The main problem is that when you want to change something from the mobile app (for example you want to delete you project you created earlier), you sent an object from a frontend that contains your username and project's name, so it looks like that: ***{username: "johndoe", projectName: "BigProject"}.*** The problem is that you can modify localStorage in some way on your smartphone, so you can pretend as the other person. You can easily modify your username in your localStorage and delete someone's project, because backend checks if username **you send** and projectname have common author. I should think about doing

something similar to Principal class, that would contain authentication far away from user, as it does in Spring.

Major features and stuff that is working:

- **Authentication using GitHub oauth2, storing response in Principal, accessing it by username when you want to fetch database user**
- **A good communication between backend and frontend, sending and rendering data properly**
- **Java 8 features such as Stream**
- **Comments written to explain all the functions in Controllers**
- **MongoDB database to store all the data**
- **Mobile app (which is under development)**

Major problems and challenges:

- **It is all static. It is not reactive, so each request is followed by page reload so some features that we are used to see fast are slow. For example - sending messages.** ***Solution:*** **WebSockets.**
- **Design. Right now there is a design I copied from CodeTriage, because it is really hard for me to create a design from my own imagination. Unfortunately, I am not a painter to experience inspiration. I should create a better**

design in the future that is more *"modern"*. *Solution:* find a better UI kit and redraw all the pages.

- **API. Lots of problems and challenges I described before. There should be a safe way of authentication as it is in browser version. If it is not secure as it is right now, when you can put the whole website down by modifying data in your local storage, it shouldn't be in the production. *Solution:* find an alternative of Principal class, storage GitHub authentication response in that class, handling all the security processes far away from users so they wouldn't have access to modify the data.**

Project Structure:

Sometimes I find it hard to understand the structure of some projects I find on GitHub. Especially if that's a project that has been under development for years. And I find it more interesting how users can contribute into all that huge mass of files. So let me explain all the folders and classes of GitPals. All files can be found in ***\src\main\java\com\moople\gitpals\MainApplication*** folder.

- **MainApplication.java** - main file. It has really few lines, just main method to start the app. You should run this file in order to get the app started
- **Configuration folder**. This folder contains only **GitHubAuthConfiguration.java** file. It is a very important

file - it is required for authentication with GitHub. It is a small config that triggers authentication when you open **/login** page. It also disables **CSRF** so server doesn't throw a CORS error when you try to send an API request.

- **Controller folder.** That's the app's logic actually. So controllers are files that handle all the requests that come from a frontend. It handles **GET** and **POST** requests, for example when you send a request to change your profile information, or to send someone a message. Or simply when you try to fetch some information from the server. So as I said earlier controllers are app's brain. There is a different behaviour described on different requests. Controllers' handlers manage data from frontend, save data to database, manipulate objects and respond with a response to frontend.
- **Model folder.** So model folder contains all the objects used within the app. Think for a minute about a classroom. What does a classroom have? Students, teachers, books, tests, etc. So you need to describe all the objects within something as detailed as possible. Same with GitPals, **Model folder** contains all the objects that you can see in GitPals: Users, Messages, Projects, Responses, Forum Posts, Comments. It is all the data you manipulate with. There is a different class for each object. Some objects contain

data structures of another objects (Like Forum Post, that contains a list of Messages).

- **Service folder.** So service folder contains such files as **Data.java** that is used to create a fixed HashMap of possible technologies that user can choose whether they know the technology or not (like you choose skills you know). **ForumInterface.java, ProjectInterface.java, UserInteface.java** are files that save, fetch data from the database. They have such functions as **findAll() - returns List of objects (like List of projects or users, depends on interface class), getByTitle(), getByUsername() etc.**

Months ago I thought that work on GitPals is done, but it only seems to be starting. With that ideas I try to carry out I am sure more challenges are waiting for me. A lack of knowledge in executing features will postpone the **valid** development of some components of the app, but sooner or later I will get it done.

You feel like you can help?

You are welcome to contribute and improve some component of GitPals, whether it is API, backend controller or html page fix.

[Proceed to GitPals repository.]