Open Dynamic Interaction Network: a cell-phone based platform for responsive EMA

Gisela Font Sayeras
University of Nebraska - Lincoln, gfontsayeras2@huskers.unl.edu

OPEN DYNAMIC INTERACTION NETWORK: A CELL-PHONE BASED PLATFORM FOR RESPONSIVE EMA

by

Gisela Font Sayeras

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professors Jitender Deogun and Bilal Khan

Lincoln, Nebraska

May, 2020

OPEN DYNAMIC INTERACTION NETWORK: A CELL-PHONE BASED PLATFORM FOR RESPONSIVE EMA

Gisela Font Sayeras, M.S.

University of Nebraska, 2020

Adviser: Jitender Deogun and Bilal Khan

The study of **social networks** is central to advancing our understanding of a wide range of phenomena in human societies. Social networks **co-evolve** concurrently alongside the individuals within them. **Selection** processes cause network structure to change in response to emerging similarities/differences between individuals. At the same time, **diffusion** processes occur as individuals influence one another when they interact across network links. Indeed, each network link is a logical abstraction that aggregates many short-lived pairwise interactions of interest that are being studied.

Traditionally, network co-evolution is studied by periodically taking static snapshots of social networks using **surveys**. Unfortunately, participation incentives make surveys costly to deliver, which makes it impractical to collect snapshots at fine temporal resolution. On the other hand, collecting data at wider time intervals requires participants to perform error-prone recall about long periods of time. This creates a difficult research tradeoff between data cost and data quality.

More recently, techniques of Ecological Momentary Assessment (EMA) have been developed, involving repeated sampling of subjects' current behaviors and experiences in real time, in subjects' natural environments. This thesis project describes the design, implementation, and validation of a new platform for **responsive** EMA.

The Open Dynamic Interaction Network (ODIN) platform is a cost-effective and flexible cell-phone based platform to collect continuous time sensor data and deliver contextual surveys to a study population. ODIN allows social and behavioral health researchers to instrument study protocols by specifying both the **questions** to be asked and the **rules** governing *when* questions should be asked over the duration of the study. Researcher-specified rules can reference sensor data (e.g. time, GPS, accelerometer-based activity, Bluetooth-based proximity to other participants, etc), as well as the subject's previous answers. ODIN is composed of four backend services, two web user interfaces, and an Android application. A pilot study was conducted over the course of 30 days with 14 participants to evaluate the system. The results obtained from the pilot show that the system successfully collects relevant data for the study as well as triggering questions according to the study needs.

ACKNOWLEDGMENTS

ODIN was originally conceptualized and designed in 2013-2015 by an interdisciplinary team of social and behavioral scientists as part of National Science Foundation project SMA 1338485 (PI Khan). ODIN's development was subsequently supported by the National Institutes of Health through grant R01 GM118427 (PI Dombrowski, Khan) during the period 2016-2019, and this M.Sc. thesis was funded in part by the latter grant. As part of my thesis, I led a team of software developers through the iterative process of implementing and testing the ODIN software platform. In the course of doing this, I found that many aspects of the original design put forth in the NSF funded project had to be modified in light of new end-user requirements and emerging cellphone technologies/constraints. This thesis documents the new requirements, revised design, software implementation, and preliminary evaluation of the ODIN system.

I want to thank Jitender Deogun for his support and motivation throughout my master's degree; Bilal Khan for encouragement and guidance throughout the last 3 years working on ODIN; The other members of the social research group that were involved in the aspects of computer programming of the system. This work would not have been possible without the collaborations and input of the software development team: Alekhya Bellam, Yijie Yan, Kin Pi, Maisha Jaunering, Daniel Schaefer, Anthony McIntosh, Michael Shanahan, Kim Nguyen, Emery Tanghanwaye, Max Nguyen, Andrew Bueide, and Navya Singh; earlier versions of the ODIN system were developed by Andy Peng, Hasan Eray Dogan, Carlos Castilla, Mateusz Opalinski, and Anna Wisniewska; Devan Crawford, Patrick Habecker, and James Lindsay for their help with the design of the ODIN website and the phone application; Trey Andrews, Becca Brock, and Kim Tyler for giving feedback and making the ODIN pilots possible. These last 3 years have been challenging, but working with the SNRG team made it enjoyable.

I would also like to thank my friends, especially the Barteks, that supported me throughout this journey and Pedro Albuquerque for his love and support. Finally, I would like to thank my parents and grandparents for all their love and support throughout these years despite the distance. My father is an inspiration for this thesis because he has Parkinson Disease, and he is a participant on a research project that uses a phone application weekly for 2 years to examine his progress. I hope that ODIN would help other patients and researchers in the areas of psychology and medicine.

GRANT INFORMATION

This work was originally conceptualized and designed in 2013-2015 by an interdisciplinary team of social and behavioral scientists as part of National Science Foundation project SMA 1338485 (PI Khan). ODIN's development was subsequently supported by the National Institutes of Health through grant R01 GM118427 (PI Dombrowski, Khan).

Table of Contents

List of Figures

List of Tables

Chapter 1

Background and motivation

The study of social networks is central to understanding human behavior and social connections. There is a vast body of research on people's social conduct [2, 3], but quantitative studies involving longitudinal surveys are often hampered by data quality issues stemming from biases in a recall. In more recent years, social scientists have made use of the Experience Sampling Method (ESM), and Ecological Momentary Assessment (EMA). ESM/EMA has been particularly fruitful in psychology, sociology, and medicine because it allows for the collection of data in situ over long periods [4, 5, 6, 7, 8]. Myin-Germeys et al. (2009) state that "The Experience Sampling Method (ESM) allows us to capture the film rather than the daily life reality of patients" [9].

Traditional surveys are more susceptible to the collection of biased data since they do not incorporate contextual information about the subject's natural environment. According to Krosnick et al., participants tend to change their answers during in-person surveys to feel more "socially desirable" [10]. Another factor that produces different answers in participants is tracking the pace at which the subject is being interviewed [11]. For instance, a person could feel more stressed in the morning and more relaxed in the afternoon, and thus, taking the survey at different times, we obtain conflicting results to short-timed surveys. In all of these cases, the participant's context leads to bias and the absence of relevant data. As researchers, we want participants to feel more engaged in the study and researchers to obtain more accurate data.

Increasing the data collection difficulties is the growing need to collect more fine-grained data on populations and their interconnections, over long timescales. In the past, Social Network Analysis (SNA) has sought to render social relationships as taking place in mostly stable networks [2, 3]. Indeed, most of the 50,000 studies registered in ClincalTrails.gov [12] were conducted via in-person, which produces a static snapshot, and requires a lot of researcher and field team expenses. The financial burdens of these approaches limits the feasible sample size. To date, financial considerations have made impossible to collect high resolution data about a social network over long time scales. To alleviate the issues surrounding longitudinal data collection, methodologies like ERGM [13, 14, 15, 16] and SIENA [17] adopt an approach of analyzing a sequence of snapshots of social networks to infer conclude their simulated dynamics. However, network snapshots fail to capture the interactions which underlie them (the evidence relationships), and are thus likely to miss some of the processes at play.

Additionally, big companies started creating applications to overcome these difficulties for users to record their data. For example, Google bought Behavio[1] which is an Android phone application developed by MIT that records sensor data such as location, activity, proximity, etc and analyzes this data to draw conclusions based on anomalies in the subjects behavior. Another example, Microsoft implemented a web-based application (HealthVault[2]) for users to record their health information. However, it was officially shut down in November of 2019 due to its low popularity. The fact that it was not designed to run on a mobile platform was one of the reasons

[1]http://www.behav.io

[2]https://international.healthvault.com

for its failure.

The different methods applied for real-time data collection to understand a subject's behavior in their natural environment is known as EMA [18]. EMA is becoming more popular due to the increase of population owning smartphones and Internet access [19]. Existing research on the cell phone application Enhanced New Mothers Network (ENMN) shows that subjects who own smartphones seem to be comfortable with phone surveys [20, 21]. The main features of EMA are data collection over the subject's daily life, reporting of the participant's current state, questions prompted when there is an event of interest, and multiple surveys completed throughout a period that provide changes in the subject's behavior [22]. For example, in a study of drug abuse using EMA could involve recording GPS and Bluetooth data to know how actively the participant is using, and if there is any relationship established among other subjects within the same study. Additionally, the participant might use EMA to report when they are considering taking drugs or any other relevant events. Because EMA methodology provides more accurate data on the subject's life, it facilitates better research across a wide range of areas of study.

EMA's novel approach provides more data for researchers to analyze but brings new challenges that researchers and developers need to consider [23]. To date, there have been a large number of EMA surveys, but there is little literature analyzing the difficulties with EMA itself [23]. Van Berkel et al. state that phone-based EMA is not yet fully developed, and researchers face considerable difficulties during the use of phones in EMA studies, due to the lack of suitable software availability [23].

In this thesis, we present ODIN, a software system that overcomes these limitations and allows researchers to collect more accurate and meaningful realtime data about a population. ODIN is an innovative software platform that participants register on a phone application and carry throughout the study, which could vary from several days

	mPower	PACO	TEMPEST	Jeeves	ODIN
Researcher-editable surveys	x	✓	✓	✓	✓
Sensor data recording	✓	x	x	✓	✓
Time-based prompting	✓	✓	✓	✓	✓
Sensor-data based prompting	x	✓	✓	✓	✓
Live data review & visualization	x	x	x	x	✓
Built-in sensorial services	x	x	x	x	✓

Table 1.1: Feature comparison of EMA applications

to months. The long-time surveys result in more accurate data since many questions are asked during the day.

To date, there has been some literature based on phone applications similar to ODIN. On the one hand, there exist phone applications that collect sensor data and immediate context-based questions for particular domains. One example of this is mPower [24]; a mobile application used to study Parkinson Disease (PD). Other examples of applications are MobiClique, BlueAware, and E-Small Talker which use Bluetooth sensor proximity to form social graphs based on ad hoc social networks [25, 26, 27]. In addition to specialized platforms, there are other more general-purpose systems like MyExperience [28] and EmotionSense [29]. Unfortunately, these systems require researchers to have some programming knowledge. For example, MyExperience requires writing complex scripts in XML files. Likewise, EmotionSense facilitates the design of ESM applications but still requires some programming skills and ability to do Android development. As a result, researchers' expenses are increased due to the need to purchase software or hardware [30, 31, 32], as well as the expertise of on-staff programmers [33] to instrument the surveys.

Table 1.1 presents a list feature comparison of other systems similar to ODIN. The applications have been chosen based on the most recent and similar to ODIN; mPower, PACO, TEMPEST, and Jeeves. The main features that make ODIN stand out from the other systems are researcher-editable surveys, sensor data recording, time-based prompting, sensor-data based prompting, live data review and visualization, and built-in sensor services. Next, each of these features is described and compared with the other systems in more detail.

Ideally, researchers should be able to program their surveys without the need to hire developers. Systems like Paco [34], Jeeves [35], and TEMPEST [36] try to provide this feature by supporting a user interface that facilitates the study creation by researchers. Jeeves uses block-based visual programming for the creation of surveys. In contrast, TEMPEST uses a user-friendly interface, but researchers still need to have basic programming knowledge of if statements since they are necessary to create conditionals. ODIN provides a web user interface that does not require any programming knowledge. The researcher can create new questions and rules by simply selecting them from the dropdown and filling out the required fields. Each rule is described in a few sentences at the top of the pop-up. Then, the user has to fill in the blanks.

Unlike many existing EMA-related applications, ODIN allows constant recording of sensor data. Paco is limited because it does not support sensor data [35], relying principally on primary Android app usage and phone status: phone on and off, incoming calls, messages, and notifications. On the other hand, mPower sensor data is only collected during specific tasks or prescribed activities such as tests of memory, tapping, voice, and walking, which participants are asked to perform 3 times a day.

The most characteristic feature in EMA studies is the time-based prompting of questions. The comparison Table 1.1 shows all the applications provide time-based questions. mPower supporting asking questions only at the beginning of the study and then on a daily basis. Other systems (e.g. TEMPEST) contain a built-in function to count the number of days that have passed since the beginning of the study. Both of these features are available in ODIN. The researcher can specify the precise date and time at which to prompt with a question, or based on the amount of time that has elapsed since the participant's registration with the study.

Another essential feature in EMA studies is sensor-based based prompting. Because this is more complicated to implement than time-based prompting, it is not commonly found in existing systems. While mPower records sensor data during specific tasks, it does not ask questions based on sensor recordings. However, in ODIN, researchers can create new rules based on the values being recorded via sensors, and new sensors can be supported over time.

ODIN provides real-time visualization of the participants' data, a feature which is not available in other systems. Jeeves allows the researchers to download the data, but it is only after the participant completes the survey. A study on dietary behavior change was run using Jeeves and reported limitations on the application, such as "synchronization issues and problems with the visualization of the questions [37]. In contrast, in ODIN, each phone is periodically (every 20min) pushing its data to the server, where it becomes available for the researcher to review.

Software libraries are used in most applications to collect sensor data, but ODIN provides its own built-in sensor services. Several software libraries have been developed to facilitate the creation of EMA studies and to allow any person to be able to participate in a given study from anywhere in the world by using their mobile device [38]. Some examples of these libraries are Aware Framework [39], ResearchKit [40], ResearchStack [41], Survalytics [42], Purple [43], and funf library from MIT (http://www.funf.org). For example, mPower uses library ResearchKit. TEMPEST and Jeeves require other libraries to use some of their available sensors. By using a custom sensor library, ODIN is able to maintain fine-grained control over the data it uses to continuously assess each participants current context.

To date, there is no other system that has all the functionalities of ODIN. Some software and libraries have some of the ODIN's features, but no system contains all of them. Some systems are missing the data interpretation and visualization (e.g. TEMPEST). Others have a fixed palette of questions but they are not based on context (e.g. mPower). Others still are missing built-in sensor services (e.g. Jeeves). In designing ODIN, we attempted to sidestep all these limitations.

ODIN enables continuously recording sensor data and prompting contextual questions. It consists of 2 main applications: 1) a Web UI where the researchers can create studies with sensors, questions, and rules, and visualize the participants' data in real-time, 2) a Phone application in which participants register to a study and contextual questions are prompted. We focus on describing the structure and details of each of these applications in the system.

This thesis follows the next outline. Chapter 2 covers the requirements of ODIN. Chapter 3 gives an overview of the system architecture. Chapter 4 focuses on the backend server design. Chapter 5 describes our Android application in detail. Chapter 6 explains the Web UI design, how the information is exchanged between the UI and the phone app with the server via HTTP. Chapter 7 describes the database design (and its consistency). Chapter 8 covers the thread management in the server and the phone application. Chapter 9 explains the different testing strategies we have used as part of our development process. Chapter 10 covers the evaluation and validation of the system. Chapter 11 describes different debugging and auditing strategies, including logging. Chapter 12 focuses on security in all aspects of the ODIN system. Finally, Chapter 13 covers the experiments and results.

Chapter 2

Requirements

ODIN has different applications that enable researchers and participants to be engaged in studies. First, we give a high-level overview of the different modules in the system, and then we present each in greater detail in subsequent chapters. ODIN can be divided into three main applications.

- Web UI: Web interface using which researchers create one or more studies with researcher-customized sensors, questions, and rules. Once a study is created, coupons may be generated and are distributed to participants wishing to participate in the study. The researchers can view real-time data collected by the app, including sensor measurements and participant answers to context-based survey questions.
- ODIN App: Android application using which the users register and participate in a research study. The ODIN app prompts the subject with questions dynamically based on the study rules and the subject's context as determined by sensor measurements and prior answers.
- ODIN Cloud Server: Ubuntu Virtual Machine running a suite of custom and off-the-shelf services that are responsible for storing the data and interacting with the Web UI and all instances of the ODIN app.

This section describes the requirements of the ODIN system, starting with the use cases, followed by the extensibility for sensors and rules.

2.1 Use-cases

2.1.1 Web UI

We first present the use case of the Web UI, where the researcher can create and modify studies. Fig 2.1 shows the use case diagram of the ODIN Web UI followed by the use case descriptions. First, the Researcher enters the user credentials to log in (Table 2.1) to the Web UI. Then, a new study is created (Table 2.2). Then, modifications to the study can be made by updating the information (Table 2.3), creating a survey (Table 2.4) by adding, editing, deleting and disabling questions and rules (Tables 2.5-2.12), generating coupons (Table 2.13), viewing the participants (Table 2.14), viewing the answers (Table 2.15) and viewing the sensor data (Table 2.16), updating the consent forms (Table 2.17), or updating the contact information (Table 2.18).

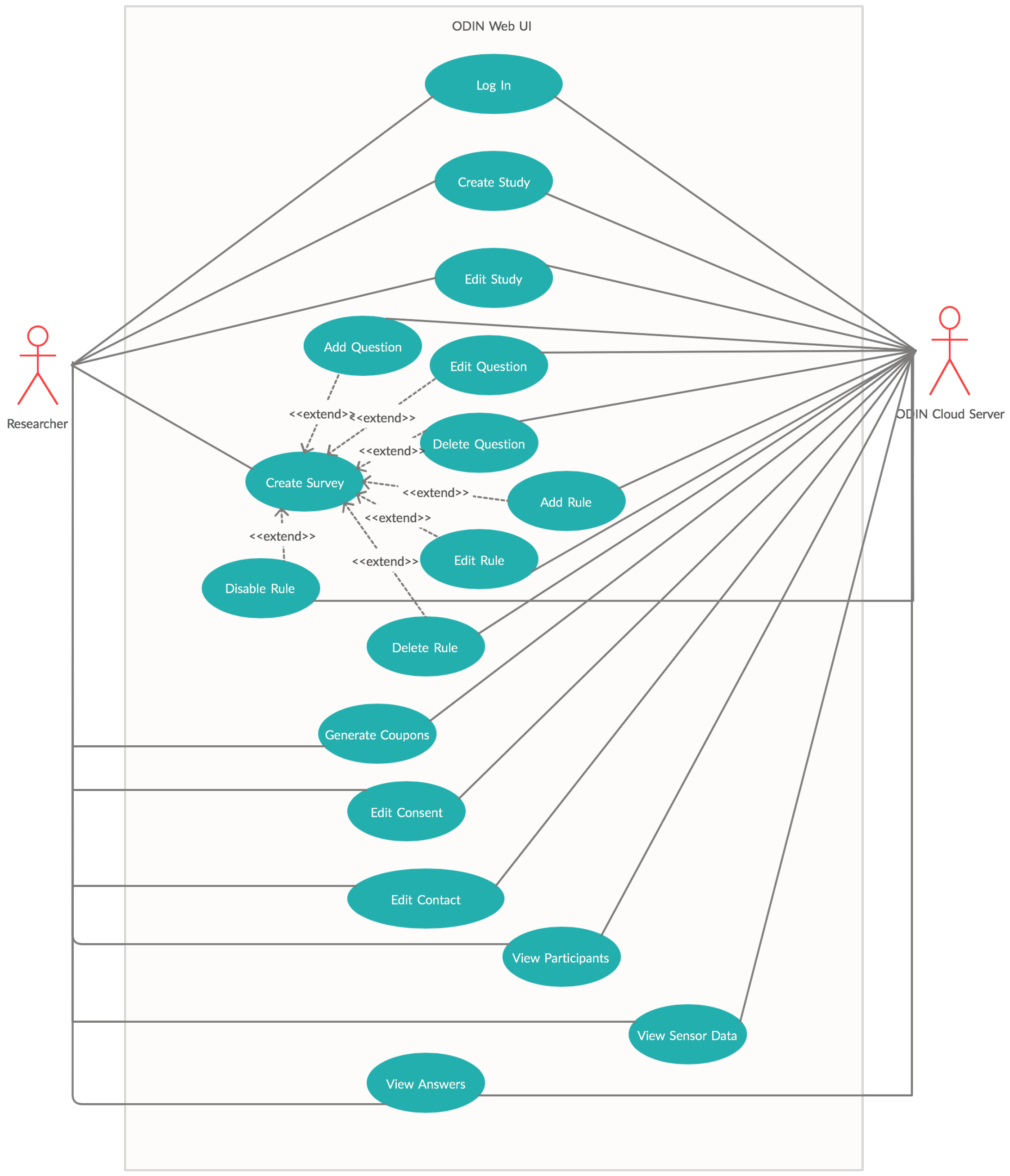


Figure 2.1: ODIN Web UI use case Diagram

Use case name	Login
Participating actors	Initiated by the Researcher
Flow of events	1. User fills in username and password on "Login" screen and clicks the *Login* button. 2. The ODIN Web UI receives the username and password information and hashes the password. The credentials are then sent to the ODIN Cloud Server. 3. The ODIN Cloud Server validates the user and stores the credentials with the password hash in the database.
Entry condition	The user is logged out of the system.
Exit condition	The user is redirected to the "Home" page.
Security Requirements	The password is hashed.

Table 2.1: Use case description for Login

Use case name	Create Study
Participating actors	Initiated by the Researcher
Flow of events	1. The Researcher selects the *Create Study* button. 2. The ODIN Web UI displays the create study form. 3. The Researcher enters the study name, duration, description, and selects the sensors they plan to use in this newly created study. This information is sent to the ODIN Cloud Server. 4. The ODIN Cloud Server creates the study if the information is valid.
Entry condition	The Researcher is logged in and in the Home page.
Exit condition	The Researcher stays in the "Home" page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.2: Use case description for Create Study

Use case name	Edit Study
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Edit* icon in a specific study. 2. The ODIN Web UI redirects the user to the “Info” page. 3. The Researcher edits the study name, description, duration, adds other researchers to the study, edits, or adds new sensors to the study and moves the study phase. 4. The ODIN Web UI sends the information to the ODIN Cloud Server, which validates the information and stores it in the database.
Entry condition	The Researcher has navigated to “Info” page.
Exit condition	The information has been updated successfully and the Researcher stays in the “Info” page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.3: Use case description for Edit Study

Use case name	Create Survey
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the Survey page in the navigation menu. 2. The ODIN Web UI displays the list of questions and rules added to the study.
Entry condition	The Researcher has navigated to “Info” page.

Table 2.4: Use case description for Create Survey

Use case name	Add Question
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Add Question* button page in the navigation menu. 2. The ODIN Web UI displays a form in which the Researcher can enter the question details. 3. The Researcher fills out the form and selects the button *Create Question.* 4. The ODIN Web UI sends information to the ODIN Cloud Server. 5. The ODIN Cloud Server validates the information and stores it in the database.
Entry condition	The Researcher has navigated to the “Survey” page
Exit condition	The information has been updated successfully and the Researcher stays in the “Survey” page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.5: Use case description for Add Question

Use case name	Edit Question
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Edit Question* button page in the navigation menu. 2. The ODIN Web UI displays a form in which the Researcher can enter the question details. 3. The Researcher fills out the form and selects the button *Update Question.* 4. The ODIN Web UI sends information to the ODIN Cloud Server. 5. The ODIN Cloud Server validates the information and stores it in the database.
Entry condition	The Researcher has created a question, and the study is in preparing state.
Exit condition	The information has been updated successfully and the Researcher stays in the “Survey” page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.6: Use case description for Edit Question

Use case name	Delete Question
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Delete Question* button page in the navigation menu. 2. The ODIN Web UI sends information to the ODIN Cloud Server. 3. The ODIN Cloud Server validates the information and updates the database.
Entry condition	The Researcher has created a question and the study is in preparing state.
Exit condition	The information has been updated successfully and the Researcher stays in the "Survey" page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.7: Use case description for Delete Question

Use case name	Disable Question
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Disable Question* button page in the navigation menu. 2. The ODIN Web UI sends information to the ODIN Cloud Server. 3. The ODIN Cloud Server validates the information and updates the database.
Entry condition	The Researcher has created a question and the study is in-progress state.
Exit condition	The information has been updated successfully and the Researcher stays in the "Survey" page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.8: Use case description for Disable Question

Use case name	Create Rule
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Add Rule* button page in the navigation menu. 2. The ODIN Web UI displays a form in which the Researcher can enter the rule details. 3. The Researcher fills out the form and selects the button *Create Rule*. 4. The ODIN Web UI sends information to the ODIN Cloud Server. 5. The ODIN Cloud Server validates the information and stores it in the database.
Entry condition	The Researcher has navigated to the “Survey” page.
Exit condition	The information has been updated successfully and the Researcher stays in the “Survey” page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.9: Use case description for Add Rule

Use case name	Edit Rule
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Edit Rule* button page in the navigation menu. 2. The ODIN Web UI displays a form in which the Researcher can enter the question details. 3. The Researcher fills out the form and selects the button *Update Rule*. 4. The ODIN Web UI sends information to the ODIN Cloud Server. 5. The ODIN Cloud Server validates the information and stores it in the database.
Entry condition	The Researcher has created a rule and the study is in preparing state.
Exit condition	The information has been updated successfully and the Researcher stays in the “Survey” page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.10: Use case description for Edit Rule

Use case name	Delete Rule
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Delete Rule* button page in the navigation menu. 2. The ODIN Web UI sends information to the ODIN Cloud Server. 3. The ODIN Cloud Server validates the information and updates the database.
Entry condition	The Researcher has created a rule and the study is in preparing state.
Exit condition	The information has been updated successfully and the Researcher stays in the “Survey” page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.11: Use case description for Delete Rule

Use case name	Disable Rule
Participating actors	Initiated by the Researcher
Flow of events	1. Researcher selects the *Disable Rule* button page in the navigation menu. 2. The ODIN Web UI sends information to the ODIN Cloud Server. 3. The ODIN Cloud Server validates the information and updates the database.
Entry condition	The Researcher has created a rule and the study is in-progress state
Exit condition	The information has been updated successfully and the Researcher stays in the “Survey” page.
Security Requirements	The size of the input is restricted and the data is validated.

Table 2.12: Use case description for Disable Rule

Use case name	Generate Coupons
Participating actors	Initiated by the Researcher
Flow of events	1. The Researcher has selected *Coupons* button in the navigation menu. 2. The ODIN Web UI displays a form to generate coupons. 3. The Researcher generates new coupons. 4. The ODIN Web UI sends information to the ODIN Cloud Server. 5. The ODIN Cloud Server validates the information and stores it in the database.
Entry condition	The Researcher has navigated to "Info" page.
Exit condition	New coupons have been updated successfully and the Researcher stays in the "Coupons" page.

Table 2.13: Use case description for Generate Coupons

Use case name	View Participants
Participating actors	Initiated by the Researcher
Flow of events	1. The Researcher selects the *Participants* option from the navigation menu. 2. The ODIN Web UI asks the ODIN Cloud Server for the data and the Server returns the data requested by the user and displays the list of coupons registered in the study.
Entry condition	The Researcher has navigated to "Info" page.
Exit condition	The Researcher stays in the "Participants" page.

Table 2.14: Use case description for View Participants

Use case name	View Answers
Participating actors	Initiated by the Researcher
Flow of events	1. The Researcher selects a coupon from the list of coupons that have registered to the study. 2. The ODIN Web UI asks the ODIN Cloud Server for the data, and the server returns the data requested by the user. 3. The Researcher selects the *Survey* option. 4. The ODIN Web UI displays the list of answers of the selected participant.
Entry condition	The Researcher has selected *Participants* button in the navigation menu.
Exit condition	The Researcher stays in the "Participants" page.

Table 2.15: Use case description for View Answers

Use case name	View Sensor Data
Participating actors	Initiated by the Researcher
Flow of events	1. The Researcher selects a coupon from the list of coupons that have registered to the study. 2. The ODIN Web UI asks the ODIN Cloud Server for the data, and the server returns the data requested by the user. 3. The Researcher selects a sensor from the list. 4. The ODIN Web UI displays the sensor data of the selected participant.
Entry condition	The Researcher has selected *Participants* button in the navigation menu.
Exit condition	The Researcher stays in the "Participants" page.

Table 2.16: Use case description for View Sensor Data

Use case name	Edit Consent
Participating actors	Initiated by the Researcher
Flow of events	1. The Researcher selects *Consent* from the navigation menu. 2. The ODIN Web UI asks for the consent form information to ODIN Cloud Server, which retrieves and returns the information. 3. The Researcher modifies the consent information and selects the *Update* button. 4. The ODIN Web UI sends the updated data to the ODIN Cloud Server. 5. The Server validates and stores the information in the database.
Entry condition	The Researcher has navigated to "Info" page.
Exit condition	The information is updated successfully and the Researcher stays in the "Consent" page.

Table 2.17: Use case description for Edit Consent

Use case name	Edit Contact
Participating actors	Initiated by the Researcher
Flow of events	1. The Researcher selects *Contact* button in the navigation menu. 2. The ODIN Web UI asks for the consent form information to ODIN Cloud Server, which retrieves and returns the information. 3. The Researcher adds new contact information, updates existing data, or deletes unnecessary contact information. 4. The ODIN Web UI sends the updated data to the ODIN Cloud Server. 5. The Server validates and stores the information in the database.
Entry condition	The Researcher has navigated to “Info” page.
Exit condition	The information is updated successfully and the Researcher stays in the “Contact” page.

Table 2.18: Use case description for Edit Contact

2.1.2 Phone application

Next, we present the use case diagram of the ODIN phone app in Fig 2.2 followed by the use case descriptions. First, the participant registers to the study (Table 2.19). There is the possibility that the study has the consent forms enabled in which case, the participant has to sign them (Table 2.20). Afterwards, sensor data starts being recorded (Table 2.21) which causes rule firings that result in prompted questions (Table 2.22). Then, the participant selects a question (Table 2.23) and has the option to either answer (Table 2.24), skip (Table 2.25) or answer later (Table 2.26).

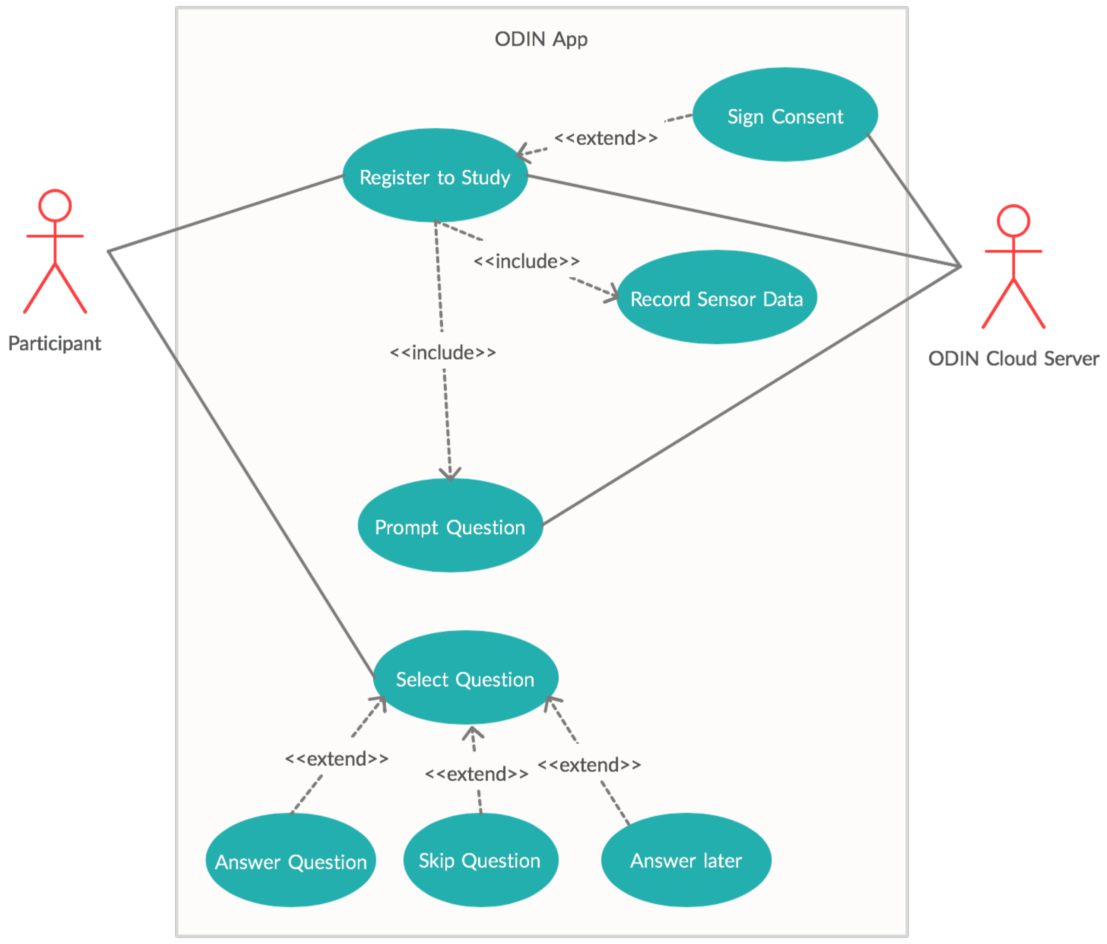


Figure 2.2: ODIN App use case Diagram

Use case name	Register to Study
Participating actors	Initiated by the Study Participant
Flow of events	1. User fills in coupon number and clicks the *Register* button 2. The ODIN App responds by sending the coupon number to the ODIN Cloud Server which uses this information to validate the Participant
Entry condition	
Exit condition	The user is redirected to either the “Consent Form” screen or “Welcome” screen.

Table 2.19: Use case description for Register

Use case name	Sign Consent
Participating actors	Initiated by the Study Participant
Flow of events	1. The Participant goes through the Consent by pressing the *Next* button and finally signs the Consent form. 2. The ODIN App responds by sending the information to the ODIN Cloud Server and receiving the study data that the Researcher created
Entry condition	The Study Participant has registered to the app.
Exit condition	The user is redirected to the Welcome activity.

Table 2.20: Use case description for Sign Consent

Use case name	Record Sensor Data
Participating actors	Initiated by the System
Flow of events	1. If the study that the participant is registered to has sensors enabled, then the ODIN App records sensor data every X minutes which is specified by the Researcher during the creation of the study.
Entry condition	The Researcher has registered and signed the Consent (if enabled).
Exit condition	The Researcher has completed the Study.

Table 2.21: Use case description for Record Sensor Data

Use case name	Prompt Question
Participating actors	Initiated by the ODIN App or ODIN Cloud Server
Flow of events	1. The ODIN App prompts a question to the user when all the conditions of the rule attached to that question have been satisfied.
Entry condition	The Researcher has registered and signed the Consent (if enabled).
Exit condition	The Researcher has completed the Study.

Table 2.22: Use case description for Prompt Question

Use case name	Select Question
Participating actors	Initiated by the Participant
Flow of events	1. The Participant selects a question from the list of questions. 2. The ODIN App displays the question selected.
Entry condition	A question has been prompted to the Participant.
Exit condition	The Participant is in the “Question” activity.

Table 2.23: Use case description for Select Question

Use case name	Answer Question
Participating actors	Initiated by the Participant
Flow of events	1. The Participant selects or types an answer for the prompted question, and selects the Submit button. 2. The ODIN App stores the answer in the system’s database.
Entry condition	The participant selected a question.
Exit condition	The Participant gives an answer to the question.

Table 2.24: Use case description for Answer Question

Use case name	Skip Question
Participating actors	Initiated by the Participant
Flow of events	1. The Participant does not select nor type any answer from the prompted question, and selects the Submit button. 2. The ODIN App stores the skipped answer in the system’s database.
Entry condition	The participant selected a question.
Exit condition	The Participant decides to not answer the question.

Table 2.25: Use case description for Skip Question

Use case name	Answer Later
Participating actors	Initiated by the Participant
Flow of events	1. The Participant decides to answer the question later and selects the *Back* button. 2. The ODIN App keeps the question in the "Questions" activity for the user wants to answer later.
Entry condition	The participant selected a question.
Exit condition	The Participant decides to answer the question later.

Table 2.26: Use case description for Answer Later

2.2 Extensibility for sensors

ODIN can grow in terms of the number of sensors since it is based on questions being prompted due to rule firings. Some of the rules use sensors as the conditions to be satisfied. By incrementing the number of sensors, we can obtain more accurate data from the participant. From date, the following sensors are available in ODIN: GPS, Bluetooth proximity, beacon proximity, activity recognition, and Empatica E4 wrist band. Each of the sensors are described in Chapter 4.2.3.3 in more detail. Our system has been designed to increase the number of sensors easily. For this reason, we plan to keep implementing new sensors so that it increases the studies that can be created since ODIN can obtain a wider range of data.

2.3 Extensibility for rules

ODIN prompts questions to the user if the rule conditions are satisfied. The rules increase as the number of sensors grows. We can obtain more data from the participant, and questions can be asked in an extended range of contexts. We have implemented time-based, report, follow-up, and sensor-based rules. The complete list of rules with their descriptions can be found in Chapter 4.2.3.1. For this reason, making the rules extensible is a feature that increases the range of studies that can be used in ODIN. We obtain more accurate data from the user with a more significant number of rules. As mentioned previously, adding new sensors in the ODIN system is relatively simple. Similarly, we have designed the ODIN system to be able to incorporate new rules easily.

Chapter 3

System architecture and overview

The main functionality of ODIN is to collect data from study participants using prompted questions based on the sensors' data. The rules engine periodically checks the data stored in the database, and prompts the user with appropriate questions whenever the corresponding conditions (i.e rules) are satisfied. The app can support an unlimited number of sensors. For the time being, we have implemented the following: GPS location, Bluetooth proximity (phone to phone), beacon proximity (phone to beacon device), activity recognition, and E4 wrist band. To provide these services, we need a stable and reliable system, which requires making use of a range of different strategies and techniques. One such example is periodic communication between phone and backend. Other aspects of the design are further discussed in later chapters of this thesis.

We developed a client/server application where the backend is implemented in Java, the frontend in ASP.NET MVC, and Java for the Android application. The client/server architecture is composed of only one server (running multiple services) and many clients [1, 44]. The server is always running, and it waits for the client to initiate the communication. The clients can communicate with other clients, but the packets need to be sent to the server first, and then the server distributes the packets to the specified destination. This architecture differs from peer-to-peer, as it does not distinguish clients and servers, allowing direct communication among the hosts [1].

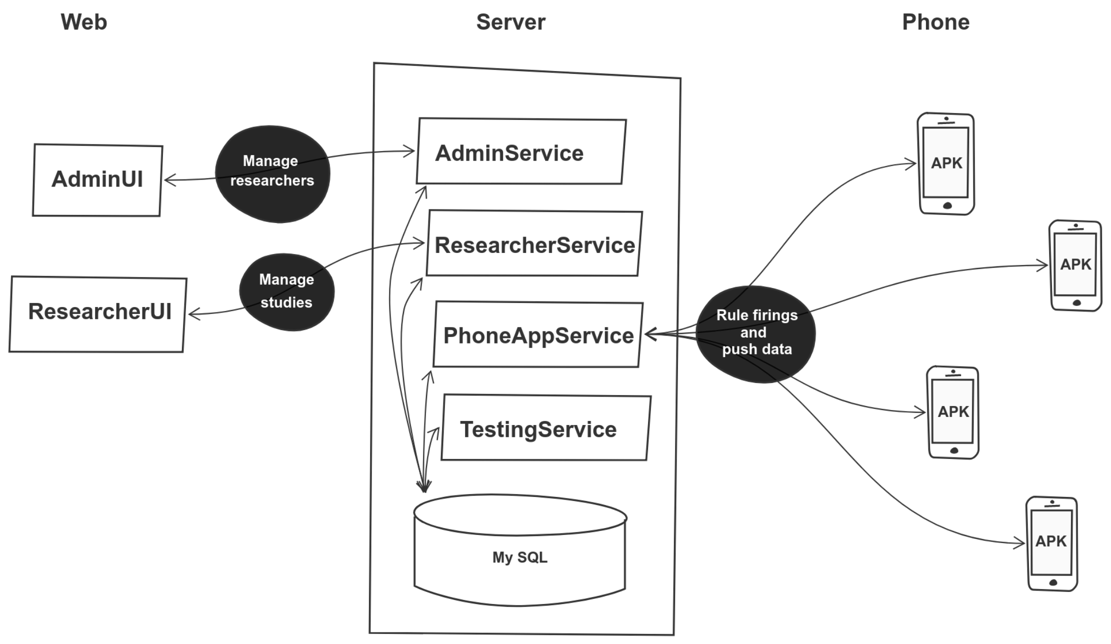


Figure 3.1: ODIN's system architecture. The left side of the figure shows the two web interfaces, the center contains the backend server, and the right side depicts the Android application.

The architecture of ODIN is depicted in Fig 3.1. Each of the terms highlighted in bold in this section correspond to an element of the Fig 3.1. The system consists of four backend services, two web user interfaces, and an Android application. Two of the backend services support their own user interfaces, and all of the services share the same database, which is also hosted in the backend server. In addition, there is another backend service for the Android application. The app itself maintains its own on-phone database. While much of the data in each apps database eventually finds its way to the server, some of the app data remains local to the phone. The Web UIs and the app make connections with the services over the network, opening an HTTP session in which there are multiple REST calls.

To make this system functional, we need researcher accounts. The administrators of the system can create, edit, and delete researchers using the web interface **AdminUI**, which talks to **AdminService** to retrieve and manipulate data from the database.

The next step is to create studies to which participants to register. The researchers (whose accounts were created by the administrators), can access the web interface **ResearcherUI** where they can manage studies. Similar to **AdminUI**, **ResearcherUI** talks to **ResearcherService** to obtain and provide a researcher-centric interface to database information. Part of creating a study involves defining its questions and rules, as well as generating coupons that the researcher can hand out to suitable participants.

The ODIN app consists of an APK that can be downloaded and installed on participants phones. Finally, when one or more studies have been set up, participants can register for the study using the ODIN app installed on their Android smartphones. Registration happens by way of a coupon number that is provided by the researcher to the participant. After the consent agreement, data collection begins using the sensors, as specified when the study was created via the **ResearcherUI**. The collected data is periodically pushed to the **PhoneAppService**. Simultaneously, **PhoneAppService** periodically notifies the **APK** when a rule firing happens at the server, or when a researcher wants to contact a study participant.

The server also contains another service that has not been mentioned previously, named **TestingService**. As the name indicates, this service includes some endpoints that are useful for testing purposes only. These endpoints are called from SOAPUI test suites, and they are used as helper functions enabling tests.

All the data from the administrators, researchers, and study participants is stored in the server database. The server contains multiple databases; the ODIN database and each study have its separate database. The backend database runs on **MySQL**

Server [45] and can be administered via PhpMyAdmin [46]. Each instance of the APK contains its own SQLite database whose schema is quite similar to the schema of the study databases in the backend. However, the SQLite database also contains some specific information that is only used on the phone. More details about the database schema in the backend can be found in Section 4.2.1 and the phone database in Section 5.2.1.

Different technologies have been used for the implementation and verification of ODIN. The backend services are part of a Java Maven [47] project using Spring framework [48] and developed in Eclipse IDE [49]. The Web user interfaces are developed in Visual Studio [50], using ASP.NET MVC [51], which includes multiple languages like C#, HTML, JavaScript, JQuery, etc. Lastly, the APK is part of an Android Studio [52] project in which includes Java and XML. All transport takes place using HTTP and makes heavy use of JSON encodings [53]. On the verification side, SOAPUI [54] is used to test endpoints from the backend services, and Android JUnit [55] and Bugfender [56] to test the APK. Jenkins [57] allows us to run daily tests on both the backend (SOAPUI) and the APK (JUnit).

Chapter 4

Backend server

As mentioned previously, ODIN follows a client/server paradigm. This kind of architecture consists of a server and one or more clients. Our backend services run either within the Tomcat web server, while the UI-related services are self-hosted by the C# Dotnet Core 2.1 runtime. Communication with these services happens via HTTP [58]. The author Mosberger et al. describes the challenges that were faced with creating a tool for calculating the performance of a Web server [59]. Distributed systems are more complex due to concurrency, and it is dynamic since most of the components such as the server, client, web content, etc. change. For this reason, designing and developing ODIN's backend server was a difficult task, but we had to make sure the design was stable since it is the base of our system.

In previous sections, we referred to the backend as the ODIN Cloud Server without making any distinctions on the services running on the Cloud. Fig 4.1 illustrates the different services running in the server and where they are hosted. The Figure also shows each of the packages that are common among all of the services. In this section, we present an overview of the server's design and a description of all the backend components.

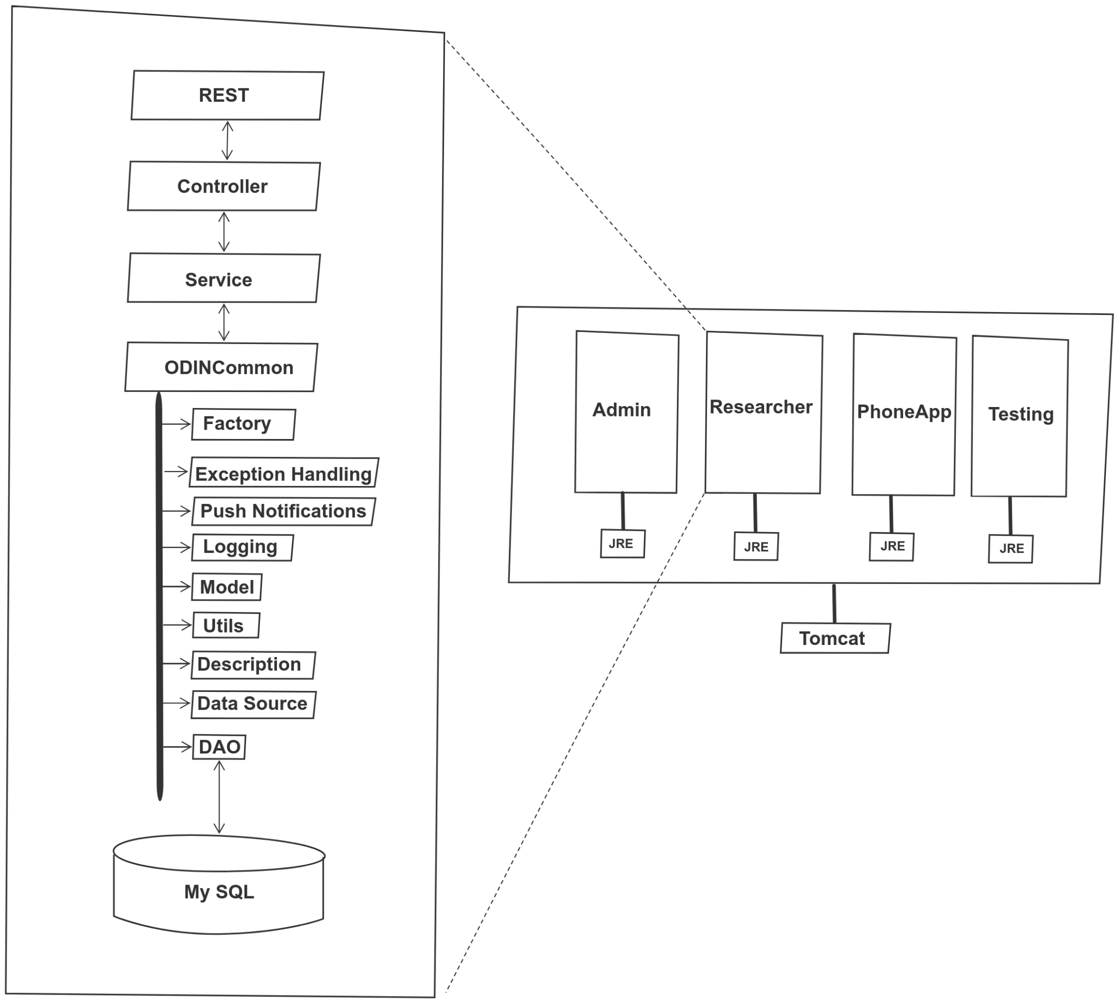


Figure 4.1: Backend services and packages. The right side of the figure displays the different services running on Tomcat, and the left side shows their structure.

4.1 Design

We can observe in Fig 4.1 that the backend is composed of 4 services: AdminService, ResearcherService, TestingService, and PhoneAppService. The first three share the same organizational structure. The PhoneAppService schema (shown in Fig 4.2), however, differs from the other three services. The subjects carrying the APK need to communicate with PhoneAppService to upload the data, and PhoneAppService needs to asynchronously notify the APKs when rules fire. Supporting this bidirectional communication creates the need for a different structure for PhoneAppService.

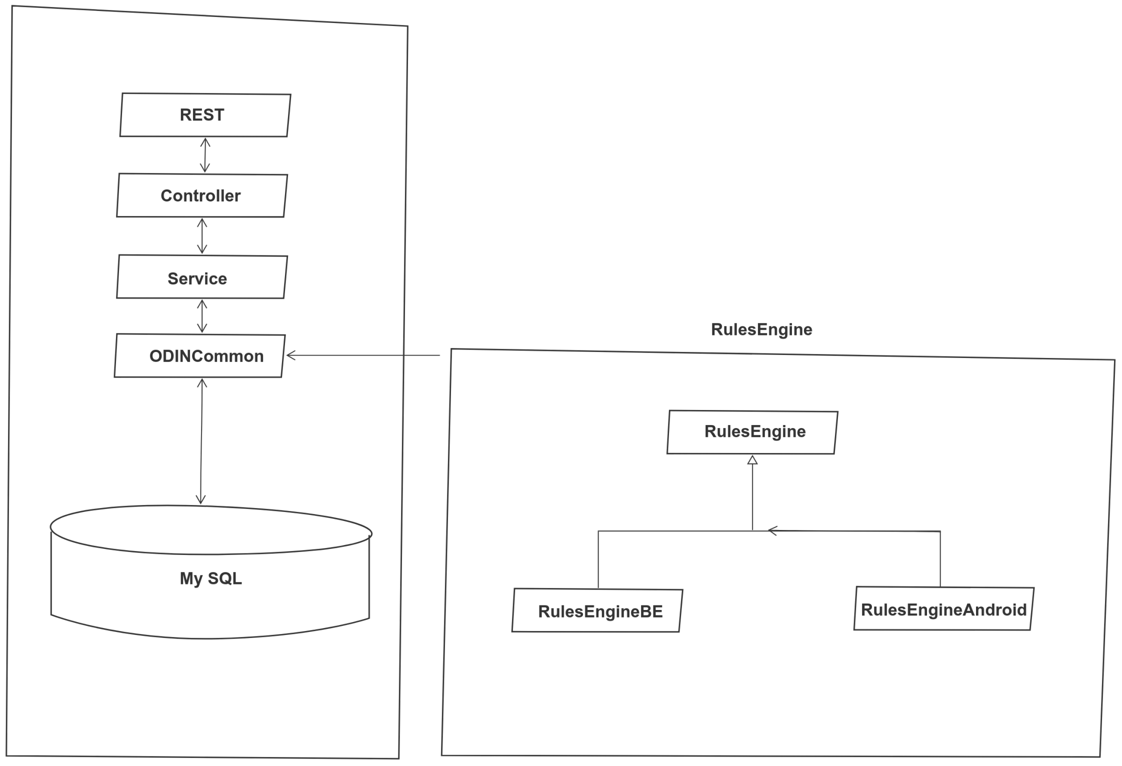


Figure 4.2: PhoneAppService structure. The left side of the figure shows the service structure and the right side displays the RulesEngine package, which uses the ODINCommon module.

The backend services are deployed into the Tomcat web server, which runs on a Ubuntu Virtual Machine. The services have been implemented in Java, and thus each runs in separate JVM under the Java Runtime Environment (JRE). WAR (Web Archive) [60] files for the different services are deployed into Tomcat [61]. We use Phpmyadmin [46] as an administration tool for the database that runs on MySQL server [45].

The services are each composed of three main packages: Controller, Service, and ODINCommon. The Controller package contains all the endpoints that are used to communicate from the Web UIs or the APK via REST calls. We need a layer between the Controller and the Data Access Objects (DAO) layer so that there is no direct communication to the database tables. For this reason, we have the Service layer, which is used as an intermediary between Controller and ODINCommon packages. Further, ODINCommon is a project that is shared among all the services and the APK, allowing for shared encodings and utilities between the APK and the backend services. Next, we describe the different packages in the ODINCommon project and their respective functionality.

- **Factory** contains generation and validation methods for specific data. For example, it contains the RuleValidation class, which is used every time the *AddRule* endpoint is called from the web UI to verify that the rule is valid and the system can proceed to insert it in the database.

- **Exception Handling** package handles notifying the developers whenever there is an exception in any of the backend services via email. It sends the IP address where the service is running, the name of the endpoint as the subject field of the email, and the stack trace is in the body.

- **Push Notifications** package handles the notifications that need to be delivered to the phone. The phone is notified in the following cases: 1) A *PhoneWakeUp* request is sent every 15 minutes to all the phones active in the study to make sure that the services continue running and 2) when there is a rule firing in the backend.

- **Logging** manages all the application logs. More details can be found in Chapter 11.

- The **Model** package is the Data Transfer Object (DTO), which is shared between the APK, ResearcherUI, and AdminUI. This package contains the database attributes that need to be transferred via REST calls as JSON objects.

- The **Utils** package contains all the helpful methods for data validation or conversion. It also has assertion classes to assist with the debugging.

- **Description** package holds the classes that generate user-friendly rule descriptions shown in the Web UI after a rule has been added.

- The **Data Source** package contains the DAO interfaces that are shared between the APK and the backend. These interfaces are an important part of the design since both the APK and the backend services (e.g. the PhoneAppService) share the same RulesEngine package, which retrieves JSON encoded data from the database.

- The **DAO** layer is in charge of retrieving and manipulating the data in the database.

The four backend services are in separate projects sharing the ODINCommon package. They follow the structure of a multimodule project: there is a parent project, and the children are the services and ODINCommon. It is complicated to maintain a standard package in 4 different services. For this reason, we use git submodules [62], which allows the developer to share the same git repository [62] among different projects. This approach makes it easier to keep the ODINCommon project consistent among the four services since we just have to make sure the project points to the correct commit [62]. Without this approach, whenever we would make a small change in ODINCommon of one project, we would have to manually propagate the changes across all 4 different projects.

We can observe from Fig 4.2 that the PhoneAppService presents a similar structure to the rest of the services with the addition of the RulesEngine package. The RulesEngine exists both within each APK and in the backend within the PhoneAppService. The same interfaces are used, but the implementation of the Scheduler is different depending on the platform where the RulesEngine is instantiated. The RulesEngine validates and then adds specified rules to the corresponding Scheduler. The RulesEngine thread runs in an infinite loop, sleeping and waking up when it is time to run the next job (rule). When it wakes up, it checks the conditions of the rules and *fires* if all the conditions are satisfied. More details about the RulesEngine can be found in section 4.2.2.

4.2 Details

In this section, we describe the following three main components in the backend: Database, RulesEngine, and Sensors. The main purpose is to show the differences between these three critical components of the system (which are instantiated both at the backend server and within each APK). Details of the APK can be found in Chapter 5.

4.2.1 Database Schema

Within the ODIN system, the MySQL database management system maintains multiple database instances. The main one is *odin* database, which contains the researcher account information along with information that is common across all studies. The odin database may be viewed as a central coordination database (i.e. even if no studies have been created, the odin database exists). Beyond this, each study that is created has a new dedicated database where just that study's data resides.

In Fig 4.3, we can observe the ODIN database schema with its corresponding tables and attributes. Next, we describe the functionality of each table.

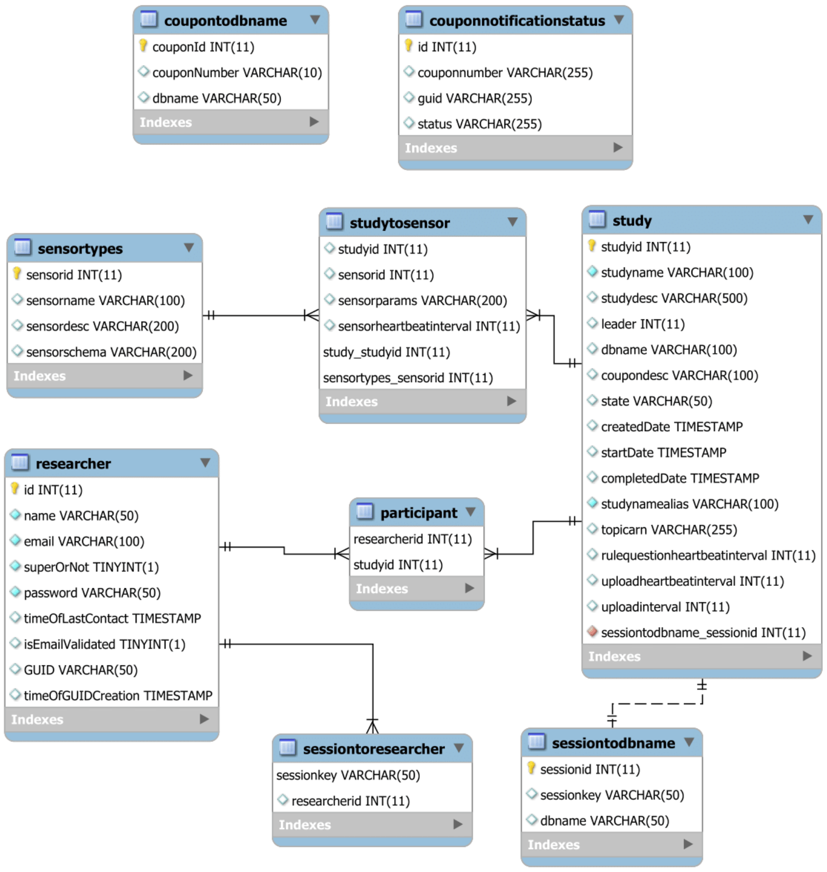


Figure 4.3: ODIN database ER diagram

- When an administrator creates a new researcher account, the information is stored in the **researcher** table.

- Afterwards, the researcher logs in to the ResearcherUI and a new *sessionkey* is generated and saved in the **sessiontoresearcher** table with the *researcherid* attribute as a foreign key to the *id* attribute in researcher table.

- The researcher then decides to create a new study, a new entry to the **study** table is inserted with the following fields: *study id, study name, study description, leader, createdDate, and studynamealias.* There are some other attributes in this table that have default values for all the studies like *uploadheartbeatinterval*, *rulequestionhearbeatinterval*, and *uploadinterval.* These values are sent to the APK as a response to a registration REST call. They represent the intervals for recording the heartbeat of the two default services on the phone: Upload, and RuleQuestion. More details about these services are explained in Chapter 5.

- Another feature supported by ODIN is that the creator of a study can add participants to a study, and can control the permissions that these collaborators have (e.g. they can manage or view the study data). Each participant is assigned a set of specific actions that they can do for that study. The study leader is stored in the study table, and all the other researchers added to that study are stored in the **participant** table. When a new participant is added, a new row is inserted into this table. This table contains a map of *researcherid*, (which is a foreign key of *id* in researcher table) to *studyid* (which is a foreign key of *studyid* in study table). Moreover, the table contains a list of *entitlements* in a JSON format, which are actions that the researcher has been granted permission to perform (by the owner of the study).

- The **sensortypes** table contains all the information about the supported sensors that can be added to a study.

- Each sensor contains some specific sensor parameters; for example, the interval between successive readings of the sensor. These parameters values are specified by the researcher when creating the study. The following information is stored in **studytosensor** table: *studyid* (foreign key to *studyid* of study table), *sensorid* (foreign key to *sensorid* of sensortypes), *sensorparams*, *sensorheartbeatinterval*. The *sensorparams* field is stored as a JSON format, and the *sensorheartbeat-interval* is a constant value set currently to 5 minutes to track the life of the sensor service in the APK.

- After the study is created, the researcher can generate new coupons, which are stored in the **coupontodbname** table with a map of *couponNumber* to *dbname*. The attribute *dbname* is a foreign key to *dbname* in the study table, which is a way to map the coupons to their corresponding studies. The purpose of this table is to allow the backend services to "route" the incoming all registrations to the appropriate study database.

- When the study has been created, and its questions and rules fully specified - it is then moved to in-progress *state*, which is reflected by changing the appropriate row of the study table (in the odin database). Afterward, subjects can register by using the coupon number given to them by the research. When the subject types in their coupon and presses the "Register" button on the phone app, the registration REST call executes, and a new session key is generated for that coupon, which maps to a study database. This information is stored in the **sessiontodbname** table. The purpose of this table is to allow incoming REST calls (all of which specify a session key as argument 1), to be multiplexed and act on the appropriate study database.

- We need a way to collect performance information from phones and use this data to classify the phone status as "ok", "pending," "timeout," or "error." For this reason, we use the "ping" and "pingAll" endpoints. When "ping" or "pingAll" is called, we insert a new entry in **couponnotificationstatus** table with an auto-generated *id* and a *guid*; the latter would be sent in the push notifications. When the phone receives the push notification, it calls the "postAcknowledment" endpoint by sending that *guid* and the *sessionkey*, which would be used to retrieve the *couponnumber*. The phone *status* is updated to either "ok", "error," 'timeout," or "pending". The first happens if the results are as expected. The second, if they are incorrect. The third one if the phone has not responded within a specified timeout interval. The pending status is used when the timeout interval has not yet expired.

When a researcher creates a new study, a new *dbname* is generated (it is a random GUID), and this unique name is stored in the **study** table. A new database is created with this name. The database schema for a study database is in Fig 4.4. Note that the schema is quite different from that of the central coordinating database. The study database contains the following tables:

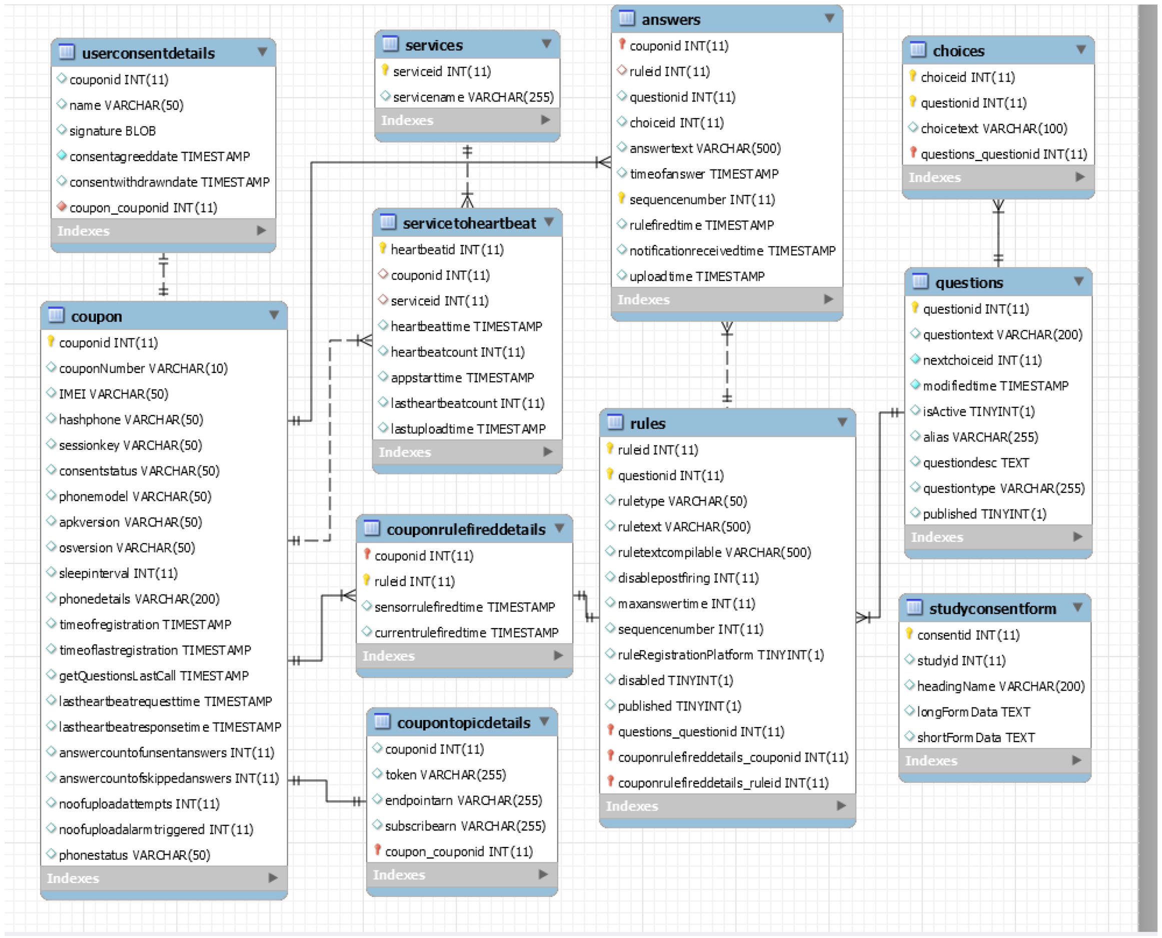


Figure 4.4: Study database ER diagram

- The main goal of the system is to prompt questions to the subject carrying the phone. When a researcher creates a new question to the study, a new row is inserted in **questions** table with values set by the researcher in the following fields: *questiontext*, *alias*, *questiondesc*, and *questiontype*. For the time being, we have implemented the following question types: multiple choice (single-select or multi-select), and fill text. There are other attributes in the questions table that are not specified by the researcher (e.g. the *questionid*, which is auto-generated in the backend). The *nextchoiceid* is a foreign key to *choiceid* in the choices table. The *modifiedtime* is generated in the backend when inserting a new row. Questions can be disabled when the study is in progress, so the *isActive* field is set to true when the question is created, and we update it to false if the question is disabled. The researcher also has the option to add questions while the study is in progress; hence, we use the *published* field to know if a question is ready to be sent to the APK as part of the next update. There are two different scenarios; the researcher can add a question when the study is in preparing state or they can add it while the study is in-progress. In the former case, we set it to true when the study is moved to in-progress state via the ResearcherUI. In the latter case, the field is set to true when the researcher selects the "Publish" button in the ResearcherUI.

- If the researcher decides to add a multiple-choice question, then response options can be added to that question. When a new response option is added, a new row is inserted in the **choices** table, with the value specified by the researcher in *choicetext*. This table also contains a *choiceid* (which is auto-generated) and *questionid* (which is a foreign key to *questionid* in questions table). There are 3 special response options; 1) "skipped" when the user decides not to answer the question, 2) "expired" when the participant runs out of time to answer the question, and 3) "unset" when the question is prompted, but the time limit never expired (this can happen, for example because the phone runs out of battery while the question is showing).

- For the questions to be prompted on the subject's phone, rules need to be attached to the question to determine when it should be shown to the subject. When a researcher adds a rule, the following fields can be specified: *ruletype*, *ruletext*, *maxanswertime*, and *ruleRegistrationPlatform*. We describe the meaning of each of these fields in more detail in Section 4.2.3.1. A new row is inserted with the values determined by the researcher for the above fields into **rules** table. There are other attributes like *ruleid* (which is auto-generated) and *questionid* (which is a foreign key to *questionid* in questions table). The fields *disabled* and *published* are inserted and updated in the same manner as the questions table described previously. There are other fields that are not being currently used like *sequencenumber*, *ruletextcompliable* and *disablepostfiring*, but could potentially be used in the future.

- The researchers have the option to add a consent form to the study or let the user start using the app without having to go through the consent process in the app (this can happen if consent is being obtained by some other means, like paper forms). The default consent form data is stored in the **studyconsentform** table. It contains an auto-generated field *consentid.* The rest of the fields are related to the consent form data, the *headingName* is the title of the section, the *longFormData* contains all the detailed description of the data, and the *shortFormData* is a short description of the section. The researcher has the option to update the last two fields. The subject first sees the short description (shortFormData) and can choose to view a more detailed description of the section (longFormData). The field *studyid* is currently not being used.

- After the researcher has created questions, rules, and decided on the consent, the study can be moved to in-progress state, and subjects can start registering using coupons. When a researcher creates a coupon, a new row is inserted in the **coupon** table. Coupon table fields are automatically generated; hence, the researcher cannot specify any values in the attributes of the coupon table. The *couponid* is auto-generated in the backend. The other fields in the coupon table are updated when the subject registers on the phone. A new session key is generated for that subject and stored in the *sessionkey* field. The fields that are updated during registration are the following: *IMEI*, *hashphone*, *consentstatus*, *phonemodel*, *apkversion*, *osversion*, *sleepinterval*, *phonedetails*, *timeofregistration*, *timeoflastregistration*, *getQuestionsLastCall*, *lastheartbeatrequesttime*, *lastheartbeatresponsetime*, *answercounttofunsentanswers*, *answercounttofskippedanswers*, *noofuploadattempts*, *noofuploalarmtriggered*,and *phonestatus*. A more detailed description of each one of these fields is presented in Chapter 5.

- If the researcher decided to have a consent form in the study, then the subject's information entered in the consent would be stored in the **userconsentdetails** table. The attributes updated with the subject values are *name* and *signature*. The *consentagreeddate* is automatically set when the user completes the study consent form. If the user later decides to withdraw from the study, the *consentwihdrawndate* field would be updated with the date and time of the user's action. This table also contains *couponid* which is a foreign key to *couponid* in the coupon table.

- After the subject has completed the registration and the consent form, questions can begin to appear (based on contextual rules). As the subject responds to these questions, the responses would be prompted, and the subject would have to respond to those questions, which would be stored in the **answers** table. For each answer, we need to know the *ruleid* (foreign key to *ruleid* in the rules table), *questionid* (foreign key to *questionid* in the questions table), *choiceid* (foreign key to *choiceid* in the choices table) and *answertext*, *rulefiredtime*, *notificationreceivedtime* as well as the *timeofanswer*. All the answers from all the phones in the study will get aggregated into a single table in the backend study database. Thus, while the APK has its own autoincremented *answerid* in its version of the answers table, we need a way to keep a unique identifier in the backend. For this reason, we use the *sequencenumber* field, which maps to the answerid on the phone. However, the phone database might have been deleted, and it could be missing some answers. Accordingly, the backend always sends a starting sequence number to the APK, which maps that specific coupon. When the answer is sent to the backend, the *uploadtime* field is updated.

- Some of the logic of the sliding window algorithm (which is used within the RulesEngine to efficiently check if a rule should fire) requires to keep track of some information like the *sensorrulefiredtime* and *currentrulefiredtime* for each *couponid* (foreign key to *couponid* in coupons table) and *ruleid* (foreign key to *ruleid* in rules table). All this information is stored in the **couponrulefired-details** table.

- All the studies have default services, but the total number of services varies based on the study requirements. Each sensor has its service. Hence, we need a way to keep track of all the services in the APK. For this reason, we have a **services** table in which we store the *servicename* and its corresponding *serviceid* which is unique across APK services and is auto-generated.

- Each subject registered in the study has an app, and within the APK there will in general be multiple services running. For ODIN to perform well, we need to make sure these services stay alive. This is achieved by having each service send itself an intent. We refer to these intents as "heartbeats". To keep track of whether a service is still running, we keep a record of the last time they received a heartbeat intent, number of heartbeats that have occurred, etc. This information is maintained in the table **servicetoheartbeat** to store the *heartbeattime*, *heartbeatcount*, *appstarttime*, *lasthearbeatcount*, and *lastuploadtime*. It also contains an auto-generated field *heartbeatid*, *couponid* (foreign key to *couponid* in coupon table), and *serviceid* (foreign key to *serviceid* in service table). All this information can be used to analyze this data and derive conclusions regarding the performance of the APK and its constituent services.

- **coupontopicdetails** table contains the details about AWS needed to notify the APK. When a coupon is registered, a *token* is sent to PhoneAppService as part of the registration REST call response. The backend uses that token to create an endpoint. This endpoint registers the coupon in the SNS service [63]. Then, SNS generates *endpointarn* and *subscribearn* when the endpoint is created, which identifies the endpoint for that coupon. We know that these details map to a coupon using the *couponid*, which is a foreign key to *couponid* in the coupons table. This information is necessary to send push notifications anonymously to the phone, delete the endpoint or unsubscribe the coupon to the endpoint.

4.2.2 Rules Engine

In this section, we go over the details of the backend RulesEngine. First, we give an overview of the structure of the system, then we describe the scheduler, and lastly, we go over each one of the rule types and filters that the system currently supports.

The user can add multiple rules to a question, and each rule can have many filters. The rules engine treats each rule as a separate job; a rule is an object that contains the rule type, all the rule parameters with its corresponding values, and all the filters attached to that rule. The main difference between a rule and a filter is that rules are primary, being triggered by applying logic to data from a single sensor, while filters are secondary checks that can cause the suppression of a rule firing based on recent data from *other* sensors (hence the term “filter).

We also have to consider the case in which the user adds questions when the study is in-progress. Recall that the researcher can add questions to a study (along with associated rules/filters) long after the study has been moved to in-progress. Because of this, we need a way to notify the PhoneAppService to register any new rules in its RulesEngine, and to disseminate these new rules to phones that are registered to the relevant study. The PhoneAppService waits for a periodic “reload” to be initiated by each phones APK. The reload action involves a REST call to the PhoneAppService endpoint named “getQuestions.” Then, the PhoneAppService adds the rules in the backend RulesEngine. The phone “reload” is explained in more detail in Chapter 5.

The Rules Engine being used in the backend and in the phone follow the same class structure, which is shown in Fig 4.5 and is described below:

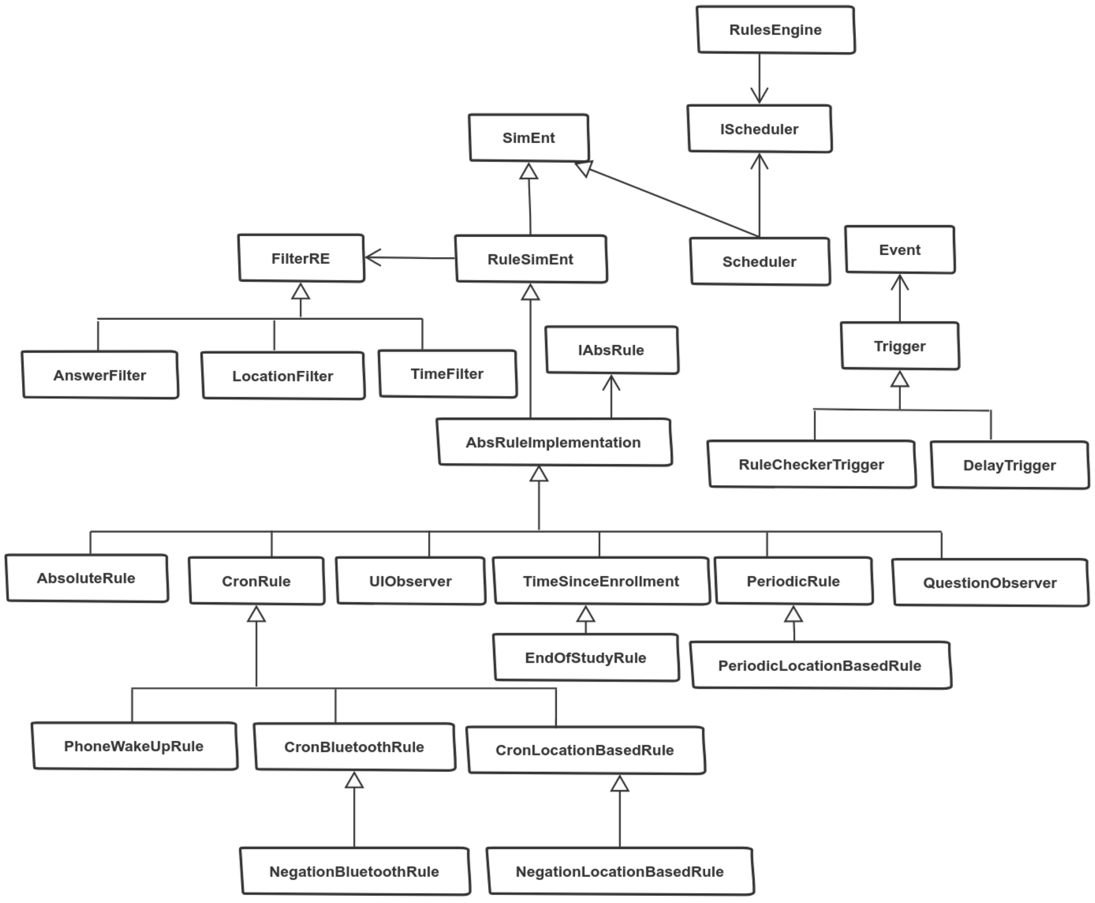


Figure 4.5: Rules Engine class diagram

- The parent class of the rules, filters, and scheduler extend the **SimEnt** class. It contains the following methods; *send*, *recv*, *deliveryAck*, and *suicide*. The first two are described in more detail in the scheduler diagram in Figure 5.5. DeliveryAck is called right after recv, and suicide when a rule has to be deregistered from the RulesEngine.

- The parent class of all the rules and filters is the **RulesSimEnt** class, which is a child of SimEnt. RuleSimEnt keeps a list of all the filters that need to be checked before the rule firing. It iterates over the list of filters and checks whether all the filters' conditions have been met. If the rule and all the filters conditions are met, then the rule fires.

- **AbsRuleImplementation** is a child class of RuleSimEnt, and it contains the abstract methods used by the rule classes. AbsRuleImplementation contains the *SlidingWindow* method, which is used in all the sensor-based rules. The rule types generated with SlidingWindow are "while", "while NOT", "on arrival", and "on departure". The first two are part of the submethod *IShape*, and the other two are part of the submethod *LShape*. More information about the sliding window, and these two methods can be found in Section 4.2.3.

- The **Scheduler** class is in charge of handling the jobs in the RulesEngine. It is explained in more detail in the next section 4.2.2.1.

- The Scheduler implements the **IScheduler** interface, which is used to hide implementation differences between the backend RulesEngine and the APK RulesEngine.

- The **RulesEngine** is the abstract class that contains the methods for registering the rules.

Now that we have the big picture of the Rules Engine, we can describe the backend Scheduler in more detail.

4.2.2.1 Scheduler

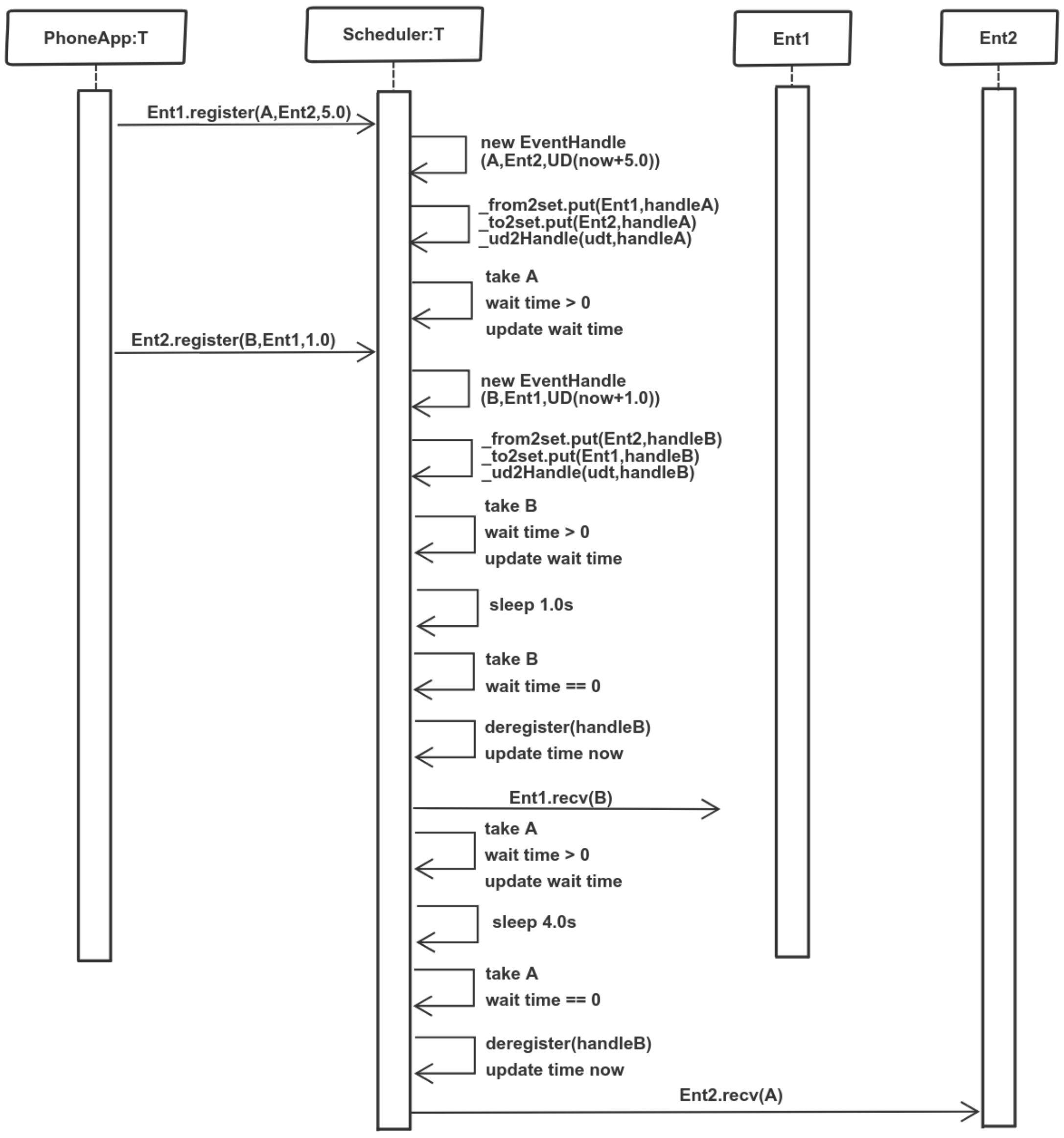


Figure 4.6: Rule Engine backend Scheduler sequence diagram. Ent1 is the sender and Ent2 the receiver of event A; Ent2 is the sender and Ent1 the receiver of event B.

We describe the RulesEngine backend scheduler. The operation follows the standard pattern of a discrete event schedule [64]. Fig 4.6 shows an example of how it works.

First, Ent1 is registered with event A, its receiver is Ent2, and it has a time of 5 seconds. Then, Ent2 is registered with event B, its receiver is Ent1, and it has a time of 1 second. This diagram represents a thread in an infinite loop. Next, we describe the sequence of events:

1. There is a new EventHandle instance created. This instance contains the event "A," the receiver entity "Ent2", and it creates a UniqueDouble, which contains time NOW + 5.0. Then is the time that the thread sleeps. After time NOW + 5.0, the thread wakes up and fires the rule.

2. Scheduler class contains 3 maps. All of the maps contain the event handle "A" as a value, and the keys are described below:

   - from2set: this map contains the registered entity "Ent1" as a key.
   - to2set: this map contains the receiver entity "Ent2" as a key.
   - ud2handle: this map contains a unique double.

3. Since A is the first event that was added to the queue; it is the first one to be taken from the queue. We check the wait time of A; it is still greater than 0, so we update the wait time and put it back in the queue.

4. Create a new EventHandle instance, just like we did for event A, but this time is for event B, and the receiver is Ent1 with 1.0 seconds.

5. Add the new event with its corresponding values to all the maps, just like we did for event A.

6. B is the new event in the queue; hence, we take B from the queue and check its wait time. We see that the wait time is greater than 0, so we update the wait time and put it back in the queue.

7. The thread sleeps for 1 second since it is the time left from event B.
8. Scheduler notifies the thread to wake up. Then, we take B since it is the top one in the queue and check the wait time. We see that the wait time is 0. The scheduler thread always takes the shortest time from all the jobs in the queue to sleep. But we need to check the wait time when the thread awakes because it could be due to a new job that was recently added into the queue. In that case, the wait time is greater than 0, and we have to put the job back into the queue.
9. Since the wait time for event B is 0, we deregister event B from the queue and update the time now.
10. Ent2 notifies the receiver Ent1 with event B.
11. Event A is the only one left in the queue. Then we take A from the queue and check the wait time; we see that the wait time is greater than 0, so we update the wait time and put event A back to sleep.
12. Sleep for 4 seconds
13. Take A from the queue, and we see that the wait time is 0.
14. Deregister event A from the queue and update the time now.
15. Ent1 notifies the receiver Ent2 with event A.

We have described the sequence of steps of the scheduler. When the receiver entity is notified, then the conditions of that rule are examined. The sensor-related rules use the sliding window algorithm to determine the rule firings, which is what is described next.

4.2.3 Sliding Window

The key problem in supporting context-based rules is knowing when to check to see if the context matches a rule. Checking all rules all the time would result in too much battery being used (for rules operating in each APK RulesEngine) and too much CPU being used (for rules operating in the backend RulesEngine). The sliding window algorithm described in this section is an attempt to check rules in a smart way so that we do not use too many resources to check if a context matches a rule specification. Laguna et al. have done similar work by using a dynamic sliding window on different sensors for activity recognition [65]. Their results in good performance because the amount of data examined is reduced. For this reason, we too choose a sliding window approach.

We can observe in Fig 4.5 that all the rules extend *AbstractAtomicRule* that extends *AbsRuleImplementation*. This class contains the sliding window methods, which are used by sensor-related rules. A more detailed description of the rules can be found in Section 4.2.3.1. Here we focus on the design of the sliding window.

The reason we need to use an algorithm like a sliding window for sensor-related rules is since we need to go back in time to check if there has to be a rule firing due to previous sensor data rows in the database. Hence, we need a smart algorithm that iterates through just the relevant data but skips the unnecessary rows. To accomplish it, we have implemented the **InterpolatedPredicate** class.

Table 4.1 shows an example of the values that are kept in the two data members (**sensorTimeToPredicate** and **sensorTimeToDuration**) given some sensor data. We assume that the sensors record data every 5 minutes, and we are trying to see if the user is at home (AH). The columns in the table represent the time the location was recorded along with the two maps values. We see the first map contains the time as the key and the predicate (boolean) as the value. In this example, all the data that contains AH would be true (T), and everything else false (F). The other map also contains the time as the key and the duration that the predicate stayed constant. For example, we can observe that the predicate at 9:30 stays F for 20 minutes = 1200 seconds, and the predicate at 9:00 is T only for 10 minutes = 600 seconds. We assume that the predicate after the recorded time stays the same until the next recording.

Time	**Data**	**Map<time,predicate>**	**Map<time,duration>**
9:00	AH	9:00 ->T	9:00 ->9:10-9:00 = 600
9:10	Gym	9:10 ->F	9:10 ->9:20-9:10 = 600
9:20	AH	9:20 ->T	9:20 ->9:30-9:20 = 600
9:30	Gym	9:30 ->F	9:30 ->(9:40-9:30)+600 = 1200
9:40	Gym	9:40 ->F	9:40 ->9:50-9:40 = 600
9:50	AH	9:50 ->T	9:50 ->(10:00-9:50)+600 = 1200
10:00	AH	10:00 ->T	10:00 ->(10:10-10:00) = 600
10:10	Gym	10:10 ->F	10:10 ->0

Table 4.1: InterpolatedPredicate example using GPS data. The predicate is the participant is at AH.

Now that we have two different maps, we can obtain the predicate and its duration only in real-time. However, there could be cases where the time is not the same as the recording time since sliding window checks at the same rate as the sensor interval, which in this example is 10 min. For this reason, we have implemented the following methods to deal with this issue:

- **getVal**(time): returns the predicate for a given time. It uses the sensorTimeToPredicate map to return the correct value. Consider the example where time=9:15. The algorithm first divides the map sensorTimeToPredicate in two such that *headMap* contains all the values before 9:15 and *tailMap* contains all the values after or equal to 9:15. Then, we get the last element from the headMap (*lowerBoundaryTime=9:10*) and the first element from the tailMap (*upperBoundaryTime=9:20*). There are a few cases that we need to consider:

  - *lowerBoundaryTime* is null which means the time given is before the first sensor data row, then we return the predicate of *upperBoundaryTime*
  - *upperBoundaryTime* is null which means the time given is after the last sensor data row, then we return the predicate of *lowerBoundaryTime*.
  - Neither of them is null, then we return the predicate of *lowerBoundary-Time*.

  Therefore, in our example of 9:15, it returns the predicate of lowerBoundary-Time so it returns True.

- **getDuration**(time): returns the number of seconds that the predicate remains the same. It uses the *sensorTimeToDuration* map to return the correct duration. Consider the example where time=9:15. The algorithm first divides the map *sensorTimeToPredicate* in two such that *headMap* contains all the values before 9:15 and *tailMap* contains all the values after or equal to 9:15. Then, we get the last element from the headMap (*lowerBoundaryTime=9:10*) and the first element from the tailMap (*upperBoundaryTime=9:20*). There are a few cases that we need to consider:

  - *lowerBoundaryTime* is null, which means the time given is before the first sensor data row, then we assume it is the same predicate of the first sensor data row, and we calculate that time until it ends. For example, if time=8:45, then it returns 9-(8:45)+getDuration(9).
  - *upperBoundaryTime* is null, which means is after the last sensor data row, then we return infinite value to break out of the sliding window loop.

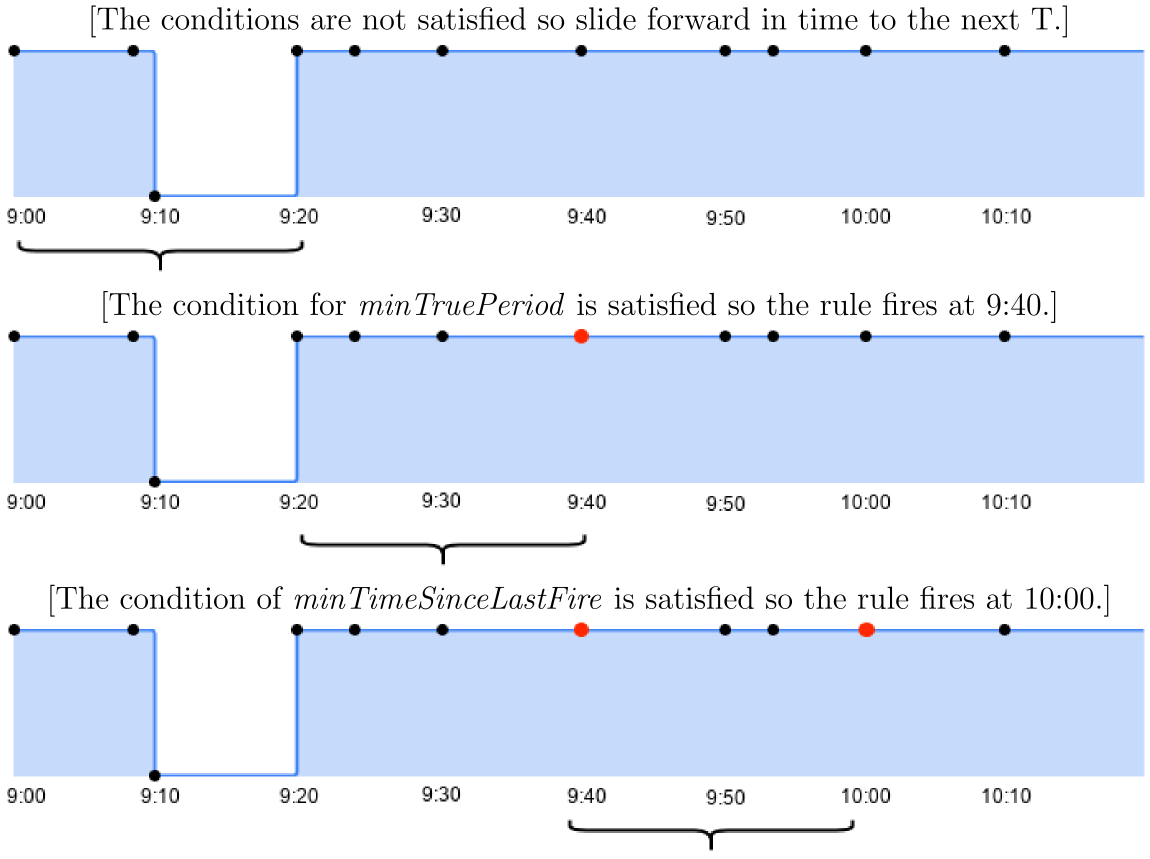
  – Neither of them is null, then if we use the example of 9:15, it returns 9:10-(9:15-9:10) = 600-(300) = 300.

In the previous section, we have mentioned that the sliding window contains two different methods. We explain the algorithm of these two methods with some examples.

- IShape: it is used to determine whether a condition is true for *minTruePeriod* amount of time. If so, then it slides the window of true values by *maxAnswerTime*, which is a parameter specified by the user. Consider the example given in Table 4.2 with *minTruePeriod=20min* and *maxAnswerTime=20min*. Fig 4.7 represents the data translated into a graph with 0 if F and 1 if T. The beginning of the sliding window corresponds to the *currentTime*. The sliding window starts at the *lastRuleFiredTime*, which in this case is 9:00. It checks the predicate at that time, which is only T for 10 minutes, hence, it slides forward in time to the next time the predicate is T (9:20) as can be seen in Fig 4.7. At 9:20, the predicate is T until 9:40, which means the *minTruePeriod* is satisfied, thus, the new *lastRuleFiredTime* is 9:40. Since the *minTruePeriod* has been satisfied, we need to check for the *minTimeSinceLastFired*. Sliding window begins at 9:40, shown in Fig 4.7, the predicate is T for at least 20 minutes. Hence, the *lastRuleFiredTime* is updated again to 10:00.

Time	Data
9:00	T
9:08	T
9:10	F
9:13	F
9:20	T
9:30	T
9:40	T
9:50	T
9:52	T
10:00	T
10:10	T

Table 4.2: Sample data for the IShape method



[The conditions are not satisfied so slide forward in time to the next T.]

[The condition for *minTruePeriod* is satisfied so the rule fires at 9:40.]

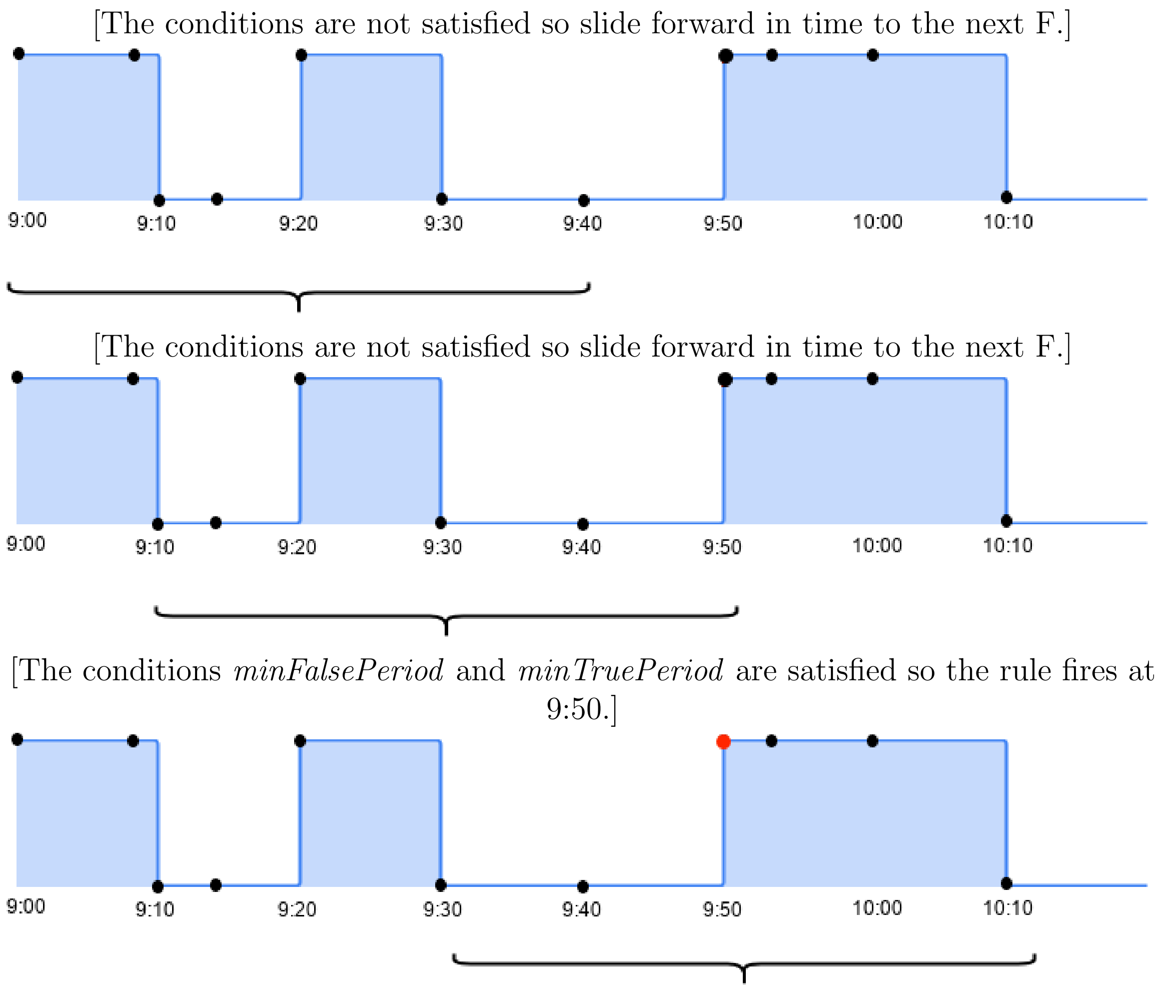
[The condition of *minTimeSinceLastFire* is satisfied so the rule fires at 10:00.]

Figure 4.7: Example of sliding Window method IShape. The mark underneath the graph represents the sliding window. The black dots represent the data which can be either 0 (F) or 1 (T), while the red dots represent the rule firings.

- LShape: it is used to determine whether a condition is false for *minFalsePeriod* and true for *minTruePeriod*. The window slides until there is a transition from false to true and checks if the parameters of *minFalsePeriod* and *minTruePeriod* are satisfied. Let's consider an example given the data in Table 4.3. We obtain the graph in Fig 4.8, which represents 0 (False) or 1 (True). We can see the transitions of the sliding window in Fig 4.8. In this example, the sliding window length is 40 minutes since it is necessary to have a F predicate for 20 minutes (*minFalsePeriod*) followed by T for 20 minutes (*minTruePeriod*). The sliding window starts at the *lastRuleFireTime*, which is 9:00 in this case. The beginning of the sliding window corresponds to the *currentTime*. The first step is to check the duration of the T predicate at 9am, since it is less than 20 minutes, it slides to 9:10 where the predicate is F as we can see in Fig 4.8. At that time, the predicate is F for 10 minutes, hence, it skips F and T since it does not matter the duration of the T predicate if the F is not at least 20 minutes. Therefore, the sliding window starts at 9:30 where the predicate is F as can be seen in Fig 4.8. In this case, it is F for at least 20 minutes and then T for at least 20 minutes. Hence, the new *lastRuleFireTime* is set to 9:50 since it is the transition from F to T. Then, the next time the sliding window begins at 9:50 since it is the *lastRuleFireTime*.

Time	Data
9:00	T
9:08	T
9:10	F
9:13	F
9:20	T
9:30	F
9:40	F
9:50	T
9:52	T
10:00	T
10:10	F

Table 4.3: Sample data for the LShape method


[The conditions are not satisfied so slide forward in time to the next F.]

[The conditions are not satisfied so slide forward in time to the next F.]

[The conditions *minFalsePeriod* and *minTruePeriod* are satisfied so the rule fires at 9:50.]

Figure 4.8: Example of sliding Window method LShape. The mark underneath the graph represents the sliding window. The black dots represent the data which can be either 0 (F) or 1 (T), while the red dots represent the rule firings.

4.2.3.1 Rules

RulesEngine contains different rule types as we can observe in Fig 4.5. Each of the rules have different parameters as well as a *maxAnswerTime* field, which is the maximum number of seconds to show the question for. When a researcher adds a new rule, the ruletype, parameters, and the filters are serialized into a JSON object. The generated object is stored in the *ruletext* field in the database.

Upon recurring time interval (Cron Rule) The time-based rule uses a Cron expression [66] to calculate the time from now until it is supposed to fire, and it fires at the time specified in the *cronString*.

Upon given day of study participation (timeSinceEnrollment) The researcher enters the number of days *duration* and the time that this rule has to fire (*hourOfday*), then when the subject registers on the app and signs the consent, it calculates the time it needs to sleep until the rule has to fire.

Upon end of study (endOfStudyRule) This rule uses the number of days the study lasts (*studyDuration*), which is entered by the researcher when creating the study. Finally, when the subject registers, the rule takes the number of days and calculates how much time the thread needs to sleep for before the study ends. When the time has passed, the rule fires and the subject has completed the study, meaning that no more questions would be asked, and no more data would be recorded.

Upon specific/any answer to an asked question (follow-up, excluding exit interview follow-up) (ChainRule) A question with this rule type is used to specify that it should be asked with a specific *delay* after the participant has answered a specific multiple-choice *question* (excluding exit interview) with specific choice *response*.

Upon specific/any answer to an asked end of study question (exit interview follow-up) (EndOfStudyChainRule) A question with this rule type is used to specify that it should be asked with a specific *delay* after the participant has answered a specific end-of-study multiple-choice *question* with specific choice *response*.

The following rules are sensor-based rules, meaning that their conditions are based on sensor data.

Location This rule type contains the following parameters: *latitude*, *longitude*, *radius*, and *minTimeSinceLastFire*, which is only used in WhileAt and WhileNotAt. The logic in which these parameters are being interpreted is based on the different types of location rules:

- **While at specified location (WhileAt)**: A question with this rule type is asked every "minTimeSinceLastFire" while the participant is WITHIN "radius" of the given location ("latitude" and "longitude"), after the participant has been within "radius" of the given location for at least "minTruePeriod".

- **While NOT at specified location (WhileNotAt)**: A question with this rule type is asked every "minTimeSinceLastFire" while the participant is NOT within "radius" of the given location ("latitude" and "longitude"), after the participant has not been within "radius" of the given location for at least "minTruePeriod".

- **Upon arrival at specified location (OnArrival)**: A question with this rule type is asked when the participant has been WITHIN "radius" of the given location (“latitude” and “longitude”) for at least “minTruePeriod”, but immediately prior WAS NOT WITHIN ”radius” of the given location for at least “minFalsePeriod”.

- **Upon departure from specified location (OnDeparture)**: A question with this rule type is asked when the participant has NOT been within “radius” of the given location (“latitude” and “longitude”) for at least “minTruePeriod”, but immediately prior WAS WITHIN “Y” of the given location for at least “minFalsePeriod”.

Bluetooth This rule type contains the following parameters: *delay*, *RSSI*, and *count*. The logic in which these parameters are being interpreted is based on the different types of bluetooth rules:

- **Upon forming/joining a group of participants (OnArrival)** A question with this rule type is asked “delay” seconds after the following: the participant was WITHIN “RSSI” of “count” (or more) other participant(s) for at least “minTruePeriod” seconds, but immediately prior WAS NOT WITHIN “RSSI” of these same participant(s) for at least “minTruePeriod”.

- **Upon someone leaving a group of participants (OnDeparture)** A question with this rule type is asked “delay” seconds after the following: the participant was NOT WITHIN “RSSI” of “count” (or more) other participant(s) for at least “minTruePeriod” seconds, but immediately prior WAS WITHIN “RSSI” of these same participant(s) for at least “minTruePeriod”.

Beacon This rule type contains the following parameters: *delay*, *RSSI*, *minor*, *minTimeSinceLastFire*, which is only used in WhileAt and WhileNotAt. The logic in which these parameters are being interpreted is based on the different types of beacon rules:

- **While in the proximity of beacon device(s) (WhileAt)** A question with this rule type is asked every “mminTimeSinceLastFire” while the participant is WITHIN “RSSI” of any beacon(s) with “minor”. Ask this question for the first time “delay” after the participant has been within “RSSI” of any (other or the same) beacon(s) with “minor” for at least “minTruePeriod”.

- **While NOT in the proximity of beacon device(s) (WhileNotAt)** A question with this rule type is asked every “mminTimeSinceLastFire” while the participant is NOT WITHIN “RSSI” of any beacon(s) with “minor”. Ask this question for the first time “delay” after the participant has NOT been within “RSSI” of any (other or the same) beacon(s) with “minor” for at least “minTruePeriod”.

- **Upon being in the proximity of beacon device(s) (OnArrival)** A question with this rule type is asked “mminTimeSinceLastFire” seconds after the following: the participant was WITHIN “RSSI” of any beacon(s) with “minor” for at least “minTrue”, but immediately prior WAS NOT WITHIN “RSSI” of any (other or the same) beacon(s) with “minor” for at least “minFalse” seconds.

- **Upon leaving beacon device(s) (OnDeparture)** A question with this rule type is asked “mminTimeSinceLastFire” seconds after the following: the participant was NOT WITHIN “RSSI” of any beacon(s) with “minor” for at least “minTrue”, but immediately prior WAS WITHIN “RSSI” of any (other or the same) beacon(s) with “minor” for at least “minFalse” seconds.

Activity Recognition This rule type contains the following parameters: *delay*, *activity*, *minTimeSinceLastFire*, which is only used in WhileAt and WhileNotAt. The logic in which these parameters are being interpreted is based on the different types of activity recognition rules:

- **While doing specified activity (WhileAt)** A question with this rule type is asked every "minTimeSinceLastFire" while the participant is doing "activity". Ask this question for the first time "delay" after the participant has been doing "activity" for at least "minTruePeriod".

- **While NOT doing specified activity (WhileNOTAt)** A question with this rule type is asked every "minTimeSinceLastFire" while NOT the participant is doing "activity". Ask this question for the first time "delay" after the participant has NOT been doing "activity" for at least "minTruePeriod".

- **Upon starting specified activity (OnArrival)** A question with this rule type is asked "delay" after the following: the participant was "activity" for at least "minTruePeriod", but immediately prior WAS NOT "activity" for at least "minFalsePeriod".

- **Upon finishing specified activity (OnDeparture)** A question with this rule type is asked "delay" after the following: the participant was NOT "activity" for at least "minTruePeriod", but immediately prior WAS "activity" for at least "minFalsePeriod".

E4 (physiology) Rules This rule type contains the following parameters: *time*, *threshold*, *delay*, *minTimeSinceLastFire*. The last one is only used in WhileAt and WhileNotAt. The logic in which these parameters are being interpreted is based on the different types of activity recognition rules.

- **While GSR above average (While)** A question with this rule type is asked every "minTimeSinceLastFire" while the participant's GSR is "threshold" ABOVE the average GSR over the last "time." Ask this question for the first time "delay" after the participant's GSR is "threshold" ABOVE the average over the last "time" seconds for at least *minTruePeriod*.

- **While GSR below average (While)** A question with this rule type is asked every "minTimeSinceLastFire" while the participant's GSR is "threshold" BELOW the average GSR over the last "time". Ask this question for the first time "delay" after the participant's GSR is "threshold" BELOW the average over the last "time" seconds for at least *minTruePeriod*.

- **Upon GSR above average (OnArrival)** A question with this rule type is asked "delay" after the following: the participant'S GSR WAS ABOVE "threshold" of the average in the last "time" for at least "minTruePeriod," but immediately prior WAS BELOW "threshold" in the last "time" for at least "minFalsePeriod".

- **Upon GSR below (OnDeparture)** A question with this rule type is asked "delay" after the following: the participant's GSR WAS BELOW "threshold" of the average in the last "time" for at least "minTruePeriod," but immediately prior WAS ABOVE "threshold" in the last "time" for at least "minFalsePeriod."

4.2.3.2 Filters

The Rules Engine contains the following types of filters:

Time A question with this filter is asked when the participants context matches the rule and all associated filters, and, additionally, the time of day is between the *startTime* and *endTime*.

Location A question with this filter is asked when the participants context matches the rule and all associated filters, and, additionally, the participant is within *radius* of the given location (*latitude* and *longitude*).

Answer A question with this filter is asked when the participants context matches the rule and all associated filters, and, additionally, the most recent response to *question* was *answer*.

4.2.3.3 Sensors

ODIN supports different sensors to collect data and store it in the database, and this data is then used to test to see if rules should be fired. Each sensor has different parameters to determine when the sensor measurements should happen. These parameters are stored in the *sensorparams* field in *studytosensor* table. We have the following sensor types:

GPS location The GPS [67] sensor is used to track the location of the subject. It records the **latitude** and **longitude** coordinates. The researcher can specify the following parameters in the ResearcherUI: *time* and *distance*. The first one is how often we need to record GPS data from the subject. The second one is the distance (meters) that the user has to move to record data. If 0 meters is selected, GPS records every selected number of seconds regardless of location and movement. The GPS sensor is only used if it is enabled in the study. If the GPS sensor is enabled, then there exists a **GPS** table in the study database. This table contains the fields *id*, *time*, *data* and *coupon*. There are multiple subjects registered in the study, each one carrying a phone which belongs to a coupon. Hence, the backend keeps all the GPS data from all the coupons associated with a study in a single table within the studys database. The *coupon* field in the table helps to differentiate them.

Bluetooth Proximity The Bluetooth [68] sensor is used to determine phone to phone Bluetooth proximity. It records the **Bluetooth name** of the other devices in the Bluetooth range and the **RSSI**. The researcher can specify the *time* in the ResearcherUI, which is how often the Bluetooth scanning from phone to phone should happen. This sensor is only used if the Bluetooth sensor is enabled in the study. If the Bluetooth sensor is enabled, then there exists a **Bluetooth_proximity** table in the study database. This table contains the fields *id*, *time*, *data* and *coupon*. Hence, the backend keeps all the Bluetooth data from all the coupons associated with a study in a single table within the studys database. The *coupon* field in the table helps to differentiate them.

Beacon Proximity The Beacon [69] sensor is used to determine beacon to phone proximity. It records the **Beacon details** and the **RSSI**. The researcher can specify the *time* in the ResearcherUI, which is how often the Bluetooth scanning from phone to beacon should happen. This sensor is only used if the Beacon sensor is enabled in the study. If the Beacon sensor is enabled, then there exists a **Beacon_proximity** table in the study database. This table contains the fields *id*, *time*, *data* and *coupon*. Hence, the backend keeps all the Beacon data from all the coupons associated with a study in a single table within the studys database. The *coupon* field in the table helps to differentiate them.

Activity Recognition The Activity Recognition [70] sensor is used to determine the different activities that the subject performs throughout the day. It records the different **activities** that the user could be performing and the **confidence level** for each activity. The researcher can specify the *time* in the ResearcherUI, which is how often the Activity recognition sensor should record the activities in the phone database. This sensor is only used if the Activity recognition sensor is enabled in the study. If the activity recognition sensor is enabled, then there exists a **Activity_Recognition** database. This table contains the fields *id*, *time*, *data* and *coupon*. Hence, the backend keeps all the Activity Recognition data from all the coupons associated with a study in a single table within the studys data. The *coupon* field in the table helps to differentiate them.

Empatica E4 (physiology) The Empatica E4 [71] sensor is used the determine specific measures from the subject. It records the **GSR** and the **skin temperature** of the subject. The researcher can specify the *time* in the ResearcherUI, which is how often the Empatica E4 sensor should record data in the phone database. This sensor is only used if the Empatica E4 sensor is enabled in the study. If the Empatica E4 sensor is enabled, then there exists a **EmpaticaE4** database. This table contains the fields *id*, *time*, *data* and *coupon*. Hence, the backend keeps all the Empatica E4 data from all the coupons associated with a study in a single table within the studys database. The *coupon* field in the table helps to differentiate them.

Chapter 5

Android

Android [52] is an Operating System; it can also be considered an open-source platform that separates the software from the hardware to make it easier for developers to design mobile applications using the Android Software Development Kit (SDK) [72]. An application is composed of the following components [72]:

- Activity: a screen that the user sees on the phone. An application contains multiple activities.
- Intent: used to communicate among the components or other applications.
- Service: runs in the background and can perform actions that do not require to show a screen.
- Content Provider: an interface to retrieve and manage the data in the phone's SQLite database.
- Broadcast Receiver: an Android implementation of an Observer pattern; it only executes when there is an event that triggers it. For example, starting an application or running low on battery.

These are brief descriptions of each component, but we see later in this Chapter better examples while describing ODIN.

In this Section, we present the structure of the ODIN Android application. We start by giving an overview of the design, and then in the next Section, we describe into more detail all the components of the application.

5.1 Design

The design of the ODIN Android application can be seen in Fig 5.1, which shows the packages included in the ODIN APK along with the different components. The main ones we can observe are the following: **Activity**, **Service**, and **Content Provider**, which is represented in the figure as the communication to the database.

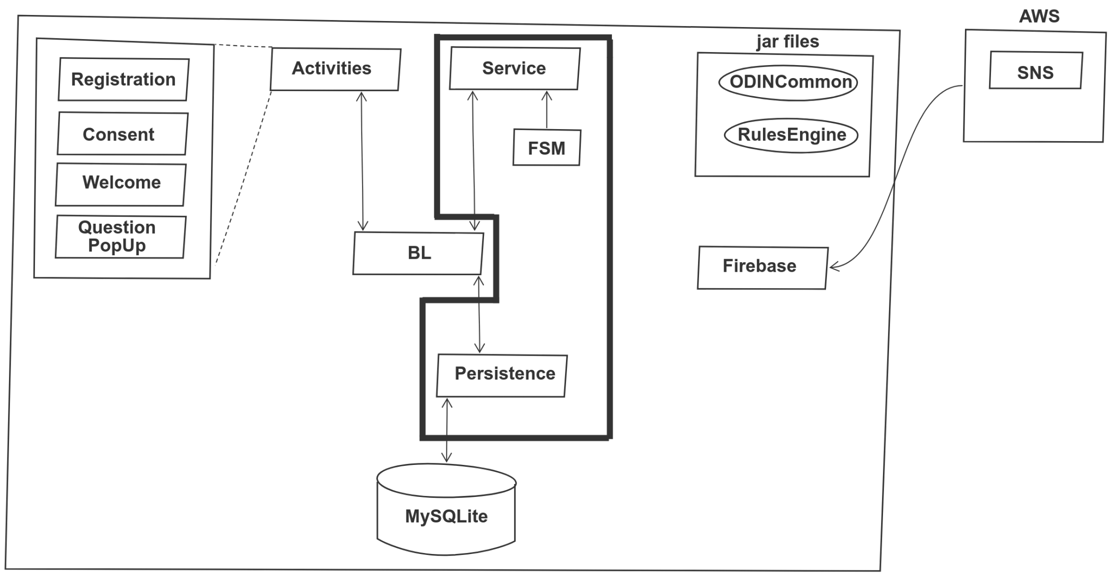


Figure 5.1: ODIN Android app overview. This diagram shows how the activities and services connect to the database. The jar files that are shared with the backend. The right side of the figure represents how the app receives notifications from the backend. The highlighted box is later used to relate it to Fig 5.18. The outer rectangle represents the APK.

The **activities** component in the ODIN app are made visible in the following order: the participant registers, signs the consent form, welcome screen, and questions.

When the participant opens the app, they see the *Registration* screen where they can enter the coupon number. After the participant selects the register button, the application redirects to the *Consent* screen. Some studies do not require a consent screen form. In that case, the participant is redirected to the *Welcome* activity right after registration. Once the participant has agreed to the terms and conditions of the study, then the *Welcome* activity is shown, and questions are prompted later on. When a rule fires, the phone is notified in the Firebase class, and then a question is prompted in the *Question PopUp* activity. There are also other activities in the menu list like *Settings*, *FAQ*, etc.

ODIN runs a different number of **services** depending on the study created by the researcher. There are some services that run by default in all the studies like *Rule-QuestionService* and *UploadService*. We say that the number of services depends on the study because the sensors vary among studies. Each sensor has its own dedicated service in the application. Each service is designed as a **Finite State Machine** (FSM) design.

The BL and Persistence layers are an intermediate layer within the activities, services, and the database. The activities and the services talk to the **Business Logic** (BL). BL is a layer between the upper layer activities and the lower layer content provider. The **persistence layer** asks for the necessary information from the corresponding database tables and sends it to the BL.

Content providers help manage the communication with the database to retrieve, insert, or update data. Each table has a content provider that contains the basic MySQL queries to perform actions such as insert, update, or delete.

Notifications from the backend are sent through Amazon Simple Notification System (**SNS**), which are received through Firebase. The application receives push backend notifications every 15 minutes to ensure all the services are running. Also, when there is a rule firing in the backend (in the backend RulesEngine), then the phone receives a notification to prompt the question associated with that rule.

The application contains two jar files to share between the backend and the phone: **ODINCommon** and **RulesEngine**. As mentioned in Chapter 4, ODINCommon is shared among all the services, and the RulesEngine is part of the PhoneAppService.

Business Logic is a layer in between the activities and the DAO to retrieve or update the database. It is in charge of making REST calls to the server to request or send information over the network. Fig 5.2 presents the diagram of the business logic layer. All the business layer classes send information periodically to the backend. They also request information from the server, for example, when the participant registers for the first time. We can also observe there is a Hardware class that goes to the mobile hardware and retrieves some information needed like the phone model, OS version, and some other relevant information. This data is sent to the server and is stored in the database.

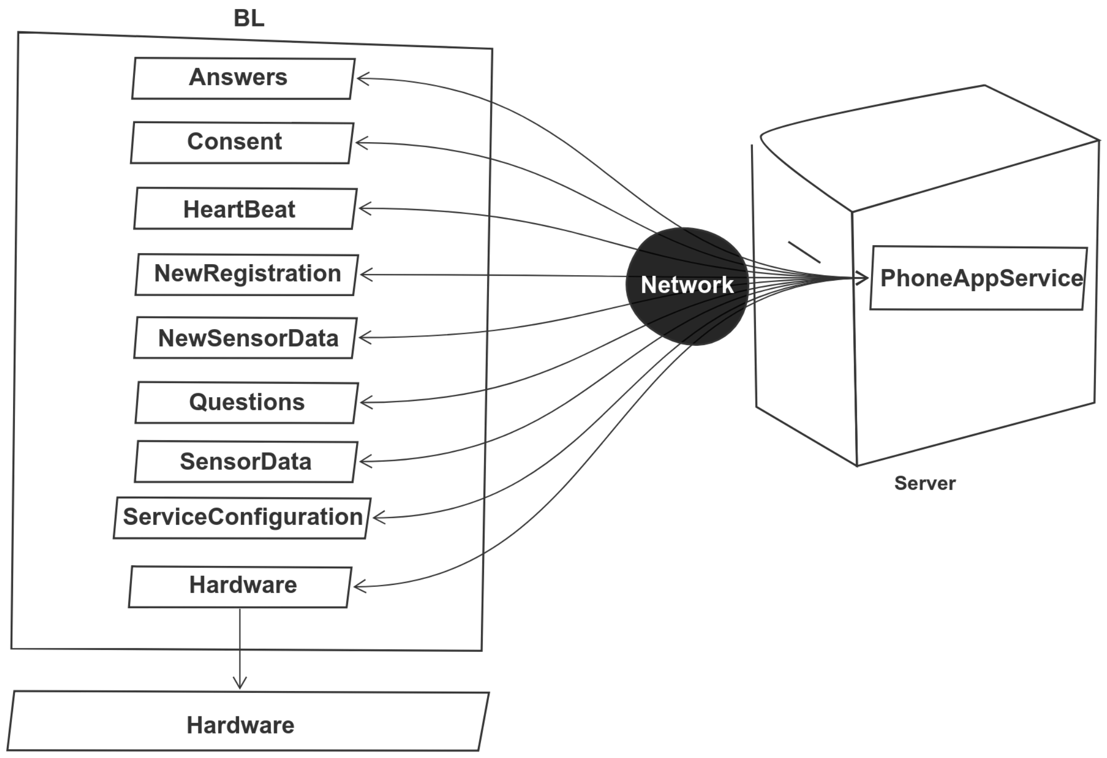


Figure 5.2: Diagram of Business Logic communicating with the backend over the network. The left side shows the BL classes in ODIN Android. The right side represents the PhoneAppService.

The Persistence Layer retrieves and updates the phone database; Its structure varies depending on the type of table: sensor or non-sensor. Each database table has a custom Table class (extends AbstractTable), as well as ReaderWriter, and ContentProvider classes. The Content Provider calls the AppDatabase, which creates and updates the tables using SQLite. We can observe in Fig 5.3 that there are two types of classes: no sensor-related (Consent or Answers) and the classes that implement the “ISensor” interface which is the GPS or Bluetooth. We differentiate these two because some methods are required if the table type is a sensor. For example, if the rule is while at a specific location, then we need to retrieve all the sensor data to check if the conditions are satisfied.

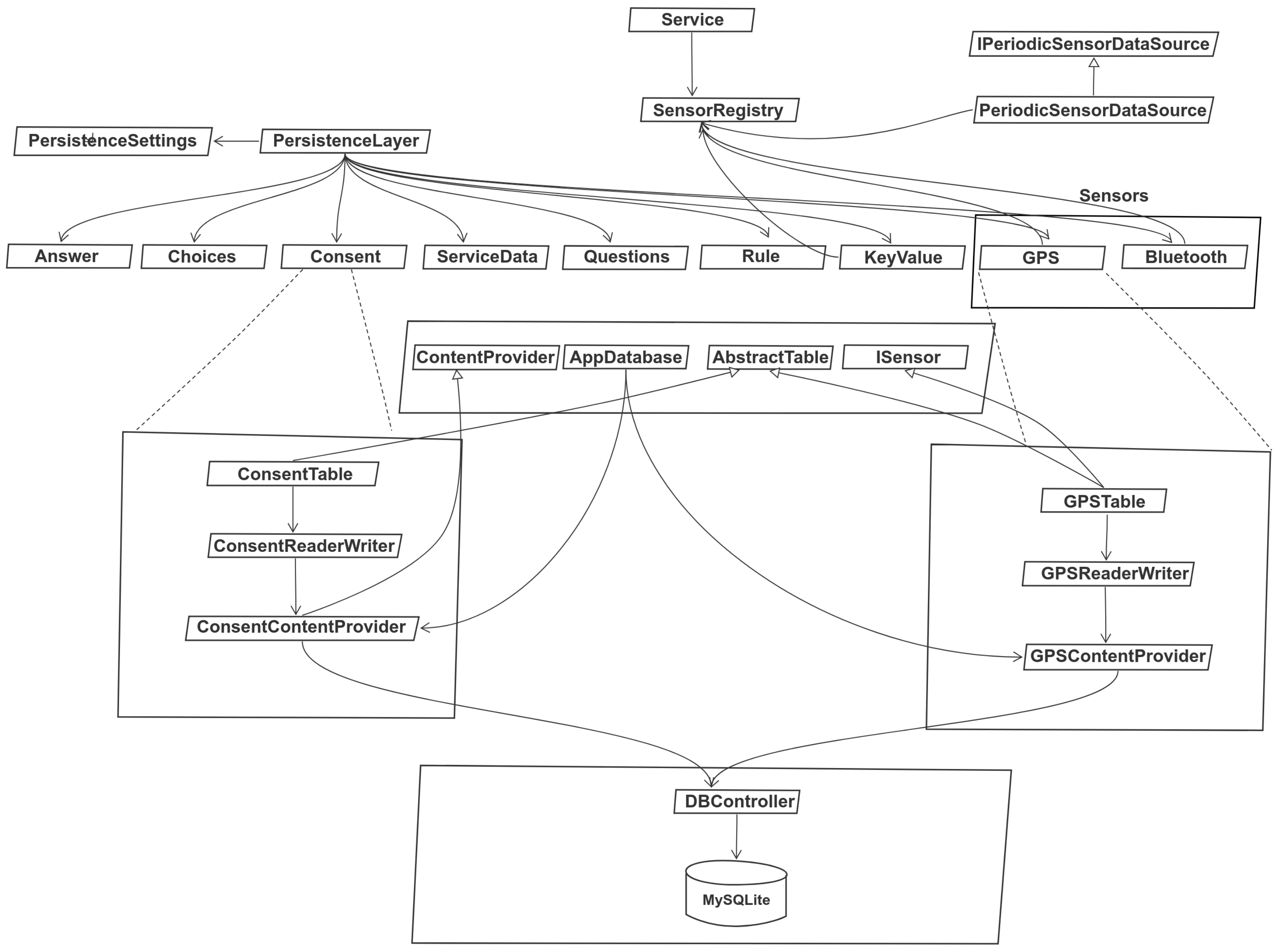


Figure 5.3: Diagram that represents the structure of the DAO layer. The sensor tables are registered into the SensorRegistry and implement IPeriodicSenorDataSource. It shows the difference between a non-senor table (ConsentTable) and a sensor related table (GPSTable).

5.2 Details

We have described the main components of the ODIN Android application. The next Section explains in more detail the database schema, the rules engine instance that is operating in each phone, different scenarios, and the services.

5.2.1 Database Schema

We can observe the database schema in Fig 5.4 of a study that contains a GPS sensor. It contains the following tables: Choices, Questions, Answer, Rule, KeyValue, ServiceData, Consent, and GPS.

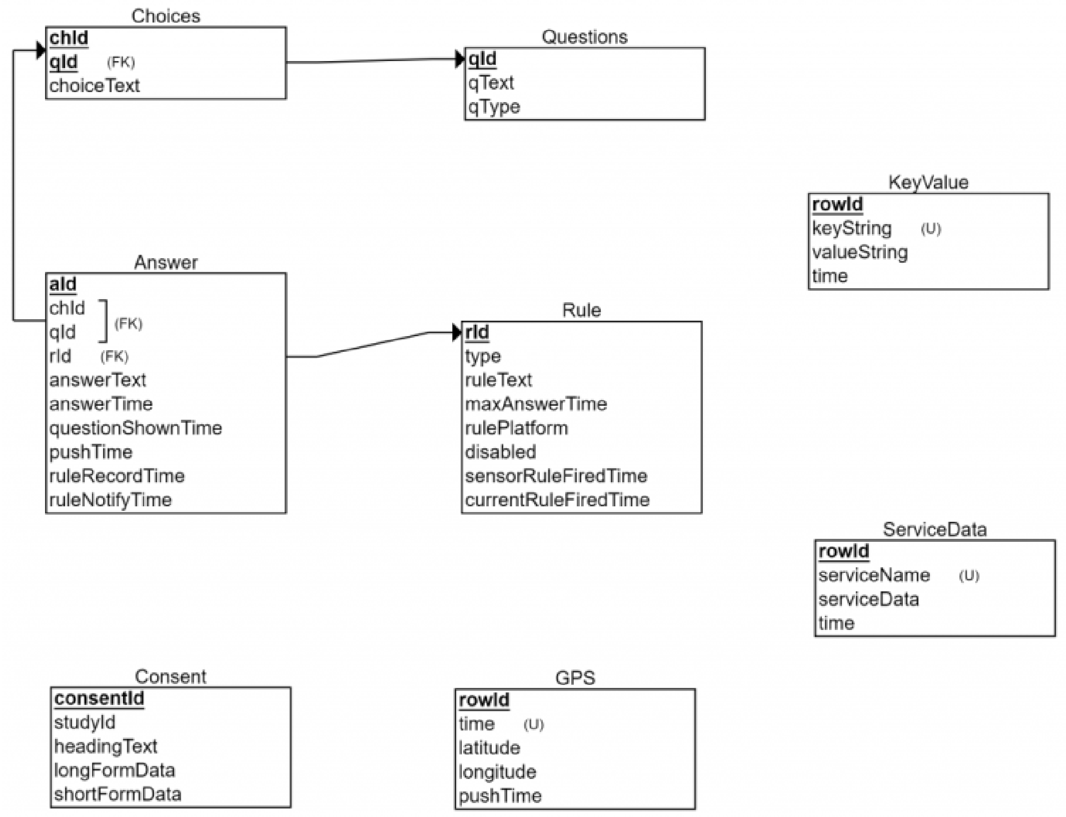


Figure 5.4: Android database schema

- When the user registers, the phone receives the consent form data, which is stored in the **Consent** tables. This table matches the Consent table in the backend.

- After the consent agreement, the phone makes a REST call to the backend to receive all the questions and rules. The questions are stored in the **Questions** table.
- Each question contains a list of choices that are stored in the **Choices** table.
- The questions have a set of rules associated with them. These rules are stored in the **Rule** table.
- When a rule fires either in the backend or on the phone, the user is prompted with a question from the Questions table. The answer is stored in the **Answer** table.
- Some of the rules may be associated with sensors. Each sensor has a different table with the fields associated with that sensor. In this example, we have a study involving use of the GPS sensor; thus, the **GPS** table contains the fields: latitude and longitude.
- The **KeyValue** table is used to store data about the services and the statistics of the phone. For example, the heartbeat count of the service, the intervals that the alarms are scheduled, the last time the phone was restarted, etc.
- The **ServiceData** table stores the instance of each service in the *serviceName* field, the *serviceData* attribute contains all the alarms for that service and the current state since the services follow an FSM design. We cannot keep this information in memory because the application could be killed or the phone

could power off. Hence, we need to store it so that if it is not in memory anymore, we can get it from disk.

5.2.2 Rules Engine (Intent-based)

Designing a functional and reliable scheduler on Android was one of the most challenging tasks in the development of the ODIN system. We tried using the same scheduler as the backend for Android, but we came across some issues due to the sleeping time of the thread. The OS was killing the thread, and rules would never fire. For this reason, we decided to use a different Rules Engine design by scheduling alarms instead of making the thread sleep.

5.2.2.1 Scheduler

Fig 5.5 represents the sequence diagram of how the Android Scheduler works. Assume we have two events (A and B) and two entities (Ent1 and Ent2). First, the sender entity registers the event with the receiver entity and the number of seconds until it has to fire. Then, we create a new event, and we add it to the queue. In this case, we have two entities registered, Ent1 with 5 seconds and Ent2 with 1 second. We take the first one from the queue and schedule an alarm passing the wait time to the OS. Since we registered A first, then we schedule an alarm for event A and then B. When the time is up, the broadcast receiver collects an intent, and it is notified that the thread needs to start its execution. In our example, it would be event B since it has only 1 second of wait time. Subsequently, the broadcast receiver initiates a new job in the Scheduler, which checks the GUID, whether it is a RulesEngine from a previous intent or it is still the same RulesEngine instance. This is done to avoid duplicate rule firings. The RulesEngine can be killed at any time by the OS, so if we see that the RulesEngine has died, we start it again, which leads to new scheduling of alarms. To keep track of the RulesEngine instance that we are currently running, we create a GUID for every RulesEngine instance. When the rule fires, we compare the GUID to know if it is from a previous instance or the current one. Finally, when the job has terminated, we deregister it from the scheduler, and the event is sent to the receiver. These methods are synchronized to avoid scheduling a new job during this process since we can have many entities scheduling alarms with the same wait time.

Despite the issues we faced in the beginning, by using the Scheduler from the backend, we have a stable Scheduler running indefinitely in our ODIN application. We accomplished it by using alarms and keeping track of the RulesEngine instance.

5.2.3 Scenarios

This Section covers important app scenarios that we need to consider to understand how all the components of the APK work together. We cover 11 different scenarios that can range from any event like the subject's first registration to killing the application. There are some common behaviors in the scenarios:

- If a REST call fails, we register the class in WifiStatusReceiver if it is not already registered, wait for Wifi and retry.
- When the user has completed the Registration and Consent activities, we prevent the participant from going back to those activities by calling the method finish().

Next, the different scenarios are described in detail.

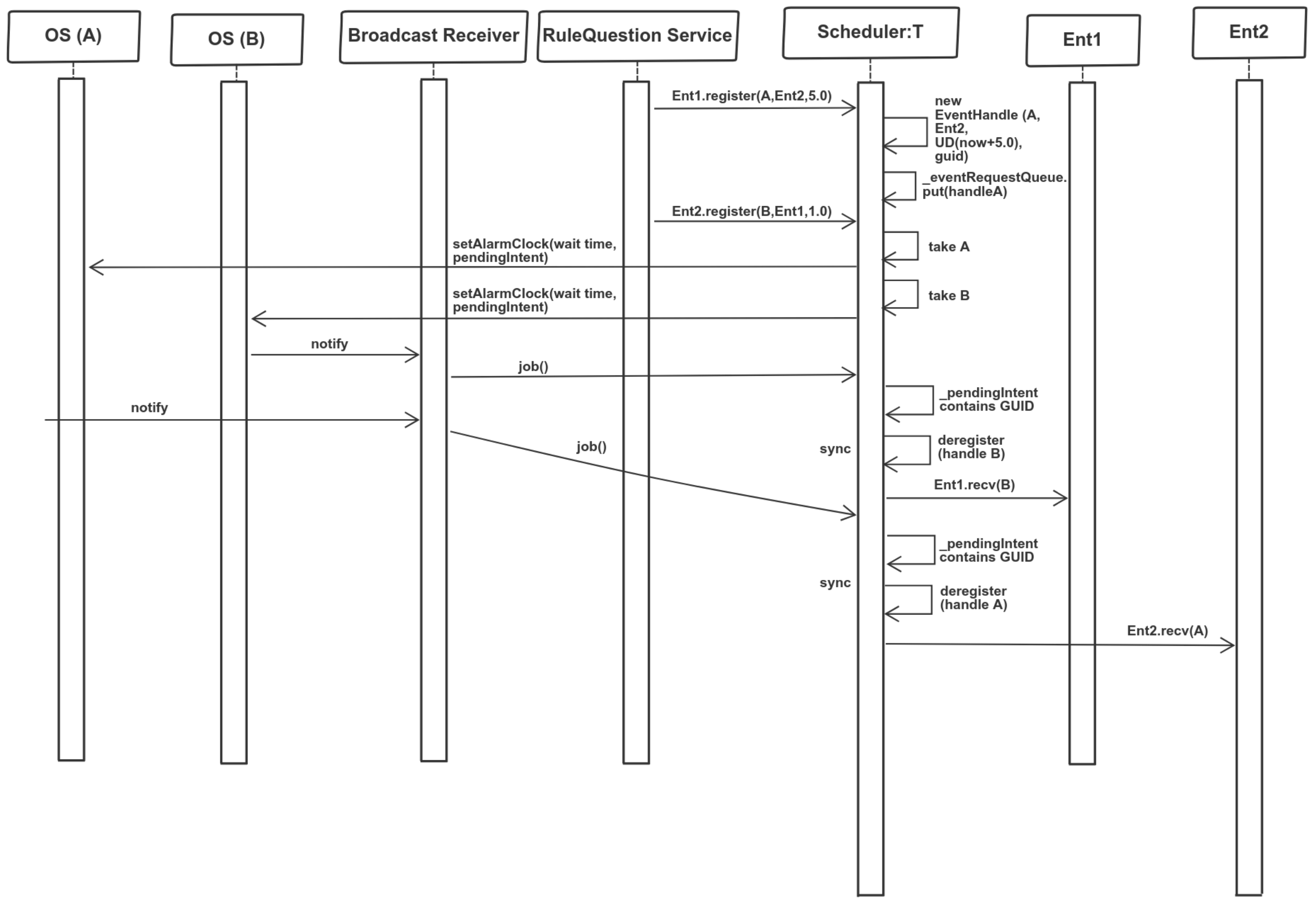

OS (A)
OS (B)
Broadcast Receiver
RuleQuestion Service
Scheduler:T
Ent1
Ent2
Ent1.register(A,Ent2,5.0)
new EventHandle (A, Ent2, UD(now+5.0), guid)
_eventRequestQueue. put(handleA)
Ent2.register(B,Ent1,1.0)
setAlarmClock(wait time, pendingIntent)
take A
setAlarmClock(wait time, pendingIntent)
take B
notify
job()
_pendingIntent contains GUID
notify
job()
sync
deregister (handle B)
Ent1.recv(B)
_pendingIntent contains GUID
sync
deregister (handle A)
Ent2.recv(A)

Figure 5.5: Rule Engine Android Scheduler sequence diagram. Ent1 is the sender and Ent2 the receiver of event A; Ent2 is the sender and Ent1 the receiver of event B.

5.2.3.1 Application launch

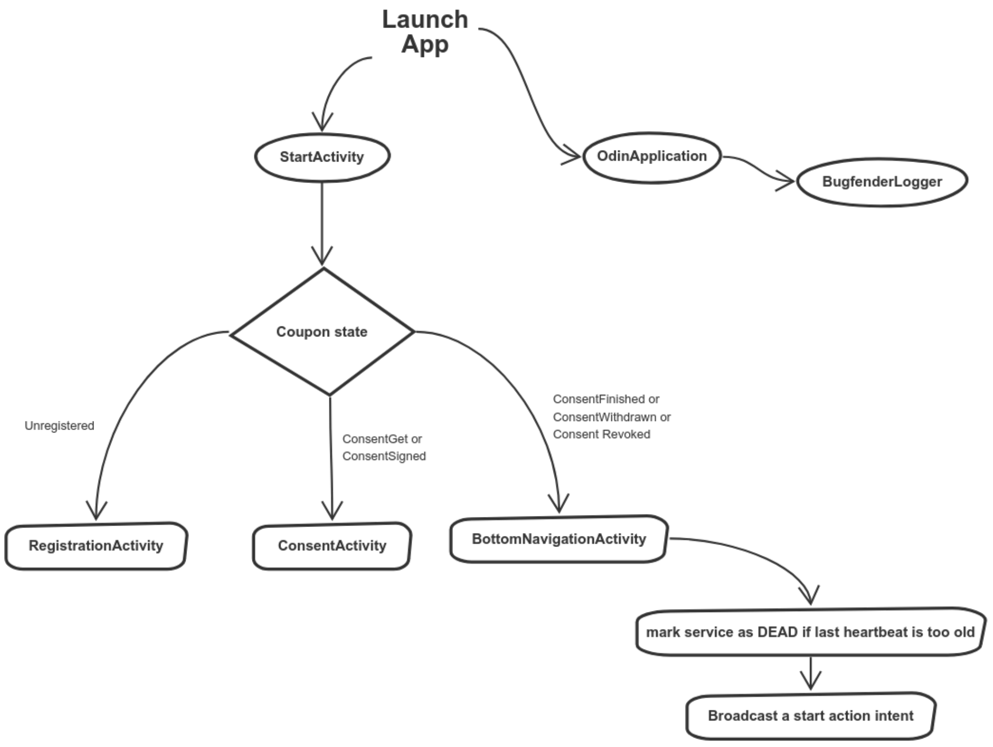

Launch App
StartActivity
OdinApplication
BugfenderLogger
Coupon state
Unregistered
ConsentGet or ConsentSigned
ConsentFinished or ConsentWithdrawn or Consent Revoked
RegistrationActivity
ConsentActivity
BottomNavigationActivity
mark service as DEAD if last heartbeat is too old
Broadcast a start action intent

Figure 5.6: Application launch scenario

The first scenario is when the user selects the ODIN icon to launch the application. We can observe in Fig 5.6 that the classes StartActivity and OdinApplication are executed in the first place. The latter initializes ResearchStack [41] and Bugfender (our logging system). The former calls the ODINCapabilities class to find out the coupon state that the participant is using to register. There are three possible states:

- First-time registration: redirects to the registration activity for the participant to enter the coupon and begin the study.

- Second-time registration: redirects to consent activity so that the user can agree to the terms and conditions of the study.
- Participation in that study has concluded: redirects to the main activity where the user can answer the questions, report events, or view other menu options. Afterward, we check all the services' heartbeat and set their status to DEAD if the last heartbeat is too old. Finally, we broadcast a start action intent to all the services.

5.2.3.2 Registration

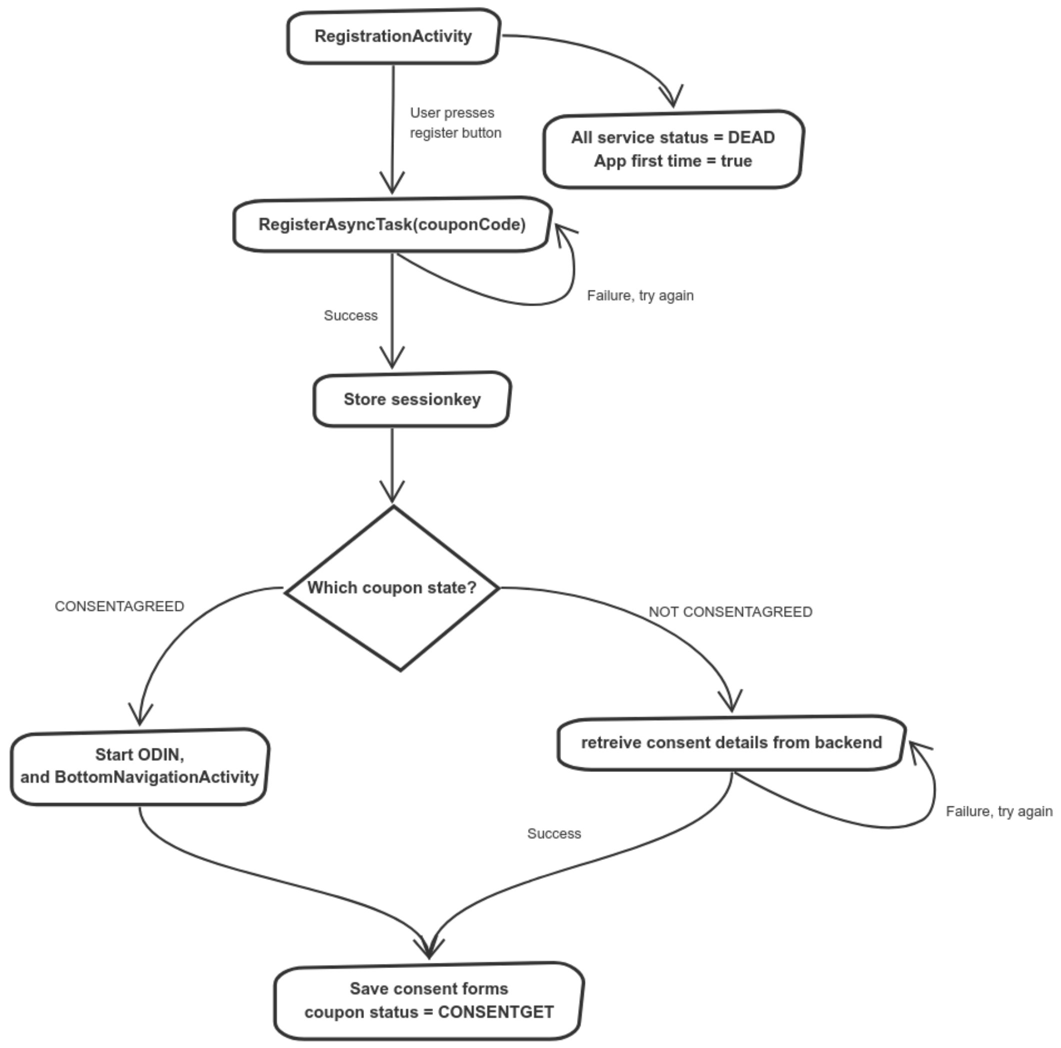

RegistrationActivity
User presses register button
All service status = DEAD
App first time = true
RegisterAsyncTask(couponCode)
Failure, try again
Success
Store sessionkey
Which coupon state?
CONSENTAGREED
NOT CONSENTAGREED
Start ODIN, and BottomNavigationActivity
retreive consent details from backend
Failure, try again
Success
Save consent forms
coupon status = CONSENTGET

Figure 5.7: Registration scenario

Fig 5.7 illustrates the sequence of events during the registration process. The participant opens the ODIN application and sees the RegsitrationActivity with an input text box to enter the coupon number. After typing the coupon number, the Register button is selected. When this happens, a new thread is created to set all the services to DEAD and set the flag for the first time, starting the app to true. In parallel, the main thread initiates a REST call to the backend server to send the registration details. If the REST call is successful, the backend returns a session key to be able to identify the phone in every REST call made after this point. This key is stored in the local database on the phone. Additionally, the coupon state is retrieved from the database and we check if the researcher enabled the consent form. Different scenarios are depending on the value of the consent state:

- CONSENTAGREED: the ODIN system is started by sending a start broadcast message in the KeepAliveOneShotReceiver class. Then, the user is redirected to the main activity (BottomNavigationActivity) that contains the prompted questions, events to report, and other menu options.
- NOT CONSENTAGREED: initiate a REST call to the backend to retrieve the consent details.

In both cases, we save the information to the local database and change the coupon status to CONSENTGET.

5.2.3.3 Consent form

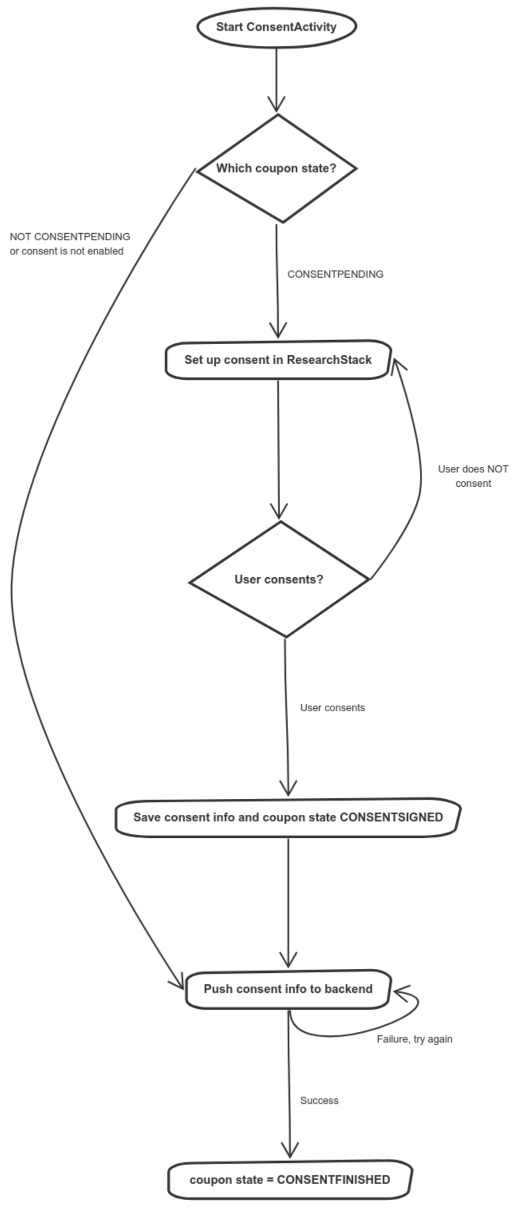

Start ConsentActivity
Which coupon state?
NOT CONSENTPENDING or consent is not enabled
CONSENTPENDING
Set up consent in ResearchStack
User does NOT consent
User consents?
User consents
Save consent info and coupon state CONSENTSIGNED
Push consent info to backend
Failure, try again
Success
coupon state = CONSENTFINISHED

Figure 5.8: Consent form scenario

After the coupon status has changed to CONSENTGET (end of section 5.2.3.2), the Consent Activity is started. There are two possible cases as can be seen in Fig 5.8:

- If the user has already signed the consent form previously or the researcher has disabled it.
- The consent status is CONSENTPENDING: set up consent details in the ResearchStack library and show the consent to the participant. If the user chooses not to consent, it goes back to the initial page of the consent. If the user agrees to the terms and conditions of the study, then the consent state is updated to CONSENTSIGNED.

In both cases, the consent information is sent to the backend. If the REST call is successful, the coupon state is updated to CONSENTFINISHED.

5.2.3.4 Start ODIN

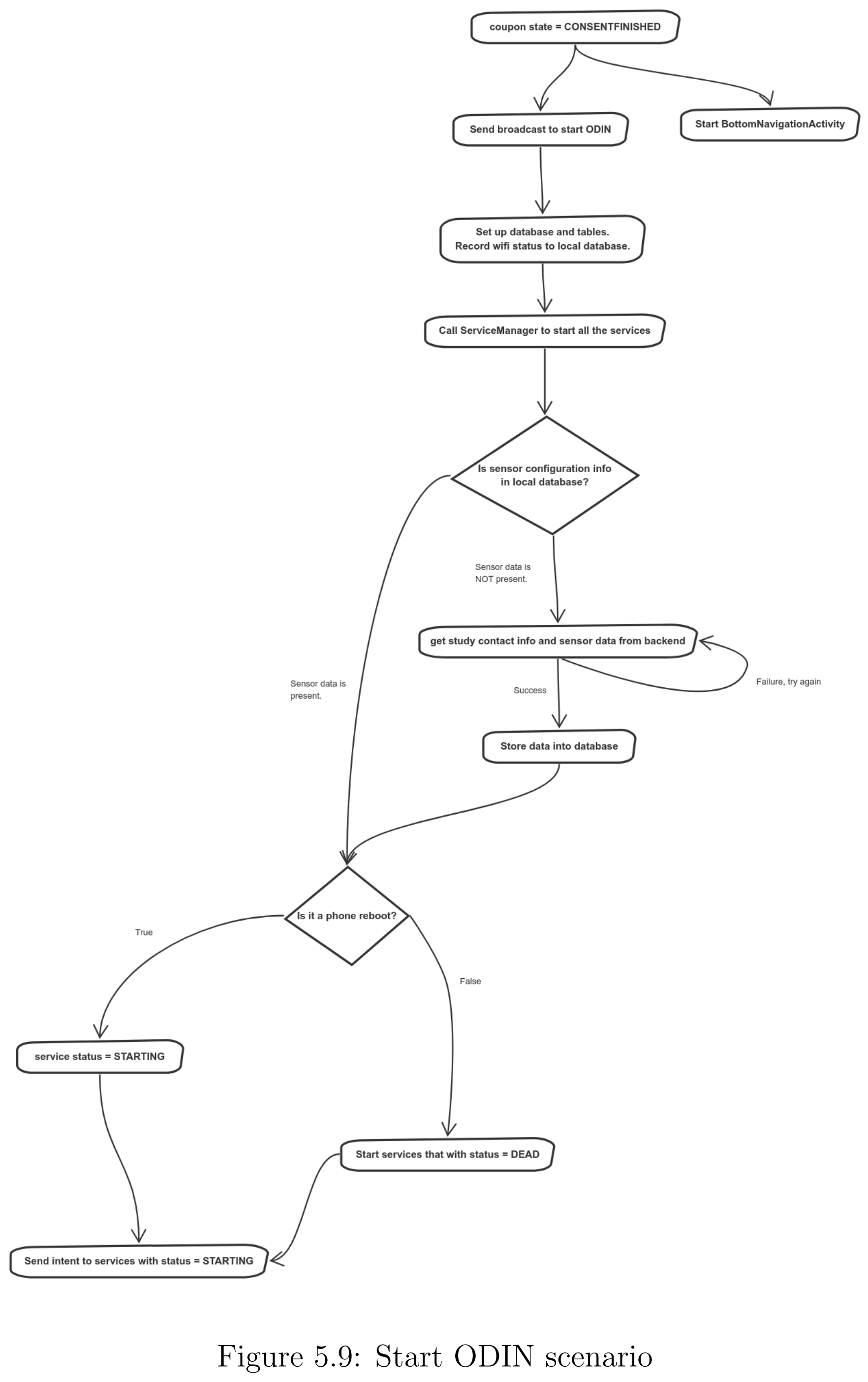

coupon state = CONSENTFINISHED
Send broadcast to start ODIN
Start BottomNavigationActivity
Set up database and tables.
Record wifi status to local database.
Call ServiceManager to start all the services
Is sensor configuration info in local database?
Sensor data is NOT present.
get study contact info and sensor data from backend
Failure, try again
Success
Sensor data is present.
Store data into database
Is it a phone reboot?
True
False
service status = STARTING
Start services that with status = DEAD
Send intent to services with status = STARTING

Figure 5.9: Start ODIN scenario

When the coupon status has been set to CONSENTFINISHED, the services can start running in the ODIN application. This process is illustrated in Fig 5.9 and described in this section.

The participant is redirected to the main activity (BottomNavigationActivity). We send a broadcast to start ODIN in the KeepAliveOneShotReceiver class that sends a start intent. Then, the database is set up, and the wifi status is recorded to the local database. The same class receives the intent and uses the ServiceManager class to start the services. It then checks if the sensor's information is in the local database.

- Sensor data is present: use the enum class to fetch all the enabled sensors.
- Sensor data is not present: execute REST call to retrieve the sensor data. The necessary application data is stored in the local database, and the enabled sensors are added.

Afterward, there are two possible scenarios:

- Phone reboot: all the services statuses are set to STARTING.
- Not a phone reboot: check all the services statuses and set them to DEAD if the last heartbeat is too old. If the service is DEAD, we need to create a new launcher, and then the status can be updated to STARTING.

Finally, send an intent to all the services enabled with the STARTING state.

5.2.3.5 Start services

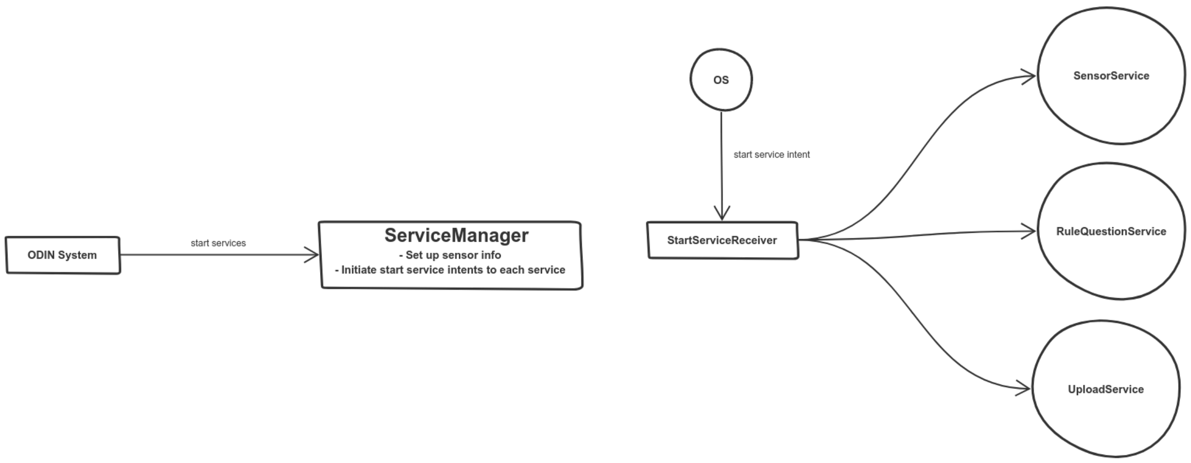


Figure 5.10: Start services scenario

Up to this point, the services are in the STARTING state, but they have not yet started actually running. To accomplish this, the ODIN System class uses the ServiceManager to start the services, as can be seen in Fig 5.10. Then, the service manager sets up the sensor information and iterates over each service and initiates an intent. The OS delivers the start service intent to the StartServiceReceiver class, which notifies each service to start.

5.2.3.6 Construction of service

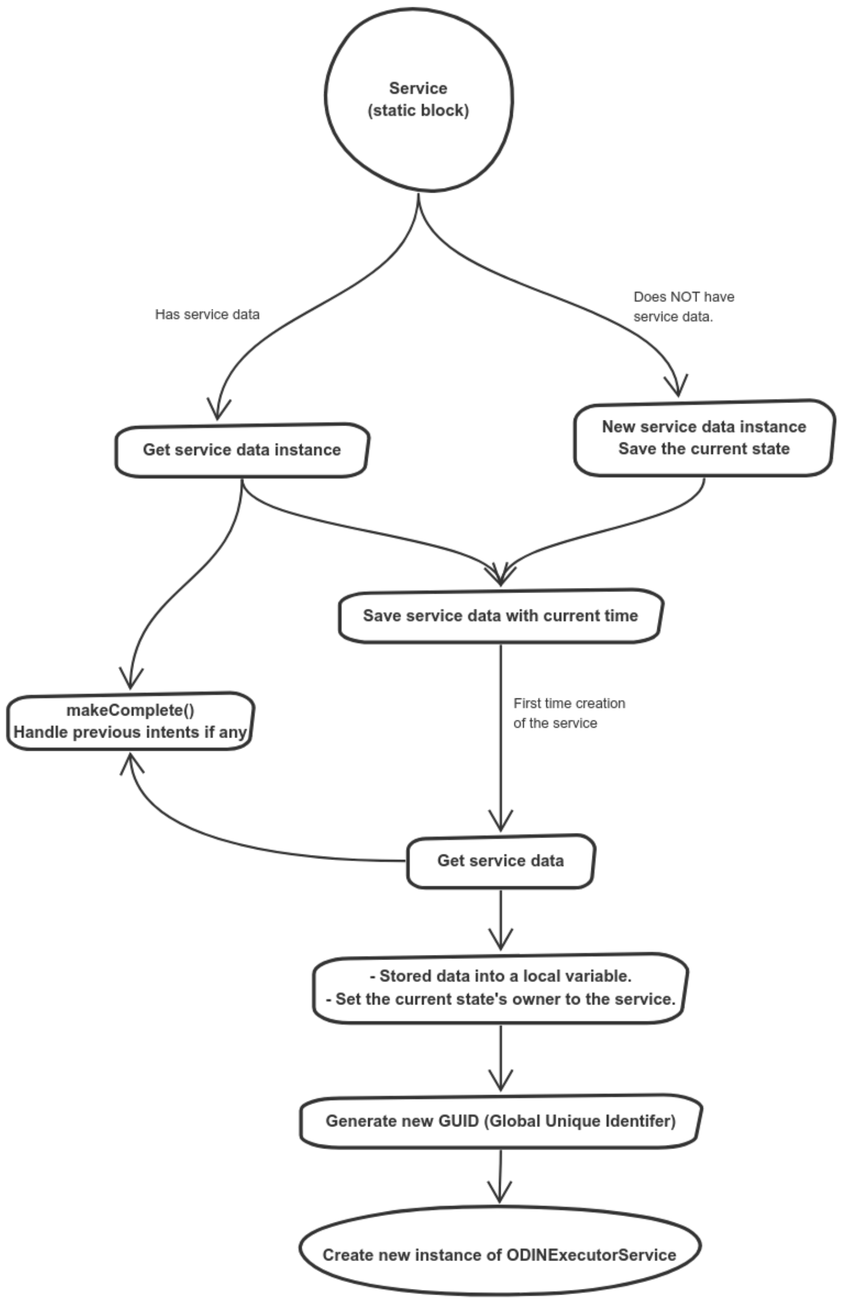

Service
(static block)
Has service data
Does NOT have service data.
Get service data instance
New service data instance
Save the current state
Save service data with current time
makeComplete()
Handle previous intents if any
First time creation of the service
Get service data
- Stored data into a local variable.
- Set the current state's owner to the service.
Generate new GUID (Global Unique Identifer)
Create new instance of ODINExecutorService

Figure 5.11: Construction of services scenario

This Section covers the construction of service, as can be seen in 5.11. In this scenario, every time we get the service data, the method makeComplete is called, which is used to handle previous intents. Each Service class has a static block that runs at the beginning of the application. In this block, we check if the Service has service data. If it does, the service data instance is retrieved. If it does not, a new instance of service data is created and this is saved to the ServiceData table within the APK's local database at the current time. If it is the first time that this service is being created, then the AbstractStatefulService class constructor is executed. We get the service data from the local database and store it in a local variable. Finally, we set the current states owner, generate a new GUID, and create a new instance of ODINExecutorService, which is the class that manages all the service jobs.

5.2.3.7 AbstractStatefulService

Fig 5.12 represents the standard workflow among all the services. Each service extends the AbstractStatefulService class, which records the heartbeat, flushes the database, and the ODINExecutorService is in that class. When a service begins its execution, the Android framework calls the service method onStartCommand, which creates a worker thread that calls onHandleIntent in AbstractStatefulService. This method checks for the action code of the intent received. There are four possible action codes that a service receives in an intent:

- ACTION_START_SERVICE: call handleStart method and flush all the necessary jobs in the queue. Then, call the current service states handleStart which calls enteringFrom. We create a new pending intent with its corresponding action code and schedule an alarm. Finally, all the information about the alarm is stored in the ServiceData table. Then we wait for the alarm to trigger and

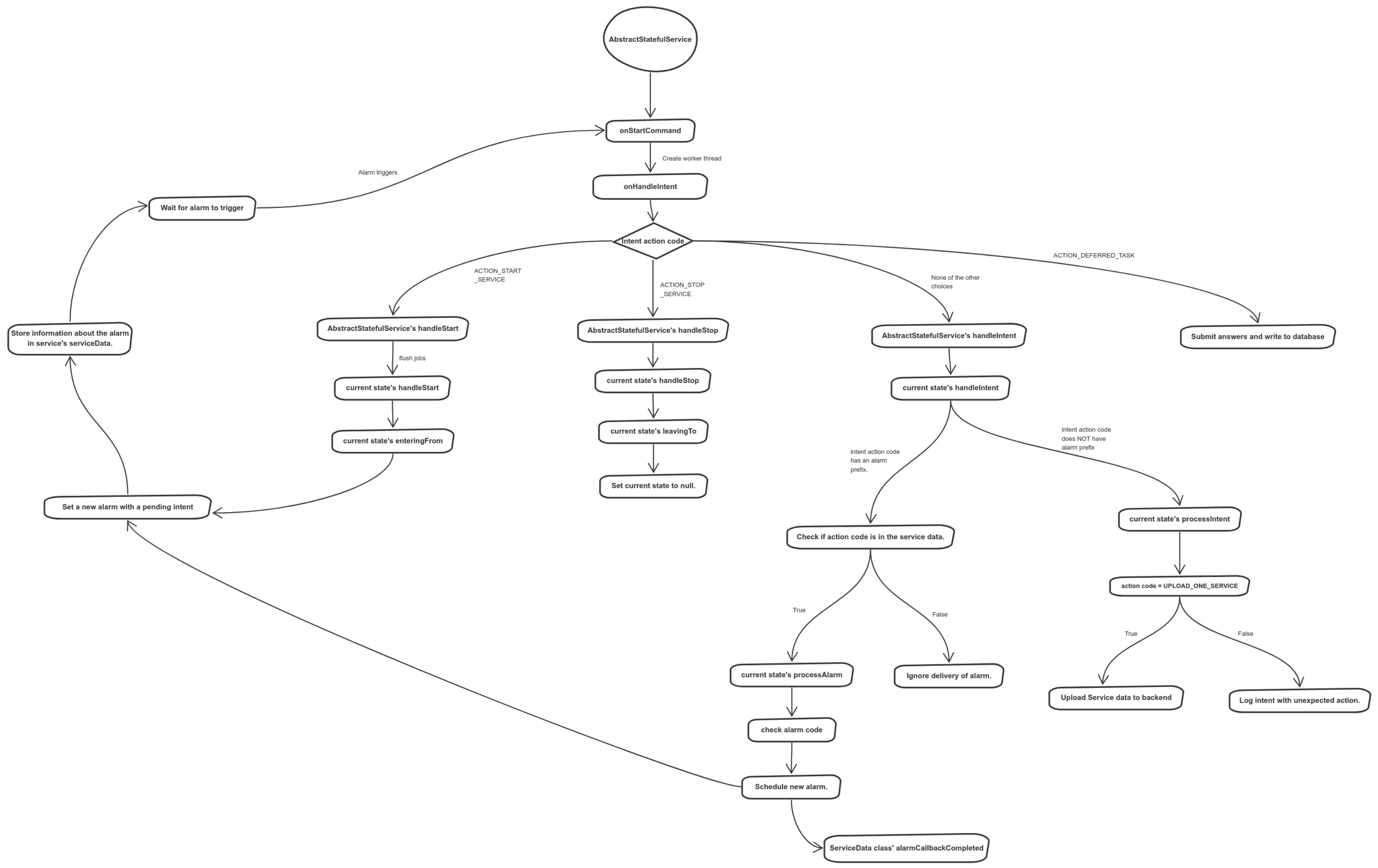

AbstractStatefulService
onStartCommand
Create worker thread
onHandleIntent
Intent action code
ACTION_START _SERVICE
ACTION_STOP _SERVICE
None of the other choices
ACTION_DEFERRED_TASK
AbstractStatefulService's handleStart
flush jobs
current state's handleStart
current state's enteringFrom
AbstractStatefulService's handleStop
current state's handleStop
current state's leavingTo
Set current state to null.
AbstractStatefulService's handleIntent
current state's handleIntent
intent action code has an alarm prefix.
intent action code does NOT have alarm prefix
Submit answers and write to database
Check if action code is in the service data.
True
False
current state's processAlarm
Ignore delivery of alarm.
check alarm code
Schedule new alarm.
ServiceData class' alarmCallbackCompleted
current state's processIntent
action code = UPLOAD_ONE_SERVICE
True
False
Upload Service data to backend
Log intent with unexpected action.
Set a new alarm with a pending intent
Store information about the alarm in service's serviceData.
Wait for alarm to trigger
Alarm triggers

Figure 5.12: AbstractStatefulService scenario

go back to onStartCommand.

- ACTION_STOP_SERVICE: call handleStop in the abstract class and the service class, then call leavingTo and set the current state to null.
- ACTION_DEFERRED_TASK: this task is used for submitting answers and writing into the database.
- If the action is none of the above then, we call handleIntent abstract method and service method. Check the intent action code if it has an alarm prefix:
  - If it does, check if the action code is in the service data.
  - If it is not there, ignore the alarm. If it is there, call processAlarm and check the alarm code. Finally, a new alarm is scheduled and follows the same workflow as ACTION_START_SERVICE after creating a new intent.

5.2.3.8 Phone power on/off

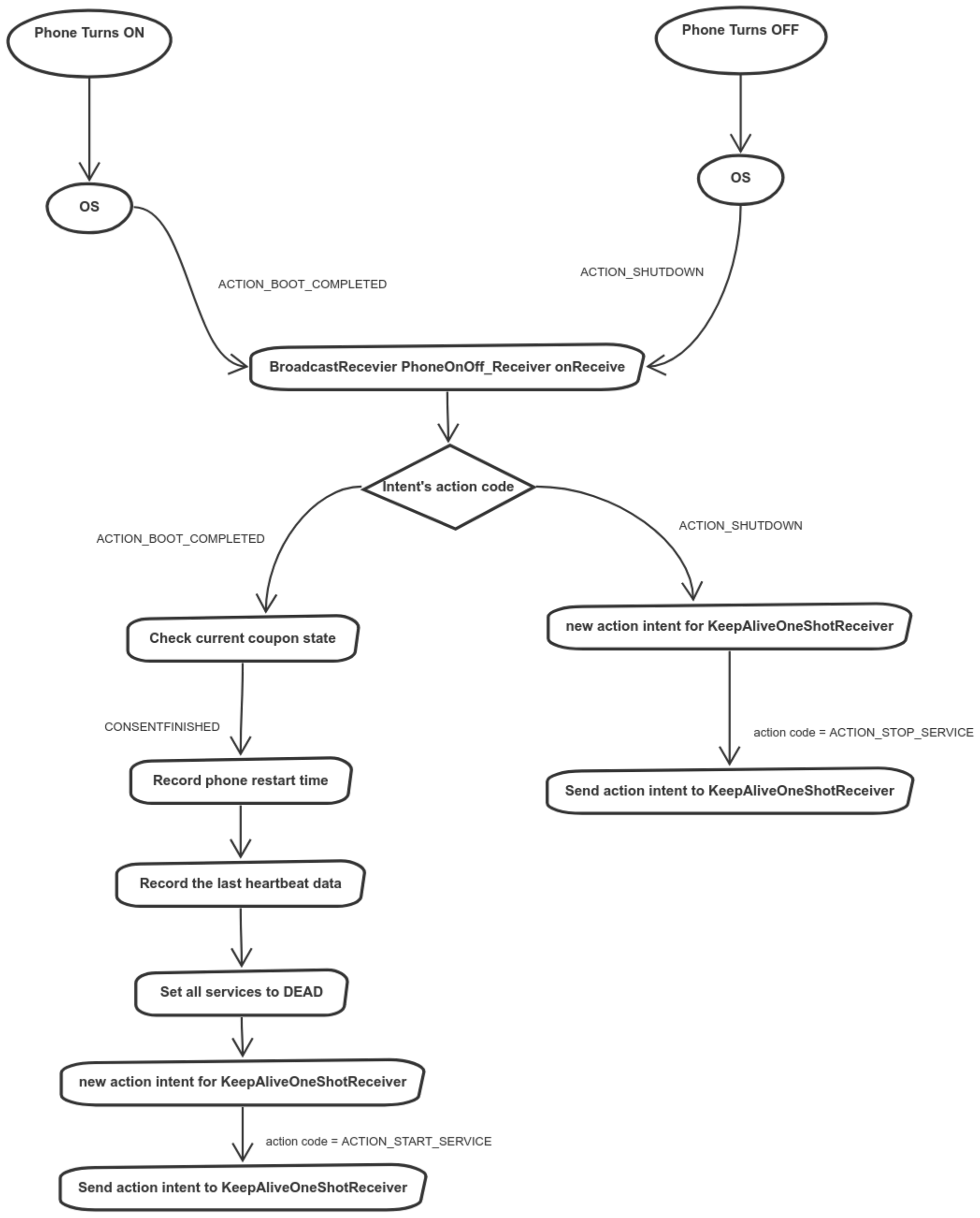

Phone Turns ON
Phone Turns OFF
OS
OS
ACTION_BOOT_COMPLETED
ACTION_SHUTDOWN
BroadcastRecevier PhoneOnOff_Receiver onReceive
Intent's action code
ACTION_BOOT_COMPLETED
ACTION_SHUTDOWN
Check current coupon state
new action intent for KeepAliveOneShotReceiver
CONSENTFINISHED
action code = ACTION_STOP_SERVICE
Record phone restart time
Send action intent to KeepAliveOneShotReceiver
Record the last heartbeat data
Set all services to DEAD
new action intent for KeepAliveOneShotReceiver
action code = ACTION_START_SERVICE
Send action intent to KeepAliveOneShotReceiver

Figure 5.13: Phone power on/off scenario

Two possible intents can initiate this scenario, as illustrated in Fig 5.13. In both cases, the OS sends an intent to BroadcastReceiver PhoneOnOffReceiver, and onReceive is called, then we get the action code from the intent.

- Phone turns on (ACTION_BOOT_COMPLETED): check the coupons state. If it is CONSENTFINISHED, record the phones restart time in the local database and the last heartbeat. Set all of the services to DEAD. Create a new intent with action ACTION_START_SERVICE and broadcast it to all the services using KeepAliveOneShotReceiver.
- Phone turns off (ACTION_SHUTDOWN): Create new intent with action ACTION_STOP_SERVICE and broadcast it by using KeepAliveOneShotReceiver.

5.2.3.9 Service is killed

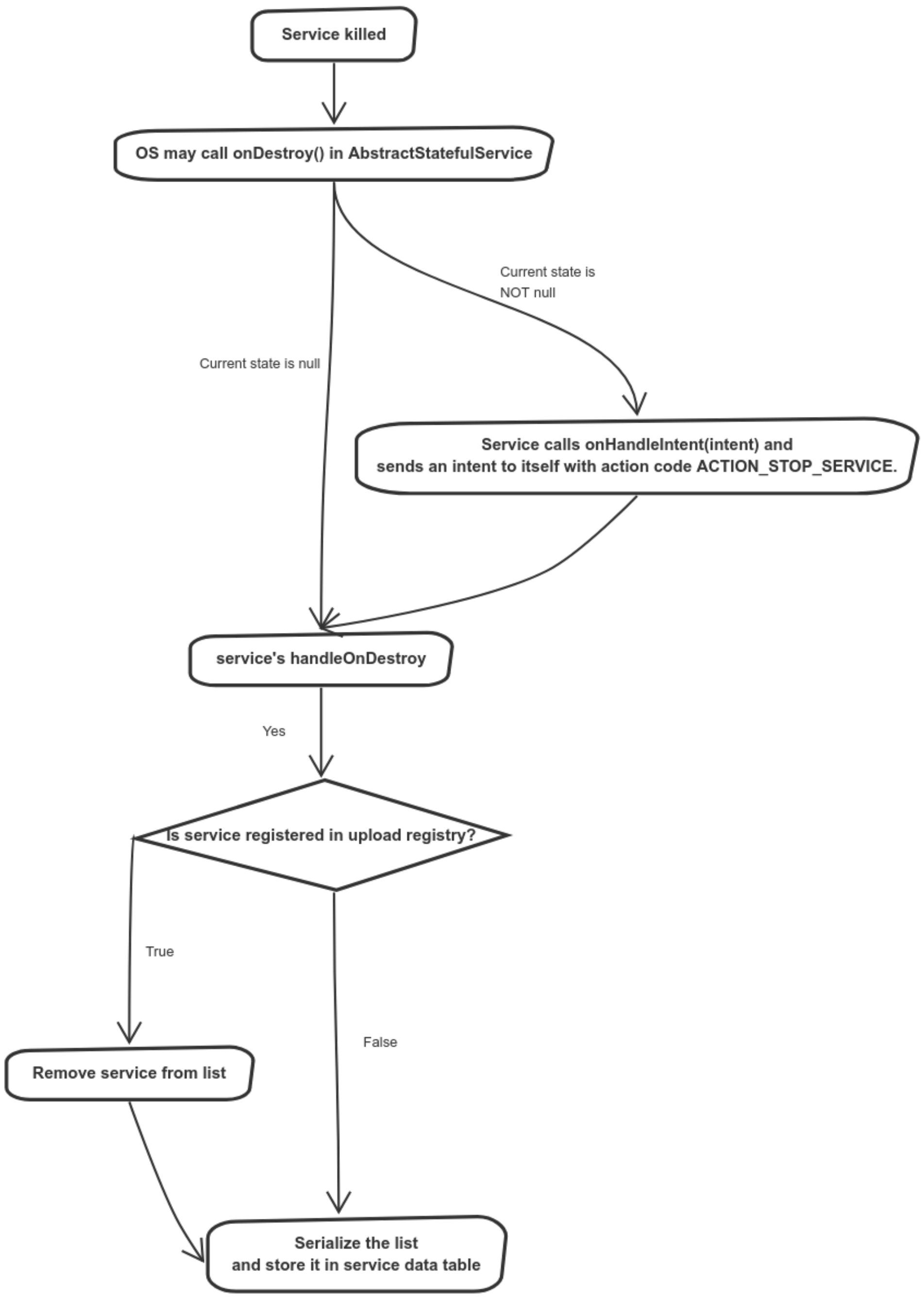

Service killed
OS may call onDestroy() in AbstractStatefulService
Current state is NOT null
Current state is null
Service calls onHandleIntent(intent) and sends an intent to itself with action code ACTION_STOP_SERVICE.
service's handleOnDestroy
Yes
Is service registered in upload registry?
True
False
Remove service from list
Serialize the list and store it in service data table

Figure 5.14: Service killed scenario

The services running in the background may be killed at any time by the OS or by the participant. We present the workflow which occurs when a service is killed (Fig 5.14. Most of the time when this occurs, the OS calls onDestroy, but it is not guaranteed. We check the value of the current state. If it is null, skip until the services onDestroy method is called. If it is not null, then the service calls onHandleIntent method and sends the intent ACTION_STOP_SERVICE. After checking for the action code in the handle intent method, the method handleStop is called, which sets the current state to null. Afterwards, the services handleOnDestroy method is called. Each service has its own implementation of handleOnDestroy. All the services unregister from the upload registry except for UploadService that does not perform any task. Each service checks if its service name is registered to the upload registry. If it is registered, we (1) remove it from the list, (2) serialize it and (3) store the serialized form in the service data table. If it is not registered, only steps (2) and (3) are executed.

5.2.3.10 User removes task

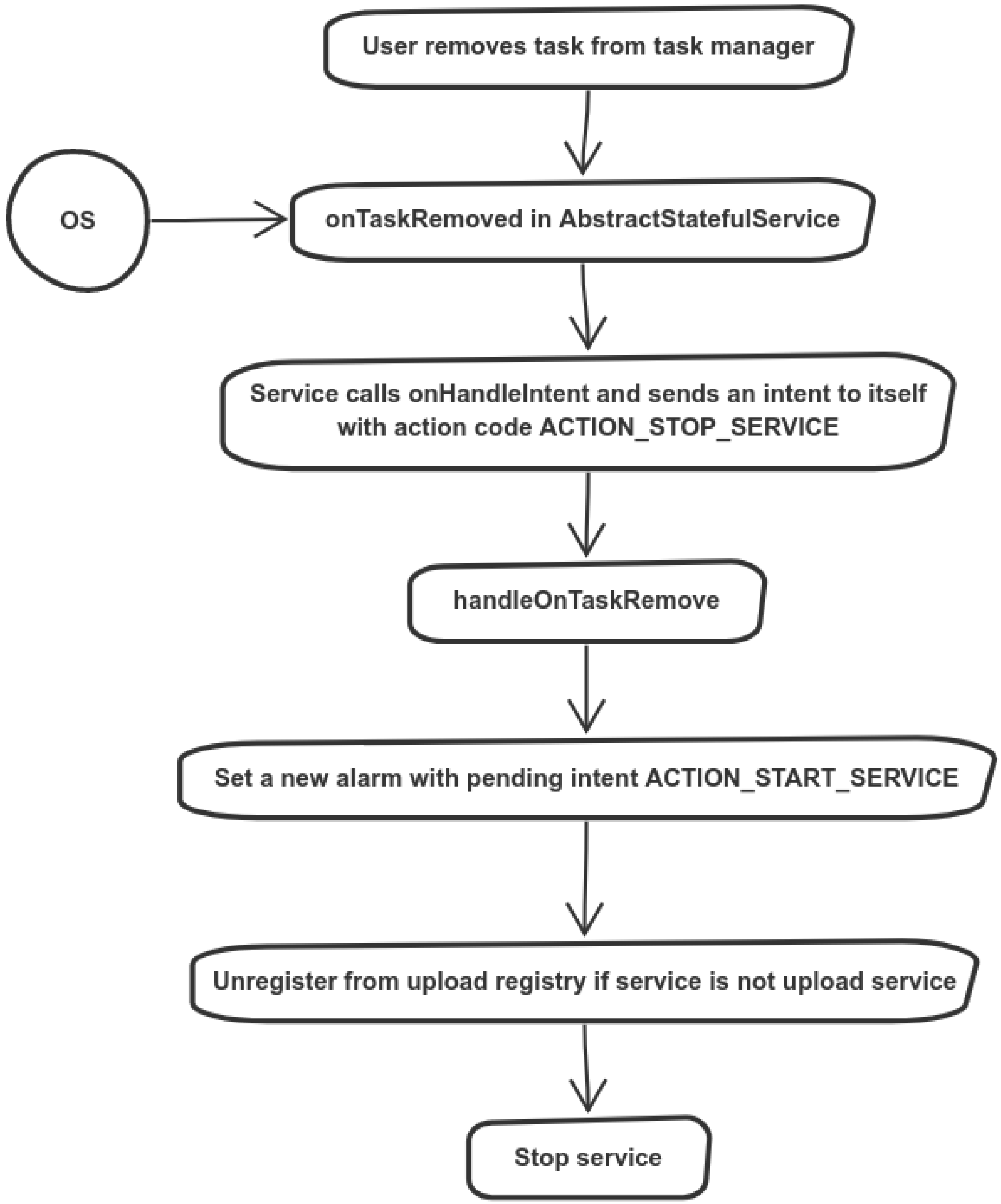

User removes task from task manager
OS
onTaskRemoved in AbstractStatefulService
Service calls onHandleIntent and sends an intent to itself with action code ACTION_STOP_SERVICE
handleOnTaskRemove
Set a new alarm with pending intent ACTION_START_SERVICE
Unregister from upload registry if service is not upload service
Stop service

Figure 5.15: Start services scenario

This scenario is initiated when the user removes the app task from the task manager (Fig 5.15. The OS calls onTaskRemove method in AbstractStatefulService. In this method, the service calls onHandleIntent and sends the intent ACTION_STOP_SERVICE. Then, the service calls handleOnTaskRemove, which creates a new pending intent to start the service and sets a new alarm with the pending intent. Finally, we call stopSelf method to stop the service. The workflow continues in Section 5.2.3.9.

5.2.3.11 Backend push notification

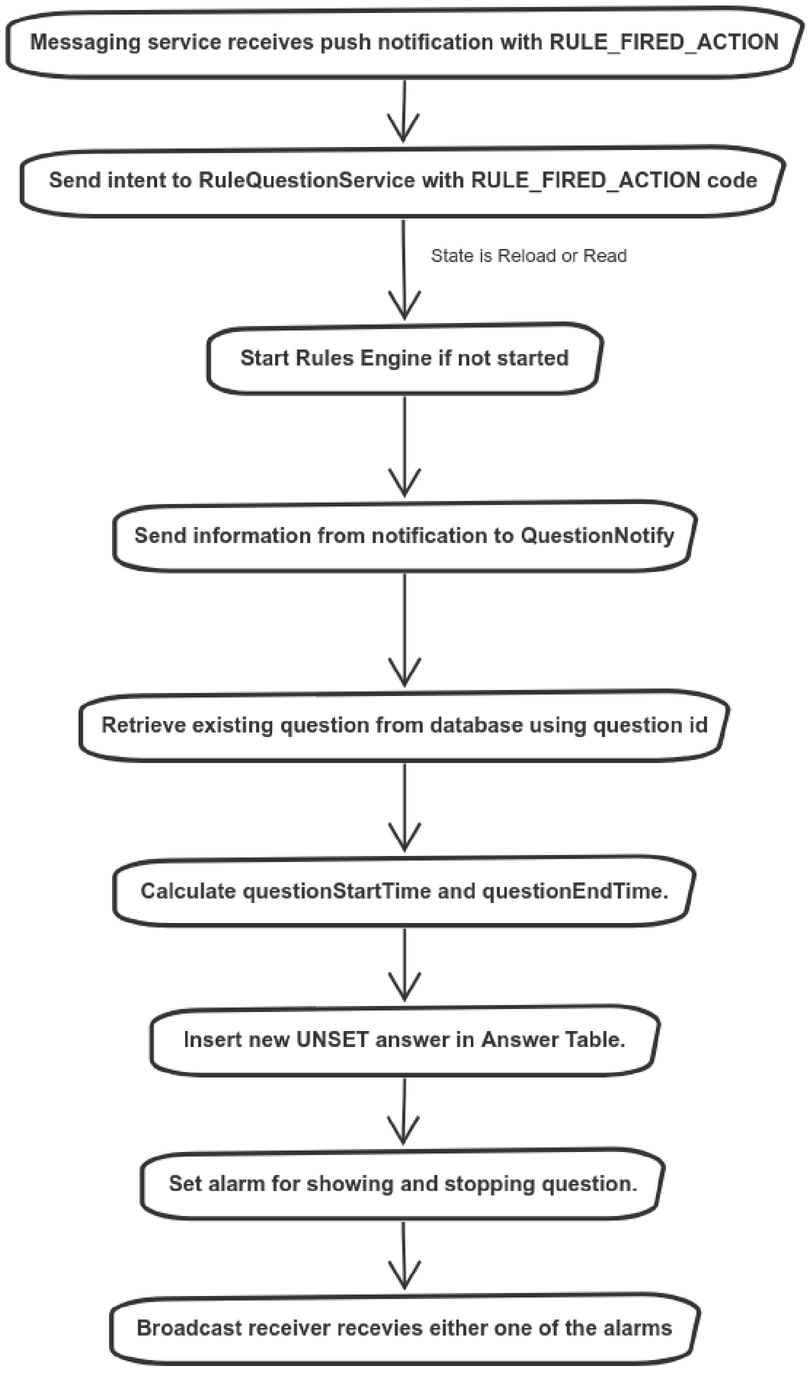

Messaging service receives push notification with RULE_FIRED_ACTION
Send intent to RuleQuestionService with RULE_FIRED_ACTION code
State is Reload or Read
Start Rules Engine if not started
Send information from notification to QuestionNotify
Retrieve existing question from database using question id
Calculate questionStartTime and questionEndTime.
Insert new UNSET answer in Answer Table.
Set alarm for showing and stopping question.
Broadcast receiver recevies either one of the alarms

Figure 5.16: Backend push notification scenario

The backend notifies the phone when a rule has fired by sending a push notification. If a rule fires in the backend, the messaging service on the phone receives a notification with RULE_FIRED_ACTION which sends a new intent to RuleQuestionService with the same action code as can be seen in Fig 5.16. If the current state is ready or reload, then we initialize the rules engine if it is not already running. The information is retrieved from the notification and sent to the QuestionNotify, then retrieve the current question from the database using the questionid. We expect the question to be present in the local database. Afterward, we calculate the question start time and the question end time, a new answer is inserted into the Answer table with the value UNSET in the local database, and two alarms are scheduled based on the questions start and end time. The first alarm starts showing the question, and the second alarm stops showing the question. These two alarms are received in the ShowQuestionBroadcastReceiver and then handled accordingly.

5.2.3.12 Consent states

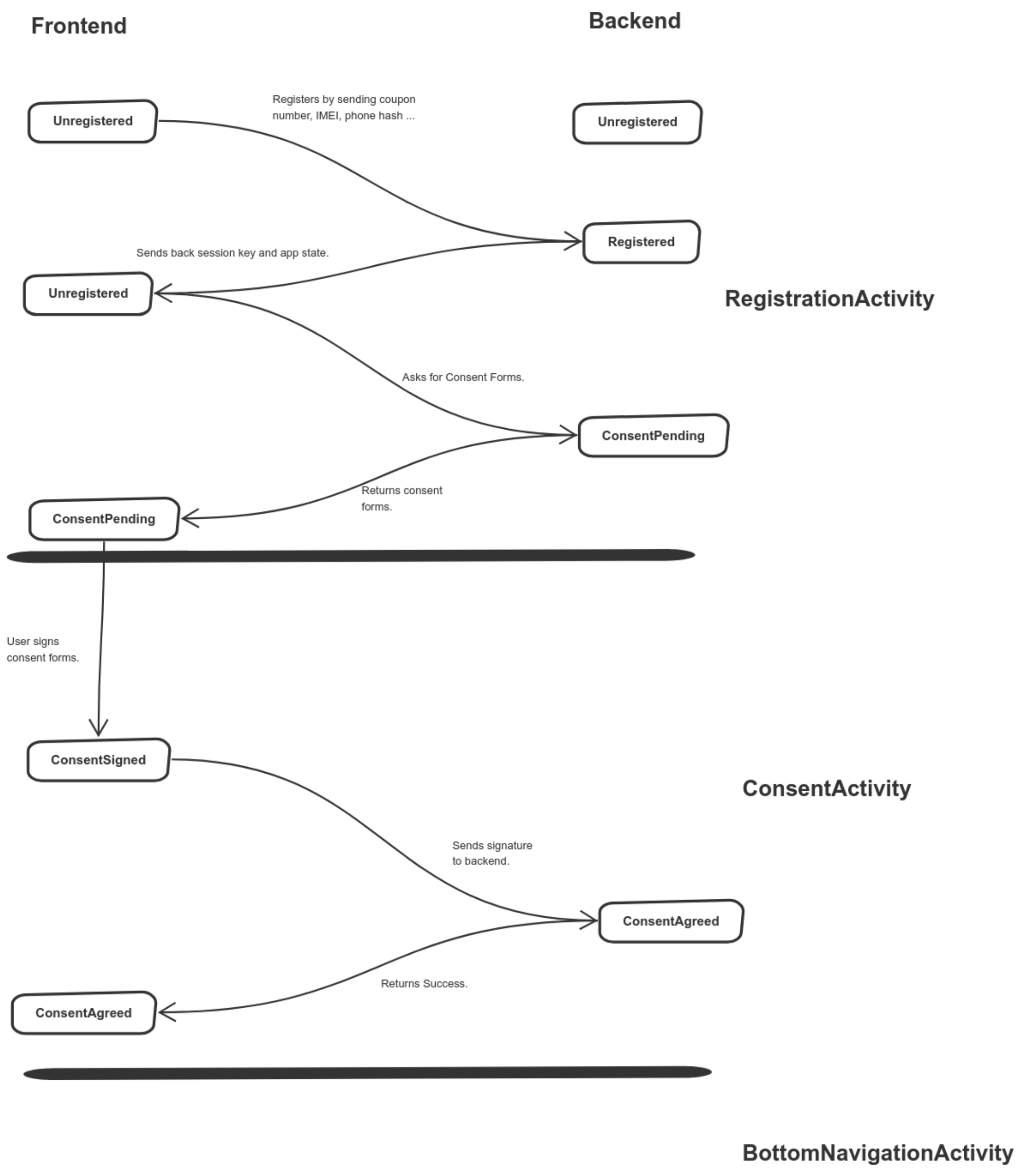


Figure 5.17: Frontend and backend consent states

Figure 5.17 shows the sequence of coupon states in the phone and the backend throughout the participation of a study. The initial state is both phone and backend, is unregistered. When the user registers, the backend state is moved to registered while the phone still considers itself unregistered. The state stays as unregistered because the phone still needs to receive the consent form information to go to the next screen. The phone asks for the consent forms, and the backend state is moved to consentPending, and when the phone receives the forms from the server, it changes its state to consentPending. The participant goes through the consent and agrees to the terms and conditions, and then the phone state is moved to consentSigned. The phone sends the participants consent signature to the backend, and the backend changes its state to consentAgreed. Finally, when the phone receives a successful response from the backend, the phone too changes its state to consentAgreed.

5.2.4 Services and Sensors

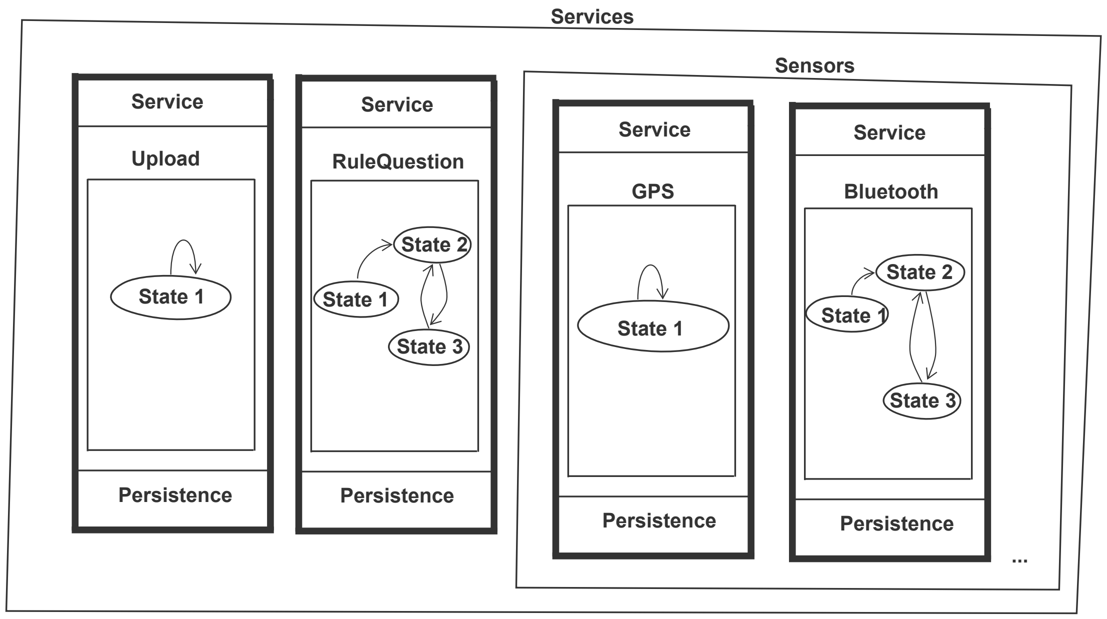


Figure 5.18: Example of the structure of ODIN Services in Android

This Section describes the different services in the Android application. Previously, we have mentioned that each sensor has a dedicated service within the phone app. Fig 5.18 shows an overview of the design of the services; each service has an FSM schema, and the sensors are also represented as services. These are the default services (UploadService and RuleQuestionService), which exist in every study and other services that are only present in the studies that have the corresponding sensors enabled. Next, we describe each of the services in detail:

5.2.4.1 Upload Service

The upload service is used to schedule periodic data uploads to the server. Fig 5.19 shows the diagram of the upload service. It consists of only one state (*PeriodicUpload*). All the other services like GPS, and Bluetooth register to this service to upload the sensor data and the RuleQuestion service registers to send the answers to the server. Then, the UploadService schedules alarms for each service to upload the data, and calls upon the registered services when it is time to upload, it retrieves the data from the database and initiates a POST request to the PhoneAppService with the data. Finally, the REST call returns a response to the request and whether the request was successful or not; the service schedules a new alarm to upload more data later.

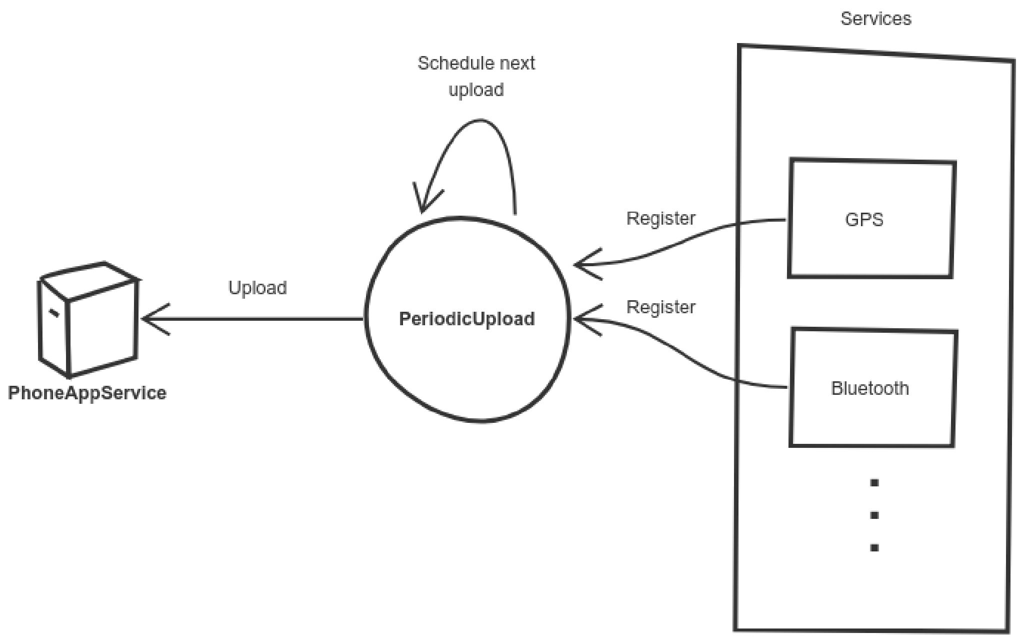


Figure 5.19: Upload Service diagram. The left side shows how it uploads the data to the server and the right side represents the other services registering to the Upload-Service.

5.2.4.2 Rule Question Service

The Rule Question service handles showing the questions to the user when there is a rule firing and makes sure that the Rules Engine is always running. Fig 5.20 represents the sequence of events in the Rule Question service. The RulesEngine starts when the RuleQuestion service is first created. The *Initial* state initiates a GET request with the PhoneAppService to get the questions when the user first registers. Then, if the request fails, we schedule a new alarm to make the REST call again later. If the request succeeds, then we move to the *Ready* state. In that state, we handle the rule firings and schedule a reload alarm. When there is a new reload alarm, it moves to the *Reload* state, which makes a GET request to get the new questions (recall the questions can be disabled and new questions can be added while a study is in progress). The reload happens to keep the questions and rules up to date in case the researcher has created new questions or previous disabled ones. If the request is successful, the service moves back to the *Ready* state, and if it fails, it schedules an alarm to retry the upload. Finally, if there is a rule firing that happens during the *Upload* state, it is also properly handled.

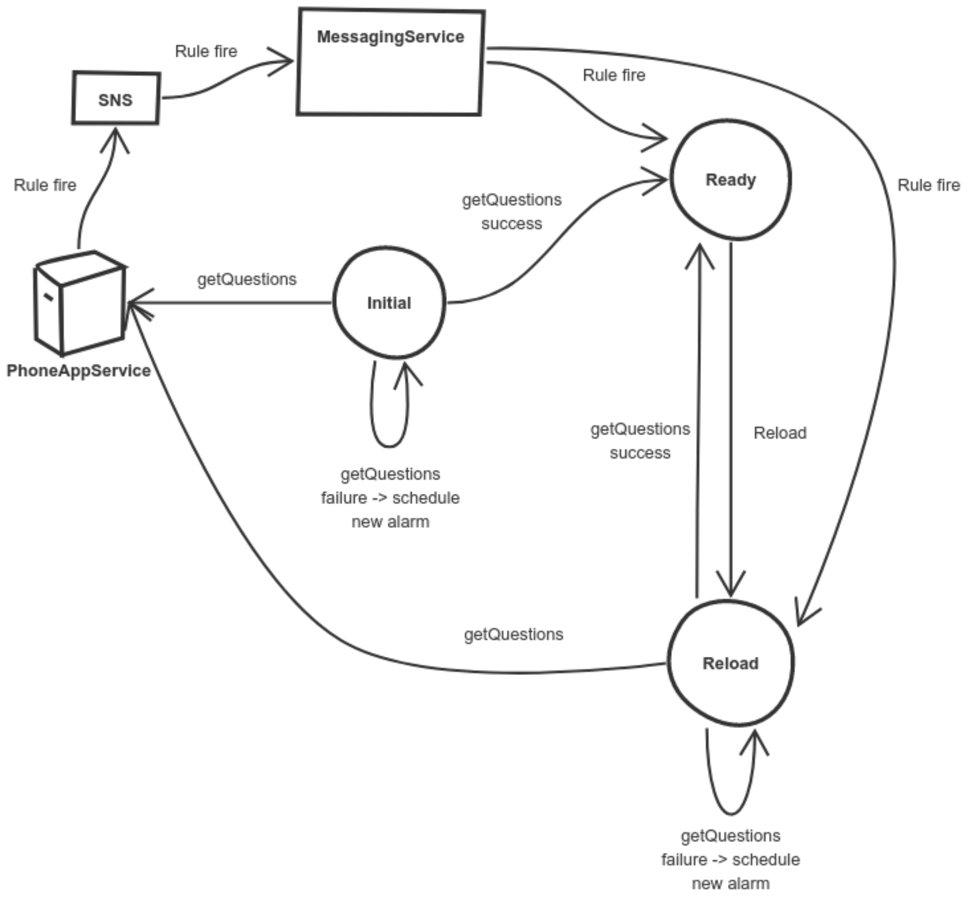


Figure 5.20: Rule Question service diagram. It shows the different states (Initial, Ready, and Reload) and classes involved in the Rule Question service. It displays how the service retrieves the questions from the server, and the server notifies the application through SNS when there is a rule firing.

5.2.4.3 GPS

The GPS sensor records the geographic history data. The GPS Service is used to track the latest participant's location periodically. Fig 5.21 shows the diagram of the GPS service. We can observe that there is only one state (*PeriodicGetLocation*), which periodically gets the location from the GPSDataCollector class, which keeps track of the latest latitude and longitude coordinates. After retrieving the location, it inserts it as a new row in the GPS database. Finally, it schedules a new alarm to get the location.

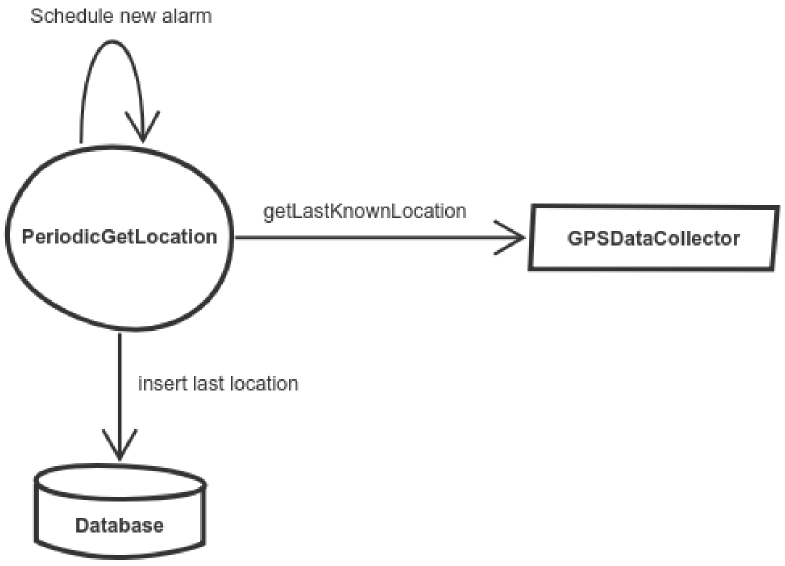


Figure 5.21: GPS Service diagram. The left side shows how the only state of the service (PeriodicGetLocation) sends data to the database and retrieves the location from the GPSDataCollector.

5.2.4.4 Bluetooth

The Bluetooth service's purpose is to find other devices nearby that are also enrolled in the ODIN study and running the ODIN app. To accomplish it, we have designed a Bluetooth service shown in Fig 5.22. This service begins in the *Initial* state in which the Bluetooth is enabled to start discovering, and the Bluetooth name is replaced by a new unique name so that other devices can identify it. Afterward, the service moves to the *Discovering* state. The Bluetooth Broadcast Receiver class notifies this state if the discovery has finished or a new device is found. In the latter, the devices found are stored in memory, and a new alarm is scheduled to start discovering again. The service moves back to the *Initial* state if the Bluetooth name is incorrect or the discovery has completed. However, if there is a timeout or if the Bluetooth has been disabled, it transitions to the *CleanUp* state. Before leaving the *Discovering* state, it writes to the database the devices found. The *CleanUp* state takes care of resetting the Bluetooth name to the original and the state (enabled or disabled). After the reset, it moves to the *Initial* state.

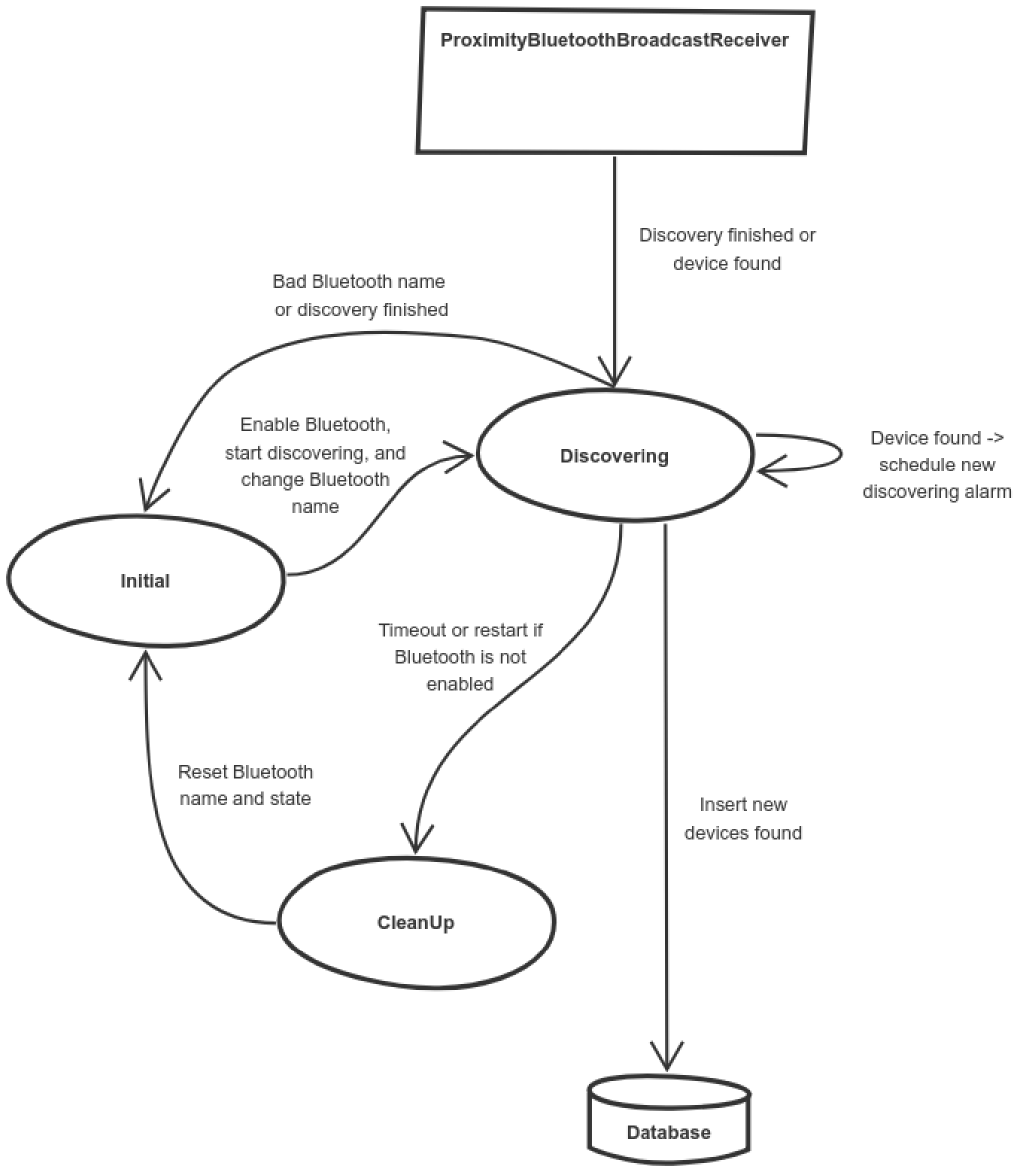

ProximityBluetoothBroadcastReceiver
Discovery finished or device found
Bad Bluetooth name or discovery finished
Enable Bluetooth, start discovering, and change Bluetooth name
Discovering
Device found -> schedule new discovering alarm
Initial
Timeout or restart if Bluetooth is not enabled
Reset Bluetooth name and state
Insert new devices found
CleanUp
Database

Figure 5.22: Bluetooth Service diagram. It contains three states; Initial, Discovering, and CleanUp. The ProximityBluetoothBroadcastReceiver class notifies the Discovering state when there is an update.

5.2.4.5 Beacon

The Beacon service scans and finds Beacon devices already included in the ODIN study. Fig 5.23 presents the diagram of the Beacon service. We can observe it contains only one state. In the *Initial* state, the Bluetooth is enabled if needed, and scanning begins. Afterward, two alarms are scheduled to restart the Bluetooth scan, stop the scan, and write the devices found to the database.

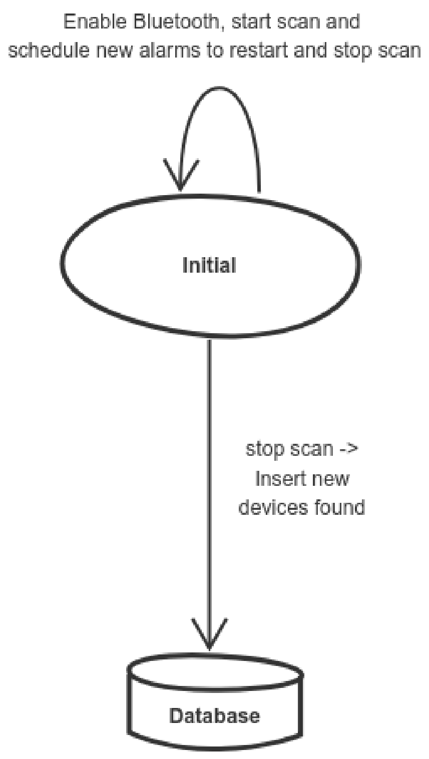

Enable Bluetooth, start scan and
schedule new alarms to restart and stop scan
Initial
stop scan ->
Insert new
devices found
Database

Figure 5.23: Beacon service diagram. It contains only the Initial state which writes to the database.

5.2.4.6 Activity Recognition

The Activity Recognition service is used to obtain the latest participant's activity. Fig 5.24 shows the diagram of the Activity Recognition service which contains only one state: *ActivityRecognitionPeriodic*. We can observe that there are two alarms, one to update the latest activity recognition and another alarm to insert the latest activity recognition into the database. This state periodically updates and inserts the activity recognition by scheduling the alarms every time this is done.

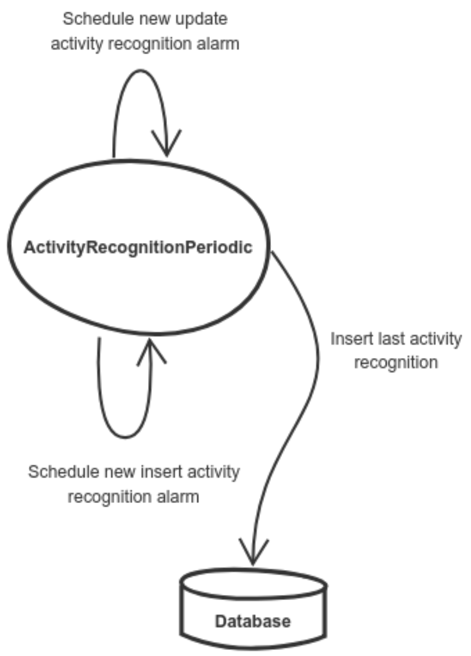

Schedule new update activity recognition alarm
ActivityRecognitionPeriodic
Insert last activity recognition
Schedule new insert activity recognition alarm
Database

Figure 5.24: Activity Recognition service diagram. It contains only the ActivityRecognitionPeriodic state which writes to the database.

5.2.4.7 Empatica E4

The Empatica E4 is a wristband that tracks physiological signals in real-time. The E4 service only takes care of recording the E4 data periodically since it is assumed the

E4 is connected previously by the user. The participant is notified of the unpaired sensors after registration. If the E4 is not connected, the user follows instructions on the application to connect it. Then, the E4 service only records the E4 data and writes it into the database if the device is connected to ODIN, and the participant is wearing it on the wrist. These actions happen in a single state *EmpaticaE4*. The service continuously schedules a new alarm to periodically go through this process of recording E4 data and writing it into the database.

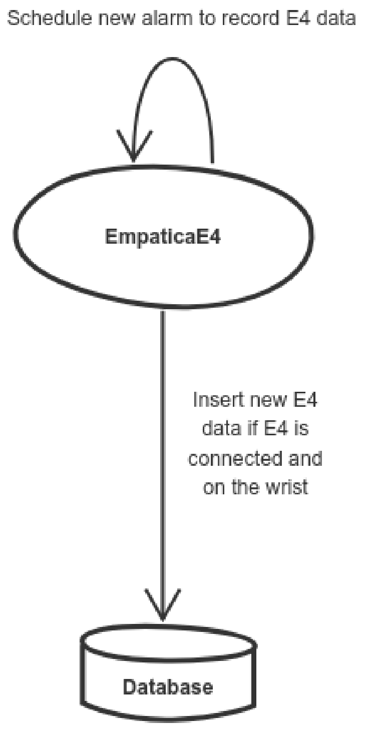

Schedule new alarm to record E4 data
EmpaticaE4
Insert new E4 data if E4 is connected and on the wrist
Database

Figure 5.25: Empatica E4 service diagram. It contains only the EmpaticaE4 state which writes to the database.

Chapter 6

Web

The Internet was first used only in academics and research, and it was unknown outside of those areas [1]. It was not until the 1990s that the World Wide Web [73] was invented. When that happened, the Web became very popular due to the many features that it provides, such as on-demand, easy to publish content, applications like YouTube, Facebook, etc. [1]. We have access to all these features thanks to the HyperText Transfer Protocol (HTTP), which is part of the application-layer protocol. HTTP/1.0 [74] was the first "sophisticated" protocol, and it was afterward improved by in HTTP/1.1 [75], which is the version that we use nowadays. We need a client and a server to be able to exchange messages over HTTP.

In our application, the client represents the two web user interfaces; AdminUI and ResearcherUI. The server is the backend service, in this case, ResearcherService. They talk to each other by exchanging messages over HTTP. In this chapter, we focus mostly on the ResearcherUI since AdminUI is a straightforward application that the administrators use to add new researchers with only a couple of endpoints. However, both of them follow the same design.

6.1 Design and User experience

The ResearcherUI has been implemented in ASP.NET MVC. We can clearly see the Model View Controller (MVC) [76] structure in Fig 6.1. There is a REST layer that contains all the REST APIs to talk to the backend server. The user submits a form, the Model class of the form is sent to the Controller with its corresponding data. Then, the Controller makes a REST call (if necessary) to update or retrieve data. Other main files that we can observe in the diagram are the appsettings and the startup. The frontend is hosted in the same Ubuntu Virtual Machine than the backend services and runs on dotnet core service. The developer uses the appsettings file to specify the IP address of the backend. The startup is the class that contains the port number of the frontend and starts the application.

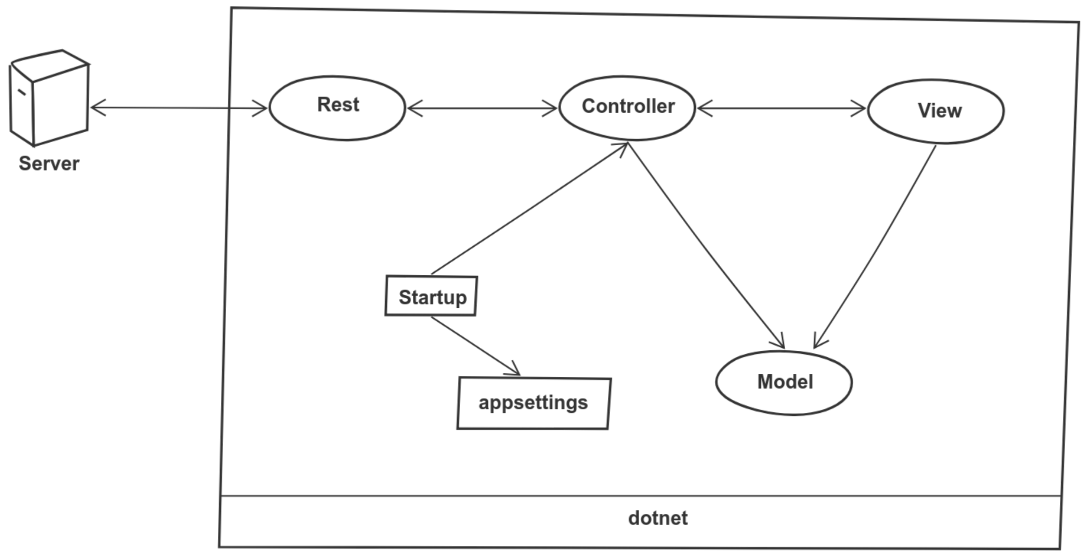


Figure 6.1: ODIN frontend MVC design and communication to the server via REST

Now that we have a high-level picture of the frontend's design, we describe all the steps that the researcher has to follow to create a study.

The first step is to Login to the application. The user must enter the username and password. There is the option to recover the password in case the user has forgotten. After logging in, if it is the first time, it redirects to a welcome page where the user can choose to create the first study. Otherwise, the user goes to the home page where he can see the list of leading and participating studies. The studies created by a researcher are its leading studies. While studies created by others that added a researcher as a member are participating studies. The leading researcher can choose the privileges of the members. When the researcher clicks on "create study" button, a pop up shows up where the user can specify the name of the study, duration, description, and select the sensors that would like to include in the study. A more detailed description of the sensors can be found in 4.2.3.3. After the study is created, the user can start editing it. Then, a new navigation menu appears at the top with the following fields: Info, Survey, Coupons, Participants, Consent Form, and Contact. Next, we describe each of the pages and the actions that the user can take in each one:

- **Info**: The user can edit the study name, description, duration, sensors, add members to the study, and move the study phase.
- **Survey**: This is the most important page because it is where the researcher creates the survey. The first step is to add a question; Multiple Choice or Fill Text. Some other fields can be specified if it is Multiple Choice questions such as randomization type for the responses, the number of choices to be selected, and adding special choices like "None of the above." The user enters a tag, which is a keyword that describes the question. The researcher can also enter a description to have more information about the question. The subject does not see either the tag nor the description. After the question is created, the user can add response options if it is a multiple-choice question. Finally, the user has

to add the rules to the existing questions. There can be many rules for a single question. The system displays different fields for a specific rule when the user selects a rule type. The user has the option to add filters to the rule. By adding filters, new conditions are being added to determine whether the rule has to fire or not. A better description of filters and rules can be found in Section 4.2.3.1.

- **Coupons**: Generate the coupons that are given to the subjects to register on the phone.
- **Participants**: The researcher can view the data of the subjects that have registered to the study. First, a coupon from the list has to be selected. Lastly, the user has to choose the data that they want to view: survey or sensors.
- **Consent**: The researcher can choose if the subject has to go through a consent form and sign it after registration. Also, the researcher can edit the text of the different sections of the default consent form text.
- **Contact**: Enter the contact information of researchers that the subject can reach out while in the study.

When the user has completed all these steps, the study can be moved to in-progress (which is done on the Info tab), so that subjects can register. Most of the study settings cannot be modified after it has been moved to in-progress. But the researcher can add new questions, rules, and disable (not delete) previous questions and rules. Also, the researcher can download data from the study on the Info page. The data is a list of CSV files with all the study tables. Finally, the researcher can move the study to a completed state when it has concluded. From that point onwards, the researcher can only view the data but cannot make any modifications.

6.2 Details

In this section, we give some background on RESTful services and explain the HTTP requests being sent over the network. Finally, we focus on a more detailed description of the structure of the system in MVC.

6.2.1 RESTful service using HTTP as a transport layer

Our application has a client-server architecture where the client is the ResearcherUI, and the server is the ResearcherService. Before going into detail on how they exchange messages over HTTP, we first need to know some necessary information about the Web.

A Web page is a document that contains a group of objects, such as files, images, etc. The URL contains the hostname and the path, which references the objects. In our application, the hostname would be the IP address of where the ResearcherService is located, and the path is the object that we want to access the server. HTTP allows clients to request objects from the servers and the servers can return the requested information to the client with an HTTP response as can be seen in Fig 6.2 [1]. HTTP uses TCP, which is explained in more detail in Section 6.3.

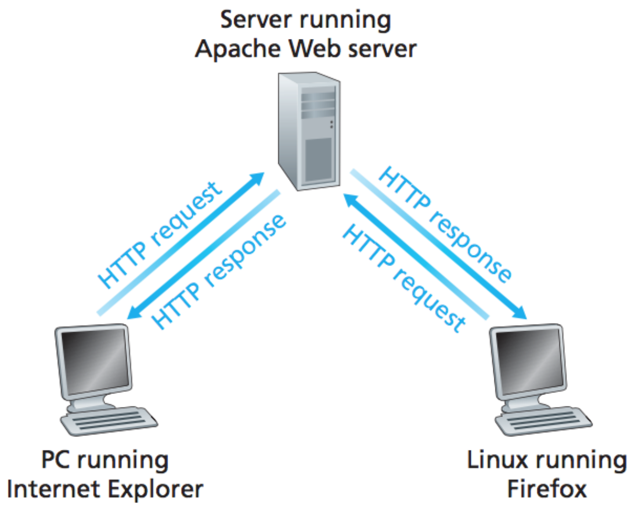


Figure 6.2: HTTP request-response behavior [1]

The way that client and server exchange information is the following; First, the client has to initiate a TCP connection with the server, then they have access to it through their socket interfaces. Hence, when the server or the client sends/receives a message, it goes through their socket interface. During the HTTP request/response, the server doesn't store any information about the client. For this reason, we say it is a stateless protocol (RESTful services) since they don't know anything about the client after the request is completed [1]

Different HTTP request methods are used to indicate the CRUD (create, read, update, and delete) function executed [77]. The methods are GET, POST, PUT, and DELETE. GET is used to retrieve information, PUT is used to insert and update stored data, POST is used to create a new object and store it in the database, and DELETE is used to delete data.

6.2.2 MVC

The Web UI follows an MVC paradigm [76]. We start describing the different Controllers in ResearcherUI, which can be seen in Fig 6.4. We can observe that we have the following: Account, Pages, Question, ResearcherHome, Study.

- **Account**: This Controller contains the Login, Logout, and other account-related actions like password reset.
- **Pages**: It contains actions of fundamental pages in the application such as the Help or Contact pages that redirect to some other website where we have the documentation.
- **Question**: Contains all the question and rule related operations like add, edit, delete, disable, etc.
- **ResearcherHome**: All the operations from the home page where the researcher sees the list of all the studies. Some examples of these are created, edit, view, or delete studies.
- **Study**: This class manages all the operations that the researcher can perform in the study, like generating coupons or modifying the consent form.

We observe that all the classes extend the ODINController class, which handles all the JSON responses from the backend. It checks whether the operation was successful or failed. In case of failure, the system shows an error to the user.

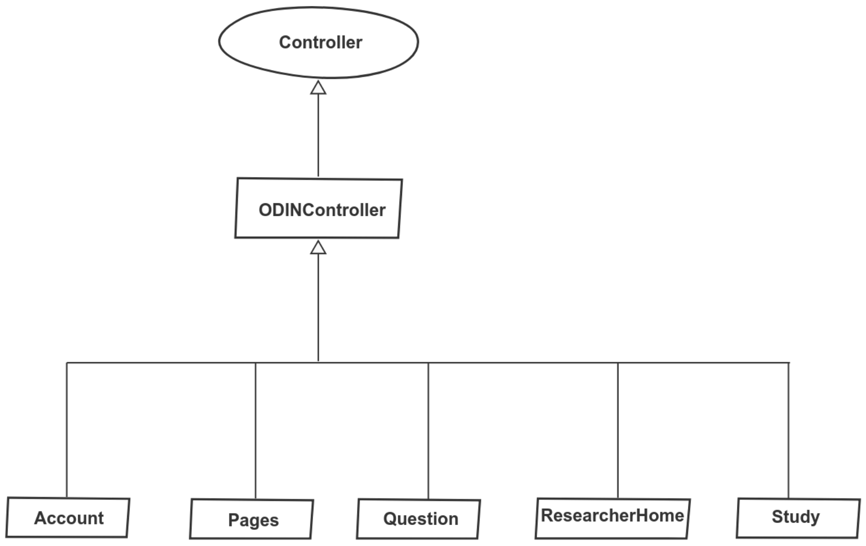


Figure 6.3: ResearcherUI Controllers design

All the information received in the Controller from the backend is stored in a Model class. Then, the Controller sends this data to the View and displays it to the user. Fig 6.4 shows all the model classes that are used to transfer the information form the Controller to the View. Also, we need a way to transfer the data to the backend in the REST calls, which is through the DTO classes.

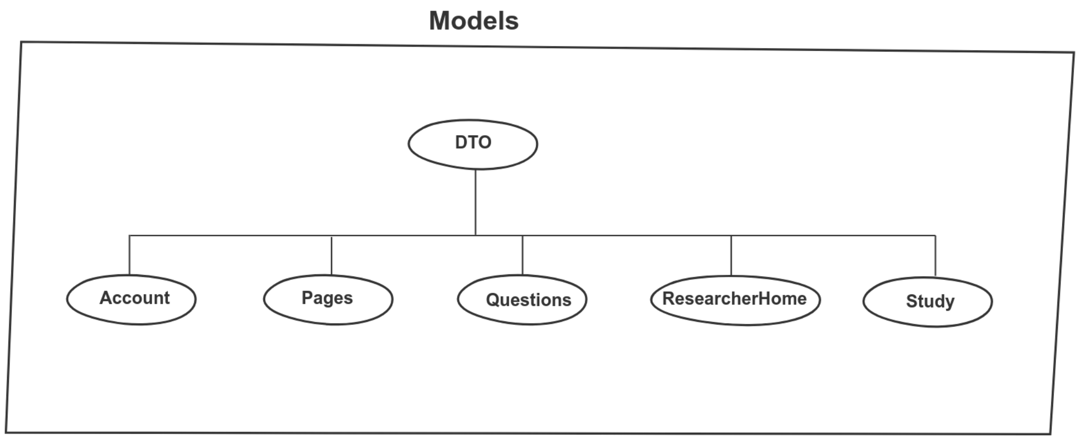


Figure 6.4: ResearcherUI Models design

After the Controller has received the data from the backend and stored it in an instance of a Model class, the View receives the data, and it can be displayed to the user. We can observe in Fig 6.5 that the View contains HTML, CSS, JavaScript and the images. In HTML, we specify the structure of the page, CSS allows to change the styling of the page, and JavaScript generates animation.

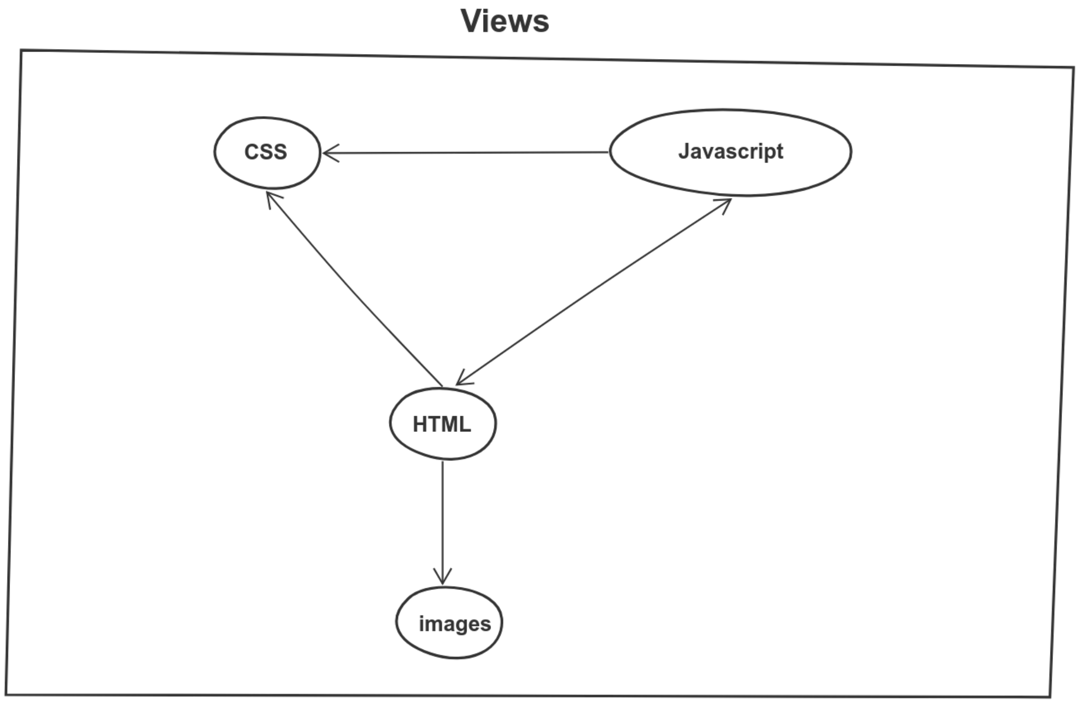


Figure 6.5: ResearcherUI Views design

The Controller initiates a connection with the backend, sends, and receives data. We need REST calls from the frontend Controller to the backend. To accomplish it, we created the same class structure from the backend in the frontend so that it is easier to write the REST calls in the frontend. Fig 6.6 shows the REST classes in the frontend that contain all the REST calls being used in the application, which is the same in the backend.

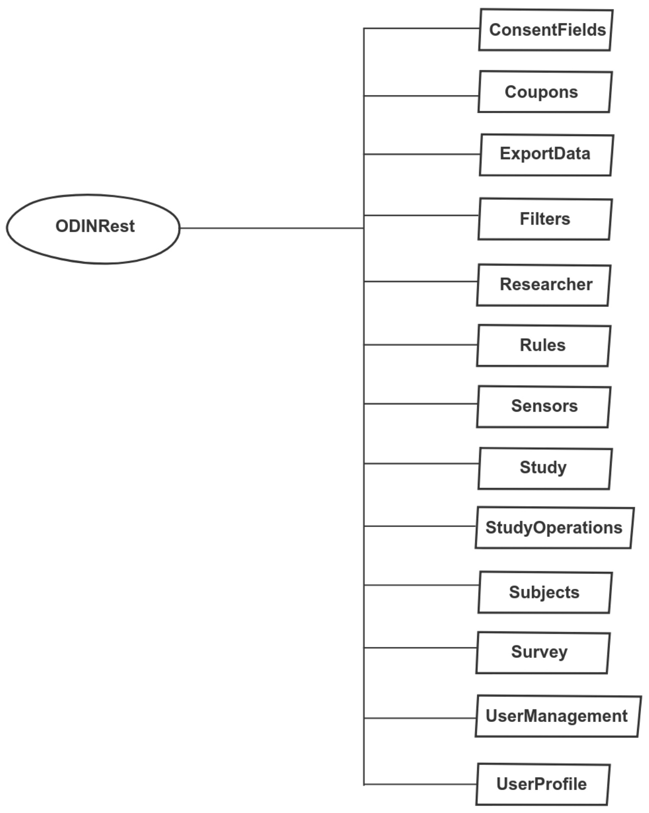


Figure 6.6: ResearcherUI REST design

6.3 Communication and Network Protocols

ODIN requires communication from the Web UI to the server and from the phone app to the server. In this section, we give a detailed description of the network protocols used to exchange messages in these two applications. We begin by giving some background information about the different protocols. There are two types of transport protocol: UDP and TCP.

TCP (Transmission Control Protocol) [1] provides a reliable, connection-oriented service to the application. TCP uses flow control, sequence numbers, acknowledgements, and timers and ensures that data is correctly and orderly delivered from the sending process to the receiving process. Furthermore, TCP provides congestion control by regulating the rate at which the sending sides of TCP connections can send traffic into the network. It also provides flow-control service to its applications to eliminate the possibility of the sender overflowing the receivers buffer. Comparatively, TCP is a more complex protocol than UDP. The TCP segment has 20 bytes of header overhead.

User Datagram Protocol (UDP) [1] is a Transport Layer protocol that is unreliable and connectionless. The connection is not established before data transfer is used for an application that is latency intolerant but loss tolerant. It is used for real-time services like computer gaming, voice or video communication, live conferences. No error checking in UDP permits packets to be dropped instead of processing delayed packets. UDP is more efficient than TCP in terms of both latency and bandwidth. UDP has only 8 bytes of overhead.

We can see the main differences between the two protocols by looking at their descriptions and some research that has been done in this area; TCP is more reliable than UDP, but UDP is latency intolerant, which is not the case in TCP. Hence, it depends on the application that we choose one protocol or the other. Some applications like video call use UDP since it cares about the fast delivery of the messages; however, some other applications like instant-messaging, cares about reliability, so we need to use TCP. There has been some research done on the network performance by comparing different transport protocols [78, 79].

Moreover, other research papers that have compared TCP and UDP on different applications by sending different types of packets. For example, The Performance Comparison of PRSCTP, TCP, and UDP for Mpeg-4 Multimedia Traffic in Mobile Network [80] it has been reported that when transmitting reliable and unreliable data TCP re-transmits all the lost frames which increase the transmission delay of the image. At the same time, UDP does not re-transmit any of the lost packets regardless of the type. Thus, worsening the quality of the image.

In ODIN, the messages are exchanged among different applications via REST calls. In other words, they use HTTP as a way to communicate, and HTTP uses TCP as the transport protocol. Therefore, the app and the Web UI exchange messages with the server via TCP. Next, we describe how this communication is handled in each application and when it happens.

6.3.1 Between App and Server

We know that TCP is reliable; however, there are external factors that make the communication between the app and server unreliable. For example, the user can run out of data, power off the phone, or the application could have died. Let's explore what happens in each of these cases. But first, we need to know how often the communication happens. As mentioned in the Android section, different services are running, and they make REST calls at different times, as we can see in Table 6.1.

Service	**REST**	**Interval**
Upload	putAnswers	20min
Upload	insertSensorsData	20min
RuleQuestion	getQuestions	1h

Table 6.1: REST calls between APK and Server

Other REST calls happen when the user registers when the consent is signed, and right afterward, getQuestions is called for the first time. However, we don't expect issues with these REST calls since the user is actively using the application at that time. The problem begins when the user stops using the application and the phone. The main issue is that the services running in the background could die at any time. For this reason, the server sends push notifications to the phone via SNS to keep the services alive. However, the user could run out of data. In this case, the phone would not receive the push notifications from the server. If the application is still running, we keep collecting data, but we fail to send the data to the server. Finally, the worst case is if the phone is off, then there is nothing we can do, no data would be collected, and there would be no communication with the server. Therefore, even using a reliable protocol like TCP, we cannot guarantee reliability in our application since many external factors could make it unreliable.

6.3.2 Between Web UI and Server

Fortunately, the communication between the Web UI and the server is more straightforward than the app and the server. In the previous section, we showed the different REST classes. The endpoints from the backend reside in those classes. The Web UI initiates a communication with the backend every time the user makes a request. For example, when the user Logs in or creates a new study. Every action the researcher makes it requires at least one REST call. Sometimes, we make more than one REST call in one request. For example, when the Survey page loads. In that case, we need all the labels, questions, rules, and filters, which require multiple REST calls. Since all of these requests happen via TCP, and there are no external factors, we can guarantee that they are delivered.

There are some other details about the communication to the server in the Web UI that we need to be aware of. When the user makes a request, it goes through two different servers; the Web server and ResearcherService. The Web server takes care of all the Controller classes in the Web UI, and those classes send requests to ResearcherService. We can see an example in Fig 6.7. The user selects the Login button; then, a POST Login request is initiated with the Web server, which hashes the password and sends the credentials in a GET request to ResearcherService to authenticate the credentials. The server generates a session key, which is stored in the database and send sent back as a response to the Web UI. Finally, it proceeds accordingly with the message received.

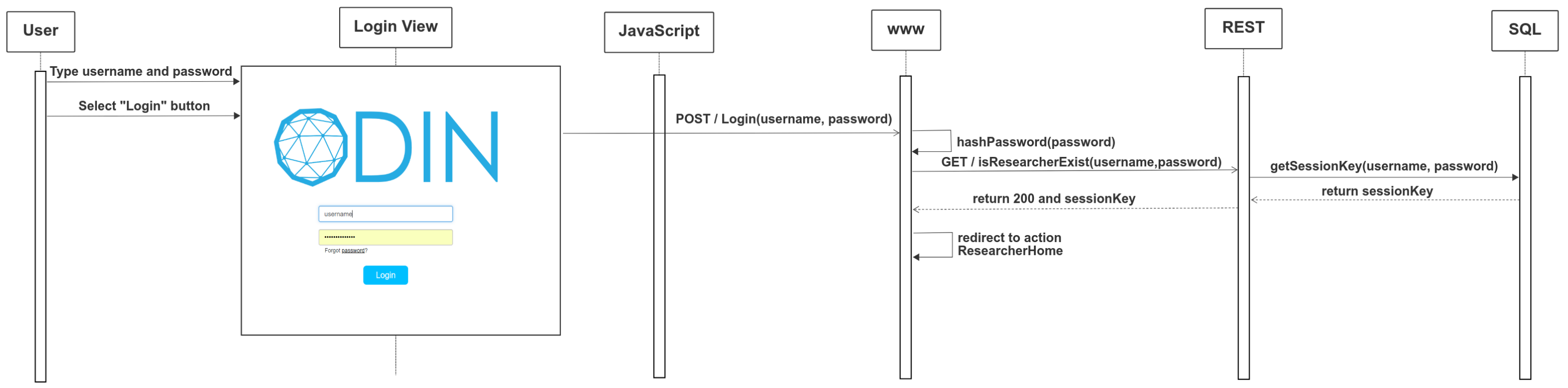

User
Login View
JavaScript
www
REST
SQL
Type username and password
Select "Login" button
DIN
username
Forgot password?
Login
POST / Login(username, password)
hashPassword(password)
GET / isResearcherExist(username,password)
getSessionKey(username, password)
return sessionKey
return 200 and sessionKey
redirect to action ResearcherHome

Figure 6.7: User Login sequence diagram of Web UI communication to ResearcherService

Chapter 7

Consistency across app and server database

The developer faces many challenges when implementing a Distributed Database System (DDBS) since it is is difficult to ensure both correctness and performance [81]. One of the reasons is because the same data resides on the phone and the backend database. Hence, we need to have an efficient protocol to ensure that we do not end up with inconsistencies between the phone app and the backend databases, and we do not lose any data during the process.

In previous sections, we have described the phone and backend databases, and we can observe the similarity in their database tables. They share the same Questions, Choices, Rules, Answers, and Sensor table. The main goal is to ensure all the data on the phone reaches the backend since the phone is a temporary storage device.

We have implemented some protocols to guarantee no data loss nor duplications. These are some challenges that we face that led us to have the protocols. We make sure that the phone only sends the data that has not been uploaded to the server yet. The phone uploads the data every 20 minutes. For example, if, for some reason, the phone does not receive the response from the backend, it does not know that the data was uploaded successfully. When the phone uploads again the data, it sends the same data that is already on the server. If we do not have a protocol we could end up with duplicate data in the backend. As a result, each sensor data has an id which is used in the backend to avoid duplicates. When the data is sent to the backend, then it checks all the entries and makes sure that everything inserted in the database is new data. The server sends back all the ids that have been successfully inserted in the database. It is more complicated for the answers because the phone stores them when the question is first prompted with a unique answer id. In this case, the backend allows the same id to be inserted because it could have been updated later by the user.

Keeping a consistent database is challenging due to many different factors like data duplication or edition. For this reason, we have our protocol to ensure the system does not end up with missing data in the backend and no inconsistencies in the data.

Chapter 8

Concurrency

Concurrent programming can be defined as multiple independent tasks being executed in parallel [82]. It is a challenging task to debug a concurrent system because the output may vary every time the program executes with the same input. For this reason, there has been some research on ways to help programmers with this. One such example is Kendo, which provides deterministic multithreading of parallel programs [83]. This software makes it easier for developers to debug and test their applications.

First, we define two terms that we are going to use throughout the chapter: deadlock and race-condition. Deadlock happens when a thread A is waiting for resource X while holding on to resource Y, but some other thread B is holding on to resource Y while waiting for resource X. In effect the two threads A and B are caught waiting for each other indefinitely [84]. A race-condition happens when multiple threads are accessing the same data simultaneously, but the side-effects depend on the order that the threads access it, which is not deterministically controlled. The usual manner in which these tricky situations are avoided is by careful and deliberate thread synchronization [84].

Concurrency plays a big task in the ODIN system. In Android, multiple services are running in parallel in an infinite loop. In the server, many researchers might be using the ResearcherUI, and many subjects have registered on their individual phones. REST calls originating at each phone wind their way through the backend server codebase and all hit the MySQL database layer. Therefore, it is crucial to managing our multithreading system to avoid deadlock in our applications. In this section, we describe the thread management mechanisms that have been used in the server and the phone.

8.1 Thread management and deadlock avoidance in the Server

Every REST call in the backend, either from the researcher or the participant, starts a new thread. There exists a thread pool in each backend service, handled by the JVM, and whenever a there is a new in-coming HTTP request, then a thread is taken from the thread pool and starts its execution in the REST call. Hence, we can have many threads being executed at the same time.

Global variables with unsynchronized methods increase the risk of race condition. The backend can have many threads being executed simultaneously. Accordingly, not using global variables in ODIN services decreases the chances of having multithreading issues. However, there are classes in which we need to have global variables (public singletons). In such situations, we make sure that the methods which access the global variables are synchronized, or the public static data member is surrounded by a mutex so that it can only be accessed by one thread at a time. Therefore, we use local variables and synchronization to have thread-safe services.

The other area that is important to manage multithreading properly is in the database. As mentioned previously, PhpMyAdmin holds the ODIN database. The ODIN database runs in the MySQL server in the Ubuntu VM. We keep track of the database connections happening simultaneously. We do this by incrementing the count every time a connection is opened and decrementing every time it is closed. This variable helps us ensure that the connection is always closed when the database transaction has been completed. The number of database connections is a constant which can be incremented in the service properties file. Each REST call initiates one connection, which is closed at the end, and the database queries happen in separate methods, so there is no risk of a race condition.

These mechanisms help us ensure that ODIN would not suffer from race conditions nor deadlocks and, therefore, it operates as a multithreaded, yet thread-safe system.

8.2 Thread management and deadlock avoidance in the App

During designing and implementing the Android application, deadlock avoidance has been one of the biggest challenges we have faced. The application would run for a certain amount of time, and then it would stop working without showing any exceptions or errors. Even by adding and enabling all the logs, we could not figure out what would be the cause of the issue. The only solution is by reading and understanding the code and trying to find the bug. We have done this many times during the implementation of ODIN. We believe we finally have a thread-safe application.

The challenge in the phone app is that many services are running in parallel. In the Android Chapter 5, we explained all the services that are running in the application, and each service has its own FSM. Therefore, each service runs independently with no shared variables, so there is no risk of deadlock. However, the challenge is when the services use the DAO layer to access the app's SQLite database.

Like the server, the Android App also has to read and write to the database. Multiple services can read the same table simultaneously, and each table has its class. Also, the PersistenceLayer class exposes methods from the table classes. These methods have signatures that are shared with the backend. PersistenceLayer and the Table classes have all their methods synchronized to make sure that we are not reading and writing simultaneously to the same table. Finally, the app also keeps track of the number of connections opened and closed. One connection is opened in the startup of the application, and then it follows the same algorithm described previously for the server in Section 8.1. Then, it asserts that the number of connections is never 0. This way, we are aware if there are any multithreading issues in the DAO layer.

Multithreading in Android has been one of the biggest challenges in developing ODIN. However, after several months of debugging and re-designing the structure of the application, we finally have a working thread-safe system.

Chapter 9

Testing Strategies

Software development is not like other businesses where the client might appreciate our effort and enjoy the final product [85]. In software development, the client expects a perfect system and would complain if there are bugs, it could potentially mean losing the client [86, 87, 88].

It is essential to distinguish between testing and debugging. Debugging implies finding the bug in the code that resulted in a test case failure, while testing is based on coming up with different scenarios and making sure that the system works as expected [89, 85].

ODIN is a sophisticated infrastructure in which many external factors can break the system. Accordingly, testing is a crucial task to ensure the system is working as expected.

There are different testing techniques, but we focus on correctness testing in this section. Sawant et al. describe the different forms of under correctness testing [89], which depends on how much the tester knows about the software [90]: white box, grey box, and the black box. In the white box, the tester knows everything about the implementation, while in the black box, the developer does not know anything, and the grey box is somewhere in between. These testing techniques differ on the testing strategy used; unit testing, integration testing, system testing. In this section, we give some background on each testing strategy and we describe in detail how it is applied into our system.

9.1 Unit Testing

Unit Testing is the smallest one of the testing strategies since it requires to test the smallest number of lines of code. It is known as white box testing since the tester must know the insights of the implementation to test it [89]. Next, we describe how we apply this strategy into different applications.

- **Backend services**: We use SOAPUI as a software tool to do unit testing on the endpoints from all the services. There are four SOAPUI projects, one for each service. SOAPUI allows the tester to create test suites that contain different test cases that can have multiple test steps. For unit testing, the tester only creates one test step, which is the endpoint that is being implemented. These are the steps that need to be followed to create unit test cases in SOAPUI. Consider the Login endpoint as an example:

  1. Think about the different test cases. In this case, we have a successful login and failure.
  2. Document the test cases. The input is a username and password, and the expected output is a JSON response with code and message. Then, we write to them as follows:
     - Login(username="correct", password="correct") − > code=200, message="ok"
     - Login(username="incorrect", password="incorrect") − > code=500, message="error"

If we run these test cases and all the outputs match, the test suite has passed, if there is any mismatch, it fails.

- **Web**: Unfortunately, we cannot test the Web services (ResearcherUI and AdminUI) using SOAPUI. Instead, the tester pretends to be a researcher using the Web UI and checks if there are any issues with the implementation. For example, the tester goes to the Login page and tries to login with a strong password and an incorrect password and checks if it is going to the correct Controller (Account) and Action (Login) with the expected input and output.
- **Android**: JUnit is the framework used for testing the ODIN phone application. We have two types of testing: Unit and UI testing. The main difference is that with UI, we can test Android code while on Unit, we test general Java classes. An instance of Unit testing would be to assert that we receive a successful message from the backend. On the other hand, checking the error message displayed on the screen if the coupon is less than 10 digits would be a UI Testing example.

9.2 Integration Testing

The next testing strategy is integration testing, which requires putting together some parts of the system while checking for errors in the interface [89]. Again, we divide it into different applications:

- **Backend services**: We use SOAPUI as a software tool to do the integration testing, but it is a bit more complex. For integration testing, the tester creates multiple test steps, which are the endpoints that are being implemented. Let's give a simple example if we want to create a test case for the CreateQuestion endpoint. The first step is to create a CreateQuestion test suite. Then, we come up with different test cases: successful and failure. For the researcher to create a question, there is a sequence of steps that need to happen: the researcher first needs to login, then create a study, and finally create a question. Each of these is a step under the test case. Hence, we follow the same protocol as unit testing described previously. If we run and all the outputs in each test step matches, then the test case has passed, if there is any mismatch in any test step, then the test case fails.

- **Web**: We follow a similar procedure than the unit testing. The main difference is that the tester checks that it redirected to the correct page instead of checking that it went to the correct Controller and Action. In other words, it successfully connected to the backend, and it performed accordingly with the response.
- **Android**: For integration testing in Android, we need to put together the PhoneAppService and the phone application. The tester registers on the phone and runs different scenarios to make sure that it is successfully talking to the backend and acting accordingly with the responses received.

9.3 Acceptance Testing

Acceptance testing is done to ensure that the product meets the criteria specified by the customer [91]. This type of testing is considered black-box testing since the user doesn't know anything about the implementation. Hence, we let the researchers interested in our system play with it and make sure it satisfies their needs. They start by creating a study on the ResearcherUI and then register on the phone. Finally, they make sure that all features meet their criteria.

The users carry the phone throughout the study, so we use Bugfender [56] to look at the logs since our access to user phones is restricted. Bugfender facilitates finding and fixing the bugs [56]. One of the most useful features that we use from Bugfender is being able to set a coupon code for each phone. As a result, we can search for the desired participant by typing their coupon number. Then, we can see the number of logging lines by level, which is helpful to determine if a phone is continually crashing because the number of error logs would be very high. Bugfender is a handy tool to analyze the phone performance for each participant. It helps the developers to find errors during the acceptance testing period and system testing, which is described next.

9.4 System Testing

The last type of testing also falls in the black box testing, and it is done after a fully functional system already exists [89]. The main goal is to run different test cases on the system to make sure it is not fragile, and confirm that it is stable. For this type of testing, we have users and developers that do not know everything about all parts of the code to run different test cases on the entire system to ensure its correctness.

On top of these techniques, we use Jenkins to run daily automation tests. The backend tests in SOAPUI execute on a JUnit test suite so that they can be deployed into Jenkins. The Android tests are already implemented in the JUnit framework so they can automatically be deployed into Jenkins. Jenkins is running on an Ubuntu Virtual Machine, which is different from the backend and the frontend. There exist some scripts that automatically deploy the binaries after building them into the IDE. Hence, it is simple to update the latest binaries into Jenkins and run the test cases to make sure the system is continuously stable.

Chapter 10

System Evaluation and Validation

System evaluation and scalability is a complicated job. Liu et al. state that it takes five years for software engineering to develop the expertise skills for scalability [92]. However, it is a crucial feature to ensure the satisfaction of the client. For this reason, ODIN must be scalable. In this section, we describe the scalability of different ODIN features: participants, researchers, questions, rules, and sensors.

10.1 Scalability for the number of participants

The number of participants increases as more users register to the ODIN phone app. Each participant is running its instance of ODIN on the phone. Hence, the backend database is the only part of the system that is shared among the participants. For this reason, we need to look into increasing the load in the REST calls, which can be accomplished using SOAPUI, and the system successfully ran a big load of REST calls simultaneously.

10.2 Scalability for the number of researchers

The number of researchers using the Web UI increases as we have more users interested in ODIN. The ResearcherUI talks to the ResearcherService in the backend. We need to follow a similar protocol to the scalability of participants. SOAPUI can help us perform a load test to simulate multiple users using the Web UI.

10.3 Scalability for question instances

Increasing the number of questions in a study affects the ResearcherUI and the phone application. We can show that those two are scalable in terms of question instances. If the researcher has created many questions, the main issue is retrieving them from the database. We have tested adding many questions to the database, and we can see that the retrieving time does not increase drastically as the number of questions increases. On the phone side, increasing the number of questions does not affect the user experience as long as the user has enough storage on the phone to support all the questions. Each question is prompted independently of the others, so increasing the number doesn't have any effect on the APK.

10.4 Scalability for rule types

The most complex scalability is the rules since it affects the rules engine. The rules engine registers the rules at the beginning, which the performance decreases as the number of rules increases. We have designed an efficient architecture to be able to support a new rule type within ODIN. In the backend, we need the following changes:

- 1. Update two properties files in ODINCommon: sensorNameToRuleType.properties and ODIN-RuleLabels.json.
- 2. Create a new class that extends CronRule, and implement the following methods:
  - **notifyCouponAdded**: registers the rules in the scheduler

  - **handleTrigger**: get the sensor rows from last rule fired time to the last sensor row and calls the sliding window.
  - **checkPredicate**: checks if the condition for the rule is satisfied. This method is not needed if the rule is non-sensor related. In the case that we want to write a negation rule, we modify the checkPredicate method.

- Update generateRuleSimEnt method in the RuleFactory class, which is the class that creates the rule object.

We can observe that new rules can be easily added to the system, and it does not affect the rest. Hence, even by increasing the number of rule instances drastically, it would not take a long time to process them.

10.5 Scalability for sensor types

Increasing the number of sensors does not affect the performance either of the Web UI or the phone application. In our ODIN design, each sensor is treated independently. In the backend, we create a separate table for each sensor, and on the phone is a separate service. The following are the changes needed in the backend to add a new sensor:

- 1. Update four properties files: sensortypes.properties, sensorNameToSensorHeartBeatNames.properties, and sensorHeartBeatIntervals.properties, and ODINSensorLabels.json
- 2. Create three new classes: Model, DAO, and Params. The Model class should extend the SensorData class. The DAO class should contain all the methods to read and write into the database. The Params class contains all the

parameters of this sensor. These parameters should be validated in the methods isValidParams and isValidJsonObject().

- 3. Add three new fields in the Constants file: sensor name, sensor DAO package, and sensor param package path.
- 4. Update the constructor in the StudyDao class to add properties of the DAO package and param classes for the new sensor.

On the phone side, we need to create a new service, a new table, and add the sensor type in the Sensor enum class. We can see that adding new sensors does not affect the system. Hence, the system is reliable as we increase the number of sensor instances.

Chapter 11

Distributed Logging and Error Detection

Logging is an essential task to recognize failures in the system. The developer uses logging to investigate the programs' behavior during runtime and decide whether there is any failure. However, deciding the logging statements is not an easy job. On the one hand, if we choose to log very little information, the system might fail, and the developer would have a hard time finding the cause of the failure [93]. On the other hand, logging too many causes other issues such as slower performance due to writing to CPU [94], or making it difficult to find the cause of the error due to many irrelevant logs [95]. For this reason, there has been some research done on this area to help developers automate logging. An example is the Microsoft Research Team that has developed a tool that learns how to log from existing logging instances [96]. However, there is still more work to be done to be able to automate logging fully.

In this section, we describe the logging practices we have designed and implemented in the system. Logging has been one of our most valuable resources to debug ODIN. We have faced many issues in the development process, and logging has been the most useful tool to overcome those difficulties. Android and Web logging are pretty straightforward, each application has a log file in the file system, and they write to it. Although, things become more complicated in the backend, which is not as simple as printing all the logs in a single file. We have designed a distributed logging system that consists of many log files, which makes it easier for the developer to track down the problem.

The motivation behind having distributed logging is that it gives us some hope to manage the complexity of the ODIN system. In PhoneAppService, multiple users are making calls to the backend regularly, so if there is an issue in one of the phones, it is nearly impossible to find anything if all the logs are written on a single file. In ResearcherService, there could be many researchers creating studies daily. If one of them encounters an issue in a study, it is easier to find the problem if we have a dedicated log file for each study than looking through a single file with all the logs. Hence, we have chosen to use a distributed system to simplify the task of finding the cause of the failure.

First, we describe the big picture of the logging system. Fig 11.1 shows the overview of the Logging structure. We can observe that the Controller sends the log file names (studyID, coupon, and mainLog) to the LoggerManager class. Then, LoggerManager calls MyLogger class to write to the file. MyLoggerFactory is used to determine the application that is logging, which could be the phone or the backend. This approach was because the Logging classes reside in ODINCommon, which is a jar file used on the APK. Hence, the phone and the backend have their implementation of the ILogger and ILoggerFactory interfaces. As mentioned previously, the phone only logs to a single file while the backend follows the distributed logging design.

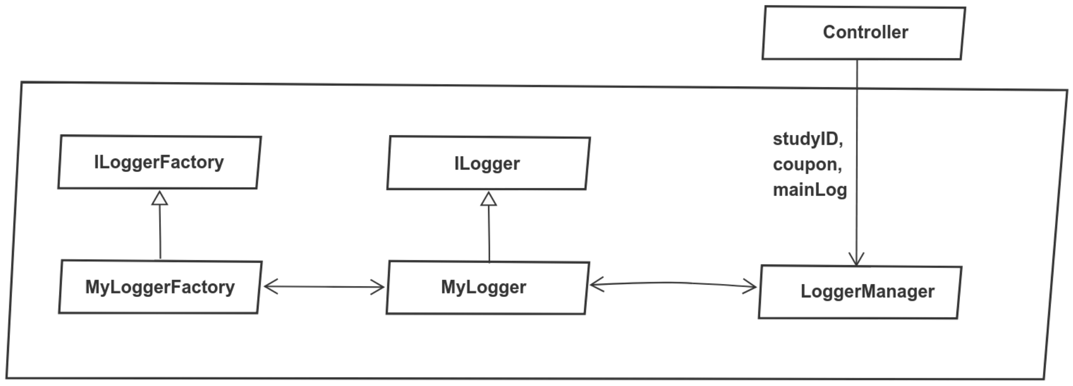


Figure 11.1: Class diagram for logging. The Controller sends a list of log files to the LoggerManager. MyLogger implements ILogger interface, and MyLoggerFactory implements ILoggerFactory interface.

Now that we have a high-level picture of the logging system let's dive into more details of the Logger Manager class. We can observe in Fig 11.2 that the Logger Manager contains three maps as data members; fileToCount, threadToFiles, and fileToLogger. We need to keep track of the threads because each REST call initiates a thread execution and terminates when the REST call concludes. Thus, in the beginning, and at the end of every REST call, we call the method "beginThread" with the log file names and "endThread" respectively, which reside in MyLogger class. At the beginning of the REST call, a new entry is inserted in threadToFiles Map. The map contains the currentThread as the key and the list of file names as the value. Additionally, one entry per file name is inserted in fileToCount, and the count for each file is increased. Finally, at the end of the REST call, the thread is removed from threadToFiles map, and we check the files that were linked to this thread in fileToCount. If the count becomes 0 after decreasing it at the end of the REST call, then we remove that entry from the map, and we also remove the entry that matches the files where the count is 0 in fileToLogger.

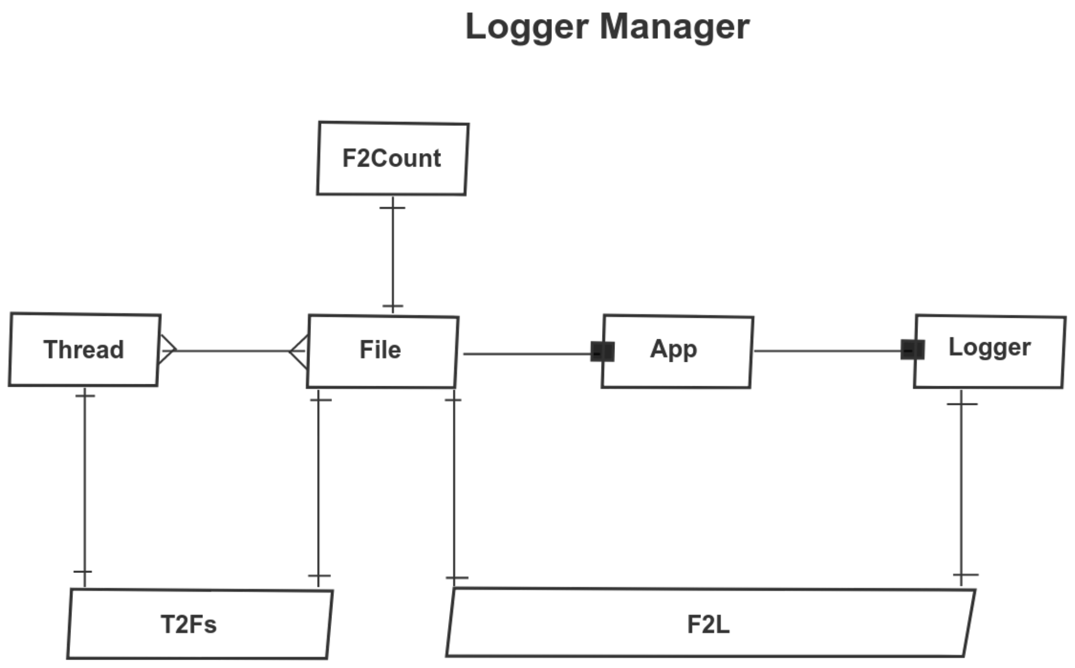


Figure 11.2: Logger Manager data members objects

We have mentioned the map fileToLogger, but we do not know how it is generated. As described previously, the Controller first sends the list of file names to MyLogger, and generates all the Maps. Afterward, we make a call to the Logger object to log, Fig 11.3 displays the process followed in the execution of that call. First, MyLogger makes a call to LoggerManager which checks the fileToLogger map to see if we already have an instance of the Logger for that file. There are two possible outcomes:

- The Logger exists, then we add it to a list of MyLogger instances, iterate through each of them and write to the file.
- It is the first time writing to this file, so we create a new Logger instance and add an appender with the name of the file that we want to write. Afterward, it adds the new instance to MyLogger list. Then, iterates through each of the

instances and writes to the file.

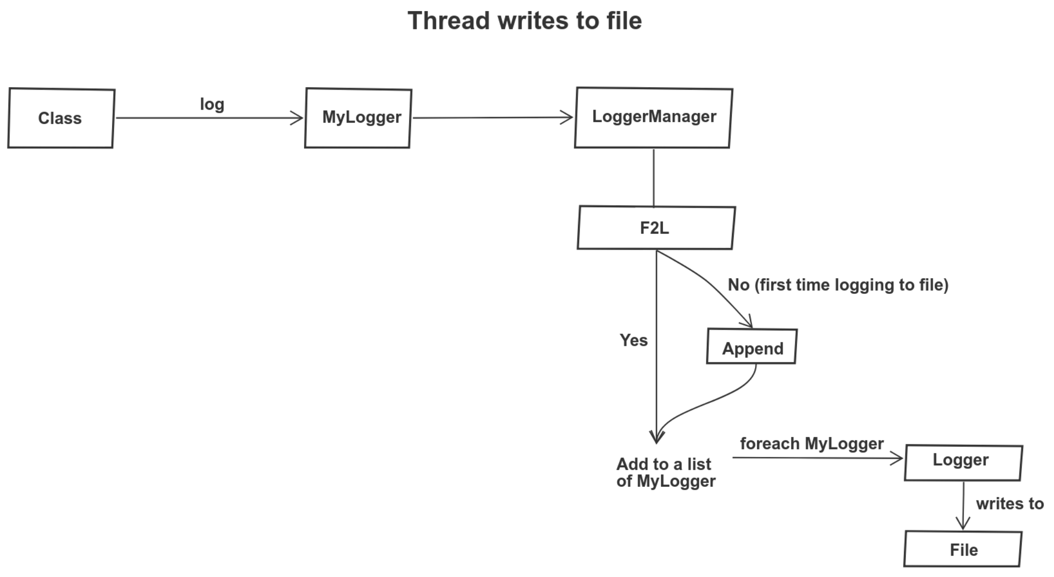


Figure 11.3: Diagram of a thread logging into a file

Even using these logging practices is complicated to detect errors if there is no difference from an info log than an error. For this reason, we have multiple levels of logging; debug, info, warning, and error. By having different levels, we encounter a couple of benefits. 1) It is easier to remove unnecessary logs in production like debug and info, and 2) simpler to search for errors. Therefore, the logging levels are critical to performing error detection successfully.

If we are aware that there is an issue in the system, it is easier to detect by using distributed logging; however, we might not know that there is an error in the first place. Accordingly, we have incorporated exception notifications in our system. When there is an exception in any of the backend services, we retrieve the list of developers to notify them. The list is in the properties file of each service. It makes life easier for the developers because they do not have to check the logs and search for errors. Instead, they can keep working on the development of the application and receive notifications when there is a failure in the system.

Chapter 12

Security

The definition of security in the software engineering field the practice of writing code in a way that will prevent malicious attacks and thwart the attackers ability to steal user's private information. The developer has to design and implement software in such a way that it is secure and reliable. In 2004, software security was considered a relatively new research topic. According to [97], the first book on security in software engineering was published in 2001. Hence, there were many years in which the software had flaws. Bellovin et al. state that any program, even if it seems to be safe, can have security deficiencies [98]. In the last two decades, there have been many malicious attacks in trendy applications such as Facebook with 50 million profiles affected [99], the IRS with 330,000 taxpayers of data compromised [100], and the biggest data breach ever happened in Yahoo which compromised 1 billion accounts [101]. Developers have become aware of the situation, and are devoting more time to learning how to build more secure software. Companies are spending more money on security. However, by looking at and understanding all the data breaches that have happened in the last years, it is not enough. Computer scientists still have to learn to design and implement better more secure software [102].

ODIN records and stores personal data; thus, we want to prevent data breaches in our system by designing a reliable security system in each of our applications. In this chapter, we start by describing the security in the Network transport layer during the REST calls, then we focus on the Android security and finally the Server Security.

12.1 Network transport Security

ODIN system transfers much data over the network; from the Web UI to the Web Server, from the Web Server to ResearcherService and from Android to PhoneAppService. At present, the data is being transferred over HTTP, but we plan to transition it to HTTPs. It is an improvement that is needed so that we can ensure security from end-to-end security within the system. In other words, to provide security in the REST calls, we need to ensure that all the data that is being transferred through SSL [103] over the network.

12.2 Android Security

The main concern in an Android application is that there could be other malicious applications trying to obtain personal information from ODIN. For this reason, we need to be aware of these possible attacks and develop a safe application to prevent them. In this section, we describe three security methods: build variants, prevention of ICC attacks, and participants.

There are multiple releases of our application; developer, researcher, and participant. Developers have access to the database, status of the services, and the option to crash the application. Neither the researcher nor the participant should have access to this sensitive information. The researchers have access to the same features as the participants and viewing all the questions to make sure the formatting is as expected. Accordingly, we need to be able to maintain different versions of the application and also make sure that participants do not have access to the developer's features. To achieve these multiple goals, we use build variants [104] that allow the developer to choose the version of the APK to build, which is determined from the different packages. Therefore, build variants allow better app security because we ensure that each type of user gets access to the appropriate subset of features.

In Android, the most common attacks are the Inter-Component Communication (ICC) Attacks. There are two main categories:

- **Unauthorized Intent Receipt**: In this scenario, a vulnerable application sends an intent which is intercepted by another malicious application that declared it as an Intent Filter.

- **Intent Spoofing**: In this case, it is the malicious application that sends the intent, and the vulnerable application then receives it.

Each of these attacks can happen in the different Android components; service, activity, content provider, and broadcast receiver. Therefore, it is necessary to think about these malware scenarios and prevent them. Next, we describe what we do in our application to prevent those attacks from happening.

The best way to prevent these attacks is to use action and alarm codes. The former is added to the intent and the latter to the scheduled alarms. In our Android application, there are many alarms and intents being sent simultaneously. For this reason, it is crucial to prevent an ICC attack. On the one hand, we always check the action code of the intent when it is processed. On the other hand, the alarms have an alarm code attached to them that is also checked every time an alarm is processed. This mechanism ensures that a malicious application would intercept neither the intents nor the alarms since they have a unique key attached to it. Moreover, by checking all the intents and alarms, we ensure the application does not suffer from intent spoofing. Therefore, the ODIN application should be secured from ICC attacks, which are the most common ones.

Participants may attempt to obtain more incentive from the study, and they can try to re-register the application with a different coupon or on another phone with the same coupon. The backend stores the phone's IMEI in the registration REST call to prevent form these kinds of attacks. Every time a participant registers with a coupon, the backend checks that the combination of studyid, couponnumber, and IMEI is unique. This avoids the user attempting to issue a registration with the same coupon on different phones or a duplicate registration on the same phone with the same coupon. Thus, we prevent participant attacks on the ODIN system by using the phone's IMEI to detect such attacks.

12.3 Server Security

The server database is hosted in PHPMyAdmin, and it is secured with two different login credentials. Moreover, we have multiple accounts, and the only account that is granted all the privileges is root. Hence, an attacker would need to have both login information, and only with the root account would be granted all the privileges.

To keep our users' accounts safe, we use MD5 [105] encryption for the user's password when they log in before sending it to the server. Then, the encrypted password is sent over the network, and it is saved in the server database. By using this security mechanism, we ensure that no attacker has easy access to the actual password of the user, which helps keep their accounts confidential.

Another essential security point in our system is the database access from a specific IP address. MySQL can determine what IP address is being used for logging in. Hence, each account is attached to an IP to make sure only that username coming in from a specific IP can log in. We can also specify the bind-address, setting it to the machine's IP address. We do not set it to localhost because that increases the security risks by making the database more accessible to attacks.

Finally, all the services can only be accessed from inside UNL or using the VPN, and only the PhoneAppService has an "outside" IP address. This restriction helps prevent the system from outside attackers since ResearcherService and AdminService are only accessible from the inside.

We have described the security mechanisms that we have used up to date in ODIN. However, there is still a lot more research that needs to be done in this area to ensure the security of our users. To date, we have not had a very large number of concurrent users, but as the number keeps increasing, we want to make sure we can have a safe and reliable system which can help researchers and participants have trust in our system and its ability to keep their data safe.

Chapter 13

Experiments and results

In this section, we present the experiment followed by the results obtained. First, we present the list of metrics used to evaluate the performance of the ODIN system. Second, we explain the pilot experiment. Third, the results obtained from the pilot are presented. Finally, we draw conclusions based on the results acquired. The metrics to examine the rules and sensors performance are presented below.

Note that the first three metrics in the rules section only take into account time-based rules since this is the only rule type for which we can predict the expected number of rule firings. Further research is needed to calculate these first three metrics for other types of rules which reference sensor data.

- Sensor metrics
  - **Sensor recording interval**: average difference between two consecutive sensor recordings.
  - **Sensor reliability**: average number of recordings that happened within 1% deviation divided by the total number of sensor recordings.
  - **Min time between recordings**: average smallest time between sensor recordings divided by the ideal time interval between sensor recordings (specified by the researcher).

  - **Max time between recordings**: average largest time between sensor recordings divided by the ideal time interval between sensor recordings (specified by the researcher).

- Rule metrics
  - **Subject performance**: average number of answers divided by the number of actual questions asked only (based only on time-based rules).
  - **Phone performance**: average number of actual questions asked divided by the number of expected questions only (based only on time-based rules).
  - **Pilot performance**: average number of answers divided by the number of expected questions (based only on time-based rules). Note that Pilot performance = Subject performance * Phone performance.
  - **Good ideal rule fired**: average number of "good" rule firings divided by the expected number of rule firings. A "good" rule firing is one that happens with less than 60 seconds time deviation from the time the rule ought to have fired. Note that the reason that rule firings are not all "good" is because the Android OS does not provide real-time guarantees as to when intents will be delivered.
  - **Good rule fired**: average number of "good" rule firings divided by the actual number of rule firings. We allow a 60 seconds time deviation from the time the rule should have fired.
  - **Late rule fires**: average number of times a rule fired late (by more than 480 seconds) divided by the total number of times that the rule actually fired.

- **Missed rule fires**: average number of times a rule did not fire divided by the total number of times the rule actually fired.
- **Early rule fires**: average number of times a rule fired early (by more than 480 seconds) divided by the total number of times that the rule actually fired.

A pilot study was conducted on 14 undergraduate students over 30 days at the University of Nebraska-Lincoln (UNL). Students psychology majors who were registered in the SONA Psychology Participation Program. Each student was assigned a coupon to register on the ODIN phone application. The pilot started on October 17th and concluded on December 10th. The students received class credit as incentive, which was not proportional to the number of answers; they received full credit just by participating in the study. We had to go through the IRB process which prevented us from letting them use their own phones. Instead, they were provided a Moto G3 phone to register for the study and carry it throughout the study. Hence, the participants were carrying two phones: their personal phone and the Moto G3. Using the same phone model for all the participants allows us to obtain better conclusions about results since the OS functionality varies significantly among Android phone models.

We can observe the sensor details of the study in Table 13.1. The sensors enabled in the study are GPS and Bluetooth Proximity. The GPS sensor records every 5 minutes, and the Bluetooth Proximity records every 5 minutes. This GPS sensor is used to track the participants' location to know when they are on the UNL campus. Also, we want to know when the participants are interacting socially by using the proximity Bluetooth sensor.

Table 13.2 shows the survey details of the SONA pilot. We can observe the list of questions along with the rules and filters associated with each question. The study consists of 3 questions of two different types (Fill Text and Multiple Choice). The first question is prompted either when the participant is on campus between 6 am - 8 pm or is near another study participant on campus between 10 am - 2 pm. The second question is asked 3 times a day at 9 am, 2 pm, and 7 pm, and when the participant is near another participant on campus between 10 am - 2 pm. The third question is a follow-up of the second question; it is prompted when the user provides an answer to the second question. All the questions had a maxAnswerTime of 15 minutes. In other words, the participant was given 15 minutes to answer the question before it expired.

Sensor	**Parameters**
GPS	Every 5 minutes
Bluetooth Proximity	Every 5 minutes

Table 13.1: Sensors with corresponding parameters from the SONA study pilot

Question	**Type**	**Rules and filters**
1	Fill Text	1. While on City Campus in between the hours 6am-8pm (ask every 14h) 2. Upon joining another study participant if on City Campus and in between the hours 10am-2pm (5 minutes delay)
2	Multiple Choice	1. Everyday at 9am 2. Everyday at 2pm 3. Everyday at 7pm 4. Upon joining another study participant if on City Campus and in between the hours 10am-2pm (5 minutes delay)
3	Multiple Choice	1. When user answers any response from question 2 (1 second delay)

Table 13.2: Questions with associated rules and filters from the SONA study pilot

Next, we present the results of the SONA pilot. This section is divided in two parts: sensor performance, and rule performance. In the sensor performance we describe the results of GPS and Bluetooth. On the other hand, the rule performance presents the results of all the different rules in the study which are time-based rules, GPS, and Bluetooth rules.

13.1 Sensor performance

The location and proximity to other participants are tracked using the GPS and Bluetooth sensors. Both sensors are set to record at an interval of 5 minutes. Fig 13.1 shows the average **sensor recording interval** of each participant throughout the 30 days. We can observe that, in most cases, it is above the expected interval of 5 minutes (expected interval). The sensors of participant 9 has a very high recording interval, with an average of approximately 700 seconds (11.6 minutes). We can observe in Table 13.3 that GPS and Bluetooth have an average **sensor reliability** above 90%. Most of the other sensor recording intervals stay within 5 minutes as can be seen in Fig 13.2 and 13.3. A vast majority of the outliers are above 5 minutes.

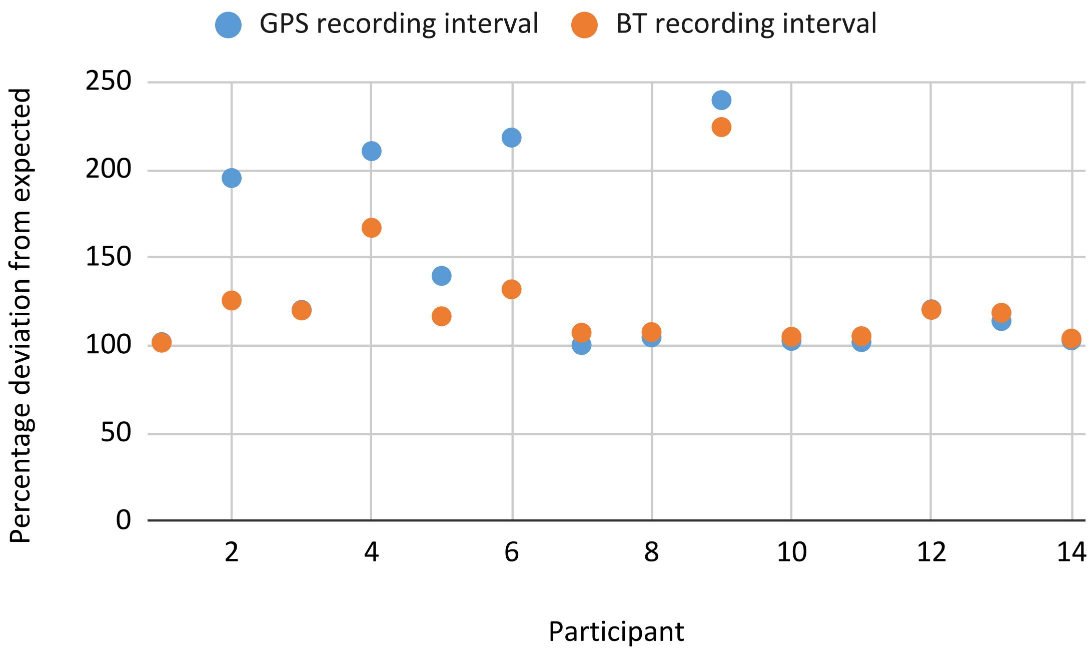


Figure 13.1: GPS and Bluetooth average sensor interval for each coupon throughout the study. The horizontal axis represents the participants of the study and the vertical axis is percentage deviation from expected interval.

Sensor	**Min time between recordings**	**Max time between recordings**	**Reliability**
GPS	-256.14%	41080.23%	93.86%
Bluetooth	-247.19%	21502.45%	90.99%

Table 13.3: GPS and Bluetooth average reliability among all the participants.

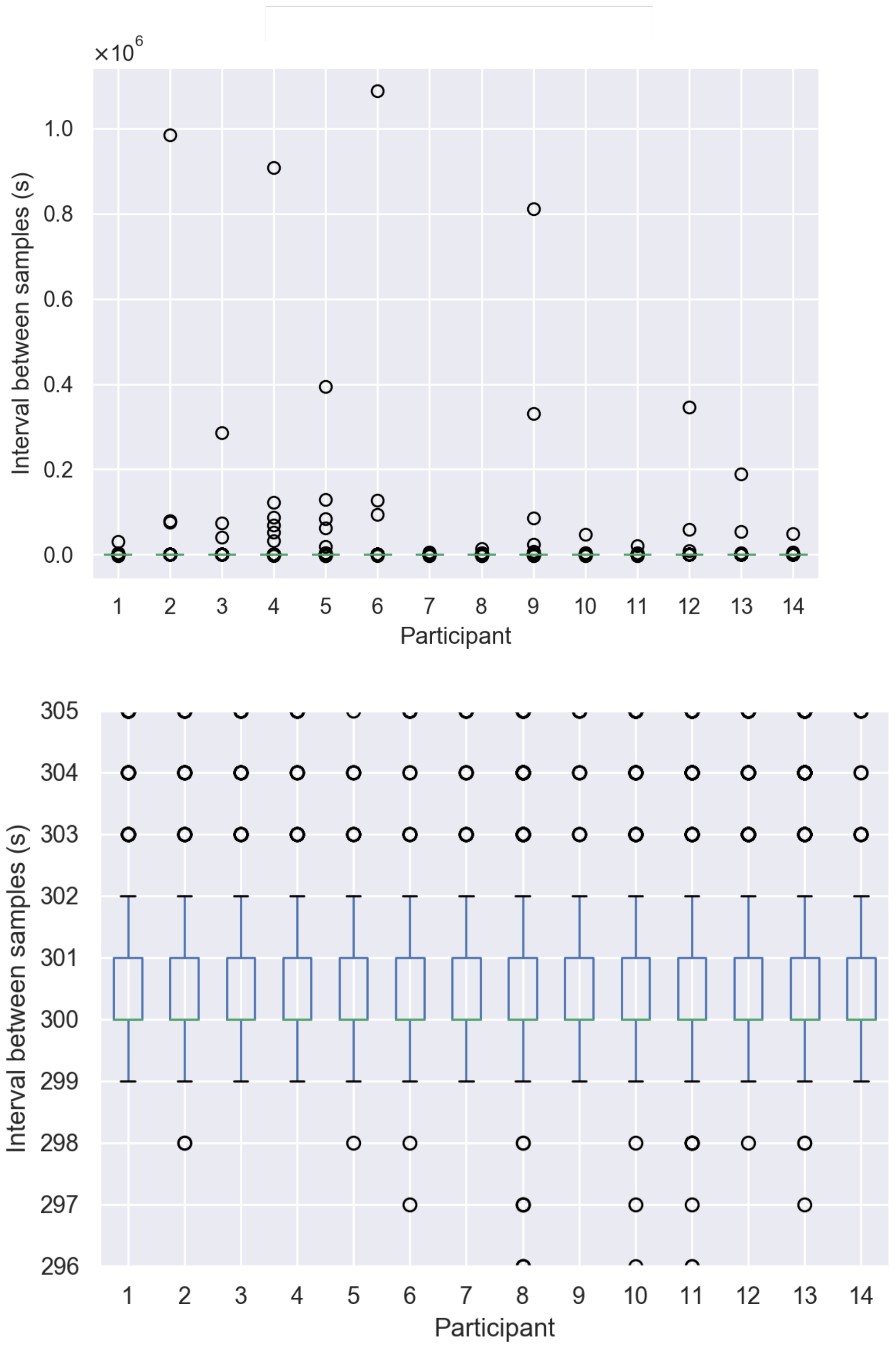
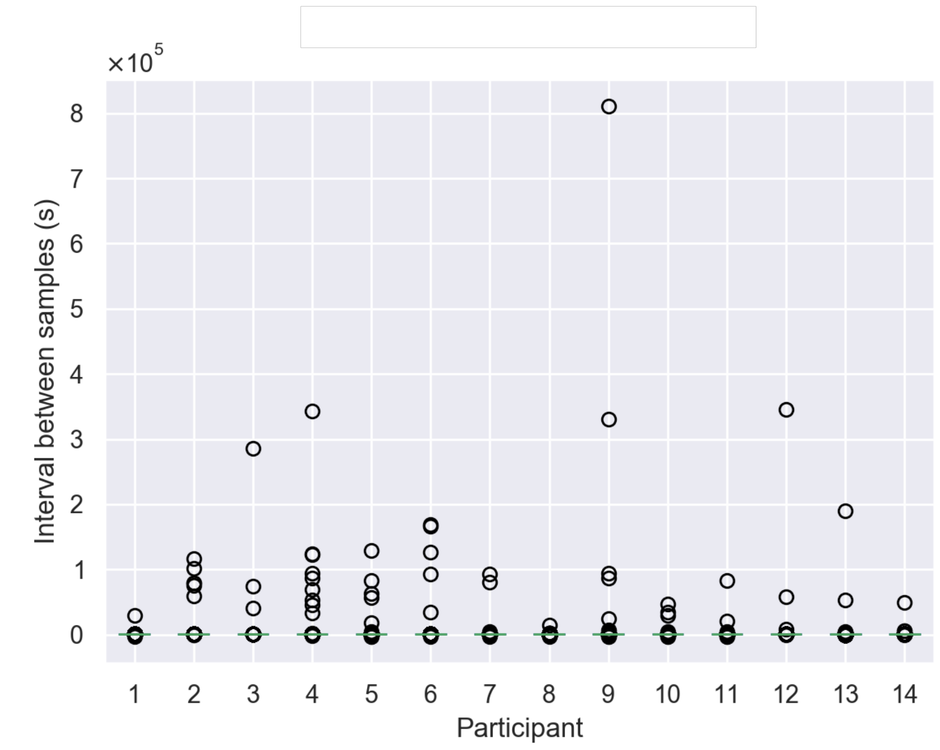


Figure 13.2: Intervals between GPS samples. The horizontal axis represents the participants and the vertical axis represents the difference between intervals. The two figures show the same data but at a different scale.



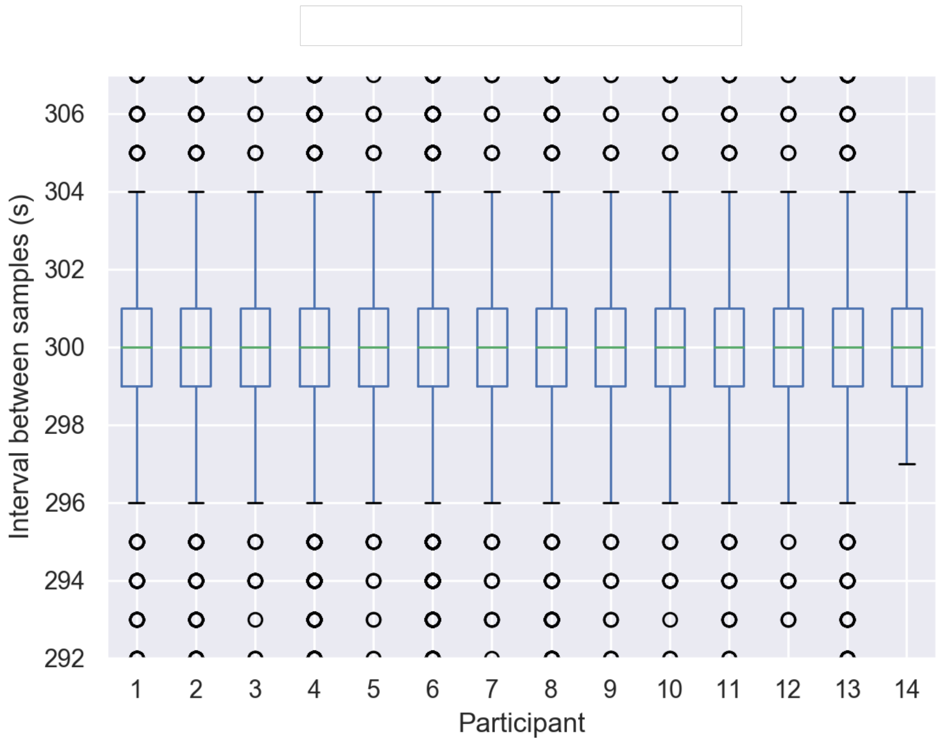


Figure 13.3: Intervals between Bluetooth samples. The horizontal axis represents the participants and the vertical axis represents the difference between intervals. The two figures show the same data but at a different scale.

These outliers are due to either the phone being off or service faults. For example, participant 9 has similar results for GPS and Bluetooth sensors since the **max time between recordings** is 8x10^6. Hence, we can assume that the user had the phone off for a few days. On the other hand, if we look at participant 6, the maximum GPS difference in sensor interval is considerably large, but it is not the case for Bluetooth. As a result, there was some fault in the GPS service, such as the service being dead or the location being off. Additionally, we found the **min time between recordings** for most of the participants to be -55 minutes. This negative distance was due to daylight time savings that happened on November 3rd. The first 11 participants registered before that date, but the last 3 completed the registration on a later date. For this reason, the discrepancy is only found in some of the study subjects data. Therefore, we can conclude that the sensor readings were successful during the pilot since most of them happened every 5 minutes.

13.2 Rules performance

Recall the rules used in the study can be found in Table 13.2 which can be summarized into four rule types: time-based, follow-up, GPS (onArrival), and Bluetooth (onDeparture). Notice that some of the rules also have filters attached to them. The filters used in the SONA pilot are time-based and GPS filters. The results of the rule metrics of the SONA pilot are presented in this section. Table 13.4 shows the rule metrics based on time questions, we also present a graph in Fig 13.4 based on the previous metrics, and Table 13.5 shows the average rule metrics for each rule type throughout the study.

Subject performance	Phone performance	Pilot performance
47.60 %	76.19 %	36.26 %

Table 13.4: Results of average subject, phone, and pilot performance metrics.

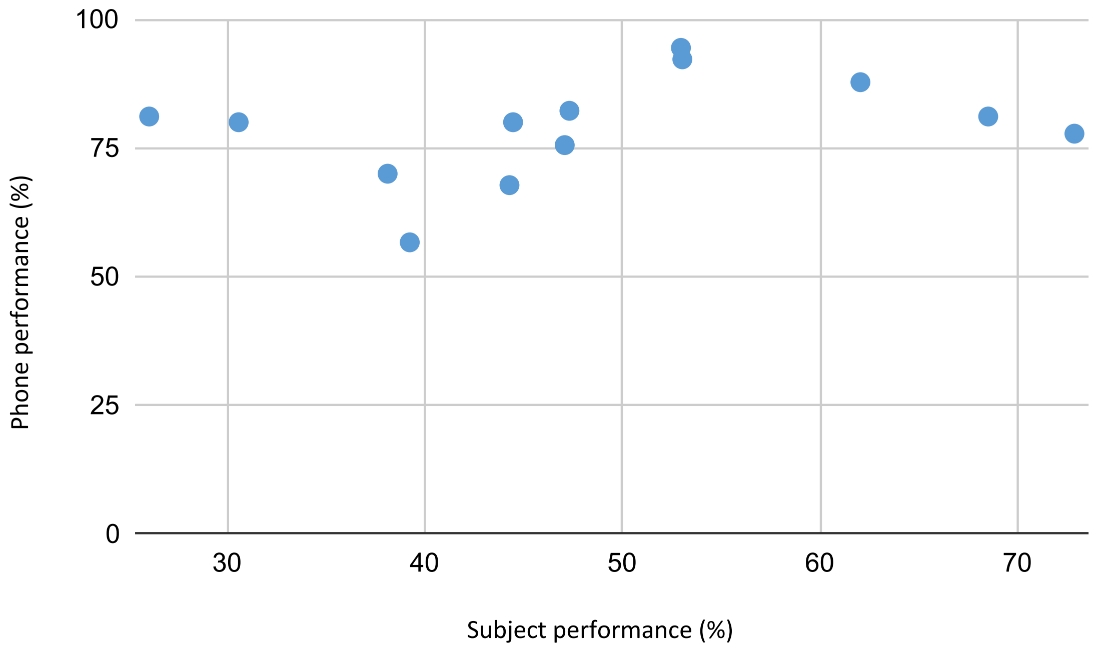


Figure 13.4: Subject and phone performance. The horizontal axis represents the subject performance and the vertical axis represents the phone performance as a percent. The dots represent the study participants.

Avg metrics	Time	GPS	BT	Follow-up
Good ideal rule fired (%)	94.5	10.64	0	100
Good rule fired (%)	90.78	10.21	0	100
Late rule fires (%)	0	70.41	0	0
Missed rule fires (%)	12.5	11.79	100	0
Early rule fires (%)	0.39	0	0	0

Table 13.5: Rules average metrics results among all the participants.

We observe in Fig 13.4, perhaps surprisingly, that the **subject's performance** and the **phone's performance** are proportional in most of the cases. Except for the 5 outliers, there is a center group which is linearly correlated. The 8 points in the center of the plot show that the two metrics are proportional. For example, a participant has a subject's performance lower than 40%, which could be the reason the phone performance is low. The phone's performance decreases if the user does not make sure the phone is on. Hence, since the user's performance is low, we could expect that the participant is not taking care of the phone, which results in low phone performance. Another example is the participant that has the highest phone performance (close to 100%); the subject's performance is relatively high compared with the other participant's phone's performance. It is rare to have a phone performance of 100% because it would mean that the user did not run out of battery nor turned off the phone for 30 days. Low **pilot performance** is more likely to happen if the participants use their phone for the study, but the users tend to forget about it since this was a secondary phone that they had to carry. Additionally, they are college students that are getting class credit only by participating in the study so it does not make any difference to answer questions, which can skew the collected data.

The results from the time-based rules were good in terms of **phone performance**, but **subject performance** was low. These conflicting results are seen in Table 13.4. On the one hand, the results show the phone performance was 76% on average. Note that external factors that can decrease phone performance, e.g the phone running out of battery or the user turning it off. On the other hand, subject performance was a much lower 47%. A reason that the subject performance is low could be due to the limited time (15 minutes) to answer a question. This time interval might not be enough since we need to consider that these are students, and they might be in class at the rule firing time. Clearly, further research is required to improve the subject performance, user experience, and make the user more engaged in ODIN, but this is beyond the scope of this thesis.

The time-based and follow-up rules **good ideal rule fire** and **good rule fire** metrics are high, but the same metrics have low values for GPS and Bluetooth rules. Closer examination led us to discover (after the pilot was over) that this was due to a flaw in the code. There was a bug in the code that calculates the distance between two latitude-longitude coordinates. This bug does not only affect the GPS but also the Bluetooth rules since they have a GPS filter (the IRB required that we only ask students questions if they were on campus, and this was achieved by attaching a GPS filter to the Bluetooth rule). Bluetooth has missed all the rules that should have fired, and we realized after analyzing the data that this was due to the GPS filter with the flawed distance calculation. We can observe in Table 13.5 that cron rule, and follow-up have a **good ideal rule fire** and **good rule fire** above 90%. The reason the cron rule's performance is slightly lower than the follow-up rules is that there is a time conversion on the server. Every time the answers are pushed to the server, the server corrects the timestamps of the phones answers to its own timeframe by adding an appropriate offset based on the difference between the server and phone clock. Currently, we are not taking into account this correction in the data analysis, and this can falsely lower the performance. For example, suppose the participants carry phones that have time set to 9 am, but the current time in the server is 9:10 am (a 10 minute discrepancy). When there is a rule firing on the phone at 9 am, the server changes it to 9:10 am, and the analysis would consider this rule to have fired 10 minute late, when in fact from the vantage point of the phone clock, it fired perfectly on time.

The results from the SONA pilot show that ODIN is a working system although there are still some flaws in the code that need to be addressed. The SONA pilot had four different rule types, and two sensors (GPS and Bluetooth). The results show that the sensor recordings happened in the intervals determined. Moreover, the rules fired as expected, except for GPS rules which also caused Bluetooth rules to not fire as expected. This behavior was due to a flaw in the calculation of the distance between the two coordinates. Finally, we compare the performance of the phone and the subject performance on the time-based questions. We can conclude based on the metrics results that ODIN is a working well, but there are metrics that need to be improved such as the GPS rules. As the ODIN system continues to be developed, we can use the framework of metrics presented in this chapter in future pilots to assess if system performance has improved.

Chapter 14

Conclusion and future work

The study of social systems has evolved throughout the years, starting from population studies using surveys, to methods that build network snapshots. Traditional surveys involve high expenses and low data quality. New techniques are necessary to collect fine-grained long-term data in order to develop mathematical models which can lead to a better understanding of the complexity of human societies and the individuals within them. For this reason, there has been a lot of effort devoted to developing new ways to collect data on population samples, including methods such as EMA which produce better quality fine-grained data over long time scales.

We have developed ODIN, a cell-phone based platform that allows researchers to create *responsive* EMA studies that yield information about both individuals and the networks of interaction between them. The system consists of a Web UI where the researcher can create a study with questions and contextual rules that determine when the questions are prompted. Then, the participant is given a coupon to enter in the phone application where contextual questions are asked based on the study requirements. A pilot study was conducted and shows that the phone's average performance is above 75%, which means that out of 100 expected questions, the system asked 75 over questions. Additionally, we perform an analysis of the sensor readings and rule firings. The sensor interval recording was exact the vast majority of the time. The time-based rule and the follow-up rules also fired as expected. The GPS and Bluetooth rules had low metric values, due to an undetected software bug. These results lead us to conclude that the platform is functioning well, but that external factors were at play in decreasing the systems performance metrics (e.g. phones running out of battery, a lack of suitable incentives to participants, etc.).

Although ODIN successfully collected data and prompted questions for 30 days, there is still work that needs to be done to improve the system and its overall performance.

One open problem for ODIN is the design of meaningful performance metrics and incentive thresholds. For complex rules, it is difficult to know how many times the rule should have fired for any given participant. Even if this number is known and computable, it may vary widely across participants in the same study. Typically, researchers specify that some minimum percentage of questions have to be answered for the participants to receive incentive payment. This is complicated in ODIN due to the uncertainty of the expected number of questions that will be asked. If the researcher only chooses time-based questions (which is essentially traditional EMA), then it is possible to calculate the performance because the expected number of questions is predictable. The implications of complex rules on performance metrics and incentive thresholds is an important area of future research.

Another open problem for ODIN is maintaining and updating the system to keep pace with new Android releases and vendor-specific constraints. A new Android OS is released every year, and some older or newer models might not be compatible with the system. Hence, there is a need to continue testing the phone application with new OS versions and examine that it is still compatible with older versions if any changes are made.

Another open area for ODIN is the support of new types of sensors and new types of rules. This is needed to enable ODIN to be used in a wider range of studies. To date, we have developed a platform with the following sensors: GPS, Proximity Bluetooth, Beacon, Activity Recognition, and Empatica E4. There exist multiple rules associated with each sensor. Nevertheless, new sensors and rules need to be added to the system corresponding to new biosensors and research use-cases. Perhaps there needs to be a way for the researcher to define new types of rules using a visual programming language.

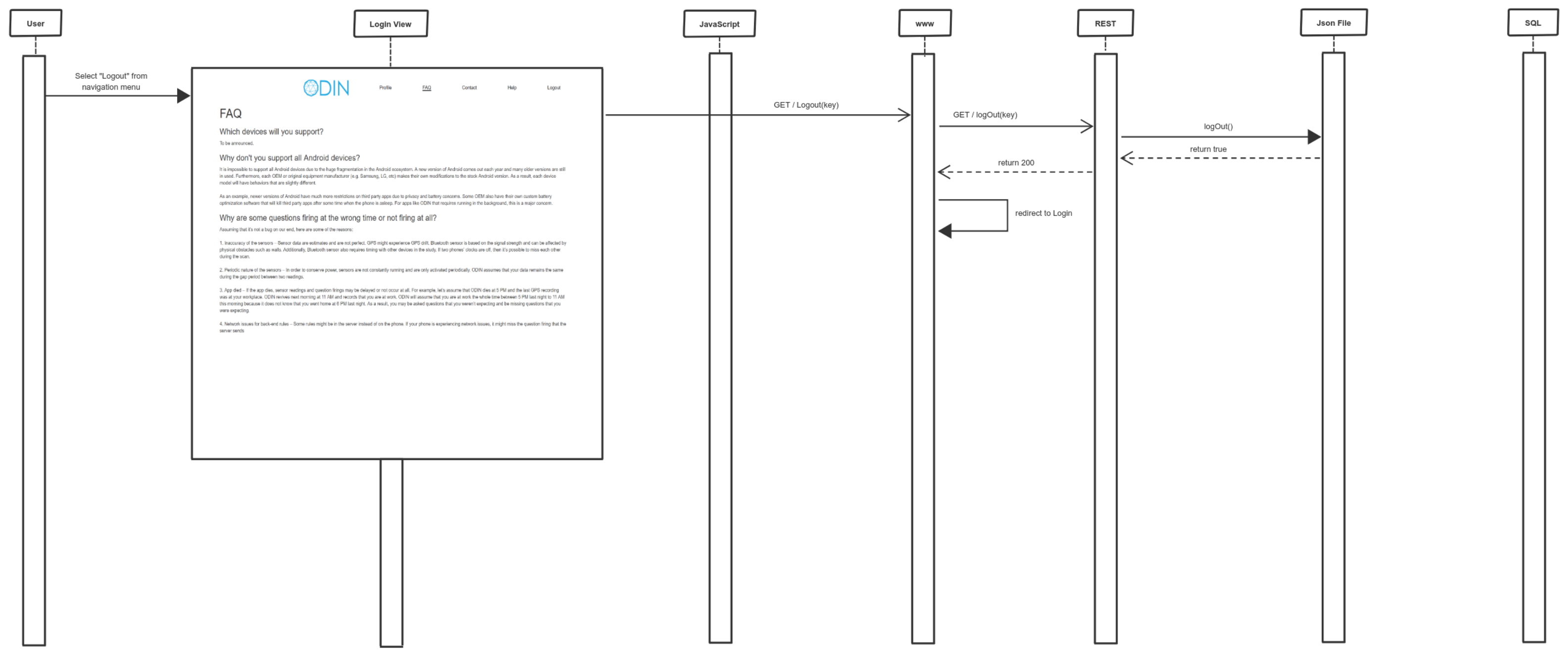
Another open area for future work is the modeling of data collected using ODIN. At the end of the study, the researcher has a large amount of sensor data and answers. Sophisticated data analysis will be required to reach rigorous conclusions from study data. Techniques to make sense of longitudinal interaction data are still in their infancy. Missing data is a statistically difficult issue in the ODIN studies, since the missingness is likely to be missing not at random (MNAR). It is difficult to imagine that the analytic tools that will be developed in the upcoming years will be universally applicable to any ODIN study (simply because of the vast variety of ODIN studies that can be made). Rather, the analytic procedures will probably have to be customized for each study.

Another open area for ODIN are issues of privacy. In general, given a rule-based survey protocol, it is very difficult to know whether information can leak out, especially if subjects come to understand the rules. A simple caricature example might be: On Monday, ask each subject in the study if they have HIV. From Tuesday onwards, whenever two people are in Bluetooth proximity who both answered yes on Monday, ask them question X, whereas whenever two people who answered no are in Bluetooth proximity, ask them question Y. Clearly, such a protocol leaks information, and if study participants knew the rules (and their own HIV status), they could figure out everyone elses. Most scientific studies require the protocol to be approved by an

IRB, which seeks to evaluate the questions to determine if the subjects can be harmed because of private information being revealed. When the study protocol consists of not only questions, but rules as well, it is not clear how such a determination can be made.

To conclude, ODIN is a novel platform that allows researchers to create responsive EMA studies. We have developed an innovative system with 6 sensors and multiple rules, and it is possible to extend it. The current version has already been tested, and it is fully functional. We expect that ODIN will be used in many other research projects in the future where it will facilitate data collection and analysis. There are, however, still many difficult open problems to be resolved in the long term.

Appendix



FAQ

Which devices will you support?

To be announced.

Why don't you support all Android devices?

It is impossible to support all Android devices due to the huge fragmentation in the Android ecosystem. A new version of Android comes out each year and many older versions are still in used. Furthermore, each OEM or original equipment manufacturer (e.g. Samsung, LG, etc) makes their own modifications to the stock Android version. As a result, each device model will have behaviors that are slightly different.

As an example, newer versions of Android have much more restrictions on third party apps due to privacy and battery concerns. Some OEM also have their own custom battery optimization software that will kill third party apps after some time when the phone is asleep. For apps like ODIN that requires running in the background, this is a major concern.

Why are some questions firing at the wrong time or not firing at all?

Assuming that it's not a bug on our end, here are some of the reasons:

1. Inaccuracy of the sensors – Sensor data are estimates and are not perfect. GPS might experience GPS drift. Bluetooth sensor is based on the signal strength and can be affected by physical obstacles such as walls. Additionally, Bluetooth sensor also requires timing with other devices in the study. If two phones' clocks are off, then it's possible to miss each other during the scan.

2. Periodic nature of the sensors – In order to conserve power, sensors are not constantly running and are only activated periodically. ODIN assumes that your data remains the same during the gap period between two readings.

3. App died – If the app dies, sensor readings and question firings may be delayed or not occur at all. For example, let's assume that ODIN dies at 5 PM and the last GPS recording was at your workplace. ODIN revives next morning at 11 AM and records that you are at work. ODIN will assume that you are at work the whole time between 5 PM last night to 11 AM this morning because it does not know that you went home at 6 PM last night. As a result, you may be asked questions that you weren't expecting and be missing questions that you were expecting.

4. Network issues for back-end rules – Some rules might be in the server instead of on the phone. If your phone is experiencing network issues, it might miss the question firing that the server sends

Figure .1: ResearcherUI Account Logout

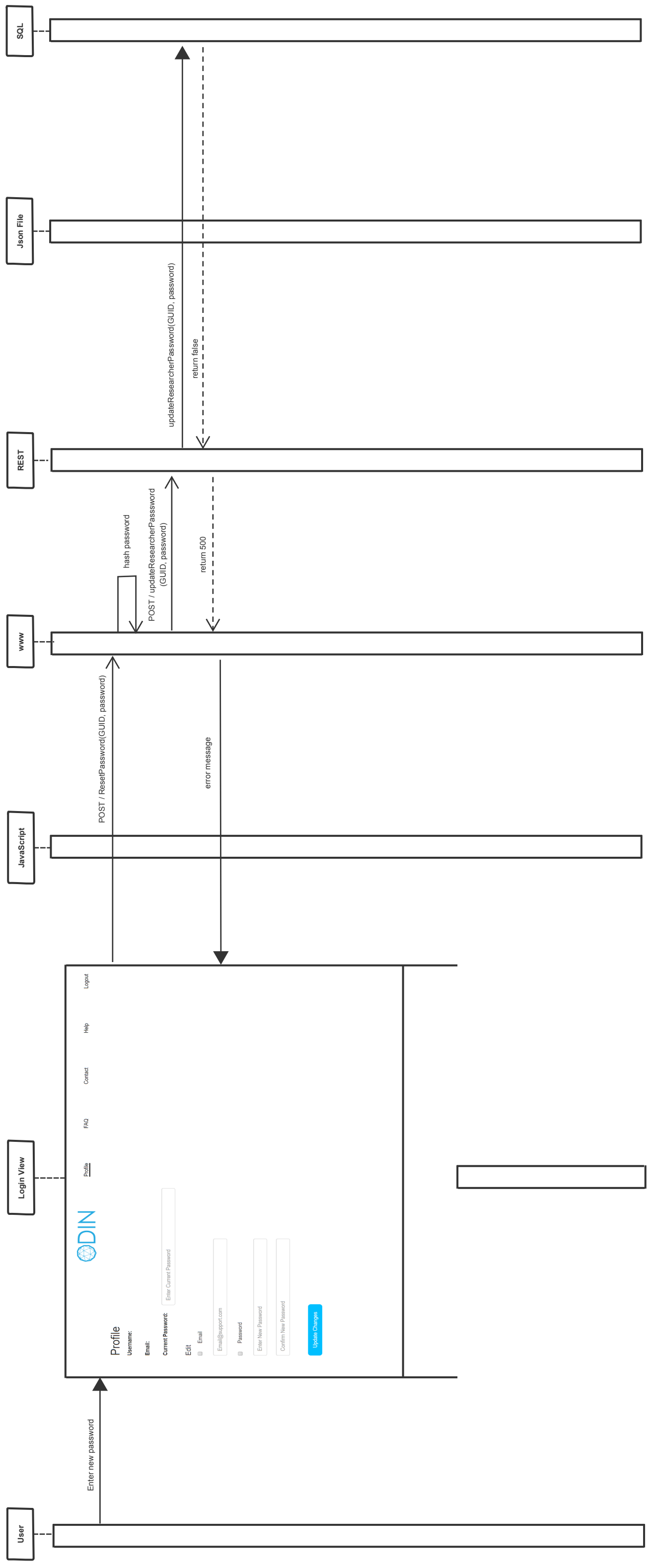


Figure .2: ResearcherUI Account ResetPassword post failure

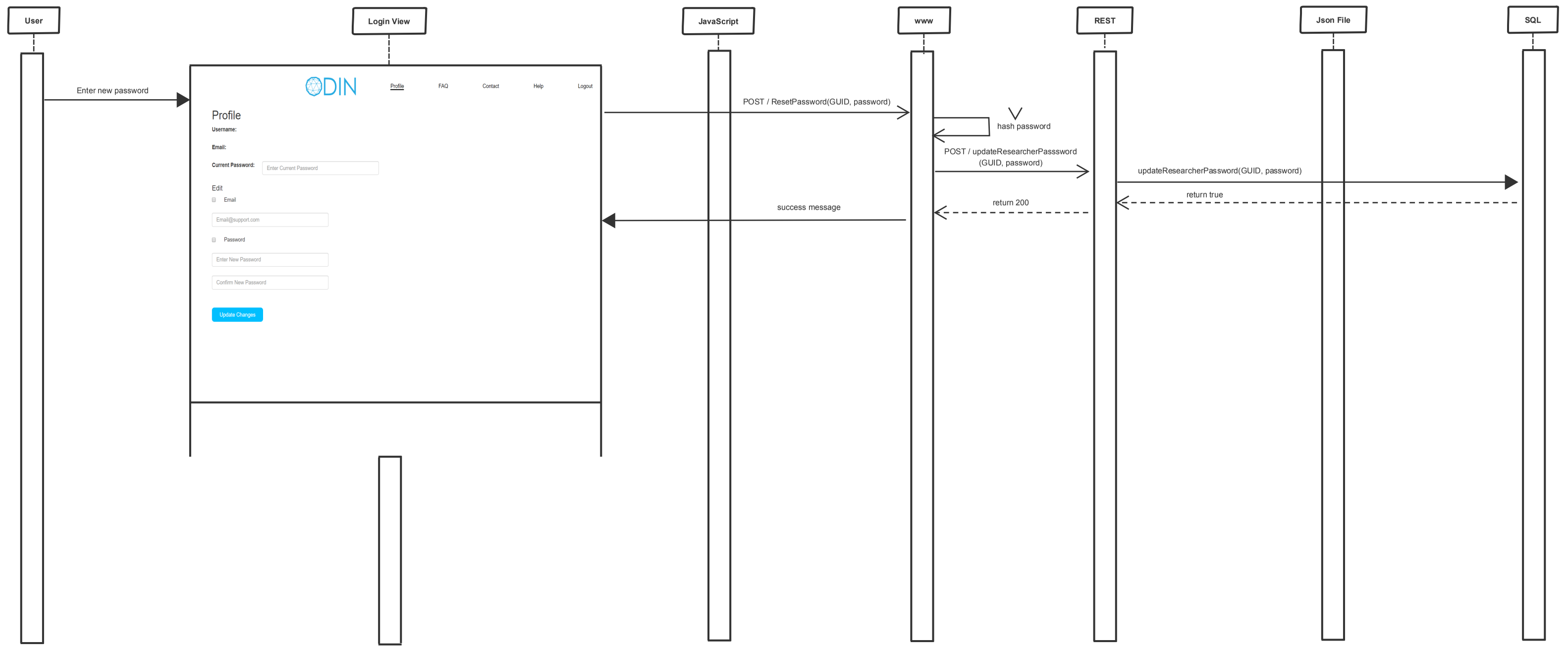


Figure .3: ResearcherUI Account ResetPassword post success

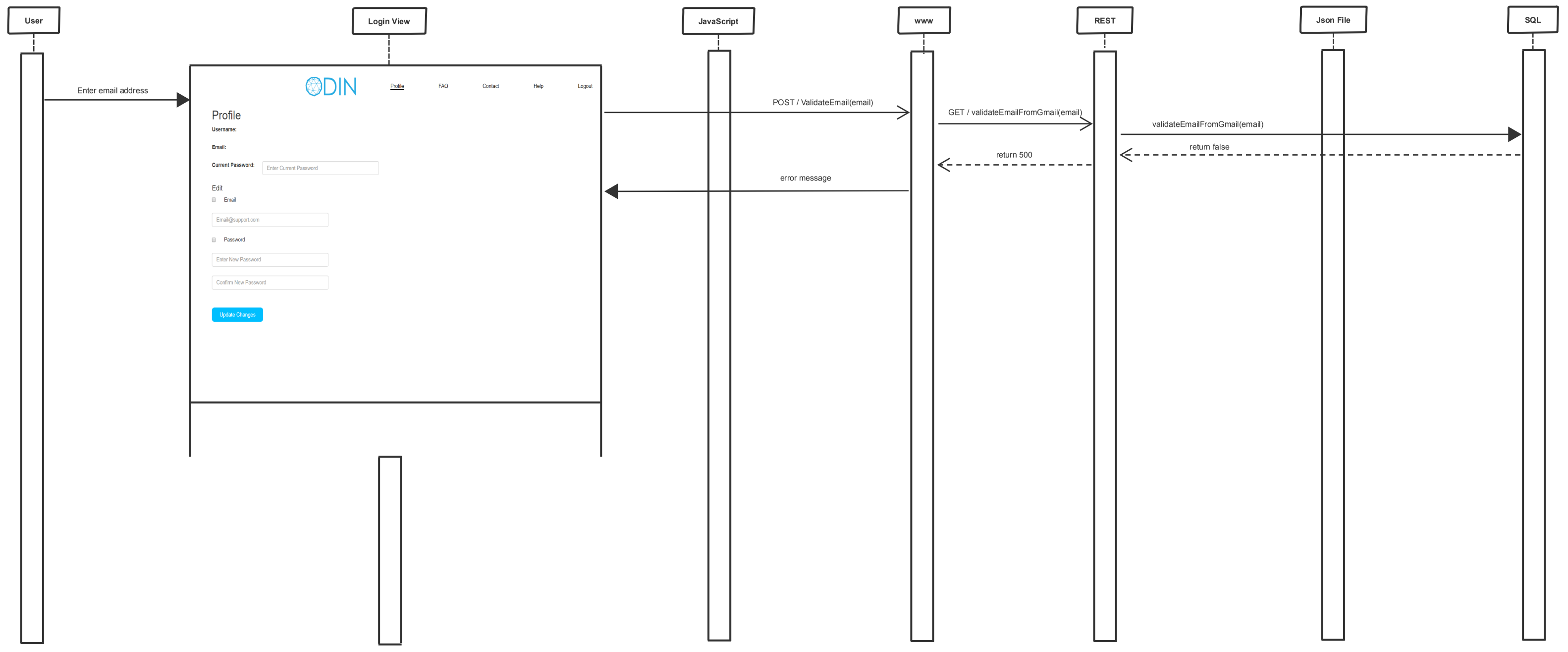


Figure .4: ResearcherUI Account ValidateEmail post failure

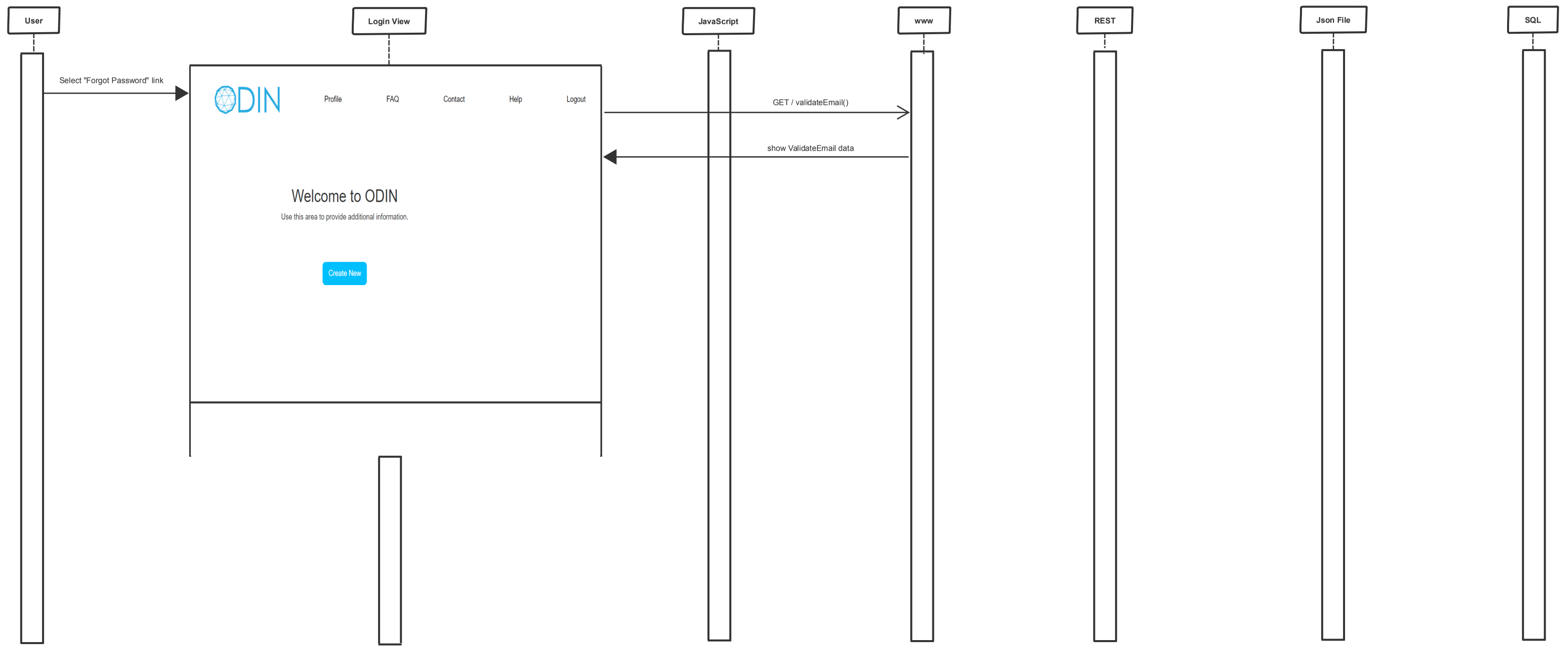


Figure .5: ResearcherUI Account validateEmail get

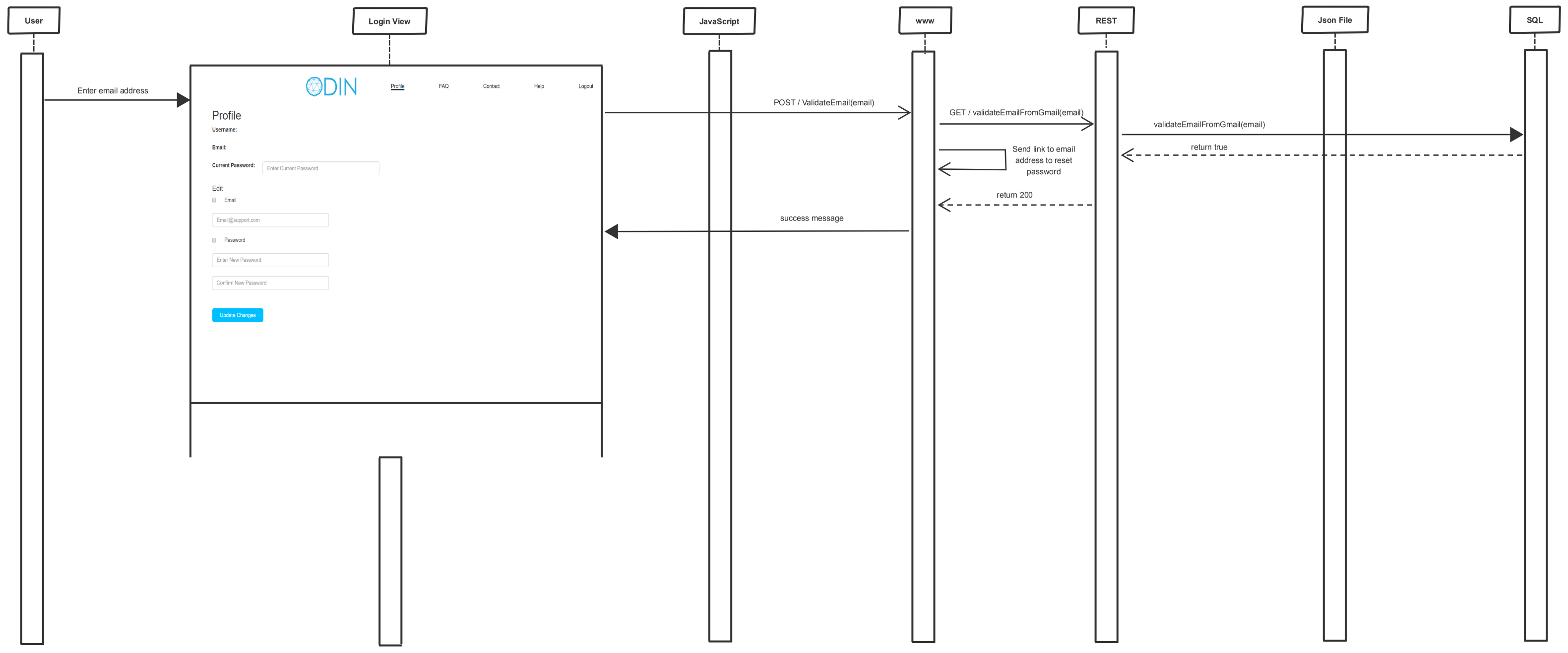


Figure .6: ResearcherUI Account validateEmail post success

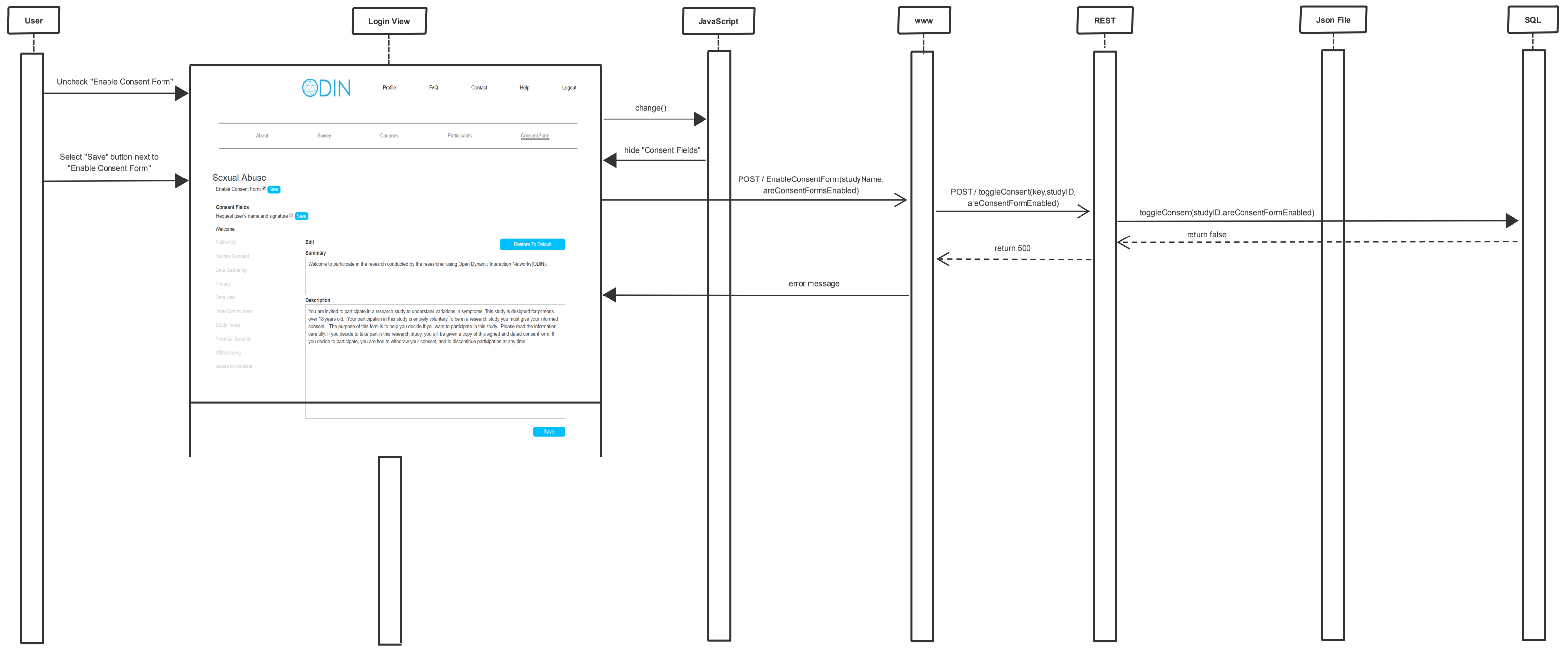
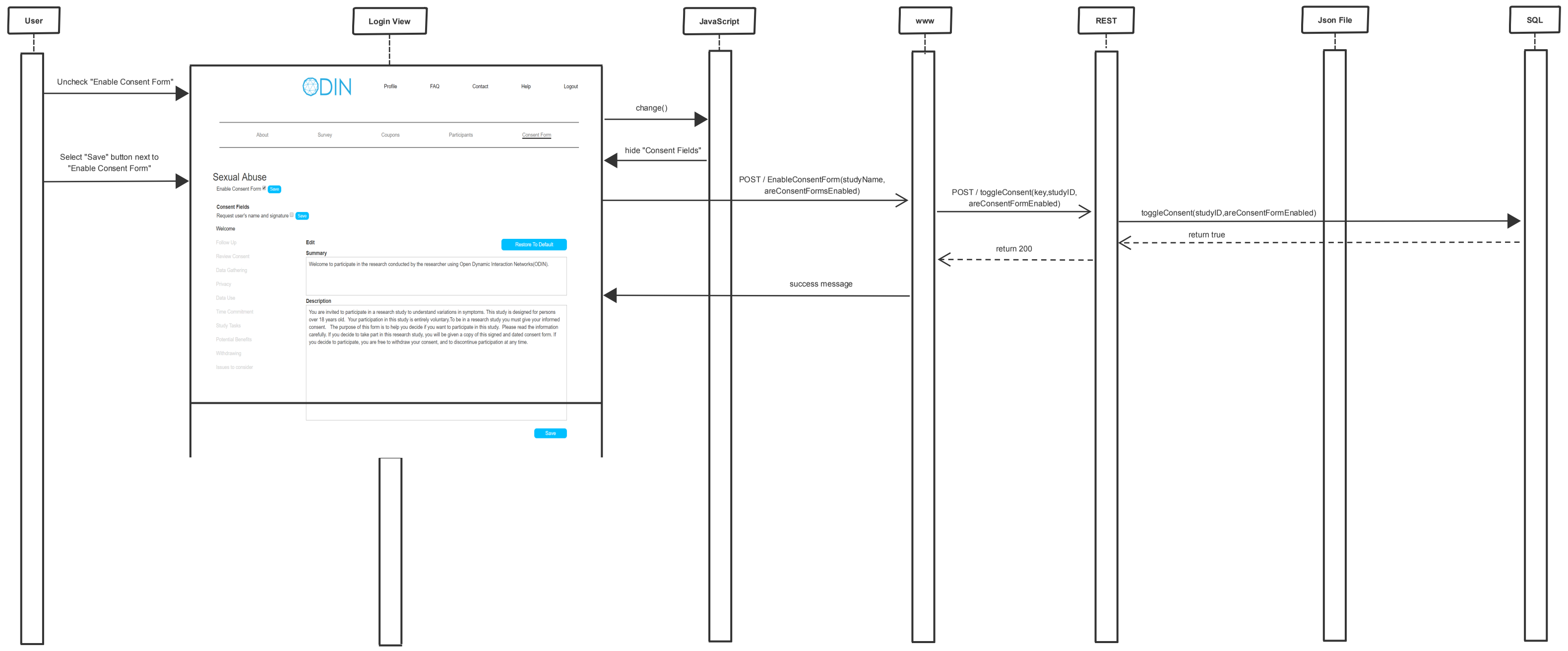


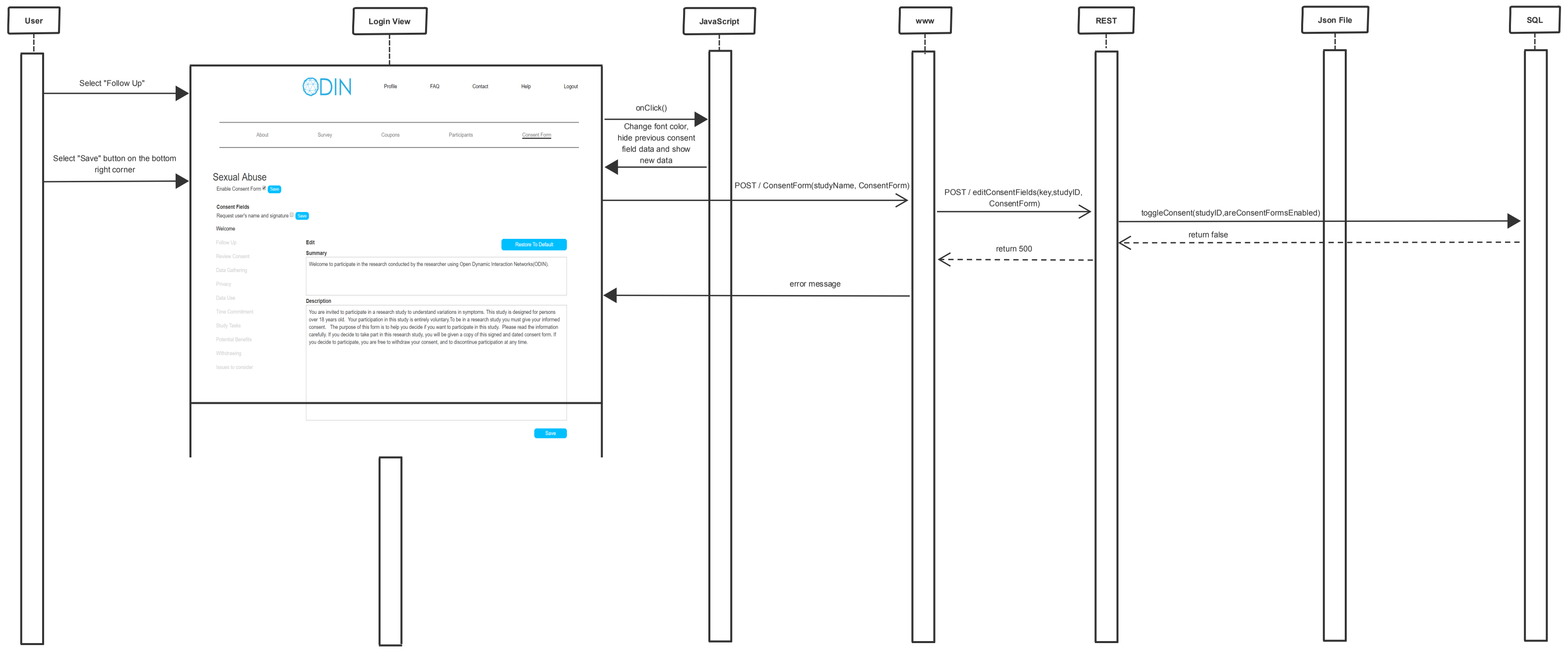
Figure .7: ResearcherUI ConsentForm EnableConsentForm failure



Welcome to participate in the research conducted by the researcher using Open Dynamic Interaction Networks(ODIN).

You are invited to participate in a research study to understand variations in symptoms. This study is designed for persons over 18 years old. Your participation in this study is entirely voluntary.To be in a research study you must give your informed consent. The purpose of this form is to help you decide if you want to participate in this study. Please read the information carefully. If you decide to take part in this research study, you will be given a copy of this signed and dated consent form. If you decide to participate, you are free to withdraw your consent, and to discontinue participation at any time.

Figure .8: ResearcherUI ConsentForm EnableConsentForm success



Sexual Abuse

Enable Consent Form ☑

Consent Fields

Request user's name and signature ☐

Welcome

Edit

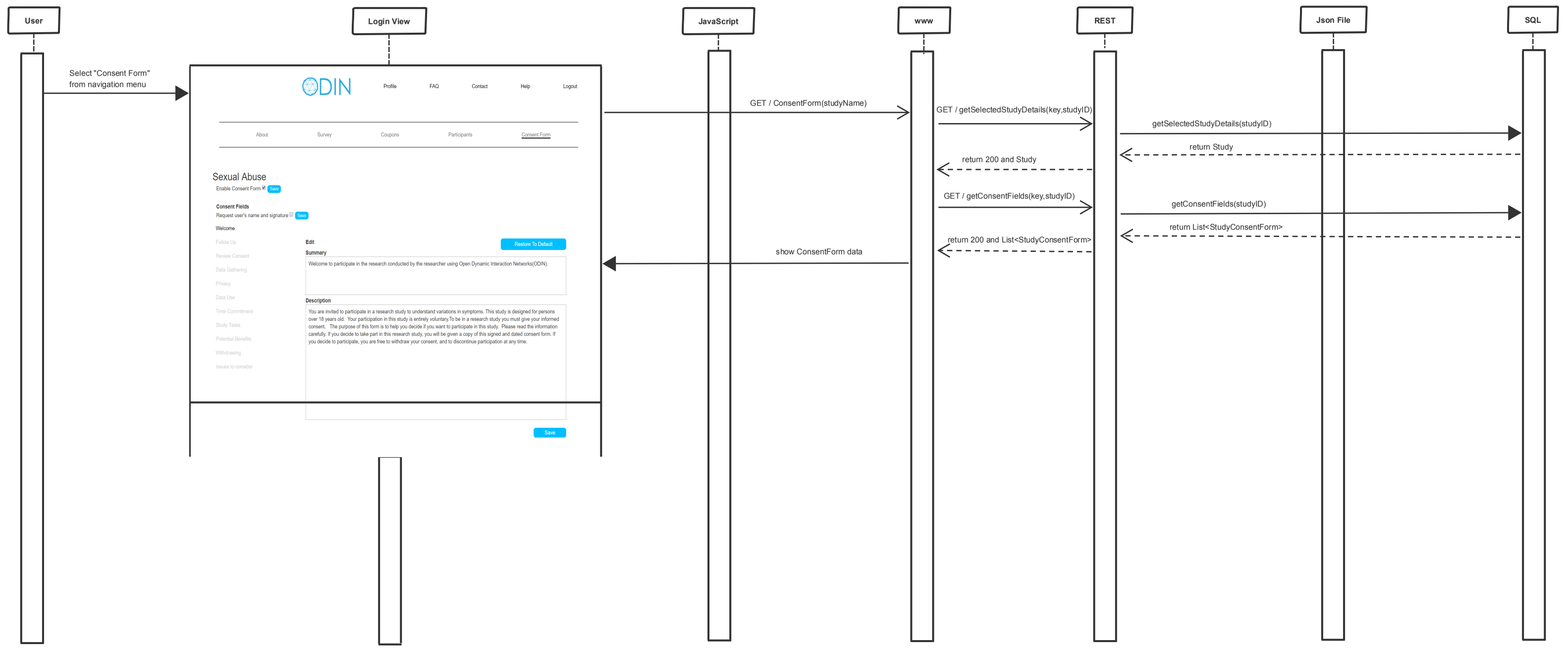
Summary

Welcome to participate in the research conducted by the researcher using Open Dynamic Interaction Networks(ODIN).

Description

You are invited to participate in a research study to understand variations in symptoms. This study is designed for persons over 18 years old. Your participation in this study is entirely voluntary.To be in a research study you must give your informed consent. The purpose of this form is to help you decide if you want to participate in this study. Please read the information carefully. If you decide to take part in this research study, you will be given a copy of this signed and dated consent form. If you decide to participate, you are free to withdraw your consent, and to discontinue participation at any time.

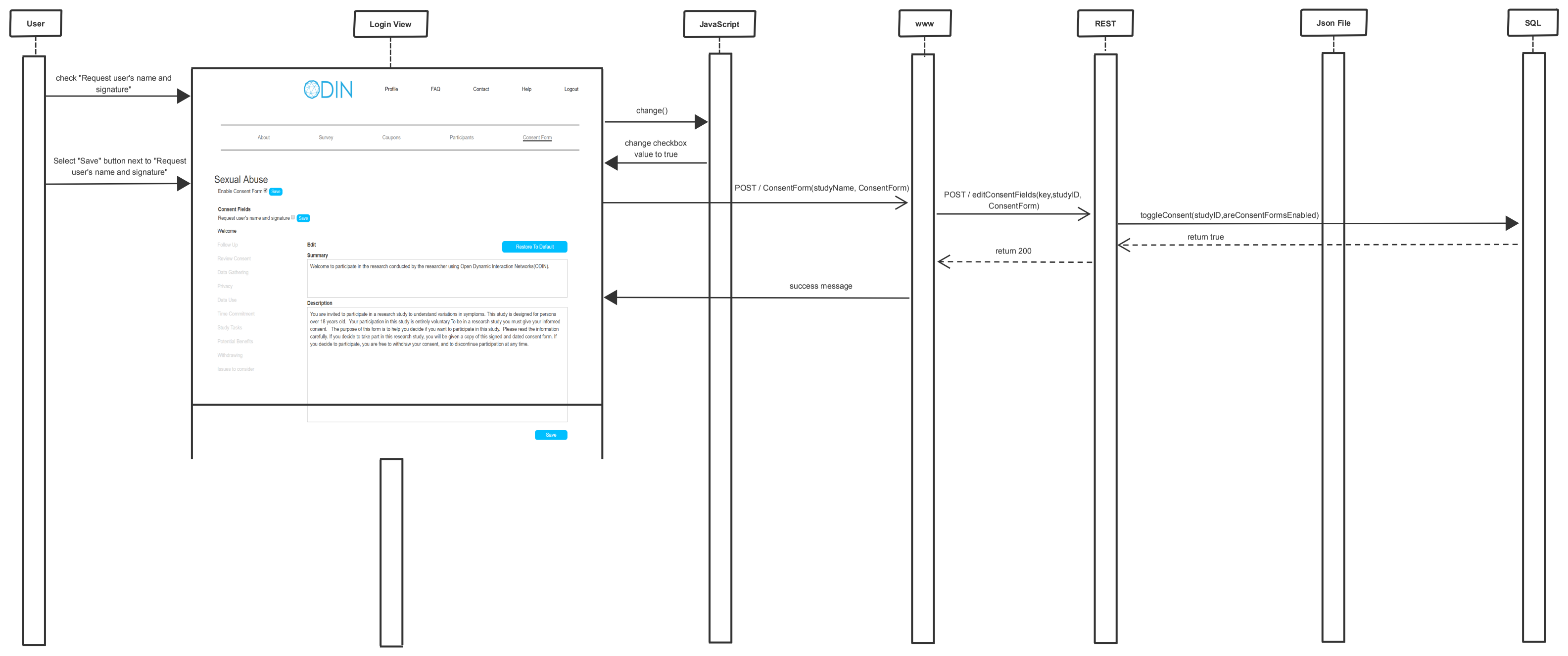
Figure .9: ResearcherUI ConsentForm failure



Welcome to participate in the research conducted by the researcher using Open Dynamic Interaction Networks(ODIN).

You are invited to participate in a research study to understand variations in symptoms. This study is designed for persons over 18 years old. Your participation in this study is entirely voluntary.To be in a research study you must give your informed consent. The purpose of this form is to help you decide if you want to participate in this study. Please read the information carefully. If you decide to take part in this research study, you will be given a copy of this signed and dated consent form. If you decide to participate, you are free to withdraw your consent, and to discontinue participation at any time.

Figure .10: ResearcherUI ConsentForm get



Sexual Abuse

Enable Consent Form ☑

Consent Fields

Request user's name and signature ☐

Welcome

Edit

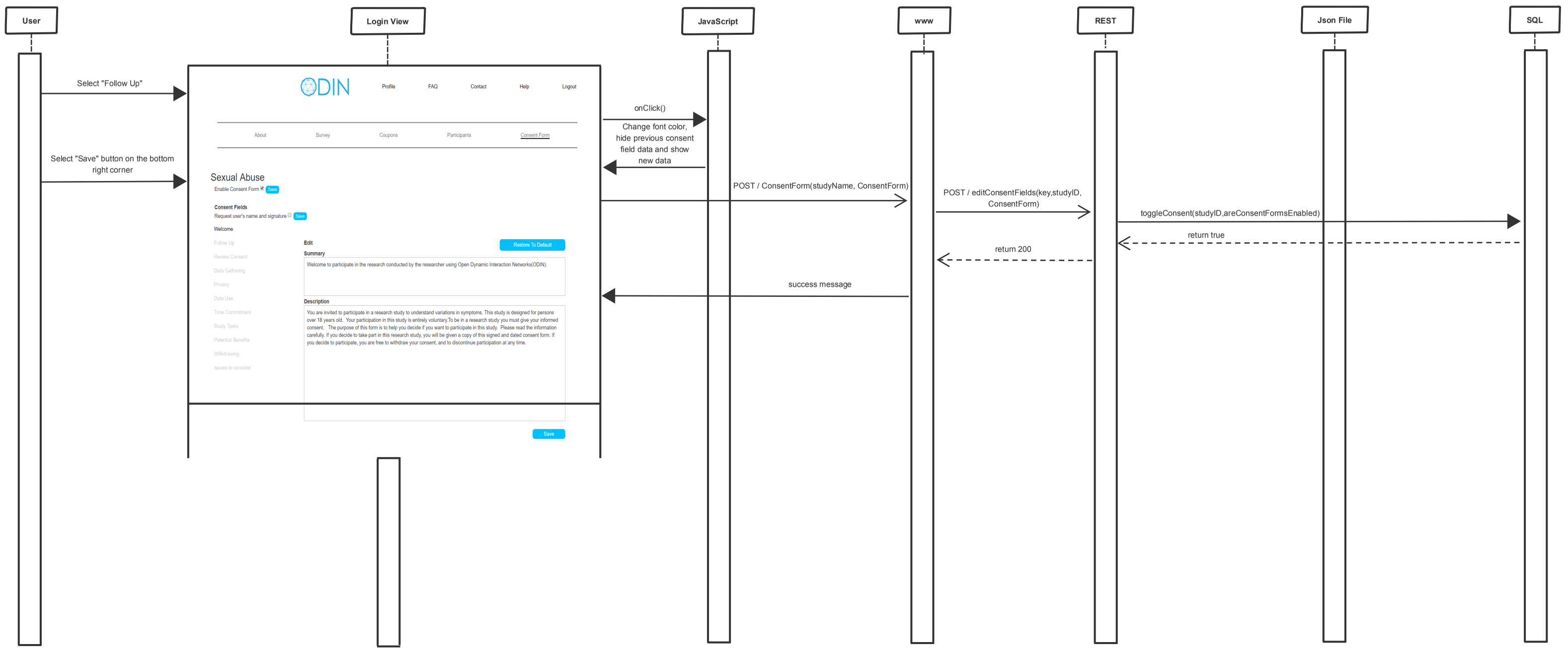
Summary

Welcome to participate in the research conducted by the researcher using Open Dynamic Interaction Networks(ODIN).

Description

You are invited to participate in a research study to understand variations in symptoms. This study is designed for persons over 18 years old. Your participation in this study is entirely voluntary.To be in a research study you must give your informed consent. The purpose of this form is to help you decide if you want to participate in this study. Please read the information carefully. If you decide to take part in this research study, you will be given a copy of this signed and dated consent form. If you decide to participate, you are free to withdraw your consent, and to discontinue participation at any time.

Figure .11: ResearcherUI ConsentForm post success 1



Sexual Abuse

Enable Consent Form ☑

Consent Fields

Request user's name and signature ☐

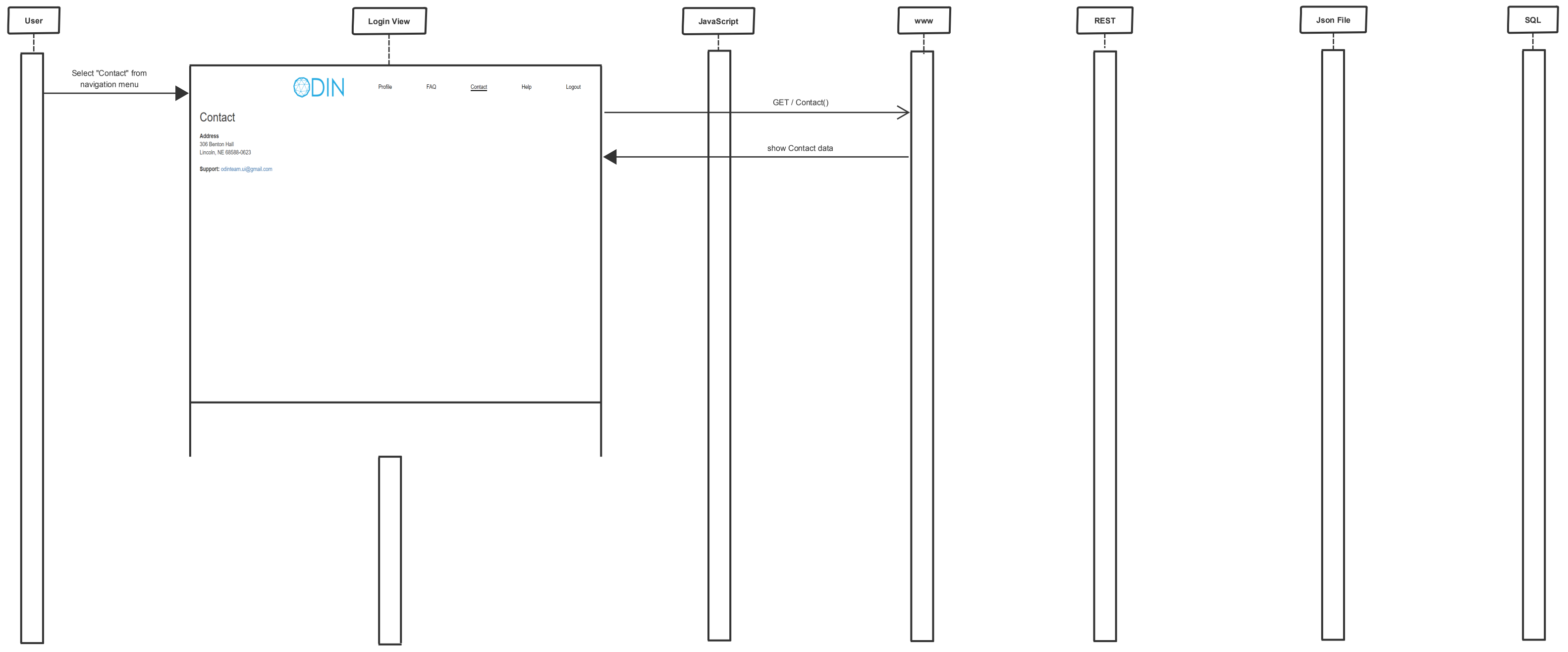
Summary

Welcome to participate in the research conducted by the researcher using Open Dynamic Interaction Networks(ODIN).

Description

You are invited to participate in a research study to understand variations in symptoms. This study is designed for persons over 18 years old. Your participation in this study is entirely voluntary.To be in a research study you must give your informed consent. The purpose of this form is to help you decide if you want to participate in this study. Please read the information carefully. If you decide to take part in this research study, you will be given a copy of this signed and dated consent form. If you decide to participate, you are free to withdraw your consent, and to discontinue participation at any time.

Figure .12: ResearcherUI ConsentForm post success 2



Contact

Address
306 Benton Hall
Lincoln, NE 68588-0623

Support: odinteam.ui@gmail.com

Figure .13: ResearcherUI Contact

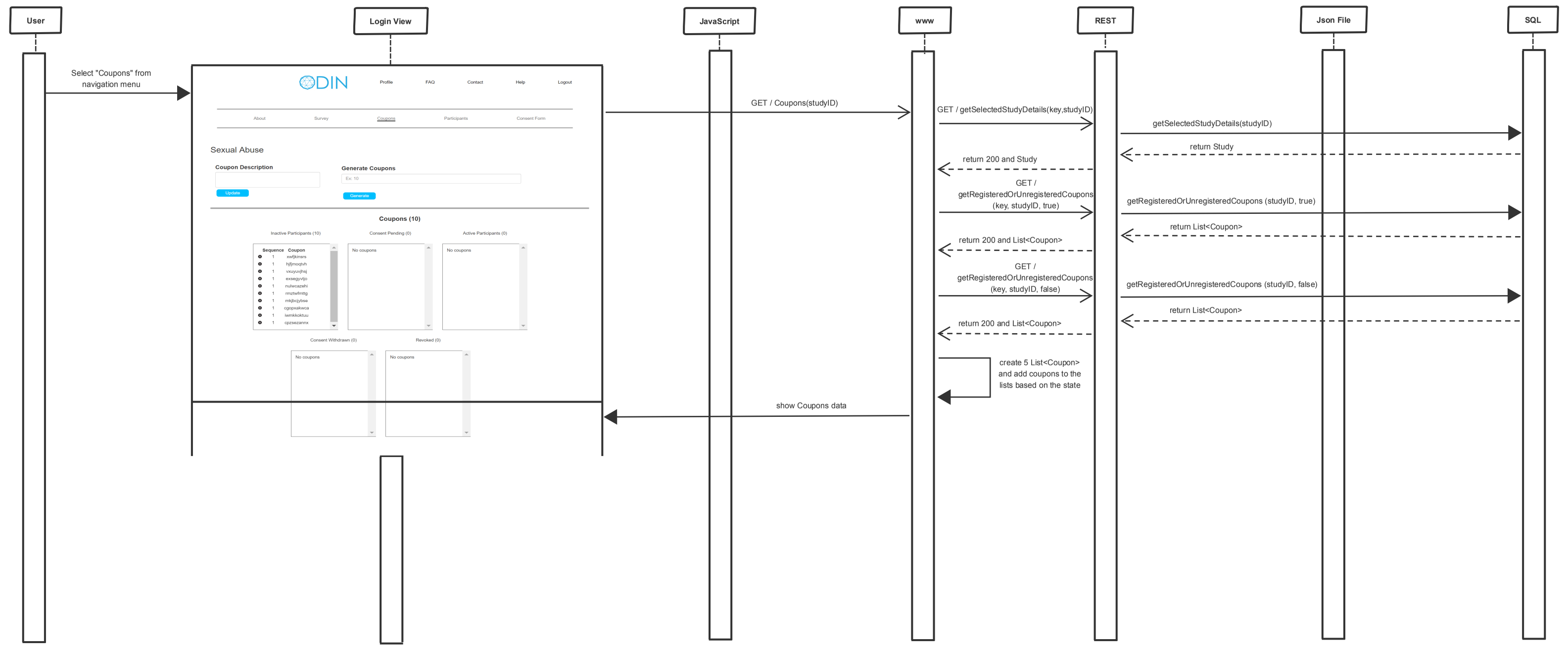

User
Login View
JavaScript
www
REST
Json File
SQL
Select "Coupons" from navigation menu
GET / Coupons(studyID)
GET / getSelectedStudyDetails(key,studyID)
getSelectedStudyDetails(studyID)
return Study
return 200 and Study
GET / getRegisteredOrUnregisteredCoupons (key, studyID, true)
getRegisteredOrUnregisteredCoupons (studyID, true)
return List<Coupon>
return 200 and List<Coupon>
GET / getRegisteredOrUnregisteredCoupons (key, studyID, false)
getRegisteredOrUnregisteredCoupons (studyID, false)
return List<Coupon>
return 200 and List<Coupon>
create 5 List<Coupon> and add coupons to the lists based on the state
show Coupons data
ODIN
Sexual Abuse
Coupon Description
Generate Coupons
Coupons (10)

Figure .14: ResearcherUI Coupon get

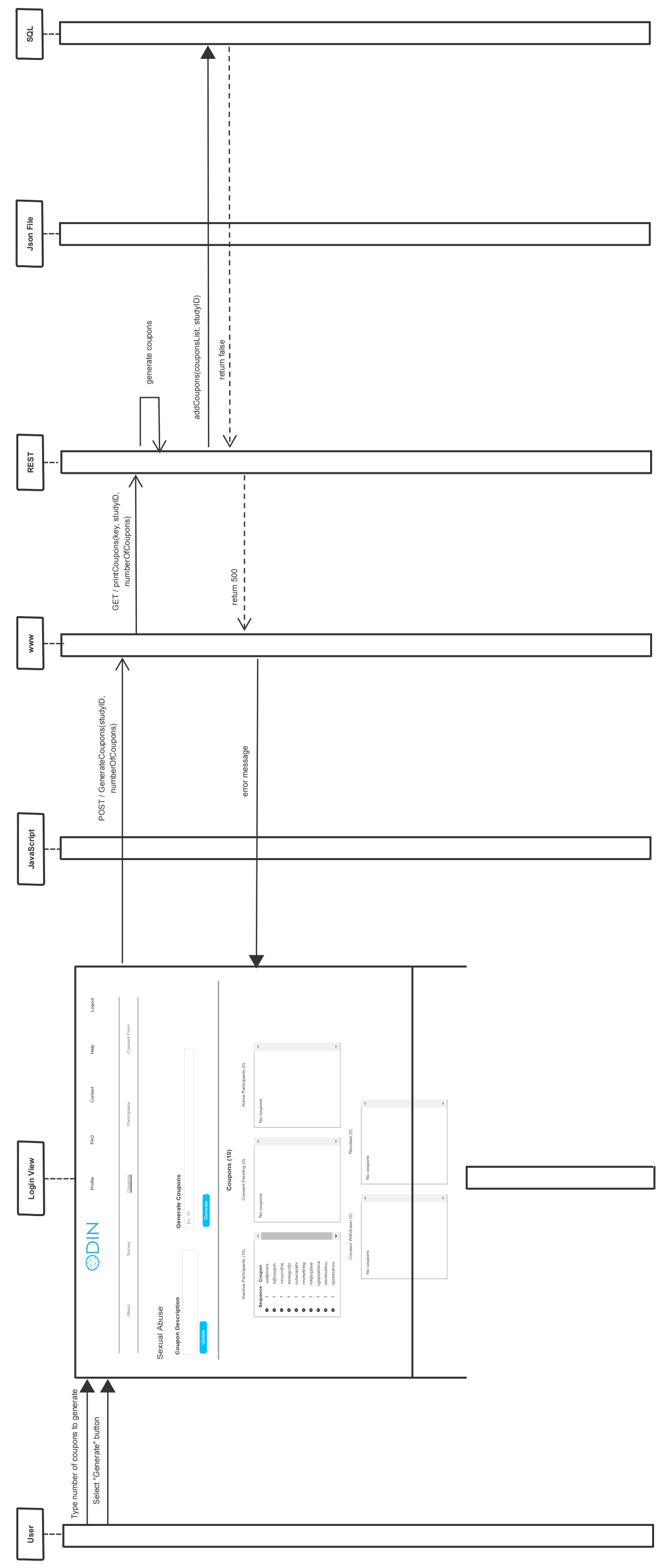

User
Login View
JavaScript
www
REST
Json File
SQL
Type number of coupons to generate
Select "Generate" button
POST / GenerateCoupons(studyID, numberOfCoupons)
error message
GET / printCoupons(key, studyID, numberOfCoupons)
return 500
generate coupons
addCoupons(couponsList, studyID)
return false

Figure .15: ResearcherUI Coupon post failure

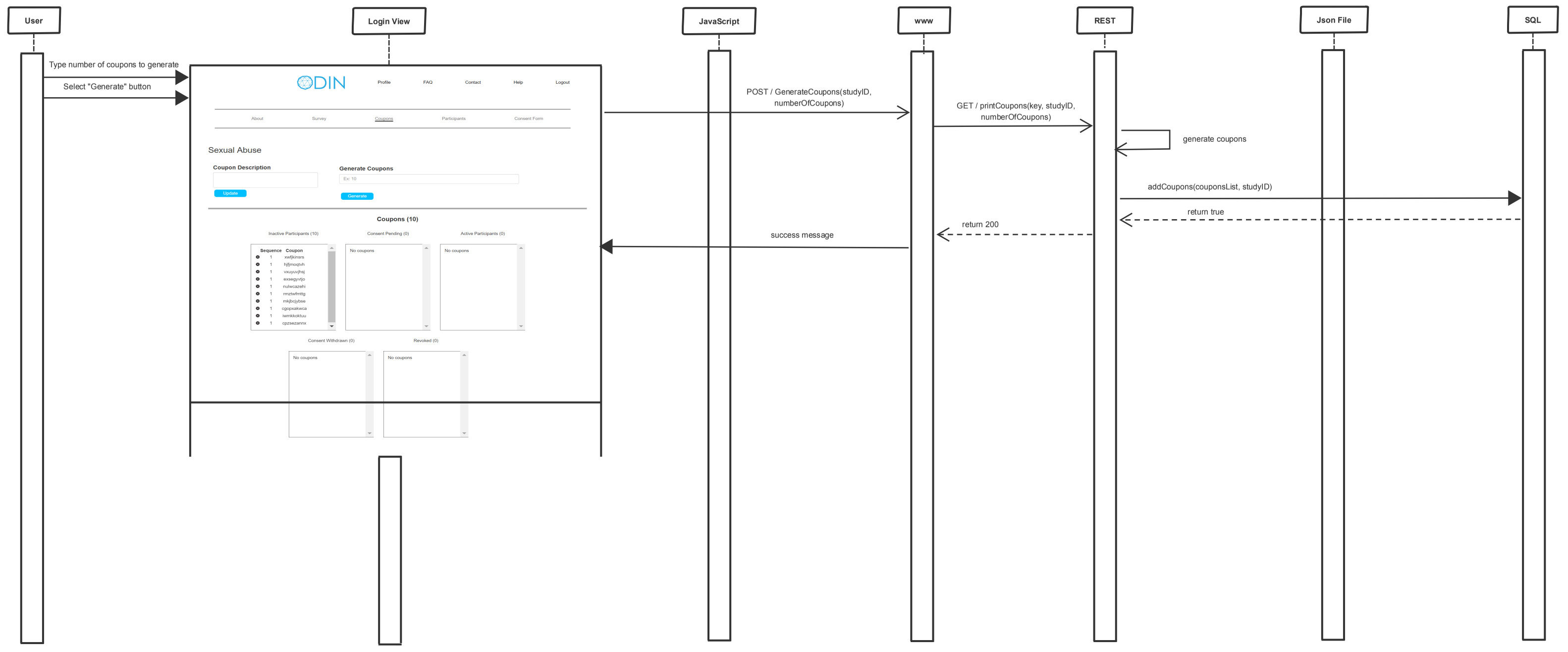


Figure .16: ResearcherUI Coupon post success

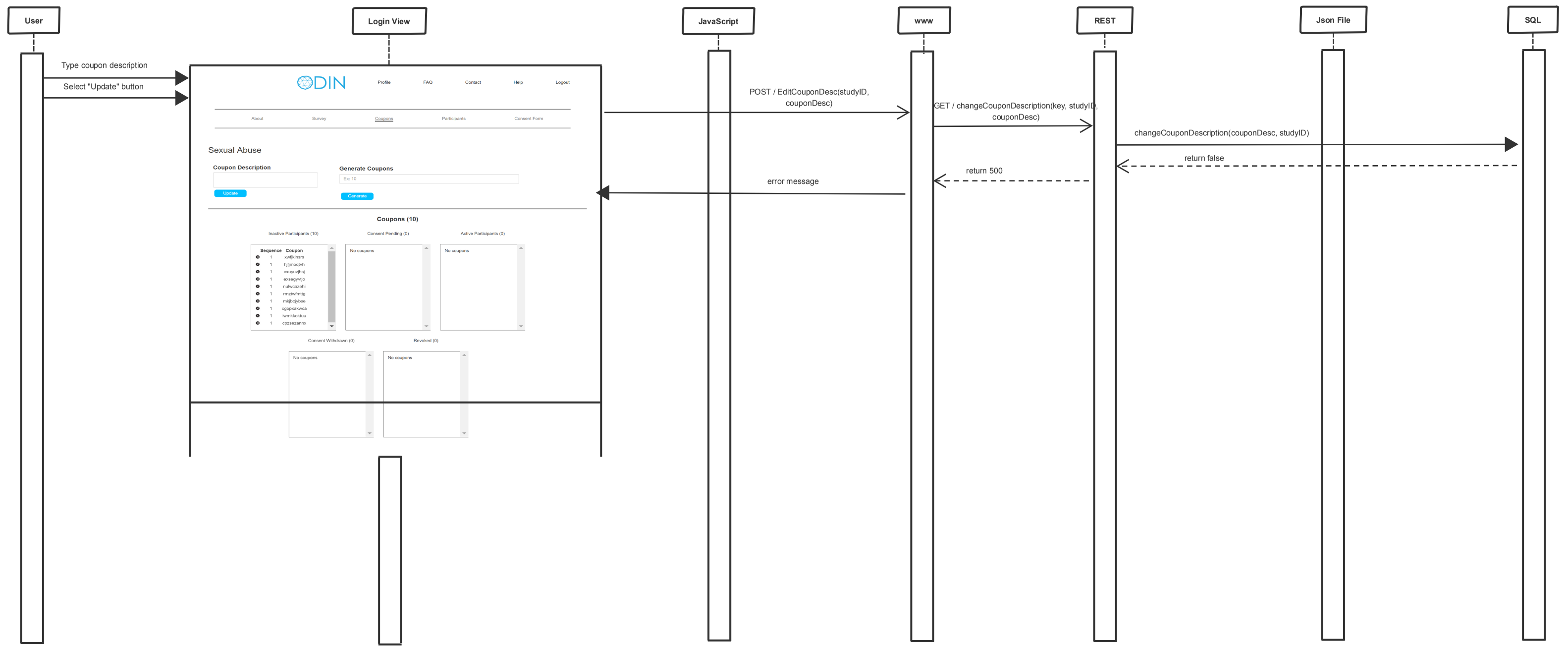


Figure .17: ResearcherUI Coupons EditCoupon failure

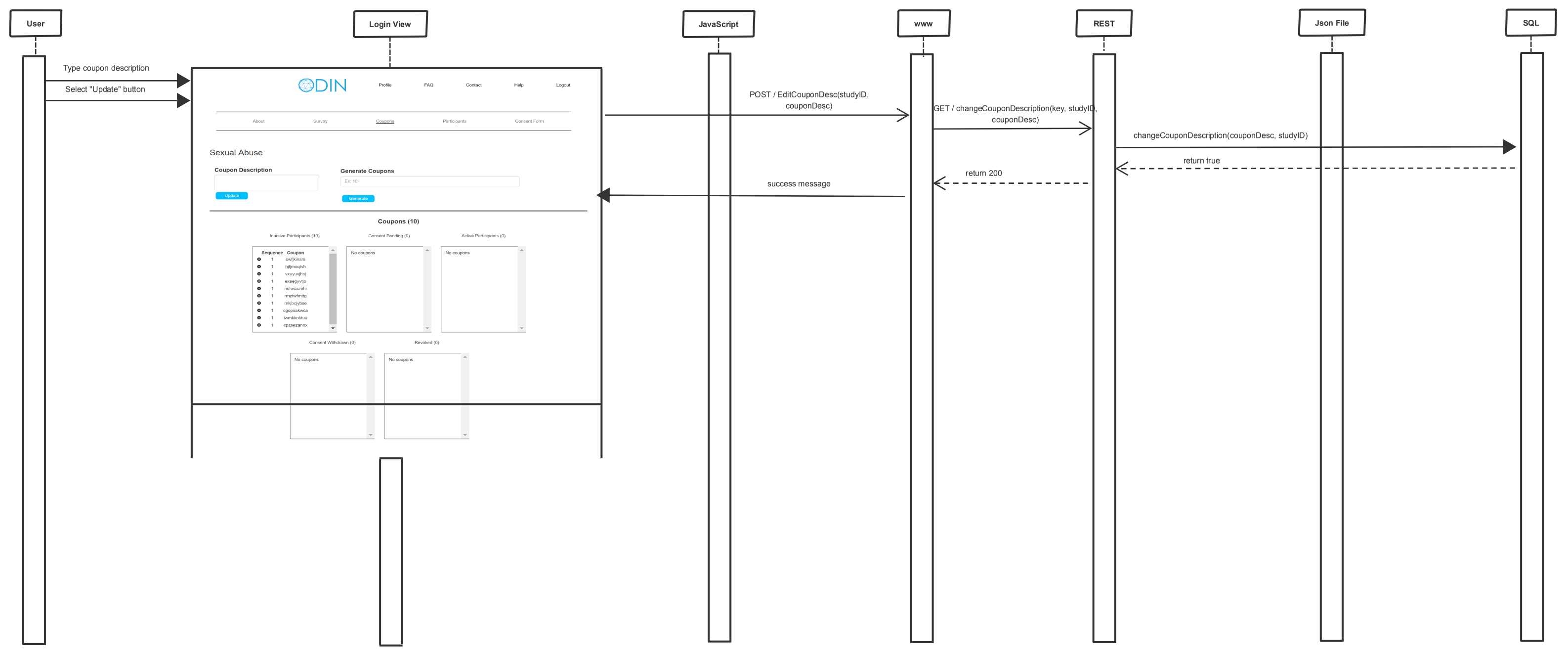


Figure .18: ResearcherUI Coupons EditCoupon success

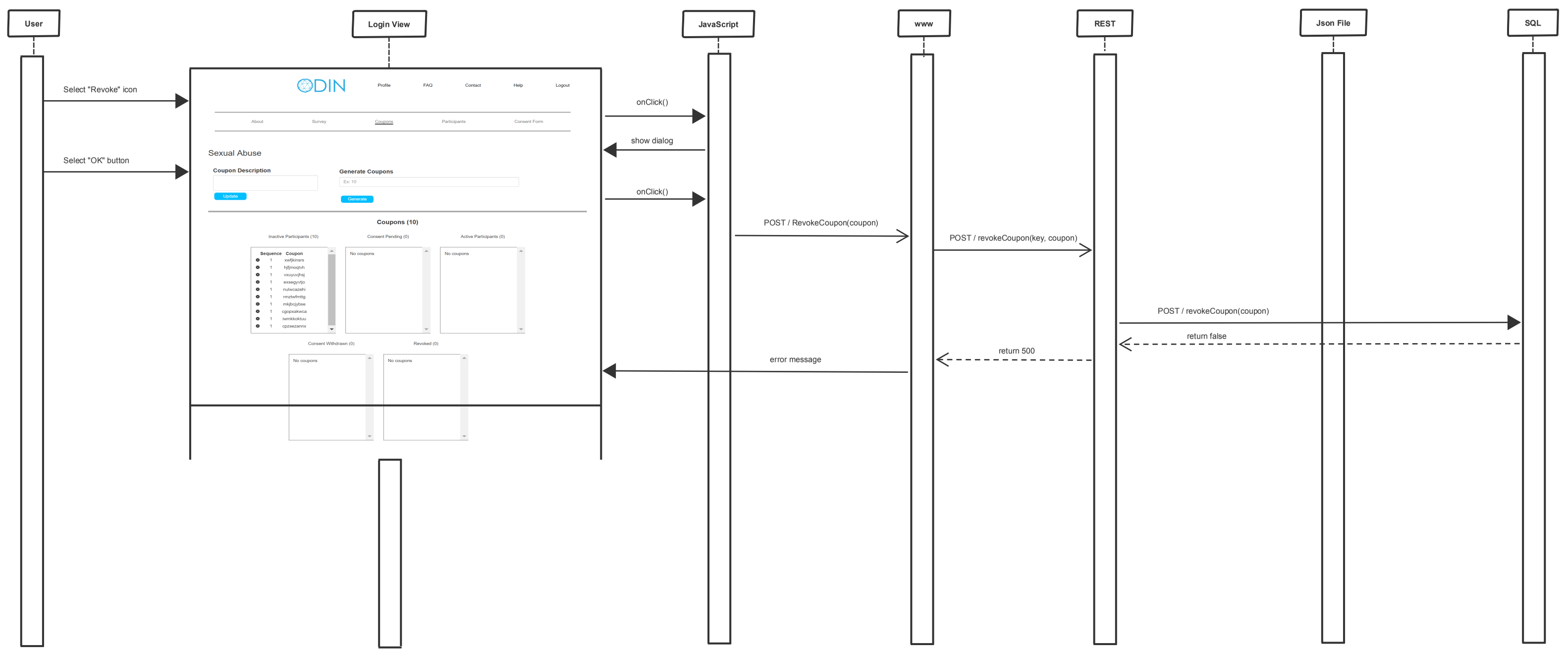


Figure .19: ResearcherUI Coupons Revoke failure

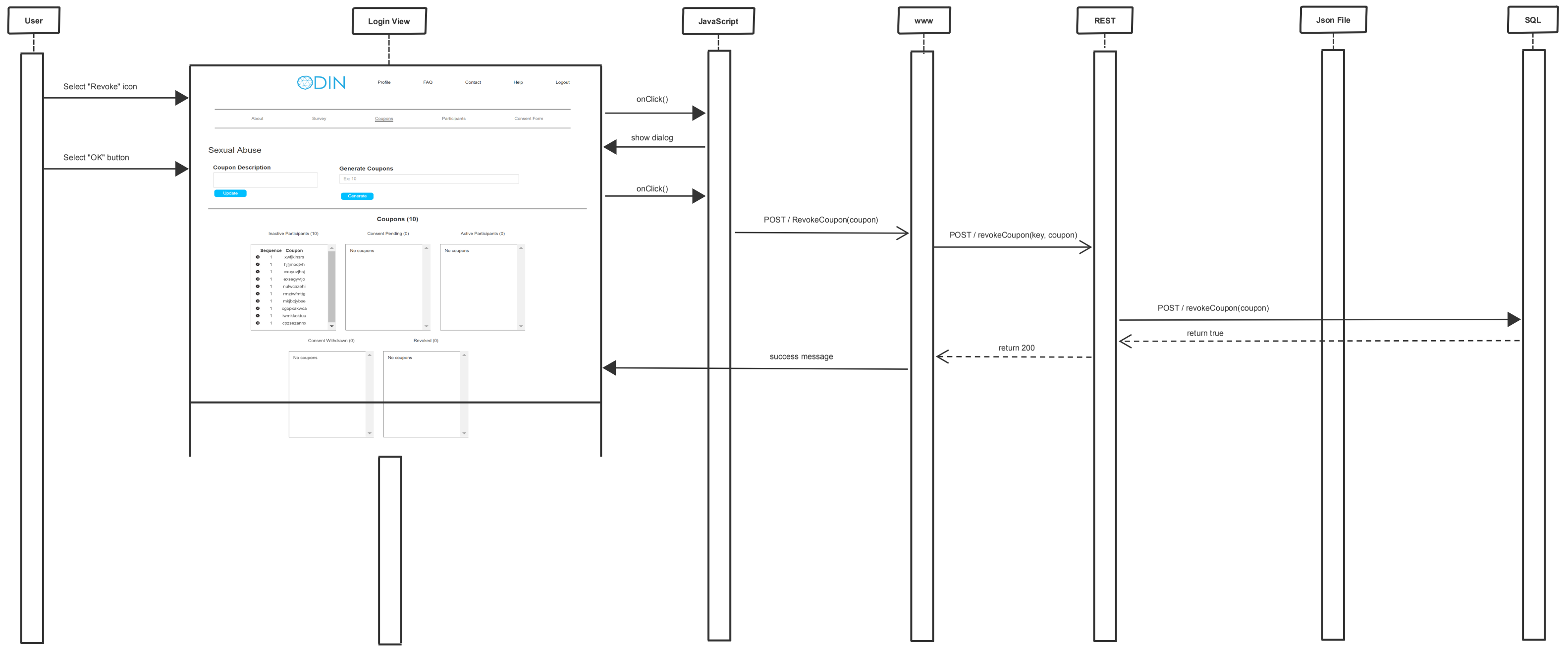


Figure .20: ResearcherUI Coupons Revoke success

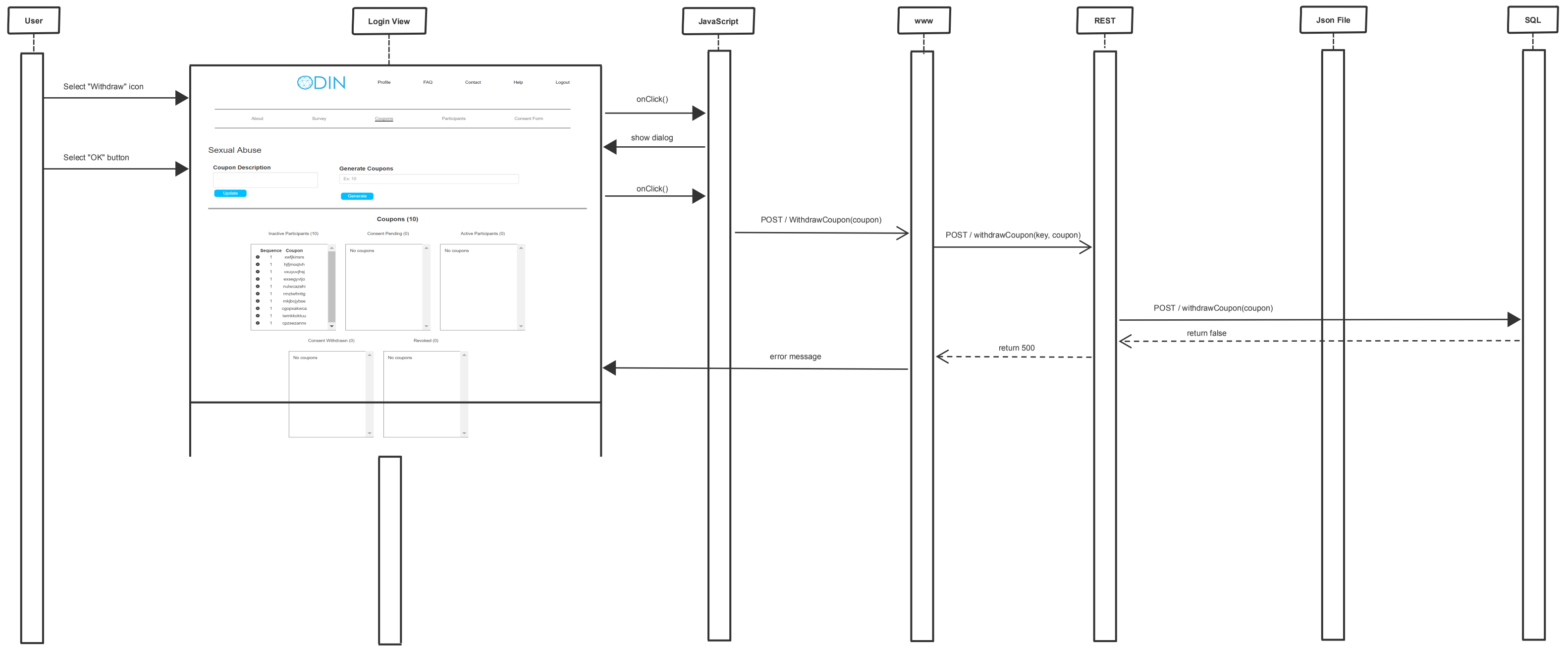


Figure .21: ResearcherUI Coupons Withdraw failure

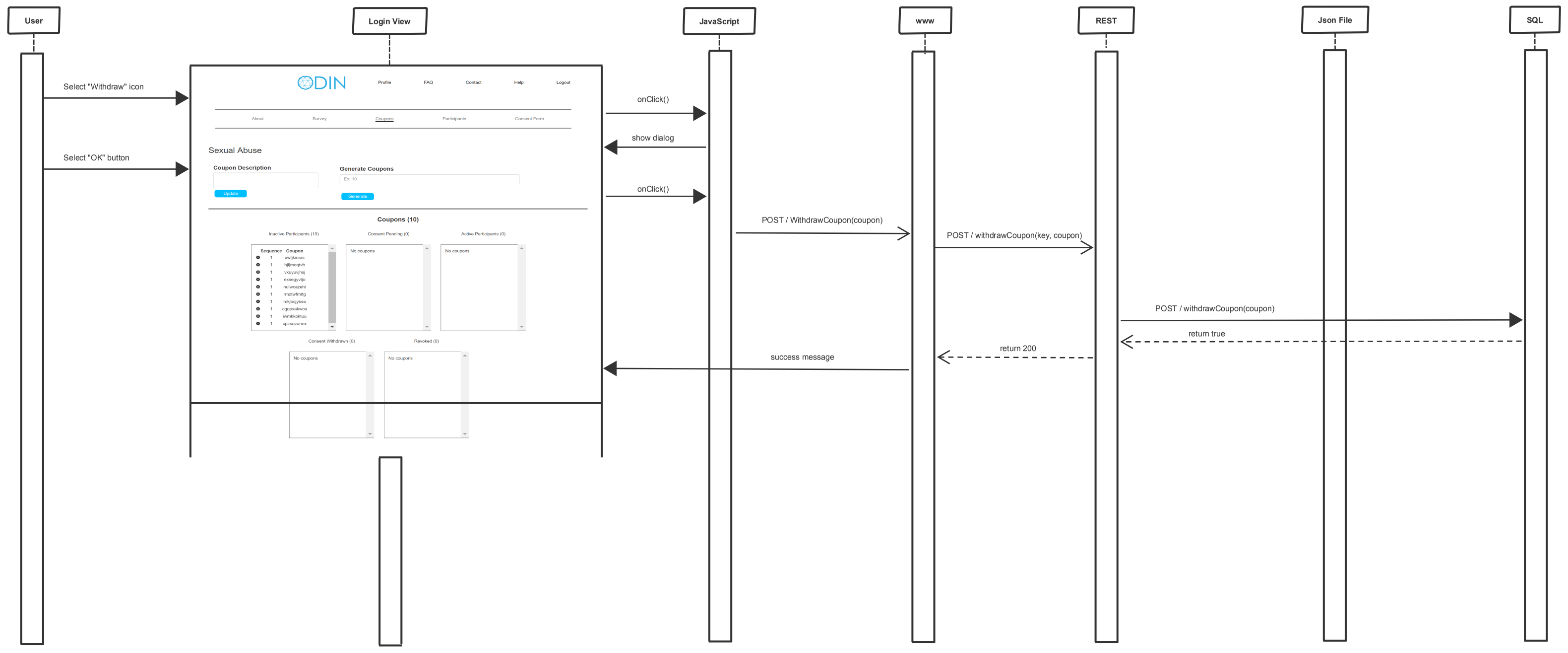
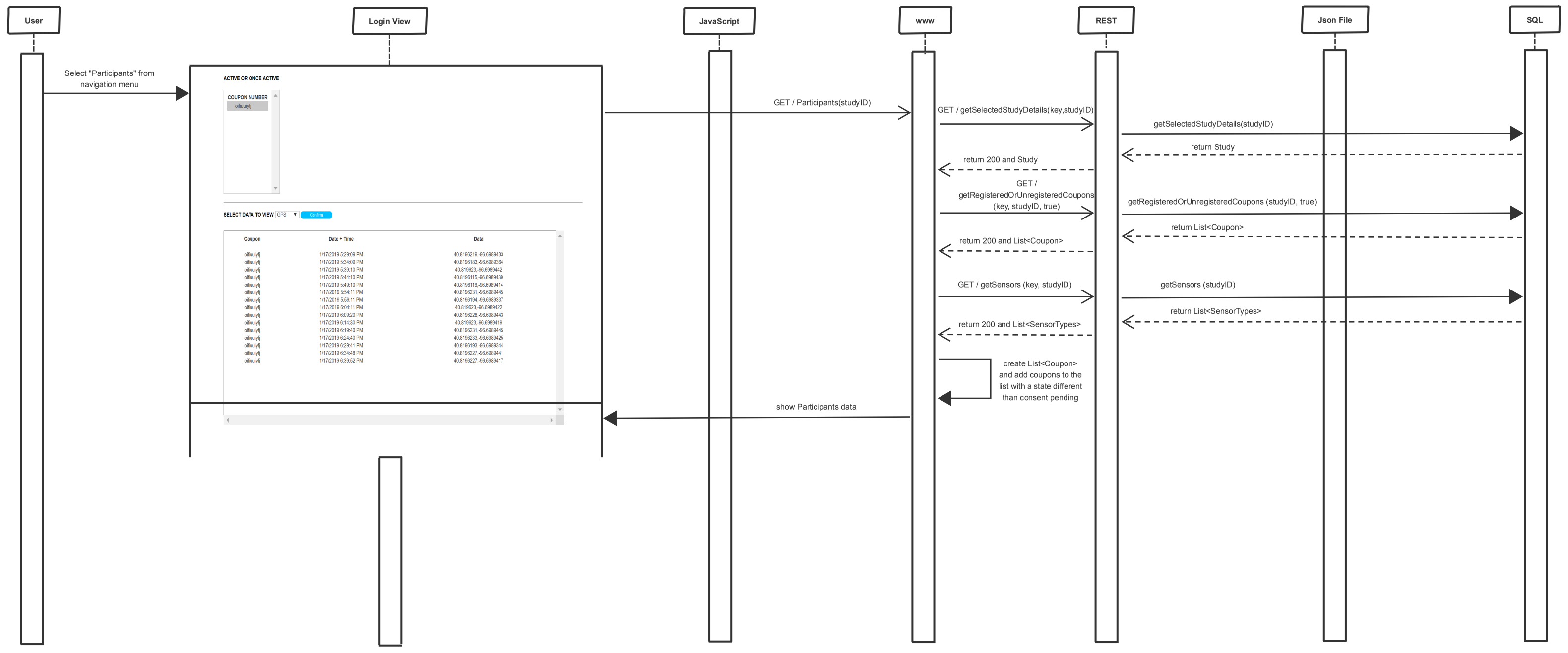
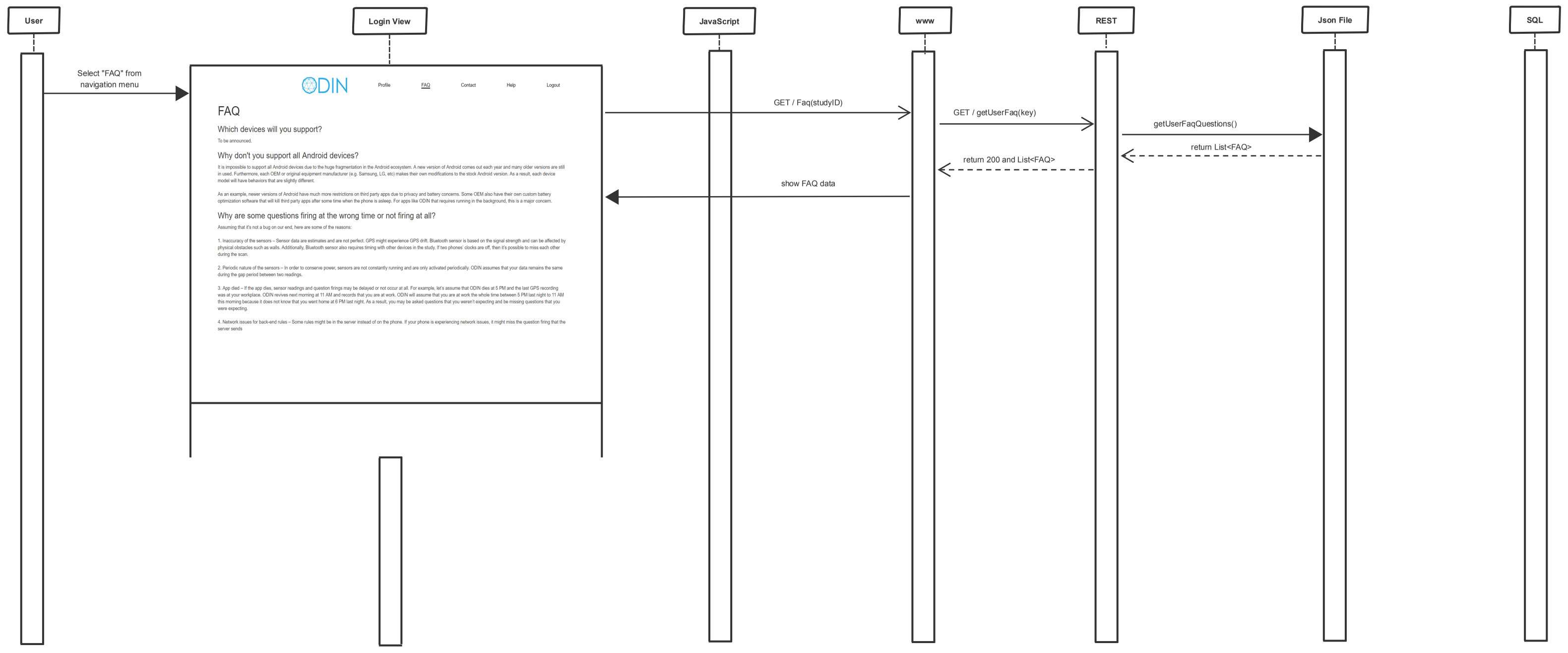


Figure .22: ResearcherUI Coupons Withdraw success


User
Login View
JavaScript
www
REST
Json File
SQL
Select "Participants" from navigation menu
GET / Participants(studyID)
GET / getSelectedStudyDetails(key,studyID)
getSelectedStudyDetails(studyID)
return Study
return 200 and Study
GET / getRegisteredOrUnregisteredCoupons (key, studyID, true)
getRegisteredOrUnregisteredCoupons (studyID, true)
return List<Coupon>
return 200 and List<Coupon>
GET / getSensors (key, studyID)
getSensors (studyID)
return List<SensorTypes>
return 200 and List<SensorTypes>
create List<Coupon> and add coupons to the list with a state different than consent pending
show Participants data
ACTIVE OR ONCE ACTIVE
COUPON NUMBER
oifiuuiyfj
SELECT DATA TO VIEW
GPS
Confirm

Coupon	Date + Time	Data
oifiuuiyfj	1/17/2019 5:29:09 PM	40.8196219,-96.6989433
oifiuuiyfj	1/17/2019 5:34:09 PM	40.8196183,-96.6989364
oifiuuiyfj	1/17/2019 5:39:10 PM	40.819623,-96.6989442
oifiuuiyfj	1/17/2019 5:44:10 PM	40.8196115,-96.6989439
oifiuuiyfj	1/17/2019 5:49:10 PM	40.8196116,-96.6989414
oifiuuiyfj	1/17/2019 5:54:11 PM	40.8196231,-96.6989445
oifiuuiyfj	1/17/2019 5:59:11 PM	40.8196194,-96.6989337
oifiuuiyfj	1/17/2019 6:04:11 PM	40.819623,-96.6989422
oifiuuiyfj	1/17/2019 6:09:20 PM	40.8196228,-96.6989443
oifiuuiyfj	1/17/2019 6:14:30 PM	40.819623,-96.6989419
oifiuuiyfj	1/17/2019 6:19:40 PM	40.8196231,-96.6989445
oifiuuiyfj	1/17/2019 6:24:40 PM	40.8196233,-96.6989425
oifiuuiyfj	1/17/2019 6:29:41 PM	40.8196193,-96.6989344
oifiuuiyfj	1/17/2019 6:34:48 PM	40.8196227,-96.6989441
oifiuuiyfj	1/17/2019 6:39:52 PM	40.8196227,-96.6989417

Figure .23: ResearcherUI Coupons get



FAQ

Which devices will you support?

To be announced.

Why don't you support all Android devices?

It is impossible to support all Android devices due to the huge fragmentation in the Android ecosystem. A new version of Android comes out each year and many older versions are still in used. Furthermore, each OEM or original equipment manufacturer (e.g. Samsung, LG, etc) makes their own modifications to the stock Android version. As a result, each device model will have behaviors that are slightly different.

As an example, newer versions of Android have much more restrictions on third party apps due to privacy and battery concerns. Some OEM also have their own custom battery optimization software that will kill third party apps after some time when the phone is asleep. For apps like ODIN that requires running in the background, this is a major concern.

Why are some questions firing at the wrong time or not firing at all?

Assuming that it's not a bug on our end, here are some of the reasons:

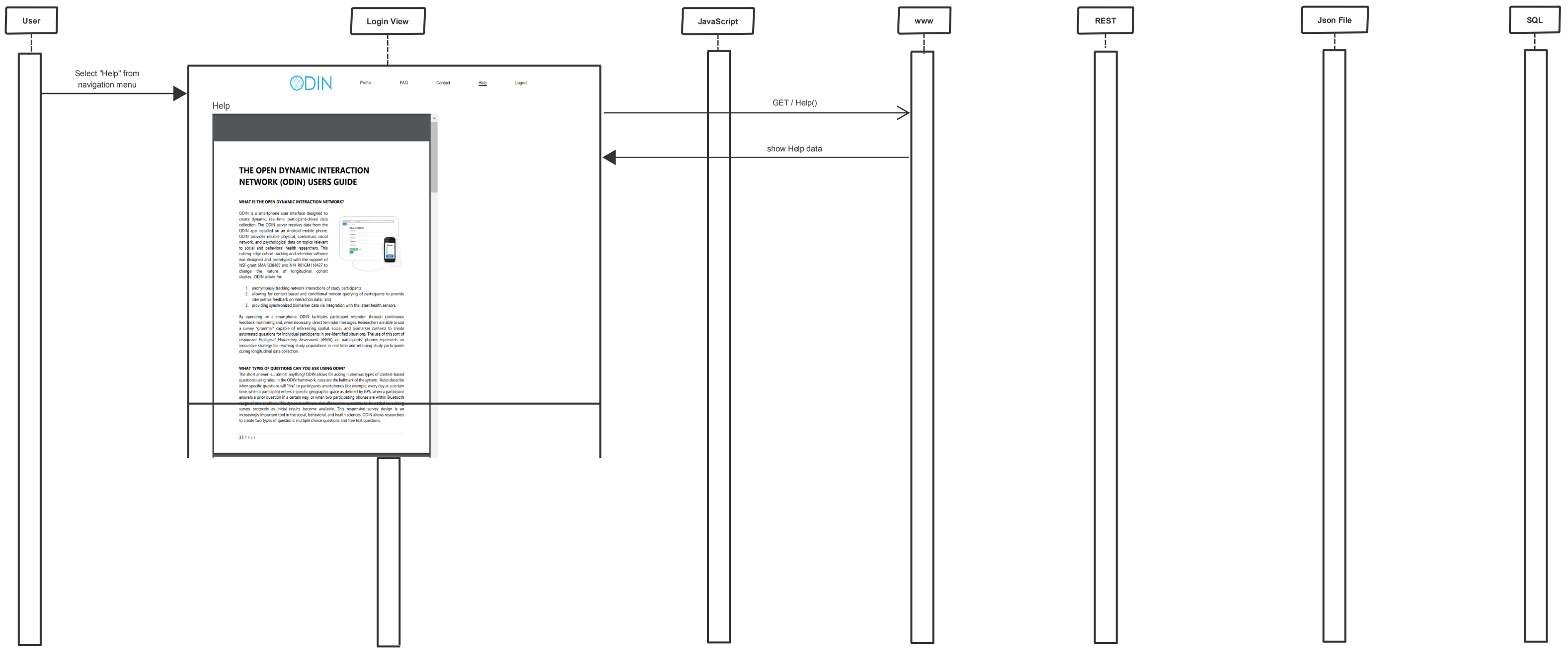
1. Inaccuracy of the sensors – Sensor data are estimates and are not perfect. GPS might experience GPS drift. Bluetooth sensor is based on the signal strength and can be affected by physical obstacles such as walls. Additionally, Bluetooth sensor also requires timing with other devices in the study. If two phones' clocks are off, then it's possible to miss each other during the scan.

2. Periodic nature of the sensors – In order to conserve power, sensors are not constantly running and are only activated periodically. ODIN assumes that your data remains the same during the gap period between two readings.

3. App died – If the app dies, sensor readings and question firings may be delayed or not occur at all. For example, let's assume that ODIN dies at 5 PM and the last GPS recording was at your workplace. ODIN revives next morning at 11 AM and records that you are at work. ODIN will assume that you are at work the whole time between 5 PM last night to 11 AM this morning because it does not know that you went home at 6 PM last night. As a result, you may be asked questions that you weren't expecting and be missing questions that you were expecting.

4. Network issues for back-end rules – Some rules might be in the server instead of on the phone. If your phone is experiencing network issues, it might miss the question firing that the server sends

Figure .24: ResearcherUI Faq



Help

THE OPEN DYNAMIC INTERACTION NETWORK (ODIN) USERS GUIDE

WHAT IS THE OPEN DYNAMIC INTERACTION NETWORK?

ODIN is a smartphone user interface designed to create dynamic, real-time, participant-driven data collection. The ODIN server receives data from the ODIN app installed on an Android mobile phone. ODIN provides reliable physical, contextual, social network, and psychological data on topics relevant to social and behavioral health researchers. This cutting-edge cohort tracking and retention software was designed and prototyped with the support of NSF grant SMA1338485 and NIH R01GM118427 to change the nature of longitudinal cohort studies. ODIN allows for:

1. anonymously tracking network interactions of study participants;
2. allowing for context-based and conditional remote querying of participants to provide interpretive feedback on interaction data; and
3. providing synchronized biomarker data via integration with the latest health sensors.

By operating on a smartphone, ODIN facilitates participant retention through continuous feedback monitoring and, when necessary, direct reminder messages. Researchers are able to use a survey "grammar" capable of referencing spatial, social, and biomarker contexts to create automated questions for individual participants in pre-identified situations. The use of this sort of *responsive Ecological Momentary Assessment* (rEMA) via participants' phones represents an innovative strategy for reaching study populations in real time and retaining study participants during longitudinal data collection.

WHAT TYPES OF QUESTIONS CAN YOU ASK USING ODIN?

The short answer is... almost anything! ODIN allows for asking numerous types of context-based questions using rules. In the ODIN framework, rules are the hallmark of the system. Rules describe when specific questions will "fire" to participants smartphones (for example, every day at a certain time, when a participant enters a specific geographic space as defined by GPS, when a participant answers a prior question in a certain way, or when two participating phones are within Bluetooth range of one another). This dynamic software also allows new questions to be added to existing survey protocols as initial results become available. This responsive survey design is an increasingly important tool in the social, behavioral, and health sciences. ODIN allows researchers to create two types of questions: multiple choice questions and free text questions.

1 | Page

Figure .25: ResearcherUI Help

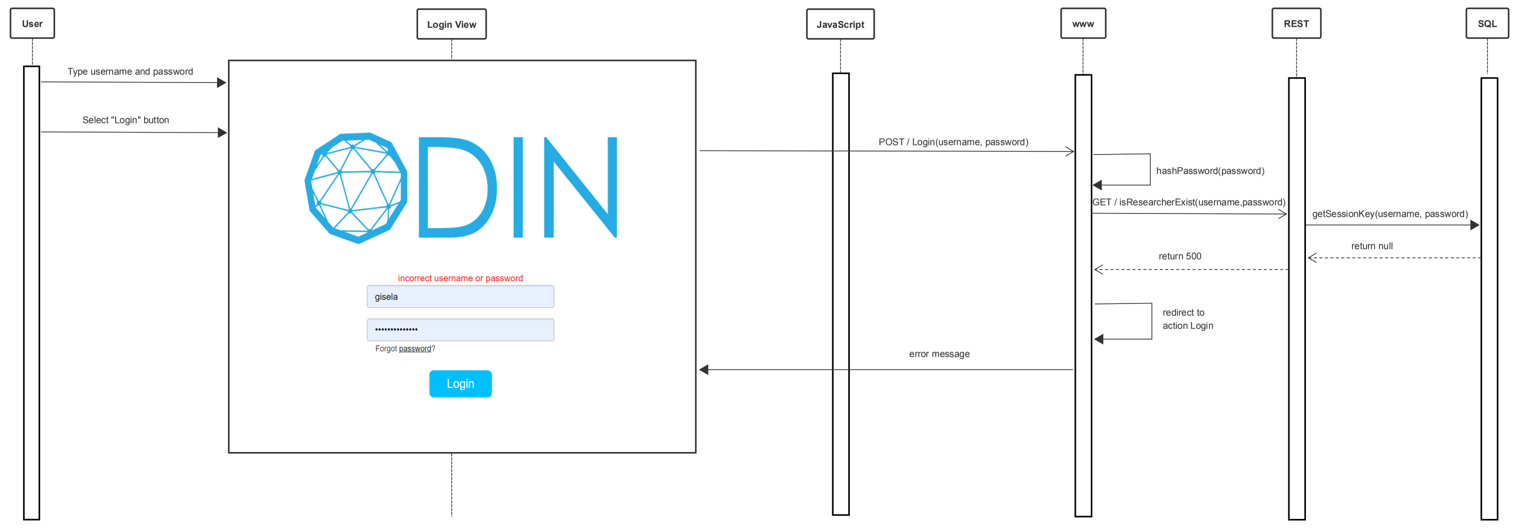


Figure .26: ResearcherUI Login failure

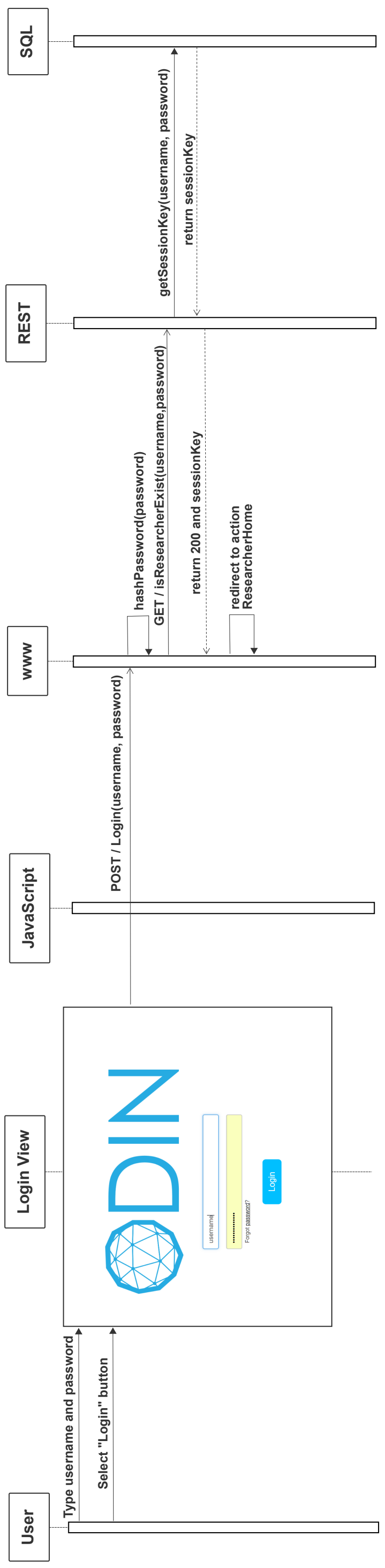
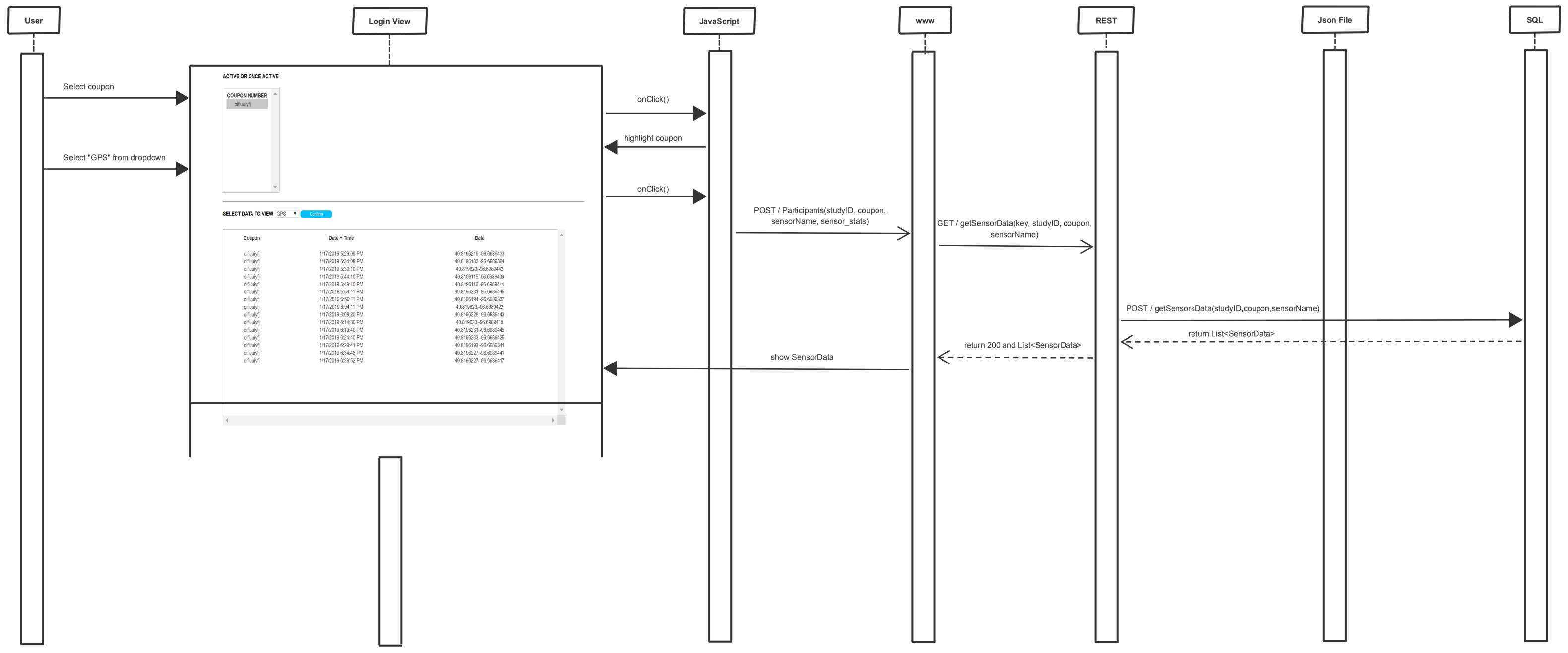

User
Login View
JavaScript
www
REST
SQL
Type username and password
Select "Login" button
username
Forgot password?
Login
POST / Login(username, password)
hashPassword(password)
GET / isResearcherExist(username,password)
return 200 and sessionKey
redirect to action ResearcherHome
getSessionKey(username, password)
return sessionKey

Figure .27: ResearcherUI Login success


User
Login View
JavaScript
www
REST
Json File
SQL
Select coupon
Select "GPS" from dropdown
onClick()
highlight coupon
onClick()
POST / Participants(studyID, coupon, sensorName, sensor_stats)
GET / getSensorData(key, studyID, coupon, sensorName)
POST / getSensorsData(studyID,coupon,sensorName)
return List<SensorData>
return 200 and List<SensorData>
show SensorData
ACTIVE OR ONCE ACTIVE
COUPON NUMBER
oifiuuiyfj
SELECT DATA TO VIEW
GPS
Confirm

Coupon	Date + Time	Data
oifiuuiyfj	1/17/2019 5:29:09 PM	40.8196219,-96.6989433
oifiuuiyfj	1/17/2019 5:34:09 PM	40.8196183,-96.6989364
oifiuuiyfj	1/17/2019 5:39:10 PM	40.819623,-96.6989442
oifiuuiyfj	1/17/2019 5:44:10 PM	40.8196115,-96.6989439
oifiuuiyfj	1/17/2019 5:49:10 PM	40.8196116,-96.6989414
oifiuuiyfj	1/17/2019 5:54:11 PM	40.8196231,-96.6989445
oifiuuiyfj	1/17/2019 5:59:11 PM	40.8196194,-96.6989337
oifiuuiyfj	1/17/2019 6:04:11 PM	40.819623,-96.6989422
oifiuuiyfj	1/17/2019 6:09:20 PM	40.8196228,-96.6989443
oifiuuiyfj	1/17/2019 6:14:30 PM	40.819623,-96.6989419
oifiuuiyfj	1/17/2019 6:19:40 PM	40.8196231,-96.6989445
oifiuuiyfj	1/17/2019 6:24:40 PM	40.8196233,-96.6989425
oifiuuiyfj	1/17/2019 6:29:41 PM	40.8196193,-96.6989344
oifiuuiyfj	1/17/2019 6:34:48 PM	40.8196227,-96.6989441
oifiuuiyfj	1/17/2019 6:39:52 PM	40.8196227,-96.6989417

Figure .28: ResearcherUI Participants Post sensor success

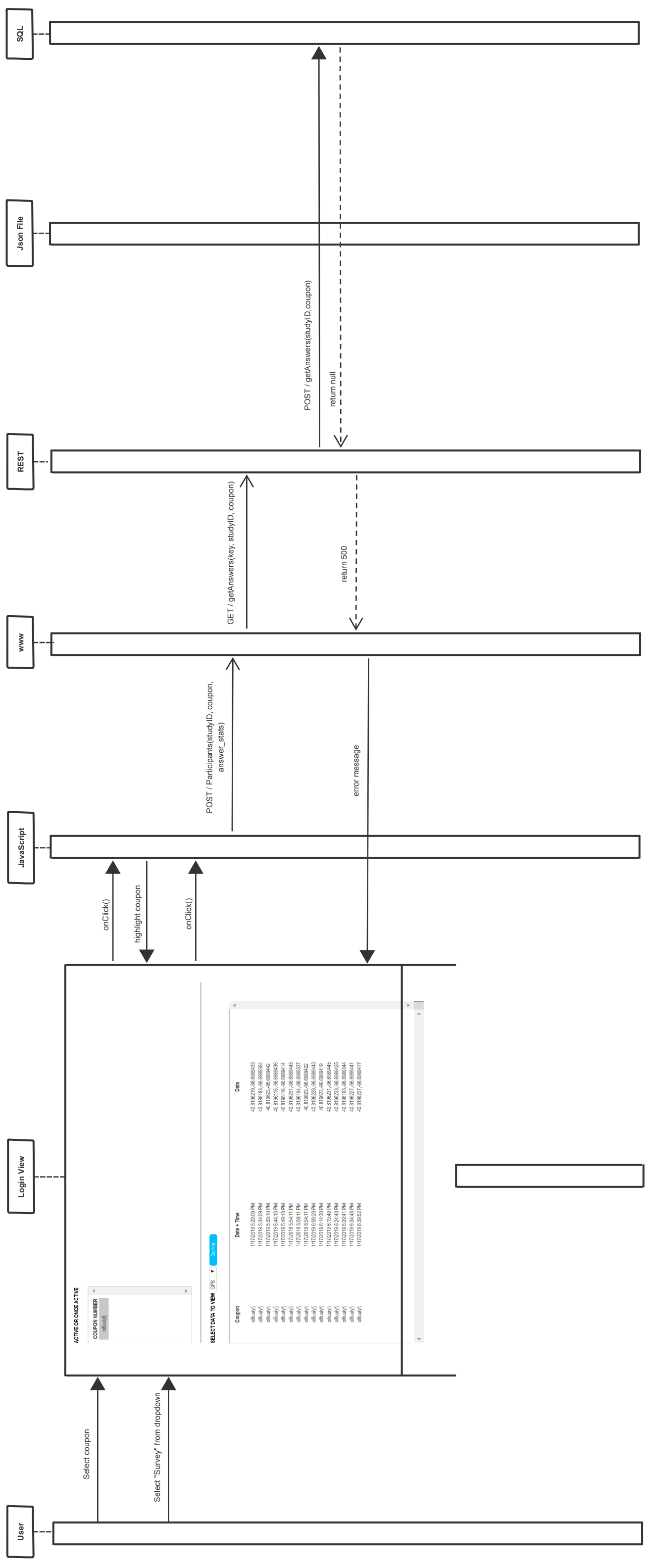
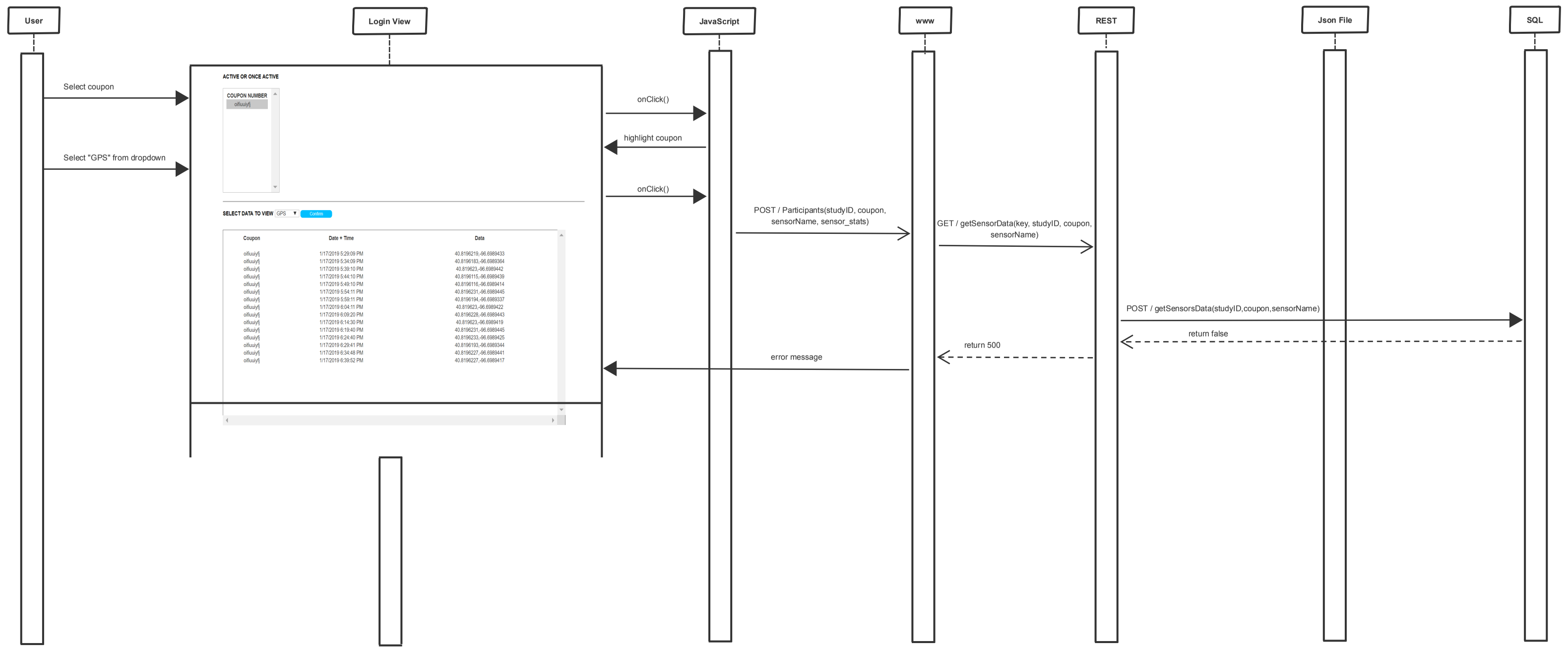
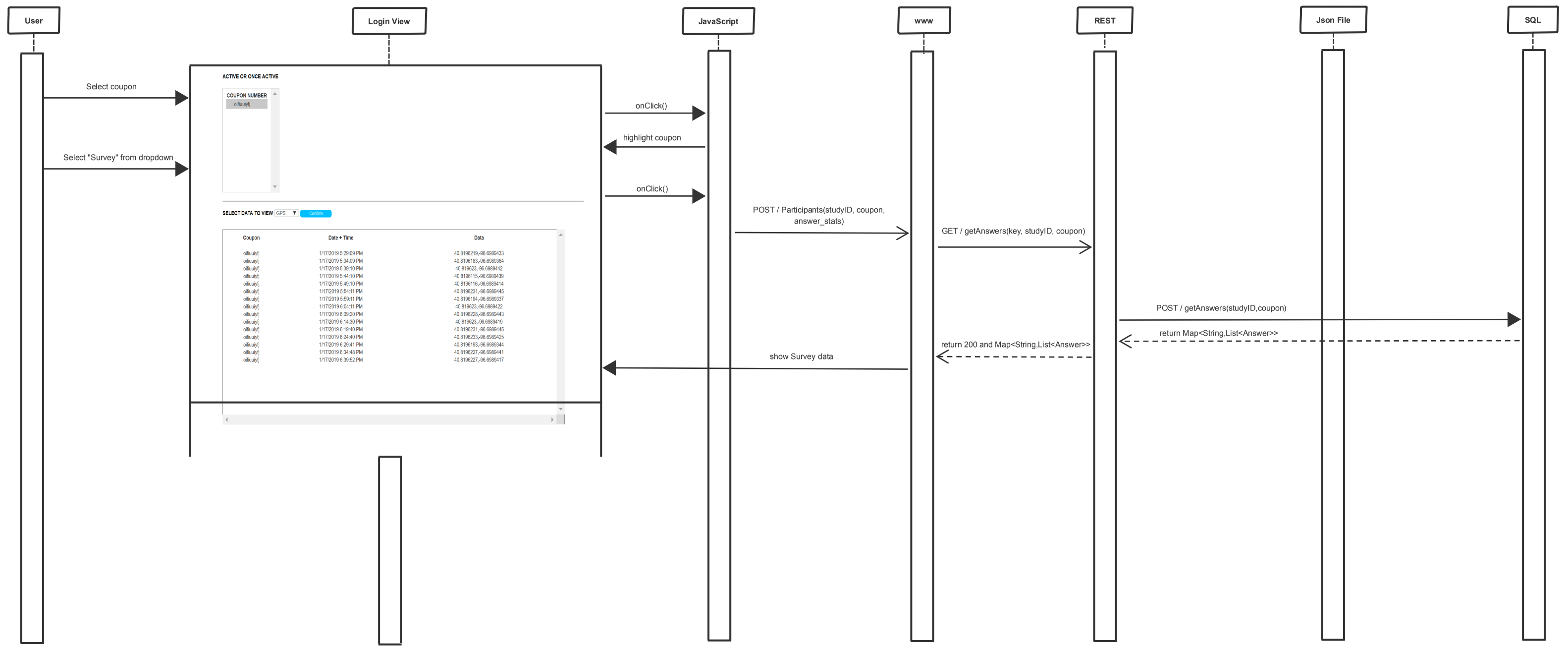

User
Login View
JavaScript
www
REST
Json File
SQL
Select coupon
Select "Survey" from dropdown
onClick()
highlight coupon
onClick()
POST / Participants(studyID, coupon, answer_stats)
GET / getAnswers(key, studyID, coupon)
POST / getAnswers(studyID,coupon)
return null
return 500
error message

Figure .29: ResearcherUI Participants post failure



Coupon	Date + Time	Data
oifiuuiyfj	1/17/2019 5:29:09 PM	40.8196219,-96.6989433
oifiuuiyfj	1/17/2019 5:34:09 PM	40.8196183,-96.6989364
oifiuuiyfj	1/17/2019 5:39:10 PM	40.819623,-96.6989442
oifiuuiyfj	1/17/2019 5:44:10 PM	40.8196115,-96.6989439
oifiuuiyfj	1/17/2019 5:49:10 PM	40.8196116,-96.6989414
oifiuuiyfj	1/17/2019 5:54:11 PM	40.8196231,-96.6989445
oifiuuiyfj	1/17/2019 5:59:11 PM	40.8196194,-96.6989337
oifiuuiyfj	1/17/2019 6:04:11 PM	40.819623,-96.6989422
oifiuuiyfj	1/17/2019 6:09:20 PM	40.8196228,-96.6989443
oifiuuiyfj	1/17/2019 6:14:30 PM	40.819623,-96.6989419
oifiuuiyfj	1/17/2019 6:19:40 PM	40.8196231,-96.6989445
oifiuuiyfj	1/17/2019 6:24:40 PM	40.8196233,-96.6989425
oifiuuiyfj	1/17/2019 6:29:41 PM	40.8196193,-96.6989344
oifiuuiyfj	1/17/2019 6:34:48 PM	40.8196227,-96.6989441
oifiuuiyfj	1/17/2019 6:39:52 PM	40.8196227,-96.6989417

Figure .30: ResearcherUI Participants post sensor failure


User
Login View
JavaScript
www
REST
Json File
SQL
Select coupon
Select "Survey" from dropdown
onClick()
highlight coupon
onClick()
POST / Participants(studyID, coupon, answer_stats)
GET / getAnswers(key, studyID, coupon)
POST / getAnswers(studyID,coupon)
return Map<String,List<Answer>>
return 200 and Map<String,List<Answer>>
show Survey data
ACTIVE OR ONCE ACTIVE
COUPON NUMBER
oifiuuiyfj
SELECT DATA TO VIEW
GPS
Confirm

Coupon	Date + Time	Data
oifiuuiyfj	1/17/2019 5:29:09 PM	40.8196219,-96.6989433
oifiuuiyfj	1/17/2019 5:34:09 PM	40.8196183,-96.6989364
oifiuuiyfj	1/17/2019 5:39:10 PM	40.819623,-96.6989442
oifiuuiyfj	1/17/2019 5:44:10 PM	40.8196115,-96.6989439
oifiuuiyfj	1/17/2019 5:49:10 PM	40.8196116,-96.6989414
oifiuuiyfj	1/17/2019 5:54:11 PM	40.8196231,-96.6989445
oifiuuiyfj	1/17/2019 5:59:11 PM	40.8196194,-96.6989337
oifiuuiyfj	1/17/2019 6:04:11 PM	40.819623,-96.6989422
oifiuuiyfj	1/17/2019 6:09:20 PM	40.8196228,-96.6989443
oifiuuiyfj	1/17/2019 6:14:30 PM	40.819623,-96.6989419
oifiuuiyfj	1/17/2019 6:19:40 PM	40.8196231,-96.6989445
oifiuuiyfj	1/17/2019 6:24:40 PM	40.8196233,-96.6989425
oifiuuiyfj	1/17/2019 6:29:41 PM	40.8196193,-96.6989344
oifiuuiyfj	1/17/2019 6:34:48 PM	40.8196227,-96.6989441
oifiuuiyfj	1/17/2019 6:39:52 PM	40.8196227,-96.6989417

Figure .31: ResearcherUI Participants post survey success

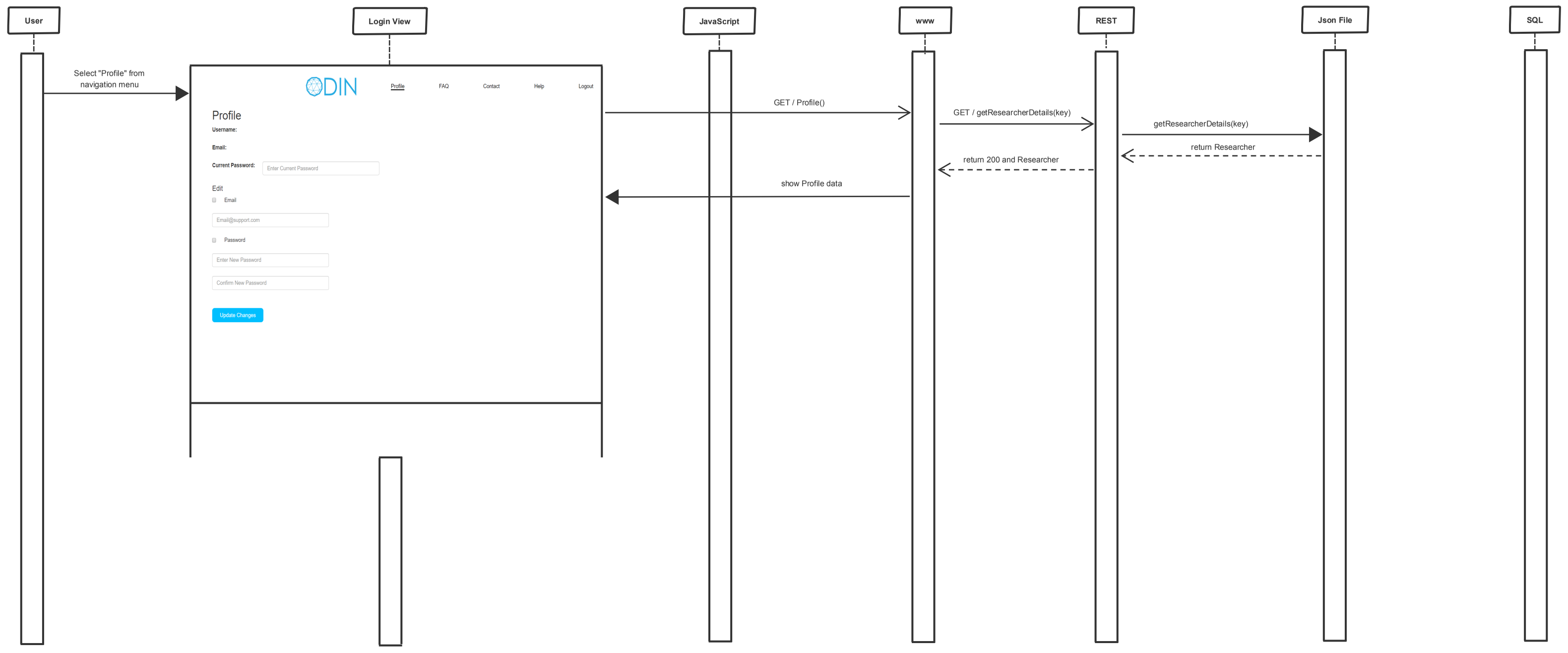


Figure .32: ResearcherUI Profile get

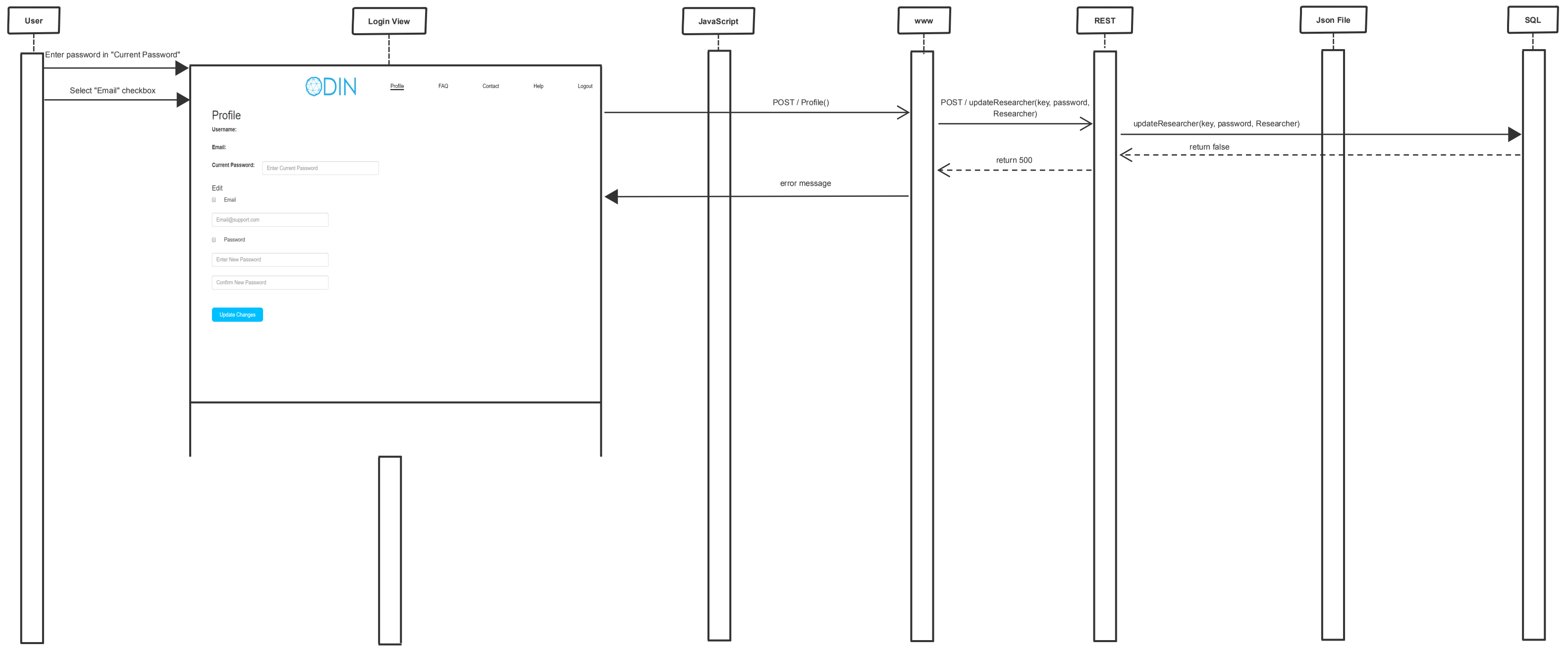


Figure .33: ResearcherUI Profile post failure

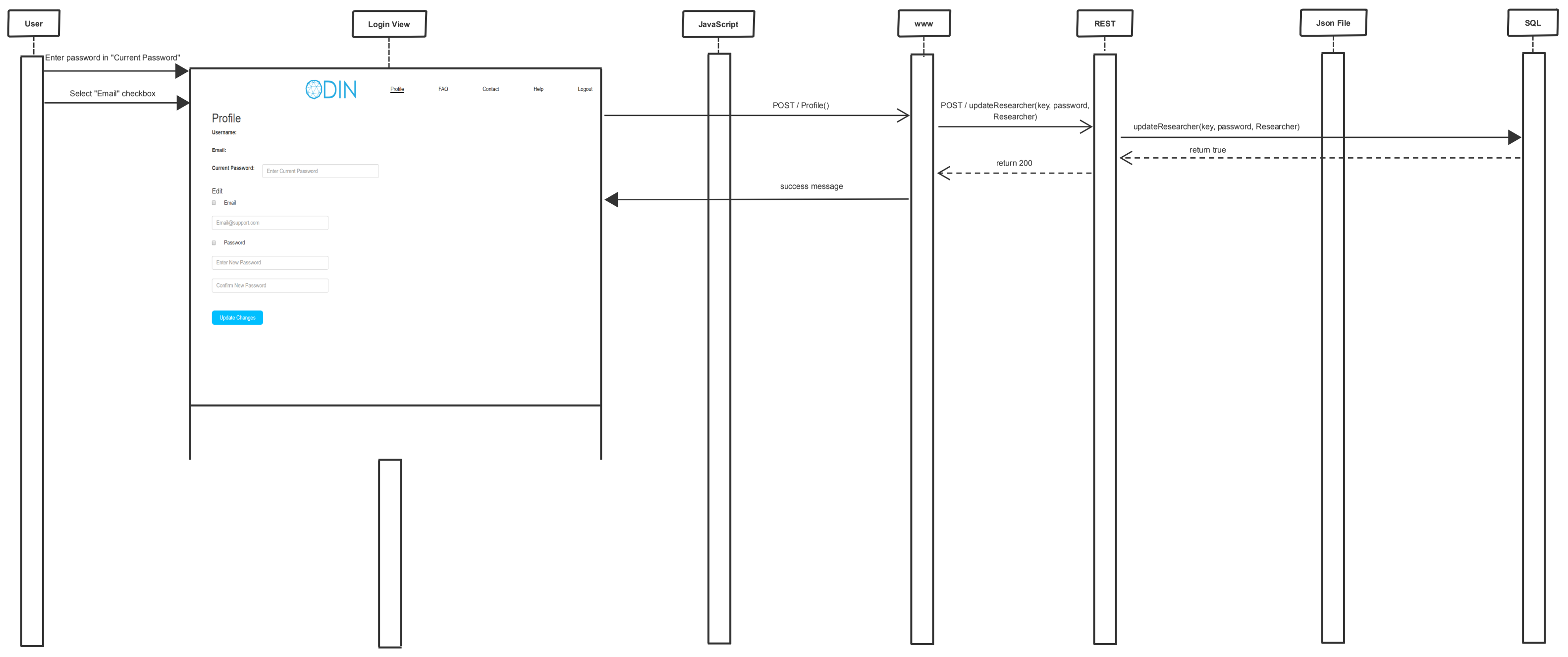


Figure .34: ResearcherUI Profile post success

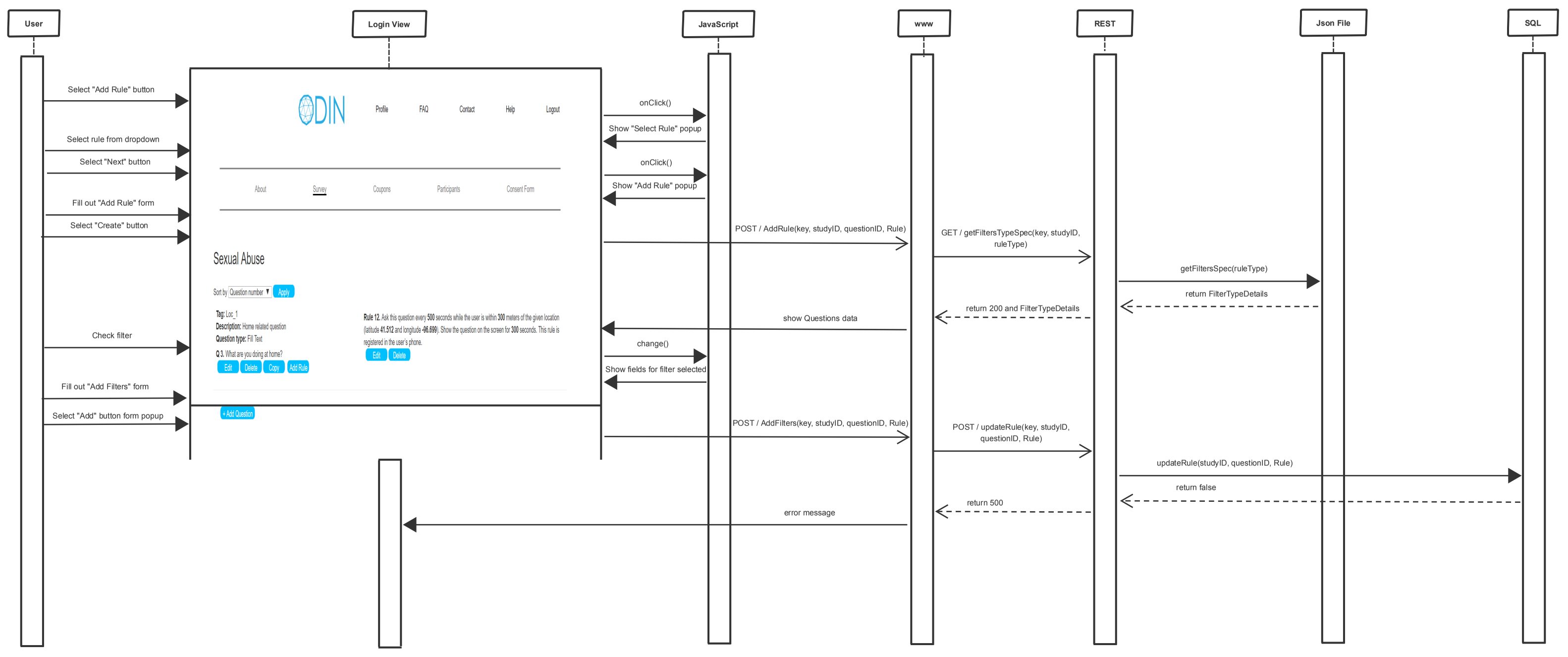

User
Login View
JavaScript
www
REST
Json File
SQL
Select "Add Rule" button
Select rule from dropdown
Select "Next" button
Fill out "Add Rule" form
Select "Create" button
Check filter
Fill out "Add Filters" form
Select "Add" button form popup
onClick()
Show "Select Rule" popup
onClick()
Show "Add Rule" popup
POST / AddRule(key, studyID, questionID, Rule)
GET / getFiltersTypeSpec(key, studyID, ruleType)
getFiltersSpec(ruleType)
return FilterTypeDetails
return 200 and FilterTypeDetails
show Questions data
change()
Show fields for filter selected
POST / AddFilters(key, studyID, questionID, Rule)
POST / updateRule(key, studyID, questionID, Rule)
updateRule(studyID, questionID, Rule)
return false
return 500
error message
ODIN
Profile
FAQ
Contact
Help
Logout
About
Survey
Coupons
Participants
Consent Form
Sexual Abuse
Sort by Question number
Apply
Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 1. What are you doing at home?
Edit
Delete
Copy
Add Rule
Rule 12. Ask this question every 500 seconds while the user is within 300 meters of the given location (latitude 41.512 and longitude -96.699). Show the question on the screen for 300 seconds. This rule is registered in the user's phone.
Edit
Delete
+ Add Question

Figure .35: ResearcherUI Questions AddFilters post failure

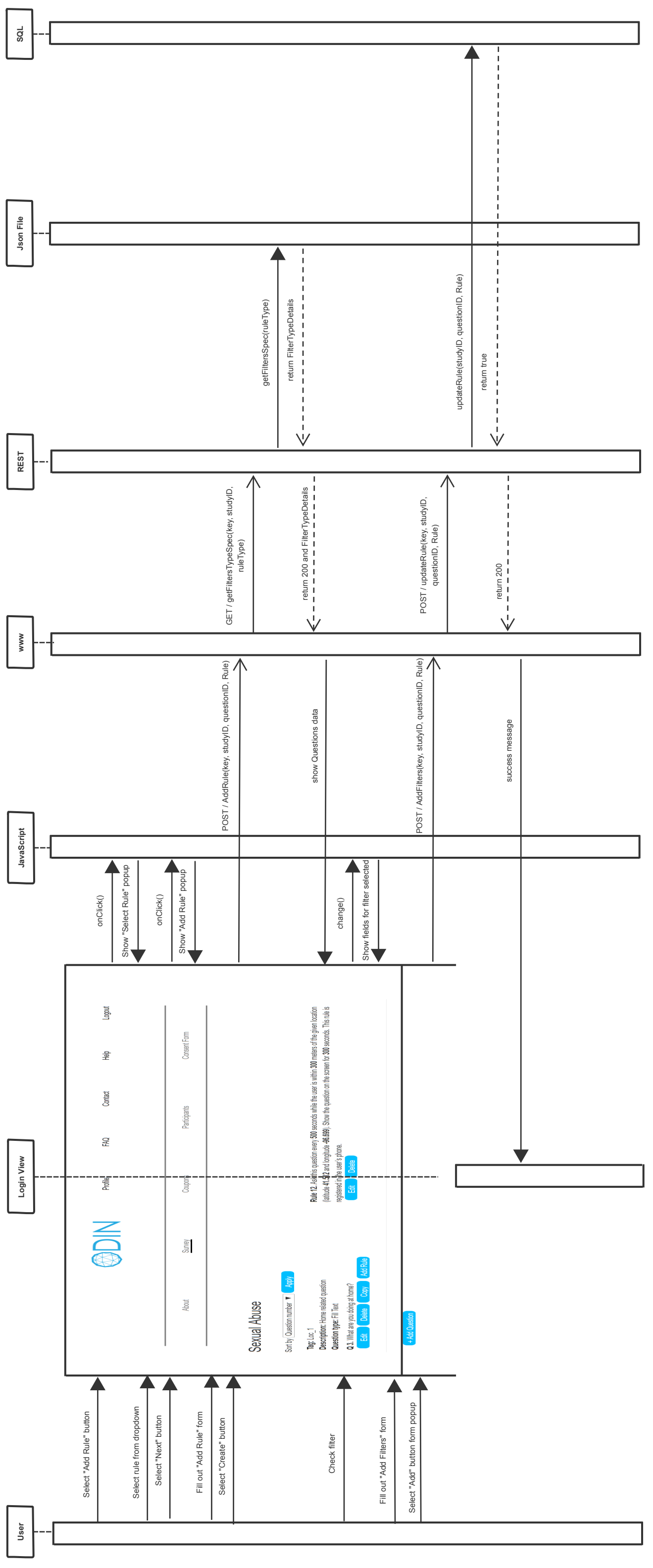

User
Login View
JavaScript
www
REST
Json File
SQL
Select "Add Rule" button
Select rule from dropdown
Select "Next" button
Fill out "Add Rule" form
Select "Create" button
Check filter
Fill out "Add Filters" form
Select "Add" button form popup
onClick()
Show "Select Rule" popup
onClick()
Show "Add Rule" popup
change()
Show fields for filter selected
POST / AddRule(key, studyID, questionID, Rule)
show Questions data
POST / AddFilters(key, studyID, questionID, Rule)
success message
GET / getFiltersTypeSpec(key, studyID, ruleType)
return 200 and FilterTypeDetails
POST / updateRule(key, studyID, questionID, Rule)
return 200
getFiltersSpec(ruleType)
return FilterTypeDetails
updateRule(studyID, questionID, Rule)
return true
ODIN
Sexual Abuse
Add Rule
+ Add Question

Figure .36: ResearcherUI Questions AddFilters post success

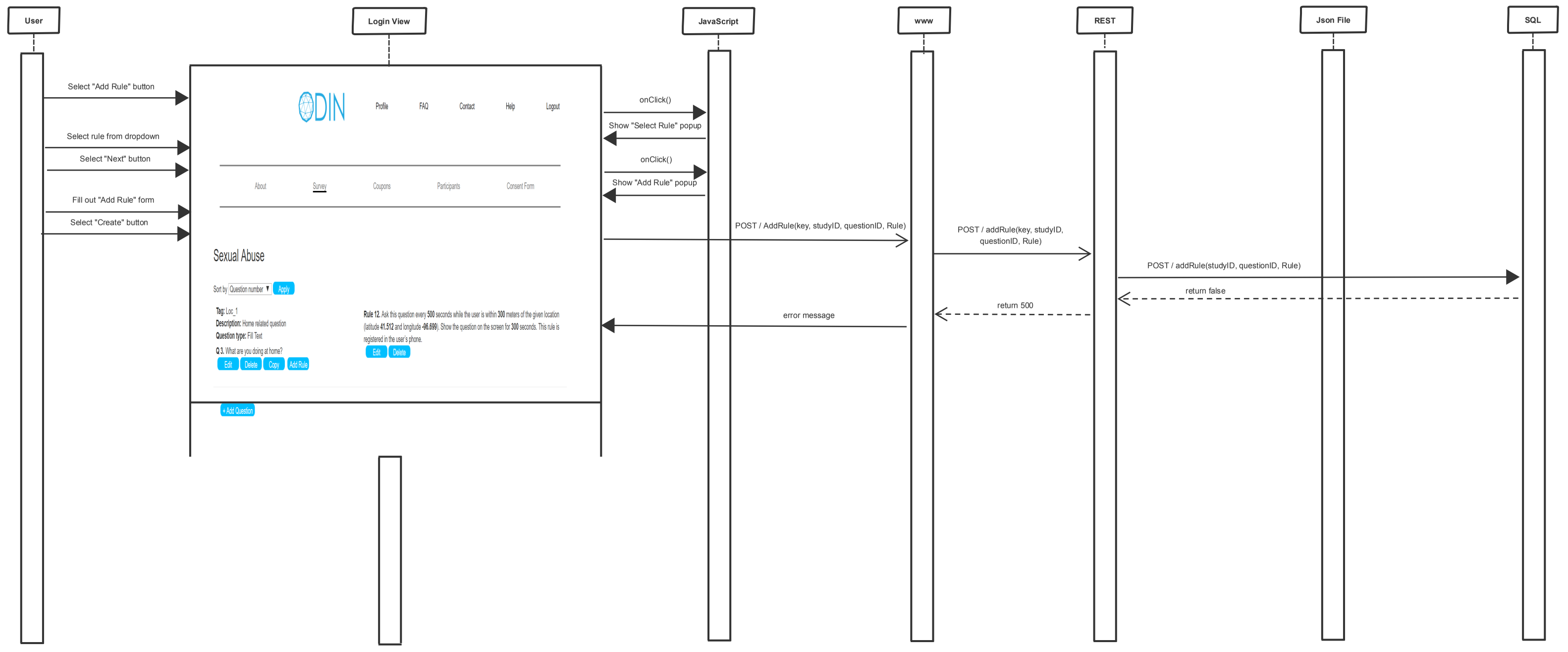

User
Login View
JavaScript
www
REST
Json File
SQL
Select "Add Rule" button
Select rule from dropdown
Select "Next" button
Fill out "Add Rule" form
Select "Create" button
onClick()
Show "Select Rule" popup
onClick()
Show "Add Rule" popup
POST / AddRule(key, studyID, questionID, Rule)
POST / addRule(key, studyID, questionID, Rule)
POST / addRule(studyID, questionID, Rule)
return false
return 500
error message
ODIN
Profile
FAQ
Contact
Help
Logout
About
Survey
Coupons
Participants
Consent Form
Sexual Abuse
Sort by
Question number
Apply
Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 1. What are you doing at home?
Edit
Delete
Copy
Add Rule
Rule 12. Ask this question every 500 seconds while the user is within 300 meters of the given location (latitude 41.512 and longitude -86.699). Show the question on the screen for 300 seconds. This rule is registered in the user's phone.
Edit
Delete
+ Add Question

Figure .37: ResearcherUI Questions AddRule post failure

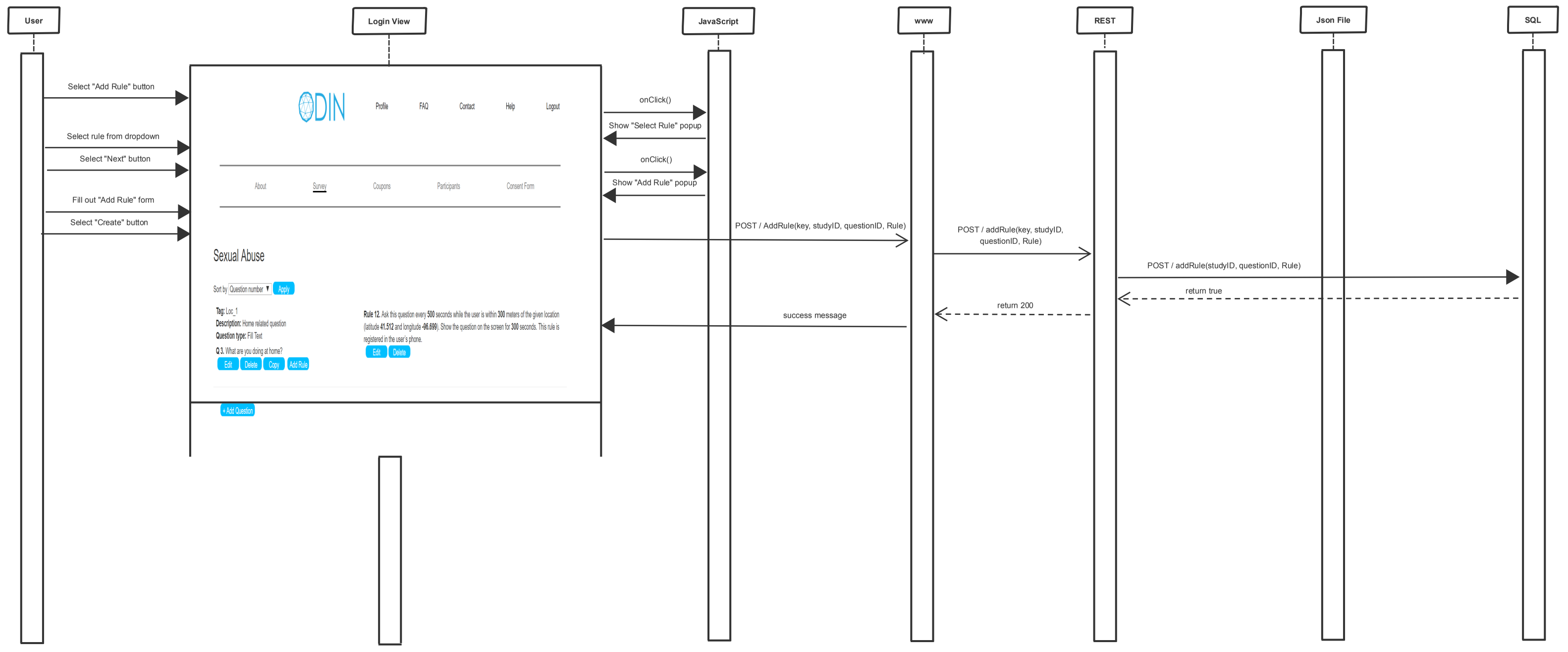


Figure .38: ResearcherUI Questions AddRule post success

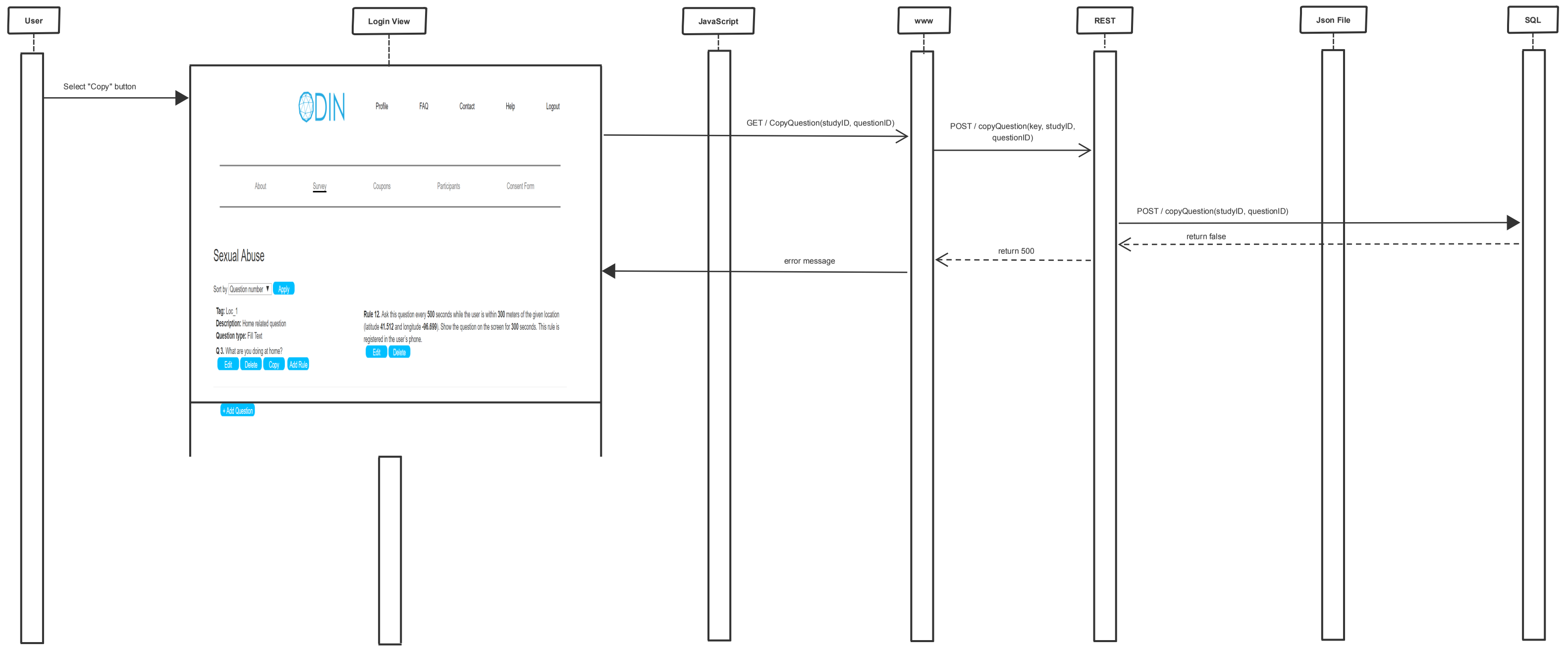

User
Login View
JavaScript
www
REST
Json File
SQL
Select "Copy" button
GET / CopyQuestion(studyID, questionID)
POST / copyQuestion(key, studyID, questionID)
POST / copyQuestion(studyID, questionID)
return false
return 500
error message
ODIN
Profile
FAQ
Contact
Help
Logout
About
Survey
Coupons
Participants
Consent Form
Sexual Abuse
Sort by Question number
Apply
Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 1. What are you doing at home?
Edit
Delete
Copy
Add Rule
Rule 12. Ask this question every 500 seconds while the user is within 300 meters of the given location (latitude 41.512 and longitude -96.699). Show the question on the screen for 300 seconds. This rule is registered in the user's phone.
Edit
Delete
+ Add Question

Figure .39: ResearcherUI Questions Copy failure

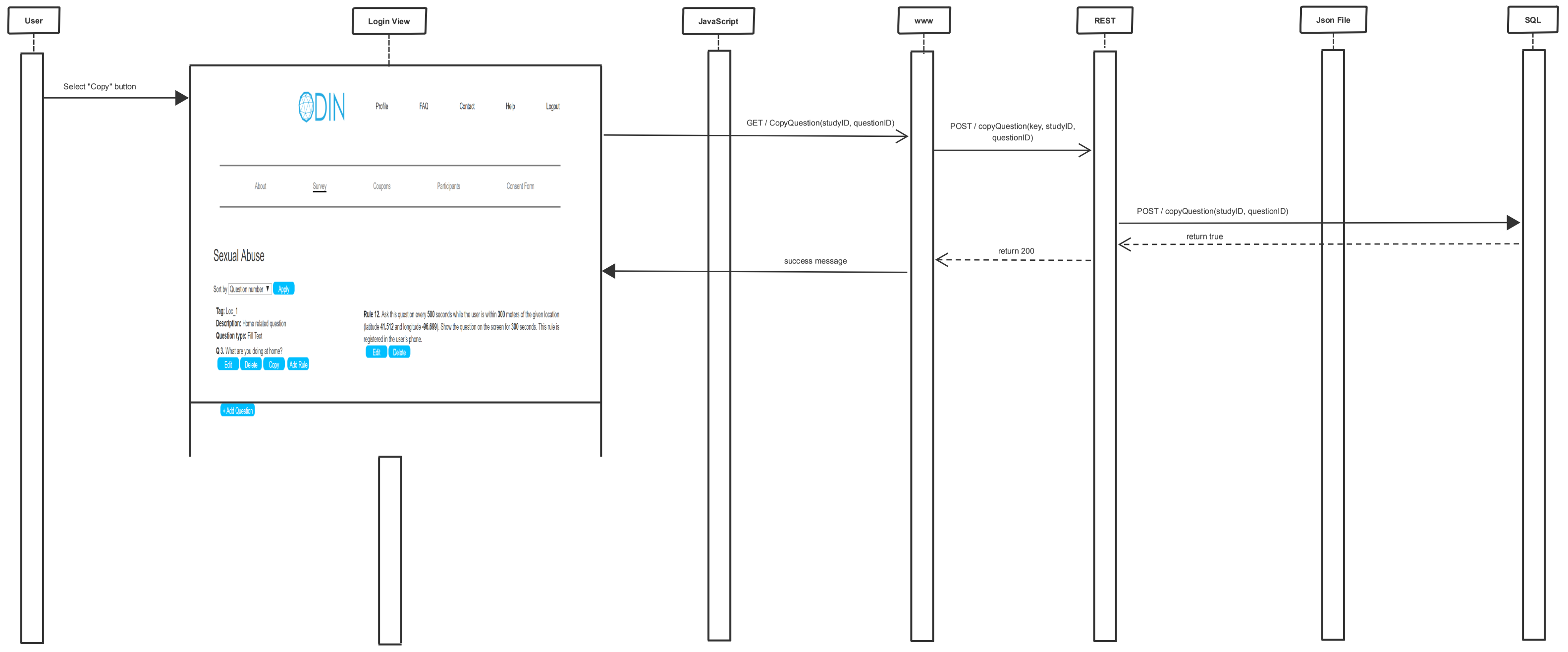


Figure .40: ResearcherUI Questions Copy success

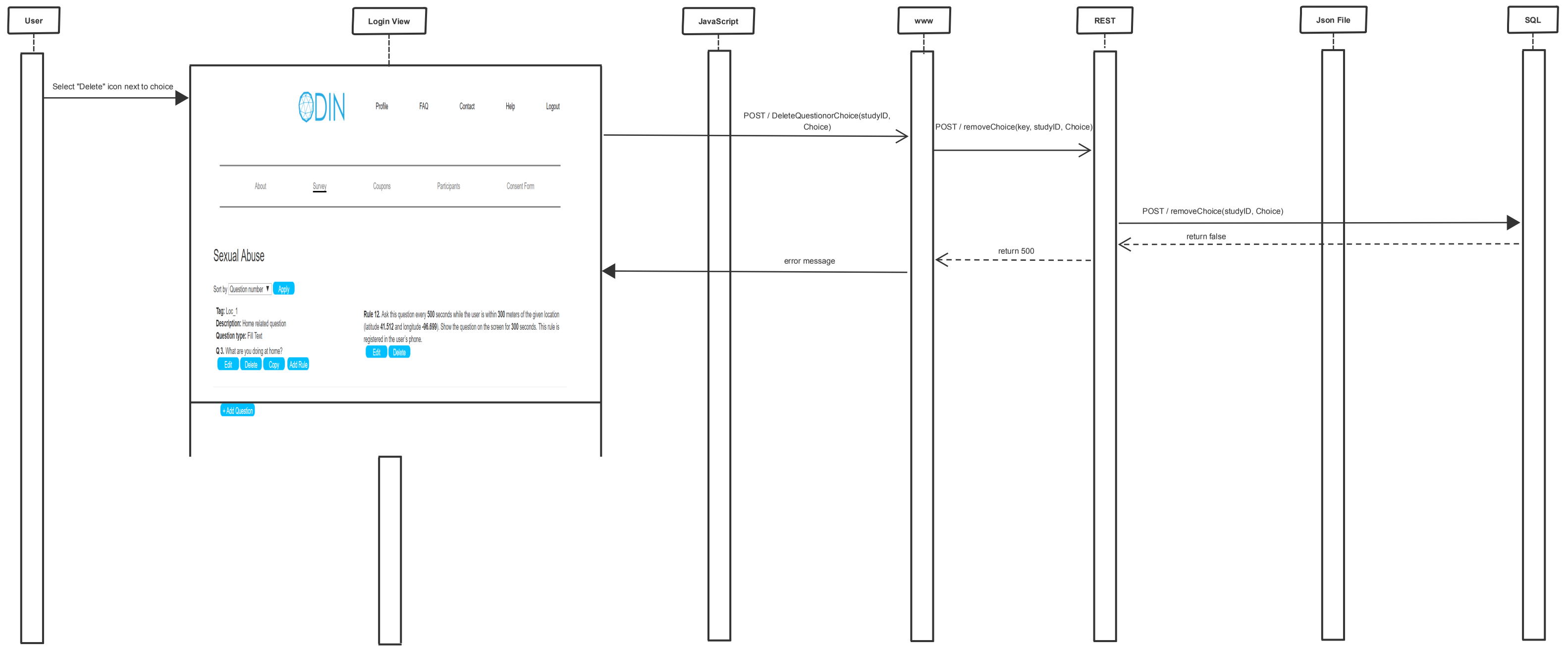
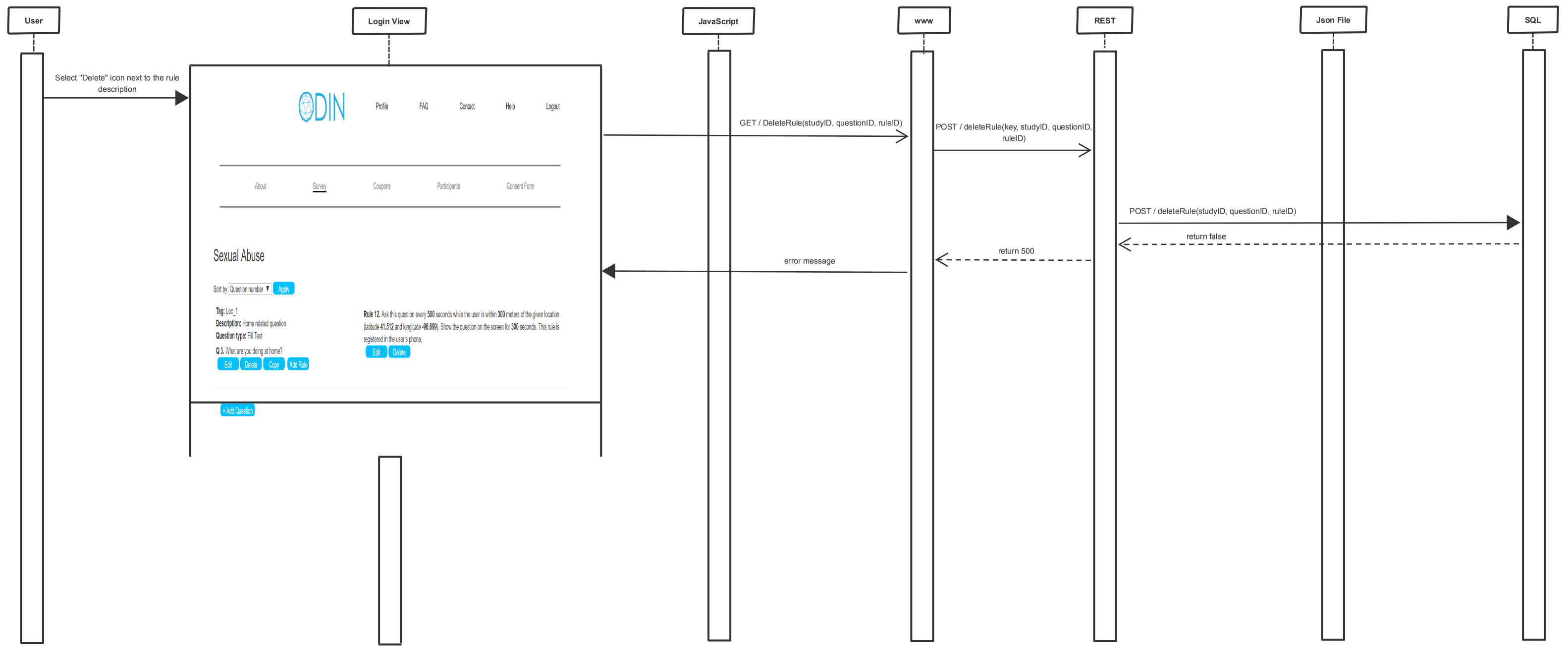


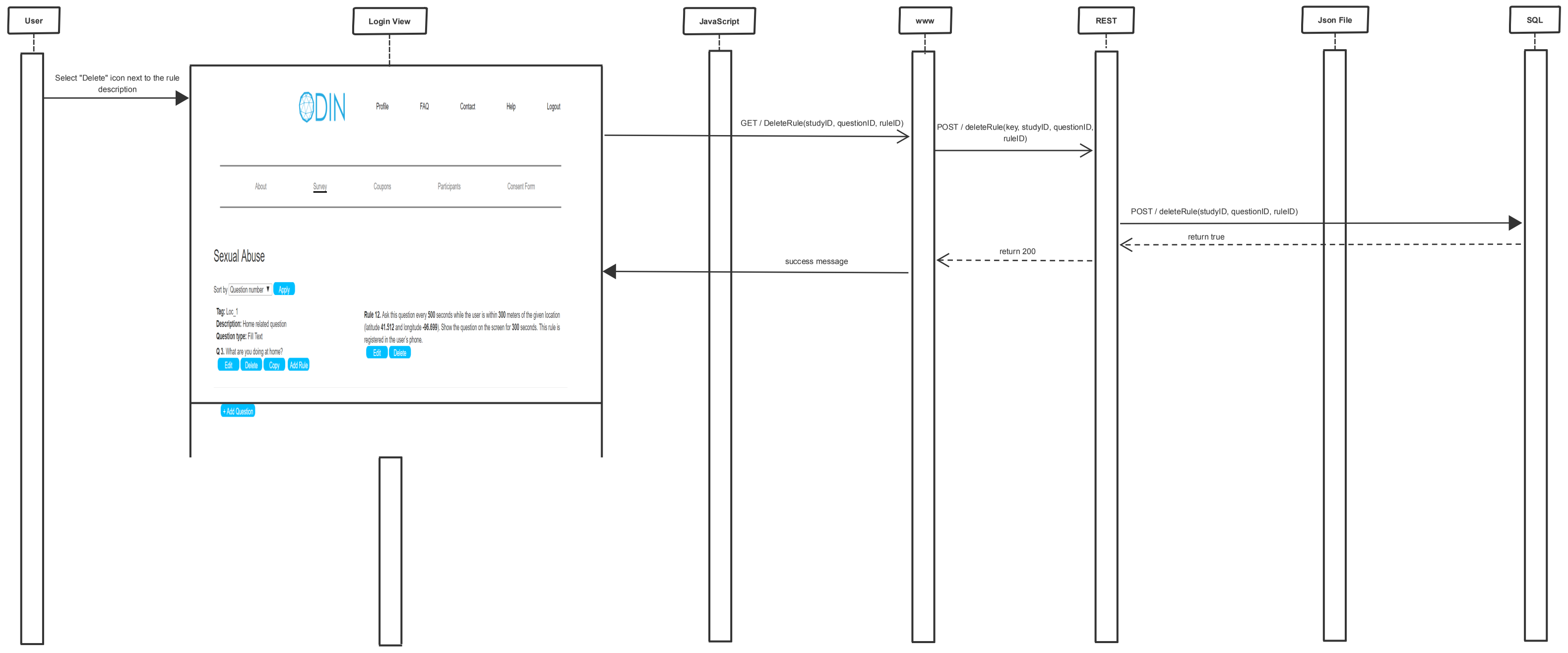
Figure .41: ResearcherUI Questions DeleteChoice failure



Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 1. What are you doing at home?

Rule 12. Ask this question every 500 seconds while the user is within 300 meters of the given location (latitude 41.512 and longitude -96.699). Show the question on the screen for 300 seconds. This rule is registered in the user's phone.

Figure .42: ResearcherUI Questions DeleteRule failure



Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 3. What are you doing at home?

Rule 12. Ask this question every **500** seconds while the user is within **300** meters of the given location (latitude **41.512** and longitude **-96.699**). Show the question on the screen for **300** seconds. This rule is registered in the user's phone.

Figure .43: ResearcherUI Questions DeleteRule success

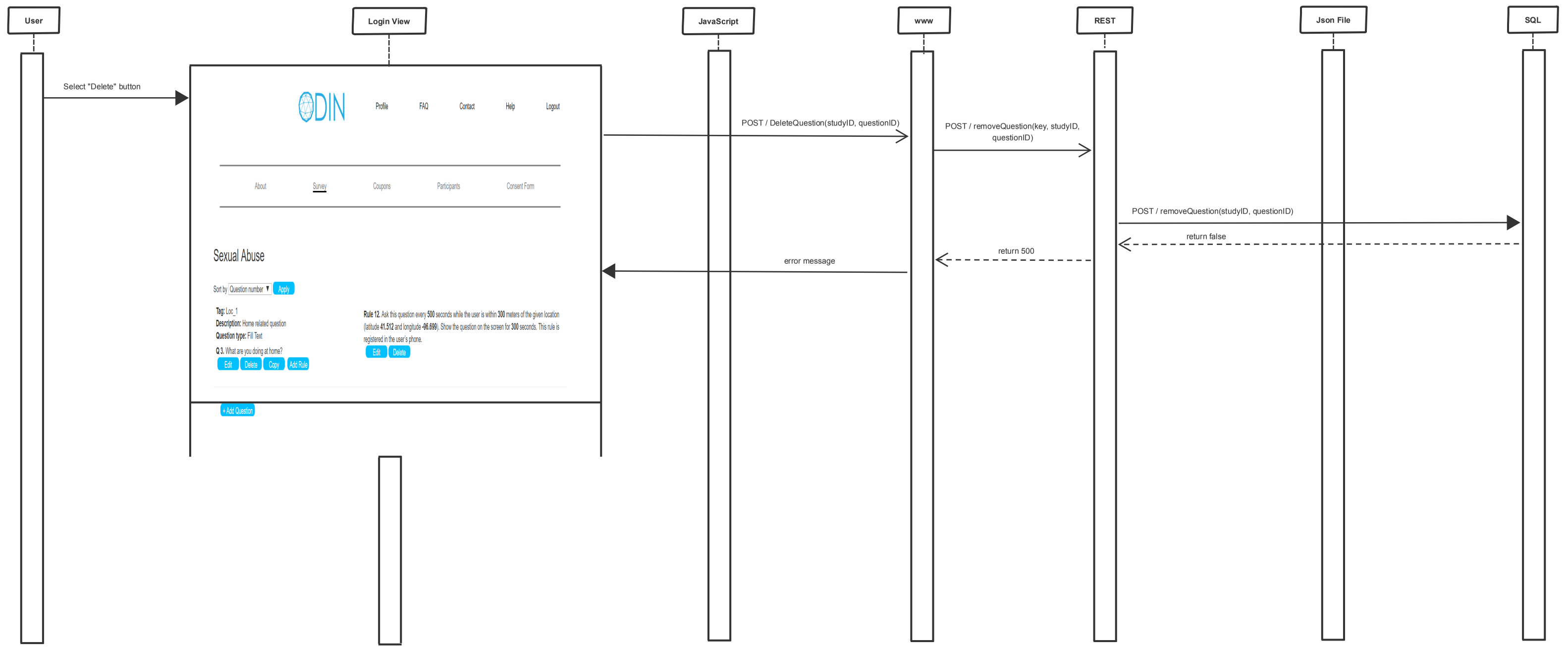

User
Login View
JavaScript
www
REST
Json File
SQL
Select "Delete" button
POST / DeleteQuestion(studyID, questionID)
POST / removeQuestion(key, studyID, questionID)
POST / removeQuestion(studyID, questionID)
return false
return 500
error message
ODIN
Profile
FAQ
Contact
Help
Logout
About
Survey
Coupons
Participants
Consent Form
Sexual Abuse
Sort by Question number
Apply
Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 1. What are you doing at home?
Edit
Delete
Copy
Add Rule
Rule 12. Ask this question every 500 seconds while the user is within 300 meters of the given location (latitude 41.512 and longitude -96.699). Show the question on the screen for 300 seconds. This rule is registered in the user's phone.
Edit
Delete
+ Add Question

Figure .44: ResearcherUI Questions Delete failure

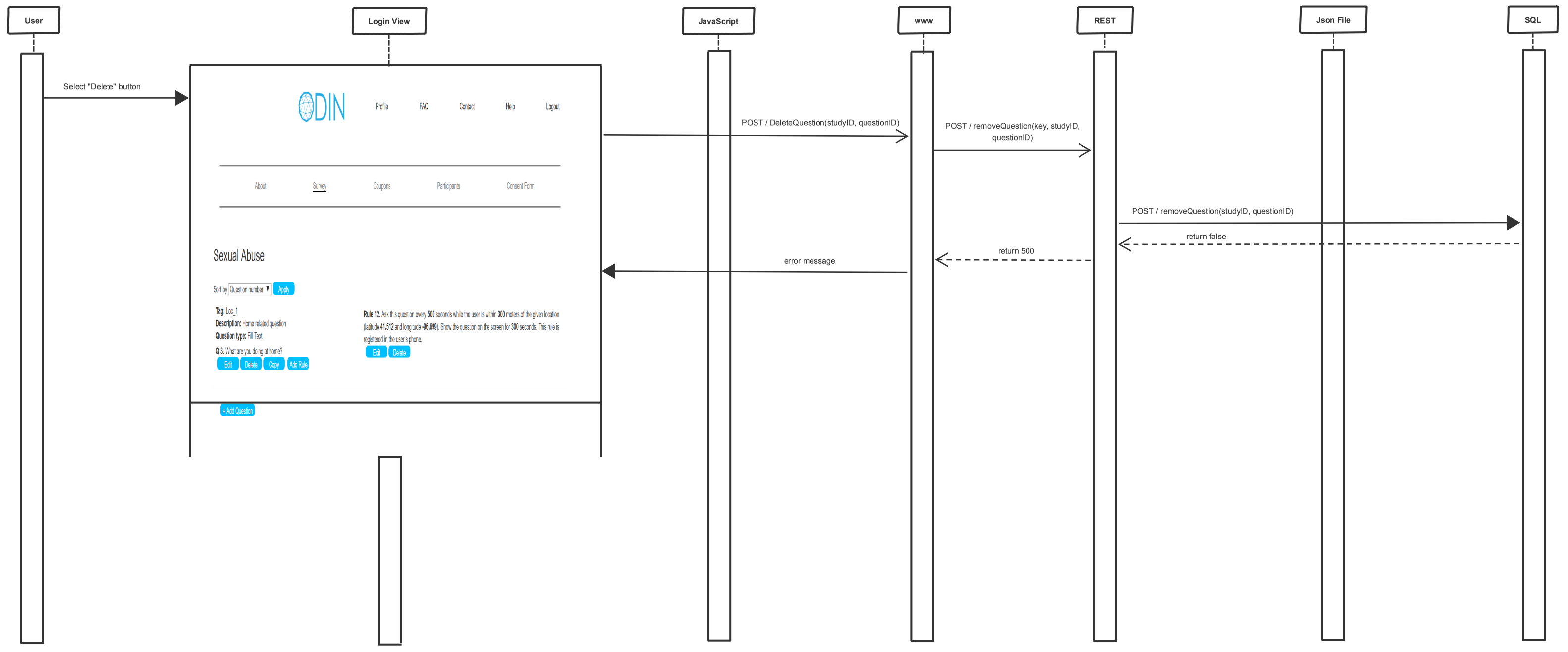
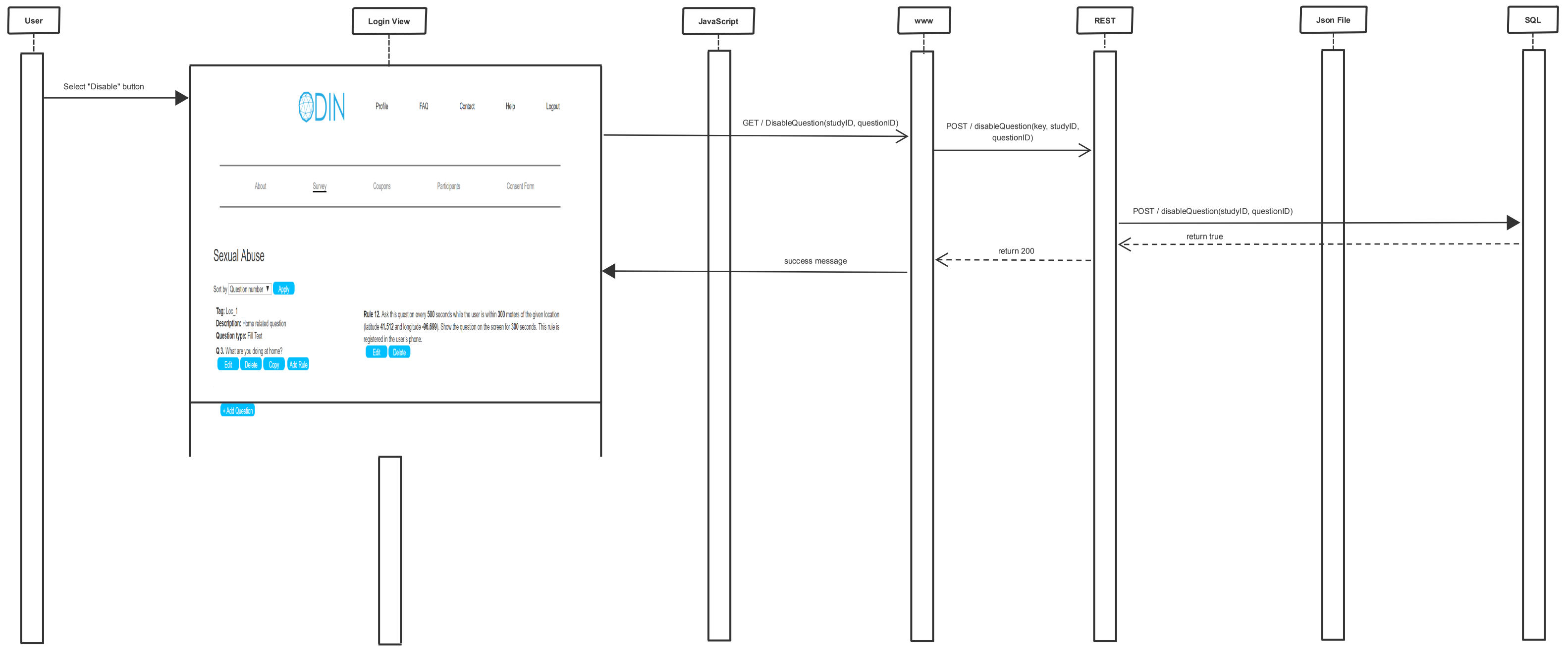

User
Login View
JavaScript
www
REST
Json File
SQL
Select "Delete" button
POST / DeleteQuestion(studyID, questionID)
POST / removeQuestion(key, studyID, questionID)
POST / removeQuestion(studyID, questionID)
return false
return 500
error message
ODIN
Profile
FAQ
Contact
Help
Logout
About
Survey
Coupons
Participants
Consent Form
Sexual Abuse
Sort by
Question number
Apply
Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 1. What are you doing at home?
Edit
Delete
Copy
Add Rule
Rule 12. Ask this question every 500 seconds while the user is within 300 meters of the given location (latitude 41.512 and longitude -96.699). Show the question on the screen for 300 seconds. This rule is registered in the user's phone.
Edit
Delete
+ Add Question

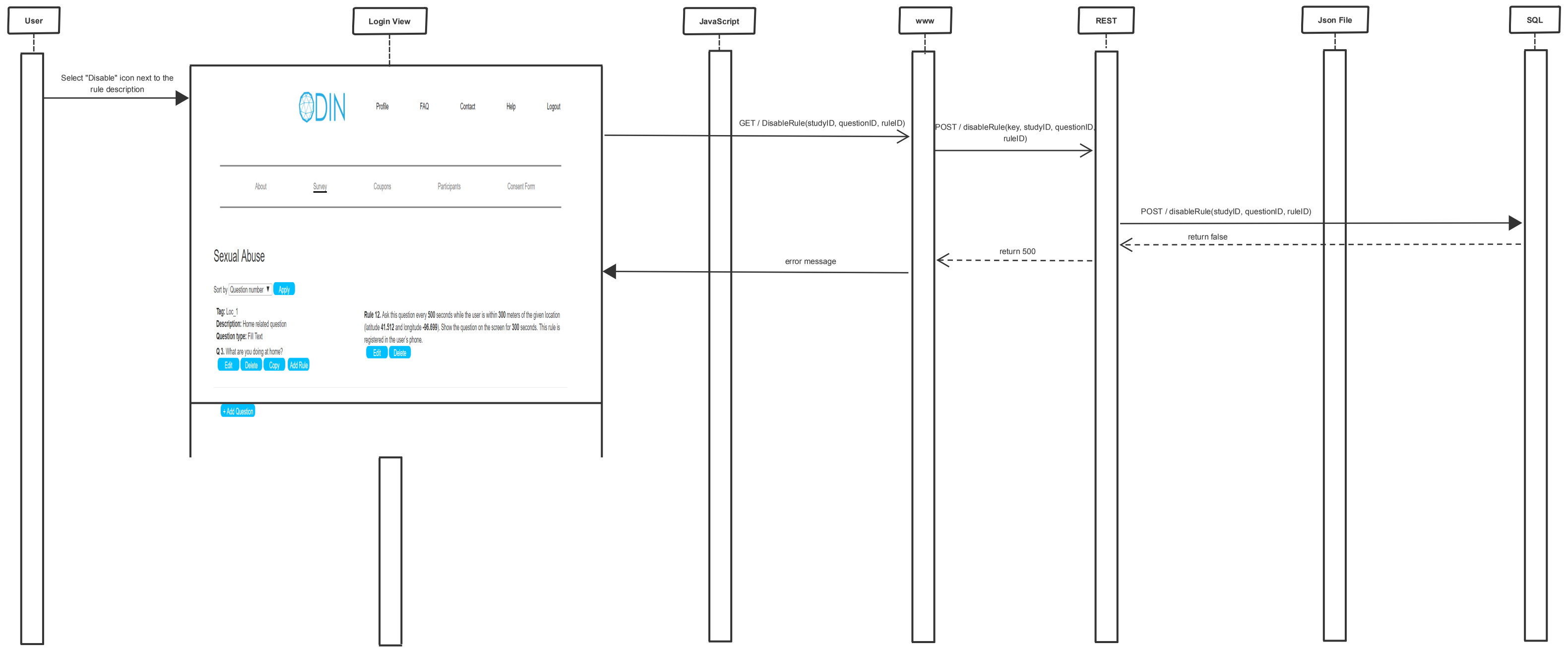
Figure .45: ResearcherUI Questions DisableQuestion failure



Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 1. What are you doing at home?

Rule 12. Ask this question every 500 seconds while the user is within 300 meters of the given location (latitude 41.512 and longitude -96.699). Show the question on the screen for 300 seconds. This rule is registered in the user's phone.

Figure .46: ResearcherUI Questions DisableQuestion success



Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 3. What are you doing at home?

Rule 12. Ask this question every **500** seconds while the user is within **300** meters of the given location (latitude **41.512** and longitude **-96.699**). Show the question on the screen for **300** seconds. This rule is registered in the user's phone.

Figure .47: ResearcherUI Questions DisableRule failure

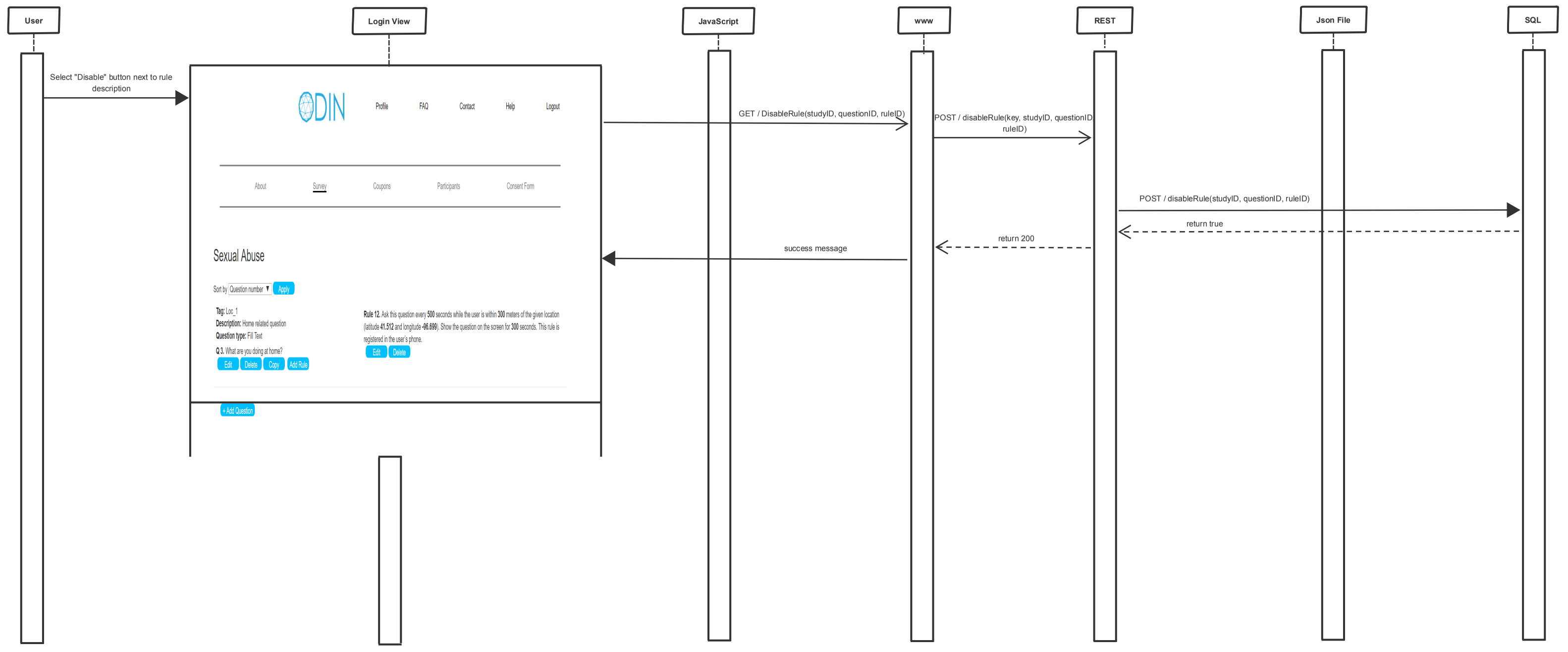


Figure .48: ResearcherUI Questions DisableRule success

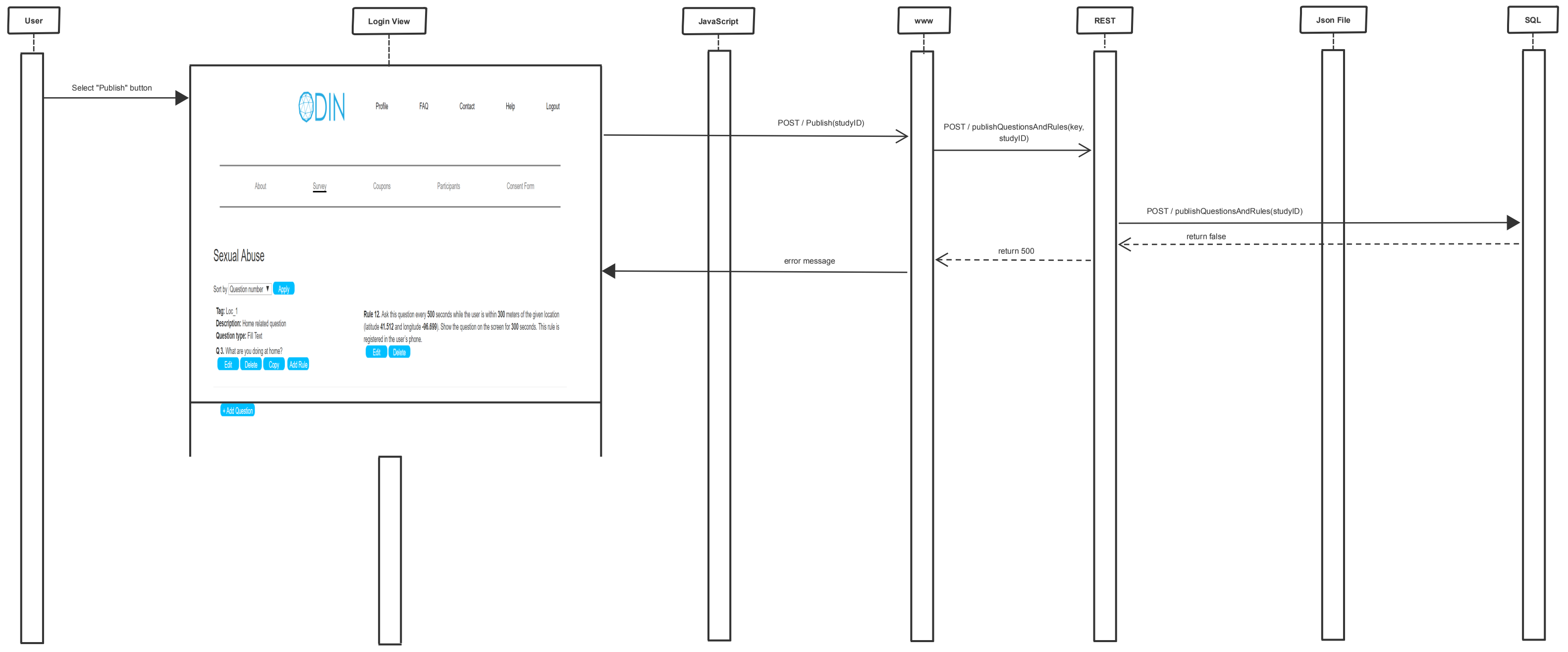
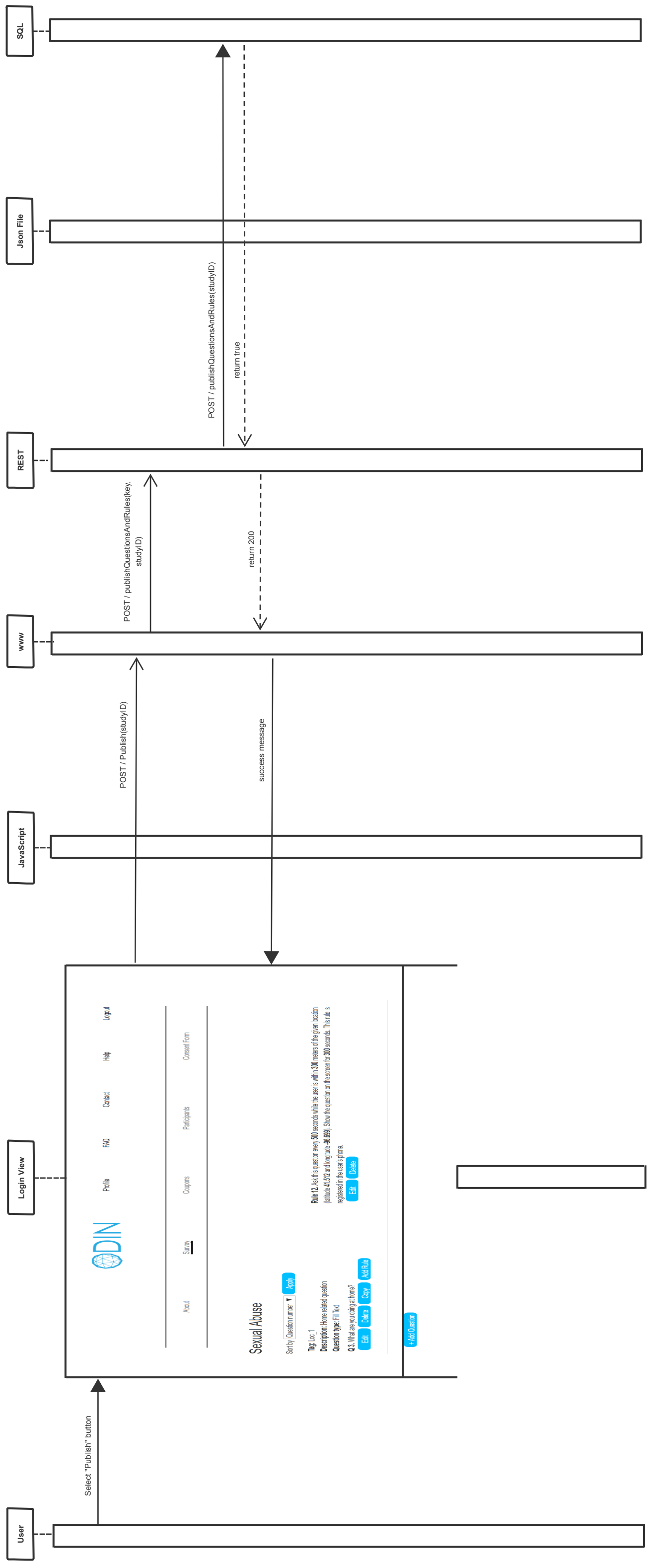


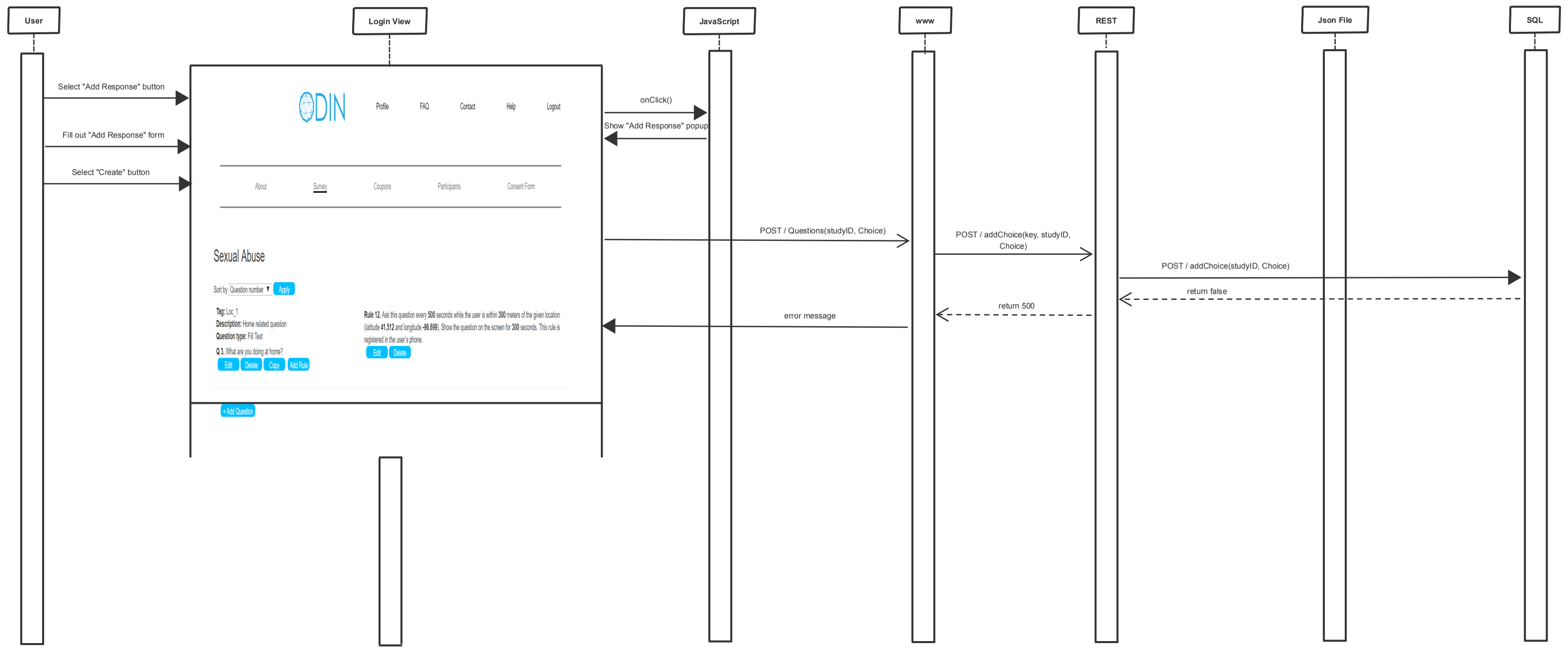
Figure .49: ResearcherUI Questions Publish failure



Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q.3. What are you doing at home?

Rule 12. Ask this question every 500 seconds while the user is within 300 meters of the given location (latitude 41.512 and longitude -96.699). Show the question on the screen for 300 seconds. This rule is registered in the user's phone.

Figure .50: ResearcherUI Questions Publish success



Tag: Loc_1
Description: Home related question
Question type: Fill Text
Q 3. What are you doing at home?

Rule 12. Ask this question every **500** seconds while the user is within **300** meters of the given location (latitude **41.512** and longitude **-96.699**). Show the question on the screen for **300** seconds. This rule is registered in the user's phone.

Figure .51: ResearcherUI Questions addChoice post failure

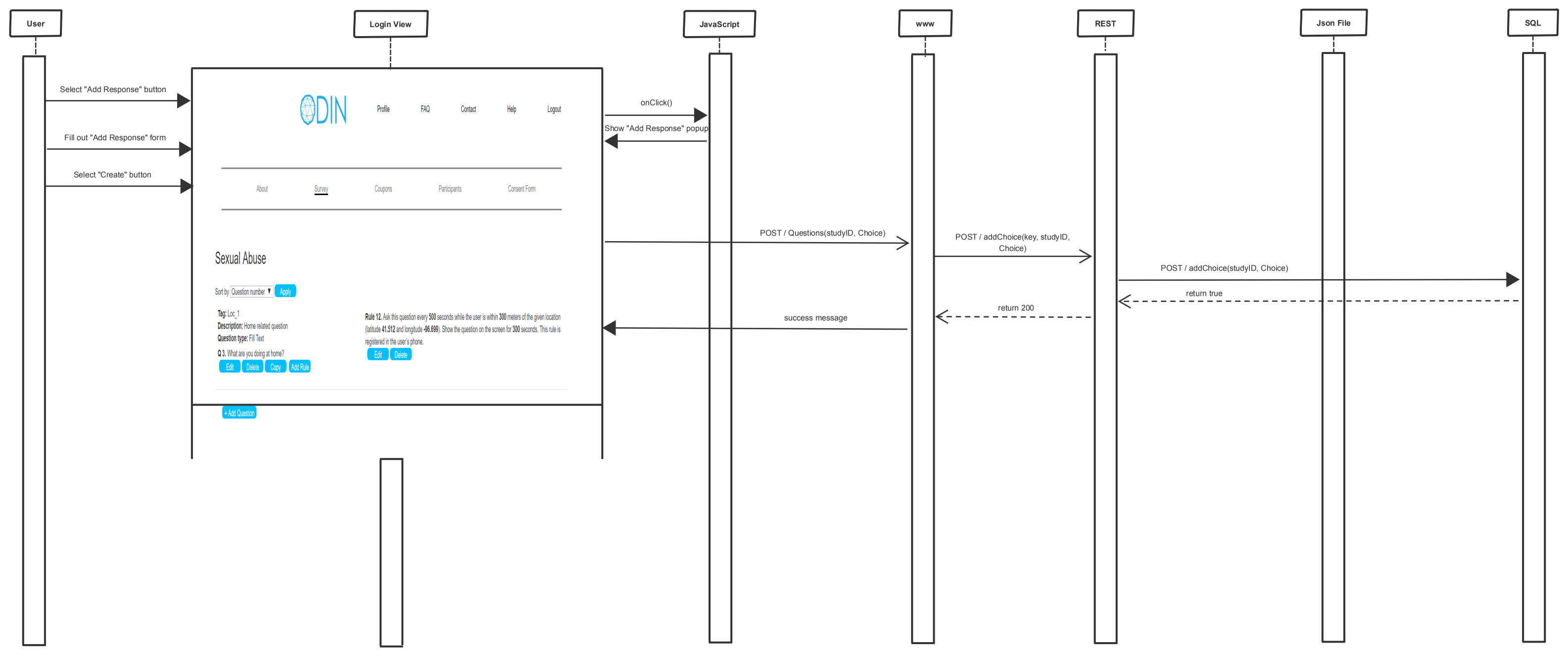


Figure .52: ResearcherUI Questions addChoice post success

Bibliography

[1] James F Kurose and Keith W Ross. *Computer networking: a top-down approach.* Addison Wesley, 2013.

[2] Carter T Butts. Revisiting the foundations of network analysis. *science*, 325(5939):414–416, 2009.

[3] H Russell Bernard, Peter Killworth, David Kronenfeld, and Lee Sailer. The problem of informant accuracy: The validity of retrospective data. *Annual review of anthropology*, 13(1):495–517, 1984.

[4] Mihaly Csikszentmihalyi and Reed Larson. Validity and reliability of the experience-sampling method. In *Flow and the foundations of positive psychology*, pages 35–54. Springer, 2014.

[5] Matthew Wolf-Meyer. Therapy, remedy, cure: disorder and the spatiotemporality of medicine and everyday life. *Medical anthropology*, 33(2):144–159, 2014.

[6] Arthur A Stone, Christine A Bachrach, Jared B Jobe, Howard S Kurtzman, and Virginia S Cain. *The science of self-report: Implications for research and practice.* Psychology Press, 1999.

[7] Scott Carter, Jennifer Mankoff, Scott R Klemmer, and Tara Matthews. Exiting the cleanroom: On ecological validity and ubiquitous computing. *Human–Computer Interaction*, 23(1):47–99, 2008.

[8] Paul Copley. *Marketing Communications Management*. Routledge, 2007.

[9] Inez Myin-Germeys, Margreet Oorschot, Dina Collip, Johan Lataster, Philippe Delespaul, and Jim Van Os. Experience sampling research in psychopathology: opening the black box of daily life. *Psychological medicine*, 39(9):1533–1547, 2009.

[10] Jon A Krosnick. Survey research. *Annual review of psychology*, 50(1):537–567, 1999.

[11] Don A Dillman, Roberta L Sangster, John Tarnai, and Todd H Rockwood. Understanding differences in people's answers to telephone and mail surveys. *New Directions for Evaluation*, 1996(70):45–61, 1996.

[12] Trends, charts, and maps.

[13] Tom AB Snijders, Gerhard G Van de Bunt, and Christian EG Steglich. Introduction to stochastic actor-based models for network dynamics. *Social networks*, 32(1):44–60, 2010.

[14] René Veenstra, Jan Kornelis Dijkstra, Christian Steglich, and Maarten HW Van Zalk. Network–behavior dynamics. *Journal of Research on Adolescence*, 23(3):399–412, 2013.

[15] Ajith Abraham, Aboul-Ella Hassanien, and Vaclav Snášel. *Computational social network analysis: Trends, tools and research advances*. Springer Science & Business Media, 2009.

[16] David J Hand. Statistical analysis of network data: Methods and models by eric d. kolaczyk. *International Statistical Review*, 78(1):135–135, 2010.

[17] Ruth M Ripley, Tom AB Snijders, Zsófia Boda, András Vörös, and Paulina Preciado. Manual for rsiena. university of oxford, department of statistics, 2015.

[18] Arthur A Stone and Saul Shiffman. Ecological momentary assessment (ema) in behavorial medicine. *Annals of Behavioral Medicine*, 1994.

[19] Sunny Consolvo, Beverly Harrison, Ian Smith, Mike Y Chen, Katherine Everitt, Jon Froehlich, and James A Landay. Conducting in situ evaluations for and with ubiquitous computing technologies. *International Journal of Human-Computer Interaction*, 22(1-2):103–118, 2007.

[20] Christie Campbell-Grossman, Diane Brage Hudson, Kathleen M Hanna, Byrav Ramamurthy, and Vishnu Sivadasan. Ease of use and acceptability of a smartphone app for young, low-income mothers. *Journal of Technology in Behavioral Science*, 3(1):5–11, 2018.

[21] Diane Brage Hudson, Christie Campbell-Grossman, Sara Brown, Kathleen M Hanna, Byrav Ramamurthy, Bhargav Gorthi, and Vishnu Sivadasan. Enhanced new mothers network cell phone application intervention: interdisciplinary team development and lessons learned. *Comprehensive child and adolescent nursing*, 40(2):126–135, 2017.

[22] Saul Shiffman, Arthur A Stone, and Michael R Hufford. Ecological momentary assessment. *Annu. Rev. Clin. Psychol.*, 4:1–32, 2008.

[23] Niels Van Berkel, Denzil Ferreira, and Vassilis Kostakos. The experience sampling method on mobile devices. *ACM Computing Surveys (CSUR)*, 50(6):1–40, 2017.

[24] Brian M Bot, Christine Suver, Elias Chaibub Neto, Michael Kellen, Arno Klein, Christopher Bare, Megan Doerr, Abhishek Pratap, John Wilbanks, E Ray Dorsey, et al. The mpower study, parkinson disease mobile data collected using researchkit. *Scientific data*, 3(1):1–9, 2016.

[25] Anna-Kaisa Pietiläinen, Earl Oliver, Jason LeBrun, George Varghese, and Christophe Diot. Mobiclique: middleware for mobile social networking. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 49–54, 2009.

[26] Rui Zhang, Yanchao Zhang, Jinyuan Sun, and Guanhua Yan. Fine-grained private matching for proximity-based mobile social networking. In *2012 Proceedings IEEE INFOCOM*, pages 1969–1977. IEEE, 2012.

[27] Adam C Champion, Zhimin Yang, Boying Zhang, Jiangpeng Dai, Dong Xuan, and Du Li. E-smalltalker: A distributed mobile system for social networking in physical proximity. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1535–1545, 2012.

[28] Jon Froehlich, Mike Y Chen, Sunny Consolvo, Beverly Harrison, and James A Landay. Myexperience: a system for in situ tracing and capturing of user feedback on mobile phones. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 57–70, 2007.

[29] Kiran K Rachuri, Mirco Musolesi, Cecilia Mascolo, Peter J Rentfrow, Chris Longworth, and Andrius Aucinas. Emotionsense: a mobile phones based adaptive platform for experimental social psychology research. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, pages 281–290, 2010.

[30] Dale S Bond, J Graham Thomas, Beth A Ryder, Sivamainthan Vithiananthan, Dieter Pohl, and Rena R Wing. Ecological momentary assessment of the relationship between intention and physical activity behavior in bariatric surgery patients. *International journal of behavioral medicine*, 20(1):82–87, 2013.

[31] Jorinde Eline Spook, Theo Paulussen, Gerjo Kok, and Pepijn Van Empelen. Monitoring dietary intake and physical activity electronically: feasibility, usability, and ecological validity of a mobile-based ecological momentary assessment tool. *Journal of medical Internet research*, 15(9):e214, 2013.

[32] Ingrid Kramer, Claudia JP Simons, Jessica A Hartmann, Claudia Menne-Lothmann, Wolfgang Viechtbauer, Frenk Peeters, Koen Schruers, Alex L van Bemmel, Inez Myin-Germeys, Philippe Delespaul, et al. A therapeutic application of the experience sampling method in the treatment of depression: a randomized controlled trial. *World Psychiatry*, 13(1):68–77, 2014.

[33] Megan A Moreno, Lauren A Jelenchick, Rosalind Koff, Jens C Eickhoff, Natalie Goniu, Angela Davis, Henry N Young, Elizabeth D Cox, and Dimitri A Christakis. Associations between internet use and fitness among college students: an experience sampling approach. *Journal of Interaction Science*, 1(1):4, 2013.

[34]

[35] Daniel Rough and Aaron Quigley. Jeeves-a visual programming environment for mobile experience sampling. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 121–129. IEEE, 2015.

[36] Nikolaos Batalas, Marije aan het Rot, Vassilis Javed Khan, and Panos Markopoulos. Using tempest: End-user programming of web-based ecological

momentary assessment protocols. *Proceedings of the ACM on Human-Computer Interaction*, 2(EICS):1–24, 2018.

[37] Giuseppe Carbonara. Mobile ecological momentary intervention (memi) for dietary behaviour change under transtheoretical model.

[38] Michael R Powell and Wilson J To. Redesigning the research design: Accelerating the pace of research through technology innovation. In *2016 IEEE International Conference on Serious Games and Applications for Health (SeGAH)*, pages 1–5. IEEE, 2016.

[39] Denzil Ferreira, Vassilis Kostakos, and Anind K Dey. Aware: mobile context instrumentation framework. *Frontiers in ICT*, 2:6, 2015.

[40] Apple Introduces ResearchKit. Giving medical researchers the tools to revolutionize medical studies.

[41] Researchstack.

[42] Vikas O'Reilly-Shah and Sean Mackey. Survalytics: an open-source cloud-integrated experience sampling, survey, and analytics and metadata collection module for android operating system apps. *JMIR mHealth and uHealth*, 4(2):e46, 2016.

[43] Stephen M Schueller, Mark Begale, Frank J Penedo, and David C Mohr. Purple: a modular system for developing and deploying behavioral intervention technologies. *Journal of medical Internet research*, 16(7):e181, 2014.

[44] Mehmet Reha Civanlar and Barin Geoffry Haskell. Client-server architecture using internet and public switched networks, November 30 1999. US Patent 5,995,606.

[45] Michael Kofler. What is mysql? In *MySQL*, pages 3–19. Springer, 2001.

[46] Marc Delisle. *Mastering phpMyAdmin 3.4 for effective MySQL management.* Packt Publishing Ltd, 2012.

[47] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Apache Maven.* Alpha Press, 2010.

[48] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, et al. The spring framework–reference documentation. *interface*, 21:27, 2004.

[49] Ed Burnette. *Eclipse IDE Pocket Guide: Using the Full-Featured IDE.* " O'Reilly Media, Inc.", 2005.

[50] Bruce Johnson. *Professional visual studio 2012.* John Wiley & Sons, 2012.

[51] Jon Galloway, Phil Haack, Brad Wilson, and K Scott Allen. *Professional ASP. NET MVC 4.* John Wiley & Sons, 2012.

[52] Belen Cruz Zapata. *Android studio application development.* Packt Publishing Ltd, 2013.

[53] Paul Bryan and Mark Nottingham. Javascript object notation (json) patch. *RFC 6902 (Proposed Standard)*, 2013.

[54] Charitha Kankanamge. *Web services testing with soapUI.* Packt Publishing Ltd, 2012.

[55] Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in action.* Manning Publications Co., 2010.

[56] A Ventayol. The 3 methods for testing your mobile app, 2016.

[57] John Ferguson Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses.* " O'Reilly Media, Inc.", 2011.

[58] Roy Fielding. Hypertext transfer protocol. *HTTP/1.1, Internet Request for Comments (RFC) 2068*, 1997.

[59] David Mosberger and Tai Jin. httperfa tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[60] Aleksa Vukotic and James Goodwill. *Apache Tomcat 7.* Springer, 2011.

[61] Jason Brittain and Ian F Darwin. *Tomcat: The Definitive Guide: The Definitive Guide.* " O'Reilly Media, Inc.", 2007.

[62] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

[63] Jinesh Varia, Sajee Mathew, et al. Overview of amazon web services. *Amazon Web Services*, pages 1–22, 2014.

[64] Mohsen Guizani, Ammar Rayes, Bilal Khan, and Ala Al-Fuqaha. *Network modeling and simulation: a practical perspective.* John Wiley & Sons, 2010.

[65] Javier Ortiz Laguna, Angel García Olaya, and Daniel Borrajo. A dynamic sliding window approach for activity recognition. In *International Conference on User Modeling, Adaptation, and Personalization*, pages 219–230. Springer, 2011.

[66] cogNiTioN.

[67] Rashmi Bajaj, Samantha Lalinda Ranaweera, and Dharma P Agrawal. Gps: location-tracking technology. *Computer*, 35(4):92–94, 2002.

[68] Jennifer Bray and Charles F Sturman. *Bluetooth 1.1: connect without cables*. pearson Education, 2001.

[69] Nirupama Bulusu, John Heidemann, and Deborah Estrin. Adaptive beacon placement. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 489–498. IEEE, 2001.

[70] Jennifer R Kwapisz, Gary M Weiss, and Samuel A Moore. Activity recognition using cell phone accelerometers. *ACM SigKDD Explorations Newsletter*, 12(2):74–82, 2011.

[71] Cameron McCarthy, Nikhilesh Pradhan, Calum Redpath, and Andy Adler. Validation of the empatica e4 wristband. In *2016 IEEE EMBS International Student Conference (ISC)*, pages 1–4. IEEE, 2016.

[72] Marko Gargenta. *Learning android*. " O'Reilly Media, Inc.", 2011.

[73] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Commun. ACM*, 37(8):7682, August 1994.

[74] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext transfer protocol–http/1.0, 1996.

[75] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[76] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings fifth ieee international enterprise distributed object computing conference*, pages 118–127. IEEE, 2001.

[77] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* " O'Reilly Media, Inc.", 2011.

[78] John Heidemann, Katia Obraczka, and Joe Touch. Modeling the performance of http over several transport protocols. *IEEE/ACM transactions on networking*, 5(5):616–630, 1997.

[79] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. Network performance effects of http/1.1, css1, and png. In *ACM SIGCOMM Computer Communication Review*, volume 27, pages 155–166. ACM, 1997.

[80] Hongtao Wang, Yuehui Jin, Wendong Wang, Jian Ma, and Dongmei Zhang. The performance comparison of prsctp, tcp and udp for mpeg-4 multimedia traffic in mobile network. In *International Conference on Communication Technology Proceedings, 2003. ICCT 2003.*, volume 1, pages 403–406. IEEE, 2003.

[81] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[82] Douglas Lea. *Concurrent programming in Java: design principles and patterns.* Addison-Wesley Professional, 2000.

[83] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[84] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[85] Sahil Batra. Improving quality using testing strategies. *Journal of Global Research in Computer Science*, 2(6):113–117, 2011.

[86] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software Second Edition*. Dreamtech Press, 2000.

[87] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[88] Edward H Bersoff and Alan M Davis. Impacts of life cycle models on software configuration management. *Communications of the ACM*, 34(8):104–119, 1991.

[89] Abhijit A Sawant, Pranit H Bari, and PM Chawan. Software testing techniques and strategies. *International Journal of Engineering Research and Applications (IJERA)*, 2(3):980–986, 2012.

[90] Mohd Ehmer Khan. Different forms of software testing techniques for finding errors. *International Journal of Computer Science Issues (IJCSI)*, 7(3):24, 2010.

[91] Ajay Jangra, Gurbaj Singh, Jasbir Singh, and Rajesh Verma. Exploring testing strategies. *International Journal of Information Technology and Knowledge Management*, 4:297–299, 2011.

[92] Henry H Liu. *Software performance and scalability: a quantitative approach*, volume 7. John Wiley & Sons, 2011.

[93] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure

diagnosis with proactive logging. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 293–306, 2012.

[94] Colin Eberhardt. The art of logging, Mar 2014.

[95] Jeff Atwood. Coding horror.

[96] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 415–425. IEEE Press, 2015.

[97] Gary McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[98] Steven M Bellovin and William R Cheswick. Network firewalls. *IEEE communications magazine*, 32(9):50–57, 1994.

[99] Carole Cadwalladr and Emma Graham-Harrison. Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach. *The guardian*, 17:22, 2018.

[100] Tim Greene. Biggest data breaches of 2015. *Network World*, 2015:1–6, 2015.

[101] Sam Thielman. Yahoo hack: 1bn accounts compromised by biggest data breach in history. *The Guardian*, 15:2016, 2016.

[102] Gary McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[103] Randall K Nichols, Panos Lekkas, and Panos C Lekkas. *Wireless security.* McGraw-Hill Professional Publishing, 2001.

[104] Hana R Esmaeel. Apply android studio (sdk) tools. *International Journal*, 5(5), 2015.

[105] Zhao Yong-Xia and Zhen Ge. Md5 research. In *2010 second international conference on multimedia and information technology*, volume 2, pages 271–273. IEEE, 2010.