URECA_Final_Research_Paper_Leow_Ken_Hing_Bryan.doc

by SCSE LEOW KEN HING BRYAN

Submission date: 30-Jun-2023 11:17PM (UTC+0800)
Submission ID: 2124788046
File name: URECA_Final_Research_Paper_Leow_Ken_Hing_Bryan.doc (911K)
Word count: 4425
Character count: 26437

Backend Development of a Multidimensional Frailty Screening Prototype

Leow Ken Hing Bryan
School of Computer Science and Engineering
Nanyang Technological University

Assoc Prof Ho Moon-Ho Ringo
School of Social Sciences
Nanyang Technological University

Mr Pham Tan Phat
Ageing Research Institute for Society and Education
Nanyang Technological University

Abstract **-** In this project, we developed a mobile application to facilitate the self-assessment and screening of multidimensional frailty among the elderly. This prototype is written in Flutter for the application and Java Spring Boot for the application server. This research paper details the methodology we used to develop the application server, including the functional and non-functional requirements we had to satisfy.

Many design decisions and trade-offs had to be made during ideation. This paper elaborates on the rationale and justification for each decision, both from 1) a technical software engineering perspective in order to meet non-functional (ISO25010) requirements, and 2) a non-technical perspective, considering practical business constraints. Through this multidimensional discussion, we hope to present a more holistic view of software development outside of traditional system design paradigms and principles, which are often technical-focused.

Design patterns were also applied during development when we found difficulty in storing information about the questionnaire, since each question has their own set of options and scoring methods which initially made it hard to capture and program for. Through applying the "strategy design pattern", we reduced the questions to a few scoring strategies (e.g., Yes/No, multiple choice, multi-part) and applied the appropriate strategy during code execution to keep our code simple and general.

In conclusion, we developed a robust application that satisfies its functional and non-functional requirements while remaining in line with the business constraints. The result is a maintainable, usable, and secure system that will support the running of our application.

Keywords – software engineering, backend development, Java

1 INTRODUCTION

1.1 BACKGROUND

The Ageing Research Institute for Society and Education (ARISE) is working on the development of an application to facilitate self-assessments of frailty across multiple dimensions. In this research paper, we uncover how a software developer would go about implementing this system through the Software Development Lifecycle (SDLC) and analyse the technical decisions a software developer might need to make throughout the process.

1.2 SCOPE

Application development consists of two interconnected but distinct parts. The front-end (frontend) is the system that faces the end-users and is what normal users will interact with for the entirety of the application. The back-end (backend) is the system that performs actions requested by the user, and interfaces with the application's persistent storage, such as a database. This research paper primarily focuses on the development of the backend and would be more applicable to other researchers looking to implement a backend or full-stack (a combination of both frontend and backend) software solution.

The SDLC process covers the following main steps: [1]

1. Planning
2. Designing
3. Development
4. Testing
5. Deployment
6. Maintenance

For this report, we will mainly cover steps 1 through 3.

While we performed testing, it is rudimentary and not worth mentioning on this research paper. Other research papers, particularly those covering Test-Driven Development [2], would be a better fit for resources covering testing in detail. Deployment and Maintenance were also things we did not cover in detail in this research paper and will thus be glossed through.

1.3 LIMITATIONS

One important limitation of this report is the different operating conditions of our team and the sources cited in this paper. Factors such as team size, programming language, level of expertise, and cultural contexts may differ from us and that of the research papers that we cite, which may impact their extent of applicability. However, we believe the general conclusion derived from the papers are more important and that is what we focused on when citing research papers.

1.4 REPORT STRUCTURE

We leverage on the steps performed in the SDLC to structure our report, covering the development process in a sequential and logical manner.

2 PLANNING

2.1 FUNCTIONAL REQUIREMENTS

Deciding on the requirements of a software system is the first step in the SDLC process. [3] In particular, there are two kinds of requirements: functional, and non-functional. We will discuss non-functional requirements in the following section.

Functional requirements refer to features which our software system must support for the end-user to perform. It is derived from the business use cases of our product and is verified by the product owner [4].

2.1.1 User Groups

In analysing our multidimensional frailty application, we identify 2 distinct user groups:

1. Patients: this is our primary user group and is defined as the elderly who are interested in, or have caregivers who are interested in, administering an assessment for multidimensional frailty for themselves.
2. Caregivers: defined as the people who are in direct care of one or more patients and are responsible for monitoring their condition of frailty. These can include clinicians, next-of-kin, and the like.

2.1.2 Functional Specifications

Next, we identify the actions that our user groups might want to perform on our application. We distil them into the following most important features:

1. As a patient or caregiver, I want to choose which dimension of frailty to assess myself for, and which questionnaire type of that frailty I wish to use
2. As a patient, I want to save my answers to a questionnaire and return to it at a later time.
3. As a patient, I want to submit my answers to the questionnaire.
4. As a patient, I want to immediately receive a frailty assessment when I submit a completed questionnaire.
5. As a caregiver, I want to retrieve the history of my patients' answers and their frailty assessments.

2.2 NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements are used to evaluate the quality of a software product and are defined in terms of characteristics.

In defining non-functional requirements, we refer to the 8 quality characteristics as defined in the International Organization for Standardization (ISO) standards, in particular ISO 25010 [5].

These characteristics all have inevitable trade-offs, and it is not possible to design a system that completely meets all characteristics. [6] Therefore, we must decide on the characteristics that are most important to us. Based on the needs of our organisation and user groups, we identify the following as most important:

2.2.1 Maintainability

With constantly evolving research in the field of frailty, the range of questionnaires offered by our application is expected to increase as our application matures.

Therefore, it is essential that our code is written in a way that makes it easy for future developments to be added, especially new questionnaires.

2.2.2 Usability

In accommodating our primary user group, we must ensure that our application is accessible to them and friendly to their needs. Some traits of the elderly we must take particular note include the following: [7]

- Language barriers: English might not be their primary language
- Eyesight: The standard font and icon sizes might be too small for them to comfortably see

2.2.3 Security

Medical data is highly confidential. The medical sector is thus prone to many cyber attacks that, in some cases such as SingHealth, cause millions of dollars of damage. [8] Due diligence must hence be carried out to secure our systems from a data breach.

2.3 TECH STACK

A technology stack ('tech stack') refers to the technologies one uses when developing a software solution. Technologies include programming languages, software libraries, dependencies, and the environment the software is deployed on.

Due to the phenomenon of technology lock-in [9], it is expensive and resource-intensive to change a tech stack once decided. Hence, selecting the most appropriate technology stack ('tech stack) is a crucial decision to make for any software development project. In particular, two factors stand out as the most essential during the selection of a tech stack: [10]

2.3.1 Supports the product's purpose

The stack must suit the business needs of the product and possess all the features needed for the product to deliver on its requirements, both functional and non-functional.

To satisfy the non-functional requirement of maintainability, we opt for strongly-typed, object-oriented programming languages. Strongly-typed languages, also known as static typing, are empirically shown to be beneficial for maintainability [11]. Object-oriented languages are also shown to be maintainable when applied in an appropriate manner but result in longer development times [12].

Based on this preliminary filtering, we are left with the following programming languages that 1) are suited for backend development, 2) are object-oriented first, and 3) provide in-built static typing support:

- ASP.NET C#
- Java Spring framework
- Go Gin framework

2.3.2 Presence of present and future resources

Resources are defined as those that are currently available in the organisation, and those that can be recruited into the team in the future [10]. In this case, the organisation refers to ARISE and the campus student body of Nanyang Technological University (NTU) at large. With reference to NTU's School of Computer Science and Engineering curriculum, all computing students would have been exposed to object-oriented languages in the course "SC2002: Object-oriented Programming", particularly in Java which consists of 75% of the course.

With reference to the 3 shortlisted languages, this makes Java as a resource much more available than the 2 alternatives. We hence settled on the Java Spring framework, particularly Java Spring Boot 2.7, which is the most recent stable version available as at the time of writing.

3 DESIGNING

3.1 ARCHITECTURE DIAGRAMS

The next step after planning is to design our application through the use of Unified Modelling Language (UML) diagrams. In the theory of software engineering, there are more than 10 types of diagrams one can draw to visualise and document a software system [13].

However, creating such documentation also comes with trade-offs, namely in the resources required to maintain them through changing business requirements. With reference to [14], I. Balosin argues that stakeholders are mainly concerned with 1-2 high-level diagrams for a surface-level understanding of the system.

In line with this principle, we proposed the following architecture diagram, which consists of both a Use Case diagram and Class diagram:

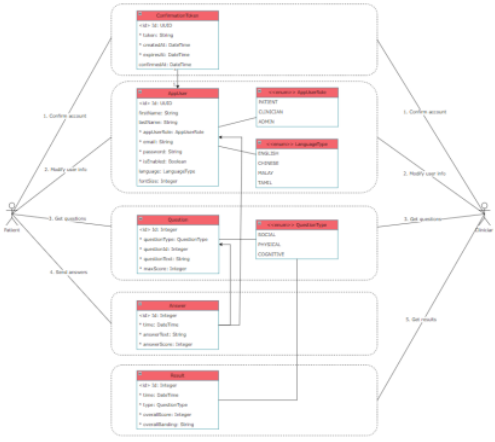

Figure 1: High-level architectural diagram for our multidimensional frailty assessment backend system.

3.2 MEETING NON-FUNCTIONAL REQUIREMENTS

In designing our systems, we must also determine the features our system is required to support in order to meet its non-functional requirements. In particular, implementing usability and security required the following features to be implemented:

3.2.1 Meeting Maintainability with Domain-Driven Design

In order to write modular, reusable, and understandable code, a popular modern paradigm in software engineering is to apply Domain-Driven Design (DDD), pioneered by Eric Evans, which consist of the following main components: [15]

- Entities: business objects that exist in the system
- Value Objects: transitory objects meant to carry data from Entities into Services
- Services: a front-facing layer that performs the business logic as required by our user groups.

One important concept in DDD is to establish what is called a ubiquitous language, a common taxonomy shared between developers and ingrained in the domain model [16]. This includes differentiating type instance homonyms, which is when the same word is used to describe 2 different objects [17].

3.2.2 Meeting Usability with Localisation

Localisation ("l11n") is the skill of adapting the user experience (UX) for different markets [18]. This is especially important due to the needs of our user group, as mentioned in 2.2.2.

3.2.3 Meeting Security with Authentication and Hashing

Authentication in the context of applications refers to the procedure of verifying a user's identity i.e. that a user is who they say they are [19]. This is important in implementing access control, where only privileged accounts are allowed to perform certain actions [20]. There are 3 main ways to provide an authentication service:

- Session State: a piece of data containing information about the user. This can be local data (cookies) or server-side data
- JSON Web Token (JWT): a signed string consisting of a header, payload, and signature. This represents the credentials of a user [21].
- Open Authorisation (OAuth): an authorisation framework where an application is granted access to the user's credentials from a third-party credential provider, such as Google [22].

We compare these 3 methods:

Session State	JWT	OAuth
Can control and guarantee your own data		Data not guaranteed from third party provider
Need to manage session location in distributed systems	Stateless, works with distributed systems	
Needs to securely store user information		Convenient for user and developer, prone to policy changes by provider

Our application will require specific user data for accessibility functions, such as collecting user's preferred language mode, font size, etc. Thus, we would prefer to control and guarantee our own data, negating the benefits of OAuth since we will still need to store user data on our end.

Between Session and JWT, Session has the added security requirement of storing session state securely in a way that is confidential. Given this, we determine JWT to be the most suitable form of authentication for our application.

Hashing, specifically password hashing, is another security measure where the user's password is put through a hashing algorithm to convert a plaintext string into one that is completely random. This is to ensure that in the event of a breach of confidentiality, any compromised passwords are unusable to the bad actors for logging in [23]. A hashing algorithm should be simple to calculate, virtually impossible to reverse, and generates a unique hash for each input [24]. Based on these requirements, bcrypt is chosen as our selected hashing algorithm, since it was purpose-built to slow down brute force attacks. It automatically comes with 'salting' in the hashing process, which adds another layer of complexity for bad actors trying to guess the password from the hash.

4 DEVELOPMENT

4.1 DOMAIN-DRIVEN DESIGN

We applied DDD in the organisation of our codebase, splitting components into Entities, Value Objects, and Services. Value Objects are also known as Data Transfer Objects (DTO), and this is the taxonomy that we go with.

On top of these components defined in DDD, Java Spring framework also provide an additional component called Repositories to interface with the underlying database [25].

Java also has many mapping frameworks available, of which we opted for MapStruct for its performance and ease of documentation [26].

In order to support the application, there are also many configuration files we needed for security and for seeding the database. Database seeding is the process of adding an initial set of data into a database [27].

Lastly, as usability is an important part of our application, we structured our codebase with an output folder, containing any components that produce an end result that is visible to our users.

The codebase is hence structured like the following:

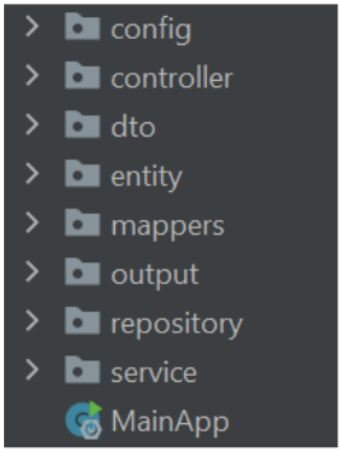

Figure 2: Organisation of our codebase into folders, guided by the principles of DDD.

As mentioned in 3.2.1, another important goal of DDD is to establish a ubiquitous language to avoid type instance homonyms. Since there were 3 dimensions to frailty – social, physical, and cognitive – and at least 3 different questionnaires that a user can do within each frailty type – e.g. Phenotype, IPAQ-E, and SARC for physical frailty – there was a need to distinguish between a single question in a questionnaire, and a questionnaire by itself. We apply this distinction carefully in our taxonomy, defining QuestionType to refer to the different dimensions of frailty as seen in Figure 3, and QuestionnaireType to refer to the different question banks that a user can attempt within a single frailty dimension as seen in Figure 4.

```
public enum QuestionType {
    1 usage
    SOCIAL,
    16 usages
    PHYSICAL,
    no usages
    COGNITIVE
}
```

Figure 3: QuestionType enum defining the dimensions of frailty.

```
public enum QuestionnaireType {
    6 usages
    PHENOTYPE( displayName: "Frailty Phenotype Questionnaire"),
    5 usages
    IPAQ( displayName: "IPAQ-E"),
    6 usages
    SARC( displayName: "SARC-F");
```

Figure 4: QuestionnaireType enum defining the different questionnaires accessing for physical frailty.

4.2 LOCALISATION

We did not manage to implement localisation with languages common in Singapore, such as English, Chinese, Malay, and Tamil. However, we implemented a scaffold for localisation by creating a localiser class and having it parse all error messages, like so:

```
if (!isValidEmail) {
    throw new IllegalStateException(localiser.invalid( fieldName: "Email", request.email()));
}
```

Figure 5: Example of the localiser class in action. Here, we call the invalid() method of localiser to return a message saying that the Email is an invalid field.

The localiser, in turn, would have had the logic to return the appropriately-translated string. In the final implementation, the function would read in the preferred language of the user and return the appropriately-translated string.

```
@Service
public class Localiser {
    2 usages  ♣ bleow +1
    public String invalid(String fieldName, String userInput) {
        return String.format("%s is not a valid %s", userInput, fieldName);
    }

    4 usages  ♣ bleow +1
    public String notFound(String fieldName, String userInput) {
        return String.format("No such %s found with %s", fieldName, userInput);
    }
```

Figure 6: Snippet of the localiser class.

4.3 JWT AUTHENTICATION, BCRYPT

4.3.1 @Configurations and @Beans

Before we explain the process of implementing JWT authentication and bcrypt encryption, we first discuss how the Java Spring framework manages classes.

Spring uses annotations for developers to label classes with certain functionalities. These annotations start with an '@' symbol, followed by the annotation [28]. In this section, we will explore the @Configuration annotation, @Bean annotation, and @Autowired annotation.

@Configuration is an annotation to inform Spring that a class is to be used to configure a component in the system. For example, in Figure 5, we annotated the SecurityConfig class to be a @Configuration. All @Configuration classes will be searched by Spring for any @Beans.

@Bean is then an annotation used to inform Spring that a method is available for use in another component, known as injection. When a @Bean method exists inside a @Configuration class, Spring knows that this @Bean method is being used to configure a specific system property and will inject this @Bean into the system accordingly.

4.3.2 JWT Authentication

With this understanding of @Configurations and @Beans, we can finally implement JWT through the following. First, we define a Security Filter Chain @Bean inside a SecurityConfig @Configuration, as seen in Figure 7. This will allow us to override Spring's default security filter chain with our own. Next, we define our security filter chain to require authentication on all endpoints, except for the registration endpoint (/api/v*/registration/**) and the Swagger specification endpoint (/swagger-ui/**). With this, we have secured our endpoints to ensure only authenticated users can access them.

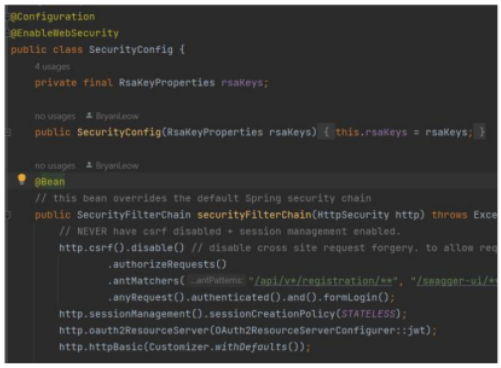

Figure 7: Snippet of the Security Chain Filter bean method in the Security Config configuration class, requiring authentication for all endpoints except for registration.

On top of implementing access control, we also need to implement a mechanism to encode a user's details into JWT, and to decode a JWT into user's details. This is where the jwtDecoder and jwtEncoder @Beans come in. As seen in Figure 6, they help us convert user details into JWT and back, and we call them using the syntax shown in Figure 9.

It is important to note that we encode and decode the JWT by making use of a Rivest-Shamir-Adleman (RSA) key – a very long and random string of characters. The strength of our encoding and decoding lies in the length, randomness, and secrecy of our RSA key.

Our RSA key must be kept in a safe location, away from bad actors. It is also best practice to rotate them frequently [29].

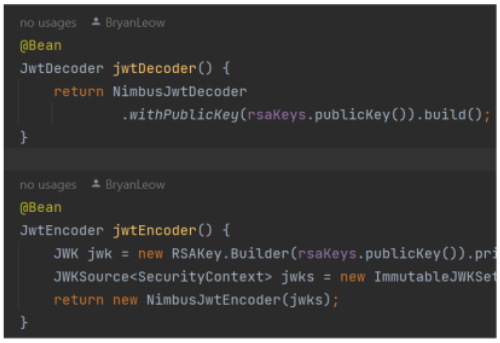

Figure 8: Snippet of the JWT Encoder and Decoder methods.

Figure 9: Example of the JWT Encoders and Decoders in action, producing a JWT for a user.

A similar concept is used in the implementation of bcrypt hashing. As seen in Figure 10, we defined a @Bean method in a @Configuration class for encoding passwords using bcrypt, and use them in our code as seen in Figure 11 to generate the hashed password for storing.

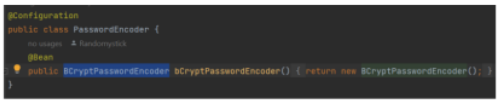

Figure 10: Snippet of the bcrypt password encoder.

Figure 11: Example of the bcrypt password encoder in action, encrypting a password for storage.

4.4 DESIGN PATTERNS

4.4.1 Variability of Questions

Due to the nature of questionnaires, there was much variability in the way different questions can be phrased. Two examples of questions include the following:

During the past week, I felt that everything I did was an effort	**Options** A ~ Rare (less than 1 day/week) B ~ Sometimes (1-2 days/week) C ~ Often (3-4 days/week) D ~ most (over 5 days/week) **Scoring** 0 = option A or B 1 = option C or D

The first question is about the time you spent **sitting** on weekdays during the **last 7 days**. Include time spent at work, at home, while doing course work and during leisure time. This may include time spent sitting at a desk, visiting friends, reading, or sitting or lying down to watch television. During the **last 7 days**, how much time did you spend sitting on a week day?	<Q1h> **hours** <Q1m> **minutes per day**

Figure 12: Two examples of questions that can be asked in the questionnaires, taken from physical Phenotype questionnaire and physical IPAQ-E questionnaire respectively.

As it can be observed, each question can have their own set of options (or not have options at all), and their respective scoring methods.

It is not practical to program our Service to contain the logic for calculating the score of every single question, as that would make the logic incredibly complex to capture and program for. It will also make the addition of future questions, potentially with never-before-seen answer types and scoring, exponentially harder to add.

4.4.2 The "Strategy" Design Pattern

First introduced in the influential book "Design Patterns: Elements of Reusable Object-Oriented Software", [30] the strategy design pattern is a behavioural design pattern whose aim is to allow programmers to create a set of algorithms, place all of them into their own classes class, and make all these classes interchangeable with one another [31].

By applying this design pattern, we can model each question to have their own 'strategy' of scoring their answers and giving a score. We hence define an interface for Scoring Strategies, like so:

```
public interface IScoringStrategy {
    1 usage  2 implementations  BryanLeow
    Double calculateScore(String answer);
    1 usage  1 implementation  BryanLeow
    Double calcMaxScore(Map<String, Double> scoreMapping);
}
```

Figure 13: Scoring Strategy interface. Each strategy must have a method to generate their own score, as well as calculate the maximum possible score that this question can provide.

In each implementation of a Scoring Strategy, we then define our own method of calculating score. For example, Figure 14 shows the method of calculating score for questions that define a range of numbers as one score, and the next range of numbers as another.

```
@Override
public Double calculateScore(String answer) {
    // we cannot guarantee that the scoreMapping is sorted.
    // verifying it is sorted takes O(n) time. sorting scoreMapping ourselv
    // either way, the best case when assuming it is sorted is O(n): verify
    // so let's just assume it is not sorted and do it naively O(n)
    Double ans = Double.valueOf(answer);
    Double res = -1.0;
    for (Map.Entry<String, Double> entry : getScoreMapping().entrySet()) {
        Double key = Double.valueOf(entry.getKey());
        if (key == ans) {
            res = entry.getValue();
            break;
        }
        if (key < ans && entry.getValue() > res) {
            res = entry.getValue();
        }
    }
    return res;
}
```

Figure 14: logic to calculate the score of a question that matches a range of numbers to a particular score, e.g. 1-2: 1 point, 2-4: 2 points, etc

When adding a question, we hence perform the following, as seen in Figure 15:

1. Define the score mapping for the question
2. Save this score mapping as a strategy. Store this in the database.
3. Take this strategy and define it together with the question, question type, questionnaire type, and question number.

```
scoreMapping.clear();
scoreMapping.put("A. No", 0.0);
scoreMapping.put("B. Yes", 1.0);
strategy = new SingleChoiceScoringStrategy(scoreMapping);
scoringStrategyRepository.save(strategy);
questions.add(new Question(QuestionType.PHYSICAL, QuestionnaireType.PHENOTYPE, ++question_number,
        questionText: "Has there a loss of >=5% in the last year?", strategy));
```

Figure 15: logic to add a question into the database.

4.4.3 Benefits of Applying Strategy Design Pattern

Through applying the "strategy design pattern", we reduced the questions to a few scoring strategies (e.g., Yes/No, multiple choice, multi-part).

The main benefit of doing so is the massive simplification at the service level. As a case in point, Figure 16 provides the logic required to calculate questionnaire score. At less than 20 lines long, it is able to calculate the questionnaire score any number of questions, for any questionnaire

```
public String postAnswers(String typeStr, String username, List<AnswerRequest> answerRequest) {
    QuestionnaireType type = QuestionnaireType.valueOf(typeStr.toUpperCase());
    log.debug("++TESTING++ USERNAME {}", username);
    AppUser appUser = appUserService.getValidUser(username);
    ArrayList<Answer> answers = new ArrayList<>();
    Double totalScore = 0.0;
    Double maxScore = 0.0;
    for (AnswerRequest request : answerRequest) {
        Question question = questionRepository.findByQuestionNumberAndQuestionTypeAndQuestionnai
        ScoringStrategy strategy = question.getStrategy();
        log.info("Question is {} {}", question.getQuestionText(), strategy.getScoreMapping().toS
        Double answerScore = strategy.calculateScore(request.answerText());
        maxScore += strategy.getMaxScore();
        totalScore += answerScore;
        Answer answer = new Answer(appUser, request.datetime(), question, request.answerTextToSt
        answers.add(answer);
    }
```

Figure 16: logic to calculate score. The red lines highlight where the strategy design pattern was used to simplify the code logic.

and applied the appropriate strategy during code execution to keep our code simple and general.

5 TESTING

5.1 POSTMAN

Basic endpoint testing was done via Postman. Postman is an API platform meant for programmers to test their endpoints. It comes with handy tools to simulate HTTP requests, complete with headers and authorisation bearers such as the aforementioned JWT.

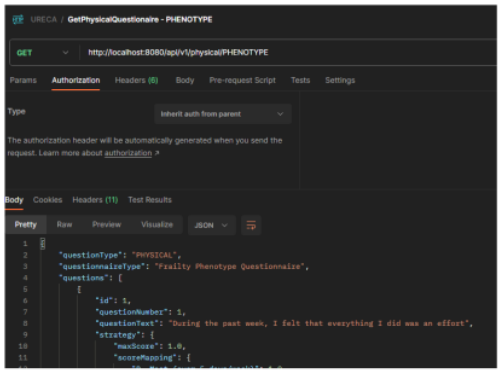

Figure 17: the user interface of a Postman request, ran against our backend. Here, we are making a GET request to the physical/PHENOTYPE endpoint that retrieves the phyiscla Phenotype questionnaire.

5.2 SWAGGER OPENAPI SPECIFICATION

Swagger is the standard way of documenting APIs [32]. It is compliant with the OpenAPI Specification, a standard format to define, structure, and syntax REST APIs [33].

It automatically generates API documentation based on the code, making it zero-maintenance code documentation.

6 CONCLUSION

In conclusion, we have developed the backend for our multidimensional frailty prototype that satisfies its functional and non-functional requirements. We applied domain-driven design and design patterns to architect a solution that is maintainable. We also made use of good software engineering practices, such as authentication through JWT, encryption-at-rest through bcrypt, localisation, and automated documentation through Swagger to ensure our solution is maintainable, usable, and secure.

Works in the near future include the full implementation of localisation for all languages and deployment onto the cloud such as Amazon Web Services (AWS). For the longevity of the project, we could also consider implementing test-driven development practices, and possibly moving to a microservices-based architecture, as both of these are also relatively modern software engineering practices that can benefit a system.

ACKNOWLEDGMENT

I would like to acknowledge the funding support from Nanyang Technological University – URECA Undergraduate Research Programme for this research project..

REFERENCES

[1] R. K. Sharma, “SDLC: What it is, Software Development Processes & Best Practices”, August 2022; https://www.netsolutions.com/insights/software-development-life-cycle/

[2] Simo M, Jürgen M, “Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies”, January 2014; https://www.researchgate.net/publication/256848134_Effects_of_Test-Driven_Development_A_Comparative_Analysis_of_Empirical_Studies

[3] Software Testing Help, “What Is Requirement Analysis And Gathering In SDLC?”, May 2023; https://www.softwaretestinghelp.com/requirement-analysis-in-sdlc/

[4] altexsoft, “Functional and Nonfunctional Requirements: Specification and Types”, July 2021; https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/

[5] International Organization for Standardization , “ISO/IEC 25010: 2011 – Systems and software engineering”, January 2017; https://www.iso.org/standard/35733.html

[6] S. Darius, A. Paris, “Quality attribute trade-offs in the embedded systems industry”, January 2020; https//pure.rug.nl/ws/portalfiles/portal/147538549/Sas_Avgeriou2020_Article_QualityAttributeTrade_offsInTh.pdf

[7] Kamal S. , Zehang C., “User Interface Design for the Asia Elderly: A Systematic Literature Review”, January 2019; https://ir.uitm.edu.my/id/eprint/46282/1/46282.pdf

[8] PriceWaterhouseCoopers (PWC), “Key lessons from the Public Report of the Committee of Inquiry (COI) on SingHealth cyber attack”, Feburary 2019; https://www.pwc.com/sg/en/risk-

assurance/assets/coi-on-singhealth-cyber-attack-201901.pdf
[9] T. J. Foxon, Technological Lock-In, December 2013; https://www.researchgate.net/publication/288174925_Technological_Lock-In
[10] H. Martinsson, V. Svanqvist, “Technology stack selection: Guidelines for organisations with multiple development teams”, June 2022; https://hj.diva-portal.org/smash/get/diva2:1681404/FULLTEXT01.pdf
[11] S. Hanenberg, S. Kleinschmager, R. Robbes, E. Tanter, A. Stefik, “An empirical study on the impact of static typing on software maintainability”, December 2013; https://pleiad.cl/papers/2014/hanenbergAl-emse2014.pdf
[12] K. S. Sand, T. Eliasson, “A comparison of functional and object-oriented programming paradigms in JavaScript”, June 2017; http://www.diva-portal.se/smash/get/diva2:1115428/FULLTEXT02.pdf
[13] Nisahda, “UML Diagram Types Guide: Learn About All Types of UML Diagrams with Examples”, September 2022; https://creately.com/blog/diagrams/uml-diagram-types-examples/
[14] I. Balosin, T. Betts, “Why Do We Need Architectural Diagrams?”, January 2019; https://www.infoq.com/articles/why-architectural-diagrams/
[15] E. Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software”, August 2003.
[16] M. Fowler, “UbiquitousLanguage”, October 2006; https://martinfowler.com/bliki/UbiquitousLanguage.html
[17] M. Fowler, “TypeInstanceHomonym”, January 2007; https://martinfowler.com/bliki/TypeInstanceHomonym.html
[18] Phrase, “UX Localization: How to Adapt User Experience for International Users”, April 2023; https://phrase.com/blog/posts/app-localization-developers-guide-to-user-experience/
[19] Oracle, “Understanding Application Authentication”; https://www.ibm.com/support/pages/understanding-application-authentication-and-authorization-security
[20] B. Lutkevich, “What is access control?”, July 2022; https://www.techtarget.com/searchsecurity/definition/access-control
[21] Auth0, “Introduction to JSON Web Tokens”; https://jwt.io/introduction
[22] R. Sobers, “What is OAuth? Definition and How it Works”, April 2012; https://www.varonis.com/blog/what-is-oauth
[23] J. Jung, “What is Password Hashing?”, May 2021; https://www.okta.com/sg/blog/2019/03/what-are-salted-passwords-and-password-hashing/#:~:text=Password%20hashing%20is%20defined%20as,unintelligible%20to%20the%20bad%20actor.
[24] Code Signing Store, “What is the Best Hashing Algorithm?”; https://codesigningstore.com/what-is-the-best-hashing-algorithm
[25] O. Gierke, T. Darimont, C. Strobl, M. Paluch, J. Bryant, G. Turnquist, “Spring Data JPA – Reference Documenation”, June 2023; https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
[26] baeldung, “Performance of Java Mapping Frameworks”, September 2022; https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
[27] Microsoft, “Data Seeding”, December 2022; https://learn.microsoft.com/en-us/ef/core/modeling/data-seeding
[28] J. T., “Spring Framework Annotations”, September 2017; https://springframework.guru/spring-framework-annotations/
[29] M. Miller, “SSH Key Management Overview & 10 Best Practices”, November 2022; https://www.beyondtrust.com/blog/entry/ssh-key-management-overview-6-best-practices
[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, October 1994.
[31] Refactoring Guru, “Strategy”, January 2023; https://refactoring.guru/design-patterns/strategy
[32] Javatpoint, “Swagger Tutorial”; https://www.javatpoint.com/swagger
[33] Stoplight, “What is OpenAPI? OpenAPI Definition and OpenAPI Standards”; https://stoplight.io/openapi

URECA_Final_Research_Paper_Leow_Ken_Hing_Bryan.doc

ORIGINALITY REPORT

3%	2%	1%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Nanyang Technological University Student Paper	2%
2	qa-eprints.qut.edu.au Internet Source	<1%
3	sdsc.sg Internet Source	<1%
4	www.coursehero.com Internet Source	<1%
5	"Systems, Software and Services Process Improvement", Springer Science and Business Media LLC, 2020 Publication	<1%
6	www.slideshare.net Internet Source	<1%
7	openprairie.sdstate.edu Internet Source	<1%
8	wiki.control.fel.cvut.cz Internet Source	<1%

Exclude quotes	On	Exclude matches	< 3 words
Exclude bibliography	On		