

Mathis Erichsen

Developing a Digital Intervention in the Context of Depressive Symptoms: The Mood Diary Application

Bachelor Thesis

at the Chair for Practical Computer Science
(University of Münster)

Principal Supervisor: Prof. Dr. Herbert Kuchen
Associate Supervisor: Prof. Dr. Markus Müller-Olm

Presented by: Mathis Erichsen [418182]
Windthorststrasse 39
48143 Münster
+49 1578 9516922
mathis.erichsen@uni-muenster.de

Submission: 31st October 2023

Contents

Figures	IV
Tables	V
Listings	VI
Abbreviations	VII
1 Introduction	1
2 Development Considerations and Technology Selection	4
2.1 Technical requirements	4
2.2 Technology Selection	6
2.2.1 General Approach	6
2.2.2 Specific Framework	7
2.2.3 Asynchronous Task Processing	10
2.2.4 Portability and Easy Deployment	11
2.2.5 Quality and Speed of Development	12
2.3 Resulting Technology Stack	12
3 Technologies and Concepts	13
3.1 Progressive Web App	13
3.2 Django	15
3.3 Celery	18
3.4 Docker	20
3.5 GitLab Continuous Integration/Continuous Delivery	21
4 The Mood Diary Intervention	24
5 Related Work	27
6 Conceptualization of the Mood Diary Application	28
6.1 Architecture	28
6.2 Digital Mood Diary Intervention	30
6.3 Counselor Access	33
6.4 Automatic Analyses	34
7 Implementation of the Mood Diary Application	39
7.1 Data Model	39
7.2 Project Structure	40
7.3 Functionality Overview	43
7.4 Implementation Details	46
7.4.1 Digital Mood Diary Intervention	46
7.4.2 Counselor Access	48
7.4.3 Automatic Analyses	50
7.4.4 Progressive Web App Enhancement	52
7.4.5 Continuous Integration/Continuous Delivery Process	53
8 Evaluation of the Mood Diary Application	55
8.1 Main Goals	55
8.2 Technical Requirements	56

8.3 Limitations and Future Work	57
9 Conclusion	59
Appendix	60
References	63
Declaration of Academic Integrity	70

Figures

Figure 1	Task Queue Pattern	10
Figure 2	Handling Push Notifications via a Service Worker	15
Figure 3	Request/Response Processing in a <i>Django</i> Application	17
Figure 4	<i>Celery</i> Task Processing	19
Figure 5	Example Mood Diary Paper Form	25
Figure 6	Mood Diary Application Architecture	29
Figure 7	Mood Diary Entry Input Form and Dashboard View	31
Figure 8	Mood Diary Entry List and Detail Views	32
Figure 9	Mood Diary Application Frame for Views and Forms	32
Figure 10	Client List View and Input Form	33
Figure 11	Client Creation Process	34
Figure 12	Notification List and Detail Views	36
Figure 13	Rule Configuration View	37
Figure 14	Data Model for the Mood Diary Application	40
Figure 15	Structure of the Project's Code Base	41
Figure 16	Examples for Shared Functionality from the Core App	42
Figure 17	Overview of User-Facing Functionality	45
Figure 18	Passing Data from a <i>Django</i> Template to JavaScript Code	48
Figure 19	Releasing Mood Diary Entries	49
Figure 20	Event-based Rule Evaluation	51
Figure 21	Progressive Web App Installation Prompt	52
Figure 22	Screenshots Desktop Computer	61
Figure 23	Screenshots Mobile Device	62

Tables

Table 1	Decision Matrix for Selecting a Web Development Framework	9
Table 2	Mood Assessment Scale	30
Table 3	Rule Preconditions Categorized by Criterion and Evaluation Type ..	38
Table 4	Excerpt of Activities and Categories for the Mood Diary Application	60

Listings

1	Example <i>Celery</i> Task	19
2	Example <i>Dockerfile</i>	20
3	Example <i>Docker Compose</i> File	21
4	Example <i>GitLab</i> Pipeline	23
9	Data Transformation Methods on the Mood Diary Model	46
10	Data Transformation Methods on the Mood Diary Model	47
18	Abstract Base Rule Class	51
21	Continuous Integration Test Job	53
22	Continuous Delivery Deploy Job	54

Abbreviations

API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DiGA	Digitale Gesundheitsanwendungen
DSL	Domain-Specific Language
GPS	Global Positioning System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JS	JavaScript
JSON	JavaScript Object Notation
MVC	Model-View-Controller
MVT	Model-View-Template
ORM	Object Relational Mapping
OS	Operating System
PK	Primary Key
PSB	Psychosoziale Beratungsstelle des Studentenwerks Osnabrück
PWA	Progressive Web App
SQL	Structured Query Language
SSH	Secure Shell
URL	Uniform Resource Locator
VAPID	Voluntary Application Server Identification for Web Push
VCS	Version Control System
WHO	World Health Organization

1 Introduction

Mental health issues are widespread. In 2017, the World Health Organization (WHO) reported the global prevalence of major depressive disorder alone (simply “depression” in colloquial language) to be at 4.4% [Wor17]. In adolescents, this prevalence even amounted to 8% in 2020, with up to 34% of them reporting depressive symptoms thus being at risk of developing a major depressive disorder [SNW22]. This state of high prevalence for psychological stress and disorders intensified in the context of the COVID-19 pandemic [Cam+20; Win+20], leading to an ever-increasing demand for mental health care services like psychological counseling or psychotherapy. On the other side of this huge demand, there are too few of such services available. This results in long waiting times for e.g., psychotherapy that can easily amount to several months [Bun18; Fun21].

Expanding mental health services is a structural change that will take years to produce a significant effect. Therefore, it is vital that the existing services are efficient and effective in order to help as many people as possible with the limited resources at hand. One possible tool to increase efficiency and effectiveness of mental health services can be digital solutions. This starts at digitizing administrative processes like scheduling appointments to reduce operational work load and reaches well into supporting concrete processes within the actual health service of for example psychological counseling or psychotherapy.

One such process that is widely used in psychological counseling and psychotherapy with regard to depressive symptoms is the so-called mood diary intervention [Hor+22; GH09]. As described in therapy manuals, an important goal of the intervention is to establish associations between mood and behavior [Hau21]. This is done by having clients log their (non-)activities together with the mood they felt when executing the activities over a period of at least two weeks. Usually, this intervention is analog, i.e., completed in a paper-pencil form. Research indicates that these forms of paper-pencil homework clients have to complete between therapy or counseling sessions often lack compliance [GLN06; GS02; HF04]. As completion of such homework is significantly associated to treatment effectiveness [PBP88; BS00; HF04], a lack of compliance endangers this very same effectiveness.

Providing the mood diary intervention as a digital application that is available on a smartphone could well ensure clients’ compliance with the intervention thus raising its effectiveness. Although some researchers note that they view compliance, also with digital interventions, in general to be an area of concern [Mos+21], others state that mobile applications can increase compliance [TK17]. In fact, a study specifi-

cally investigating compliance in adolescents for tracking mood analogously versus via a smartphone application found it to be significantly higher for the smartphone application[Mat+08]. Along with a potentially better compliance, there are other advantages linked to a digital application. These include but are not limited to reducing the risk of losing or accidentally destroying data and a better data privacy as mood diary entries can be password-protected. Also, a digital application leads to an increased convenience as most people have their smartphone with them at most places during most times so that no additional equipment for completing the intervention is needed. Additionally, there is the possibility of using push notifications to alert clients.

What is more, realizing the mood diary intervention in form of a digital application offers room for improvements that were not feasible in the analog version. First, in a digital application, it would be possible to make mood diary entries created by clients accessible to their respective counselors prior to the next counseling or therapy session. With this access, counselors could monitor the intervention process and prepare upcoming sessions based on the mood diary data instead of only accessing it during the next session when it is brought in by their clients. Second, clients could receive more timely feedback when using a digital application as compared to the analog form of the intervention. Within a digital application, mood diary entries could be analyzed automatically and thus deliver insights or guidance about clients' moods and activities in the short term, while in the analog intervention, no feedback can be received before the next counseling or therapy session.

Therefore, this project aims to (1) develop a digital application for the mood diary intervention available on a smartphone to ensure client compliance that additionally (2) makes mood diary entries accessible to counselors and (3) provides timely feedback about entries to clients through automatic analyses. The resulting application, from here on referred to as the mood diary application, seeks to help making much-needed mental health services more effective. Specifically, the Psychosoziale Beratungsstelle des Studentenwerks Osnabrück (PSB) plans to use it to replace the analog mood diary intervention they currently apply in psychological counseling.

The PSB offers easy-access, short-term, individual counseling for students who suffer from psychological stress. Students represent one of the most vulnerable groups for psychological stress as psychological disorders are highly prevalent in young adults and individuals beginning a new phase in their life [Nat08; Ziv+09; Aue+18]. Thus, for these individuals, early and easy access to mental health services is of high importance. The PSB aims to prevent the onset of these disorders or to intervene in their very beginnings. To this end, in the context of depressive symptoms, coun-

selors use, among other techniques, the mood diary intervention to help their clients understand associations between mood and behavior.

The remainder of this work is structured as follows: In section 2, technical requirements for the mood diary application will be derived based on the project's goals developed above. These requirements will subsequently be used to select a suitable technology stack. The selected technology stack along with associated concepts will be presented in greater detail in section 3. In section 4, more light will be shed on the mood diary intervention to establish a mutual basis of understanding of its underlying psychological concepts and how this intervention is typically administered in practice. Section 5 will then shortly summarize existing work in the field of digital mood diary interventions. After that, section 6 will provide an overview of the conceptualization of the mood diary application. Section 7 will then deal with its actual implementation. In section 8, the mood diary application will be evaluated with respect to the project's goals and technical requirements, while also demonstrating limitations and ideas for future work. A brief conclusion in section 9 will finally wrap up the work presented here.

2 Development Considerations and Technology Selection

In a first step, this section will derive technical requirements for the mood diary application. In a second step, it will lead through the selection of a suitable technology stack based on these requirements.

2.1 Technical requirements

First, the mood diary application should be available for smartphones so that clients can conveniently access it during their everyday life. Counselors, however, are much more likely to access the application during their office hours from their work computers. In effect, developing an application that offers access to both clients and counselors requires the application to be platform-independent, i.e., to be accessible from both smartphones and desktop computers. Besides, the mood diary application should provide easy and extensive access, meaning that it should be usable without a complex installation procedure and work on every type of device. This would ensure that preferably all clients were able to use the application in an easy-access manner, independent of the type of smartphone they possess or the Operating System (OS) it is running on.

Second, automatic analyses of the mood diary entries logged by clients have to be executed independently from user interactions. They have to run in the background, not being explicitly triggered by a user interaction like the clicking of a button, but in an event- or time-based manner. For example, an analysis could be executed every time a specific mood diary entry is logged or when a specific time limit is reached. That means that a form of asynchronous task processing must be implemented recognizing when an analysis should be run with the capability to run it independent from the application's main process.

Third, as the mood diary application is to be used by the PSB to replace the analog version of the mood diary intervention applied so far, it is likely that eventually, some of their staff members will take over the task of maintaining or even advancing the mood diary application. Anticipating this scenario, it would be best to develop the application based on a limited number of popular and, if possible, easy-to-learn technologies. This would increase the probability of finding staff members of the PSB having the necessary skills to take over maintenance and maybe even advancement of the mood diary application.

Likewise, designing the mood diary application to be easy to deploy would reduce complexity in this area, simplifying maintenance and advancement. What is more,

additionally designing the application to be portable between host systems would work towards building a long-lived application that can easily adapt to a changing OS or architecture of the host it is running on.

Another crucial aspect to consider is the type of data the mood diary application should store. Although it could be avoided to associate the data to personally identifiable information like clients' real names, the mood diary entries nonetheless contain sensitive and private information about their daily activities and moods. These circumstances demand a complete data sovereignty that is with the PSB, meaning that no data can be stored with third parties, but that all data must remain on infrastructure provided by the PSB alone.

Furthermore, the development of the mood diary application must take into account how many clients are expected to use the application. This is important to ensure that the performance of the application can handle the amount of requests or database traffic that comes with said user base. Only then, the application can run smoothly and error-free, allowing it to be used for its intended purpose. In this case, with the PSB supervising around 1,000 clients per year of which around 30% exhibit psychological problems for which a mood diary intervention is suitable, a user base of up to 300 active clients can be expected.

Lastly, there is only a time span of nine weeks available to develop the mood diary application. Despite this time constraint, the application should naturally be characterized by a high quality, resulting in a robust and well-functioning software. Therefore, the development process should be characterized by general quality and speed as to deliver a quality software solution in a limited amount of time.

To summarize, the following list of technical requirements can be derived for the mood diary application:

- **Platform-independent with easy and extensive access**
- **Asynchronous task processing**
- **Limited amount of popular, at best easy-to-learn technologies**
- **Portability with easy deployment on most systems**
- **Data sovereignty (no data stored with third parties)**
- **Application performance supporting up to 300 users**
- **Quality and speed of development**

2.2 Technology Selection

Being equipped with a list of technical requirements, as a second step, a technology stack matching these requirements will be selected. The selection process should lead to an informed decision based on a range of alternatives for each part of the stack. Selected technologies and associated concepts will be described in greater detail in section 3.

2.2.1 General Approach

There are different approaches for developing an application that is available on mobile devices like smartphones: Native applications, hybrid applications, and (responsive) web applications [SHG13]. Native applications are developed with particular tools aimed at a specific mobile device OS and run natively on it. Hybrid applications also run natively on the OS of a mobile device, but they are developed like a web application and then packed into a native application container. Web applications usually rely on web technologies like Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS) and run in web browsers.

Comparing these approaches, native applications provide better access to device features like cameras, Global Positioning System (GPS) or push notifications than web applications [Job13]. They can have a higher performance as they run natively and not through extra layers like a web browser and can offer a richer user experience [SHG13]. On the other hand, native applications must be developed specifically for each mobile device OS [Sel+13] and need to be distributed via app stores and then installed to a mobile device in order to be used. Although there are cross-platform approaches for the development of native applications like *React Native*¹ or *Flutter*², trying to lift the burden of having to develop one application per supported OS, both native and hybrid apps in the end nonetheless require to be distributed separately for each mobile device OS. Therefore, also with cross-platform approaches for developing a native application or when choosing the hybrid approach, one application per mobile device OS would have to be maintained. This would come at the risk of not covering all mobile device OS types used by clients. Moreover, there would still be the necessity to develop an extra application to run on desktop computers. Web applications however, are truly platform-independent, making it possible to develop one application that can run on all mobile devices and also on desktop computers via their respective web browsers, requiring no installation procedure.

¹ <https://reactnative.dev/>

² <https://flutter.dev/>

For the mood diary application, neither extensive access to device features nor extensive computational capabilities should be necessary, but there is the requirement of a platform-independent application with easy and extensive access. Therefore, it seems reasonable to develop a responsive web application here.

This choice however is not without downsides. It could be the case that a less rich user experience reduces compliance with the mood diary application. Also, using push notifications to alert clients about new feedback might be a good way of keeping up compliance, which is not possible with a responsive web application. Luckily, these potential shortcomings can be addressed by providing Progressive Web App (PWA) functionality to the mood diary application. PWA is an umbrella term for a set of technologies and concepts intended to make a web application look, feel, and function more like a native application [BMG18]. To this effect, a PWA can, for example, offer an improved user experience and provide push notifications. Hence, it seems promising to develop the mood diary application with PWA functionality, thereby achieving a platform-independent application with easy and extensive access while at the same time addressing some of the shortcomings web applications suffer in comparison to native ones.

2.2.2 Specific Framework

With a general approach set, it is possible to select a specific web development framework within that approach to develop the mood diary application. There is a wide range of such frameworks written in numerous programming languages that could be chosen for this task. Already in 2008, researchers compared over 80 different web development frameworks [VK08]. Therefore, a programming language is chosen first to narrow down the number of potential frameworks. This choice is guided by the requirement of employing a limited amount of popular, at best easy-to-learn technologies. As web applications make use of JS [SHG13], this programming language would be an obvious candidate. However, it is not JS but Python that is recommended as an entry point to programming for beginners and used for this purpose by a wide range of universities [She15; BZS13]. What is more, Python is extremely popular and boasts an extensive community [Sri17; SFV19]. Consequently being a popular, easy-to-learn technology, Python is chosen for the purpose of this project.

Within Python, there are again numerous web development frameworks available. The Stack Overflow Developer Survey 2023 taken by over 90,000 developers [Ove23]

lists *Django*³, *FastAPI*⁴, and *Flask*⁵ as the currently most popular ones, which will subsequently be taken into consideration. Although there have been evaluations of e.g., *Django* in academia based on a variety of criteria [Cur+19], here, in the interest of limiting the number of employed technologies, it is most relevant to select a full-stack web development framework. That means that the selected framework should provide an all-in-one solution for developing the mood diary application, thereby covering specific key aspects. Leaning onto the work of the framework comparison mentioned above [VK08], the following key aspects should be provided by the selected web development framework:

- **Presentation layer:** The framework is able to render content in a client’s browser.
- **Form handling including validation:** The framework provides means to deal with user input submitted via forms, including the possibility to validate that input.
- **Authentication:** The framework offers a secure way for users to authenticate themselves and to handle their authentication state (e.g., by providing login/logout functionality).
- **Backend-integration:** The framework includes abstractions to access backend services like databases.

Extending this list, two more key aspects should be taken into consideration:

- **Open source:** The framework provides full access to its source code.
- **Testing support:** The framework offers some integrated tools for testing applications developed with it.

Below, the three candidate frameworks (*Django*, *FastAPI*, and *Flask*) will be evaluated based on their fulfillment of these key aspects. No fulfillment will be represented as - (-1 point), partial fulfillment as *o* (0 points) and total fulfillment as + (1 point). From the resulting decision matrix, a score showing how much each framework matches these key aspects can be derived. The evaluation will be based on the respective documentations that are available online. Please refer to Tab. 1 for the results.

³ <https://www.djangoproject.com/>

⁴ <https://fastapi.tiangolo.com/>

⁵ <https://flask.palletsprojects.com/>

	Django	FastAPI	Flask
Presentation Layer	+	-	+
Form Handling	+	+	o
Authentication	+	o	o
Backend-Intergration	+	o	o
Open Source	+	+	+
Testing Support	+	+	+
Sum	6	2	3

Table 1 Decision Matrix for Selecting a Web Development Framework

As *FastAPI* scored the lowest, it becomes obvious that this framework is not the right choice for the task at hand. Although listed as a web development framework, *FastAPI* really is a framework for building Application Programming Interfaces (APIs), and not whole web applications, therefore completely lacking a presentation layer. *Flask* provides more functionality in this domain, resulting in a higher score. It is a lightweight framework with a lot of flexibility that offers many key aspects through extensions or dependencies. That is why half of them are marked as partially fulfilled in the result matrix - they can be fulfilled by adding extensions or dependencies, but not directly through *Flask* itself. *Django* reaches the highest score, as it ships with all the proposed key aspects included. It boasts its own templating engine for the presentation layer, form classes with validation methods, native authentication middlewares, an own Object Relational Mapping (ORM) with extensive relational database support for a backend-integration, is fully open source and provides various testing tools. Together with its authentication functionality comes the possibility to manage users and their accounts together with the rest of the application data in the same database, thus matching the requirement for data sovereignty. What is more, *Django* together with a relational *PostgreSQL*⁶ database has been successfully used in live applications (e.g., [V+22]), hence meeting the performance requirement of supporting 300 users with ease. Consequently being a full-stack web development framework covering all key aspects listed above, *Django* in combination with a *PostgreSQL* database will be used to develop the mood diary application.

⁶ <https://www.postgresql.org/>, the currently most popular database [Ove23].

2.2.3 Asynchronous Task Processing

The selected general approach and specific web development framework for the mood diary application already cover several technical requirements. These are namely platform-independence, data sovereignty, and application performance, all while concentrating on a limited amount of popular, easy-to-learn technologies. To also include the remaining three requirements (i.e., asynchronous task processing, portability and easy deployment as well as quality and speed of development), the technology stack must be extended a bit further.

Addressing asynchronous task processing, as described above, the mood diary application should run automatic analyses based on mood diary entries logged by clients. These should be triggered in an event- or time-based manner, being executed separately from the application's main process. Each analysis that should be executed, whether triggered event- or time-based, can be thought of as a work package, or a task. A common pattern to deal with handling such tasks and their processing is a so-called task queue, consisting of a scheduler, a queue, and one or more workers (see Fig. 1). The main application (the mood diary application in our context) sends a task to the queue when a specific event occurs, while the scheduler takes care of queuing tasks based on time intervals. The queue stores the tasks until they are executed by a worker. Like that, tasks, such as analyses of the logged mood diary entries, can be triggered in an event- or time-based manner and executed asynchronously, separated from the main application.

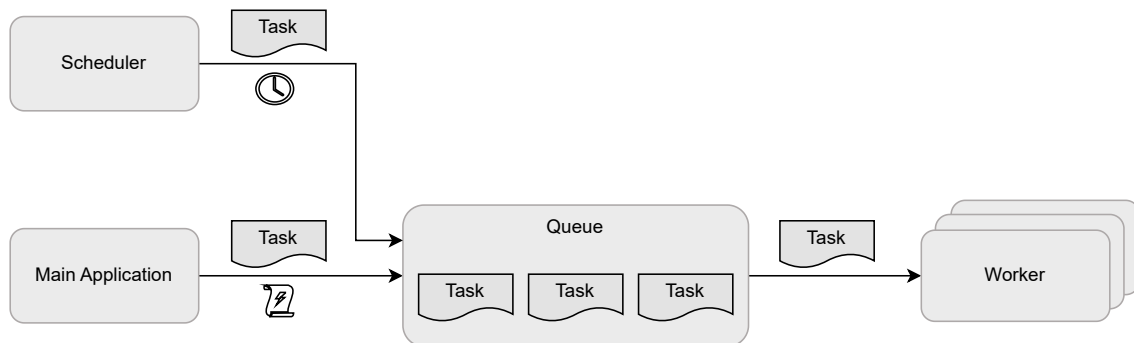


Figure 1 Task Queue Pattern

The Python ecosystem offers many different implementations of the task queue pattern. As a starting point for selecting one of those, the Awesome Python GitHub repository⁷ providing a categorized overview of tools developed by the Python community will be consulted. Although certainly only depicting a part of the tool

⁷ <https://github.com/vinta/awesome-python>

landscape available in Python, this repository can offer good guidance. In the category of task queues, it lists five different candidates: *Celery*⁸, *Dramatiq*⁹, *Huey*¹⁰, *Mrq*¹¹, and *Rq*¹². At this point, it is not conceivable that the automatic analyses for the mood diary application will require any special details apart from the regular functionality of a task queue. Therefore, selection will be based on the respective implementation’s popularity. According to stars awarded for their GitHub repositories, the most popular task queue of the five candidates is by far *Celery*. It possesses over 21,000 stars, while *Rq* in the second places reaches 9,000 stars. Beyond this popularity measure, *Celery* has also been successfully used in academic work (e.g., [New+13; Bau+19]). In effect, asynchronous task processing for the mood diary application will be implemented via *Celery*.

2.2.4 Portability and Easy Deployment

When thinking about how to develop the mood diary application to be portable and easily deployed, virtualization emerges as a valuable concept. Virtualization allows the mood diary application to be isolated from the underlying host system, meaning that it can run in its own virtual environment abstracted away from the actual hardware of the machine it is located on. This implies that no manual dependency installations or complex server setups are necessary. Instead, the application and everything it needs to work can be packaged together to run isolated from the underlying machine. Virtualization mitigates potential conflicts that could arise from running multiple applications on the same server and enables easy transfer between different host systems, thus ensuring a seamless deployment.

Among the available tools for virtualization, *Docker*¹³ stands out as a compelling implementation of this concept. It is so popular and wide-used that it reached the first place in the category “other tools” in the Stack Overflow Developer Survey 2023 [Ove23]. More than 50% of the over 80,000 respondents in this category said they were utilizing *Docker*. Demonstrating such usage statistics, Docker is selected as a virtualization solution for the mood diary application. With Docker, the application along with all necessary dependencies and configurations can be encapsulated within a so-called container, making it inherently portable and effortlessly deployable across diverse systems.

⁸ <https://docs.celeryq.dev/>

⁹ <https://github.com/Bogdanp/dramatiq>

¹⁰ <https://github.com/coleifer/huey>

¹¹ <https://github.com/pricingassistant/mrq>

¹² <https://github.com/rq/rq>

¹³ <https://www.docker.com/>

2.2.5 Quality and Speed of Development

To satisfy quality and speed of development, testing plays a vital role as it can ensure software quality [Jam+16]. Optimally, tests are a part of every stage of software development [Oul94]. The concept of Continuous Integration (CI) can help to achieve this. In CI, changes made to a software application are integrated frequently, meaning to both execute tests and to run automated build processes detecting erroneous changes quickly and reliably [Fow06]. This can also enhance development speed [Vas+15]. Closely related to CI is the concept of Continuous Delivery (CD), where a software application can be deployed automatically at any time [Fow13]. A combined CI/CD process would thus cover quality and speed of development and contribute to an easy deployment.

Several Version Control System (VCS) platforms offer integrated tools to create CI/CD processes. As version control along with code hosting is necessary for every software project, integrating a CI/CD process here precludes having to employ separate technologies for this matter. Two popular VCS platforms built around the VCS *Git*¹⁴ are *GitHub*¹⁵ and *GitLab*¹⁶. Both platforms allow to define CI/CD processes. While *GitHub* has been acquired by Microsoft [Mic18] and is only available as a software service, *GitLab* is completely open source and can be self-hosted, i.e., run on an own server. The University of Münster hosts such a *GitLab* instance for their employees and students. Consequently, the mood diary application will be developed with the help of *GitLab*, providing both VCS and CI/CD capabilities.

2.3 Resulting Technology Stack

Taken together, a technology stack satisfying all technical requirements has been selected. Developing the mood diary application as a web application featuring PWA functionality ensures it to be platform-independent with easy and extensive access. Choosing the full-stack web development framework *Django* written in Python backed by a *PostgreSQL* database to implement this web application adheres to constraining the project to a limited amount of popular, easy-to-learn technologies. At the same time, it ensures an application performance supporting up to 300 users as well as data sovereignty. *Celery* as a task queue enables asynchronous task processing and the use of *Docker* and *GitLab* pave the way for a portable, easy-to-deploy application with a CI/CD process ensuring quality and speed of development.

¹⁴ <https://git-scm.com/>

¹⁵ <https://github.com/>

¹⁶ <http://gitlab.com/>

3 Technologies and Concepts

In the following section, the technologies composing the selected technology stack alongside with their associated concepts will be described in greater detail to establish a mutual basis of understanding.

3.1 Progressive Web App

A web application can be enhanced with PWA functionality in order to look, feel and function more like a native application. The approach has been developed by Google and was first presented in a blog post in 2015 [Rus15]. It is not made up of a single technology but rather describes a set of technologies and concepts [BMG18]. Important to note here is the concept of progressiveness. The web application will be enhanced with PWA functionality if the browser it is viewed in supports the necessary features but will not break anything otherwise - in that case, it will just work like a regular web application. There are further concepts that add to the ones already embraced by regular web applications like responsiveness. For a full list of associated concepts, see [BMG18]. Here, the concepts in focus will be that of a PWA being installable, app-like, and re-engageable. These can enable the mood diary application to offer a richer user experience and to feature push notifications which were two aspects identified as potential drawbacks if developing the mood diary application as a regular web application as compared to a native one.

Illustrating these concepts, with a PWA, when visiting the application through the browser of a mobile device, it can be placed on the homescreen (via a banner popping up or through the browser's context menu). After doing so, the application is accessible through a homescreen icon just like native applications are. When tapping the icon, the application can respond instantaneously, whether an internet connection is available or not, by rendering the application shell (see below) and then show dynamically loaded content as soon as it is ready. As a result, the application looks, feels and functions like a native application to the point that it can receive push notifications. All this is made possible by employing three technologies: service workers, the application shell, and the web app manifest [MBG18].

Service Workers A service worker is essentially a JS script running in its own browser thread in the background, i.e., separated from the browser's main thread [Lee+18]. When initially visiting a web application with PWA functionality, the application registers the service worker with the browser. From then on, it runs

persistently in the background and can perform operations, even when the web application is closed. Acting as a kind of proxy, the service worker allows to intercept requests made to and from the web application. It thus enables caching of requests or handling of push notifications [BMG18]. To do this, the service worker behaves in an event-based manner dealing with events concerning the web application. These can be fetch events when the web application performs requests to a remote server or push events when a remote server sends push notifications (see also Fig. 2) to the web application [Mal+17]. With this principle, the service worker is the main source powering PWA enhancements. In order to use service workers, the web application must be served via Hypertext Transfer Protocol Secure (HTTPS). For a full specification of service workers, please refer to [W3C22b]. It is noteworthy that while previous academic work correctly observed limitations to PWA applicability as Apple iOS would not support it, at this point in time, Apple iOS and specifically the Safari browser now support all PWA functionality. The last missing part was the support for push notifications, which was added in March 2023 [Ang+23].

Application Shell Leveraging service workers, the application shell is a term for an architecture pattern where all assets making up the user interface of a web application are treated separately from the content that is displayed within that user interface [OG15]. All the HTML, CSS, JS, and other assets making up the user interface are loaded once. As they are separated from the dynamic content of the web application, these assets are static and can be cached by the service worker. Thus, the application shell is always locally available and can be rendered instantaneously when the application is started, e.g., displaying a splash screen or a navigation bar. The application shell is therefore necessary to receive a direct response when starting the application, hence being important for the user experience [BMG18].

Web App Manifest The web app manifest is a JavaScript Object Notation (JSON) file providing the ability to configure the look and feel of a PWA [She17]. Through this configuration file, it is possible to e.g., specify the icon and name that are going to appear on a mobile device’s homescreen when the application is placed there. The web app manifest can be used to make the application shell look like a native app by specifying that browser elements like the address bar are hidden when the application is started. Also, it allows to specify a specific splash screen, i.e., a screen that is shown to bridge waiting and loading time. The manifest file can be included into the web application by just linking it in the application’s HTML similar to how a CSS or JS file would be linked. For a full specification of the web app manifest, please refer to [W3C23].

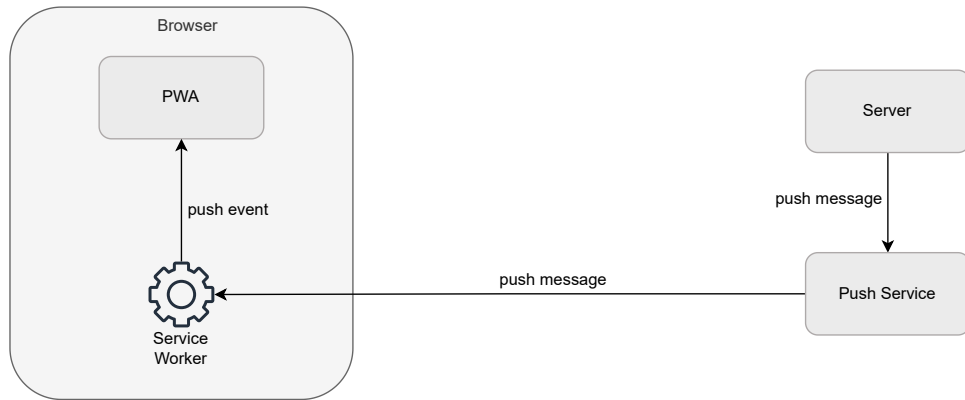


Figure 2 Handling Push Notifications via a Service Worker

In Summary, service workers, the application shell, and the web app manifest enable a PWA to be installable, app-like, and re-engageable through push notifications. These features should be incorporated into the mood diary application in order to ensure a high compliance with the mood diary intervention by providing a rich user experience and engaging users via push notifications.

3.2 Django

The web development framework *Django* follows the architectural pattern of Model-View-Template (MVT), which is a variant of the classic Model-View-Controller (MVC) design that has been described in academia [SCD06]. The model in *Django* is a Python class representing a database table. Each instance of the class represents one row of the corresponding database table. *Django's* own ORM handles the conversion between the model classes and the database tables. So-called migration files are used to propagate changes made to the models, like for example adding or deleting a field, to the database schema, thereby also building up a history of changes applied to the schema.

The view in *Django* is slightly different from that in the traditional MVC architecture. Instead of representing the output presentation, *Django* views are more akin to controllers in MVC, handling the logic of the application. They receive Hypertext Transfer Protocol (HTTP) requests, process them by for example performing appropriate database queries using the ORM, and return HTTP responses.

The template in *Django* is the presentation layer, responsible for defining how content should be displayed. A template is like a HTML file with some special syntax. *Django's* own template engine can then generate HTML dynamically by substituting variables in the template with actual values provided as context by the view.

Incoming requests are directed to the appropriate view based on their Uniform Resource Locators (URLs). To achieve that, each view is linked to an URL pattern. A router equipped with these patterns matches an incoming request against the patterns, directing the request to the view linked to the first match. For example, a view could be linked to the URL pattern *entity/<int:pk>*, possessing an integer parameter *pk*, matching a request with a URL like *some-host.de/entity/1*.

In order to further control the request/response processing, *Django* provides the concept of middlewares. A middleware can be used to hook into the processing to add extra functionality at certain points. For example, before routing the request to the appropriate view, a middleware could examine the request ensuring that the user sending the request is logged in. With such a middleware active, every request would be passed through the middleware before handing it over to the router.

An illustrating example of *Django's* request/response processing based on the MVT architecture together with routing and middlewares can be found in Fig. 3: A client performs a *GET* request to the *Django* application. The URL in this request is *django-app.de/persons/1*, indicating that the client wants to retrieve information about the person with the Primary Key (PK) of 1. The request first passes through a middleware, checking if the user associated with the request is logged in and raising an error if they are not, thereby disrupting the processing. Once the request passes the middleware, it reaches the router. The router uses the URL patterns defined in *urls.py* to determine which view should handle the request. In this case, the URL pattern *persons/<int:pk>* matches the request URL, so the request is directed to the *persons_view* function defined in *views.py*. The view function retrieves the person with a PK of 1 from the database, using *Django's* ORM to store the database record in an instance of the *Person* class defined in *model_person.py*. It then uses *Django's* template engine, providing it with the template from *person_template.html* and the *Person* class instance as context data to compose a response. In the response sent back to the client, the corresponding placeholder from the template is replaced with the actual name of the person retrieved from the database.

To summarize, *Django* is based on the architectural pattern of MVT and uses the concepts of a router and middleware. Like that, upon receiving a request, *Django* can pass it through middlewares, route it to the appropriate view and respond with HTML being dynamically composed based on the data provided by the view. This response can then be displayed in the browser.

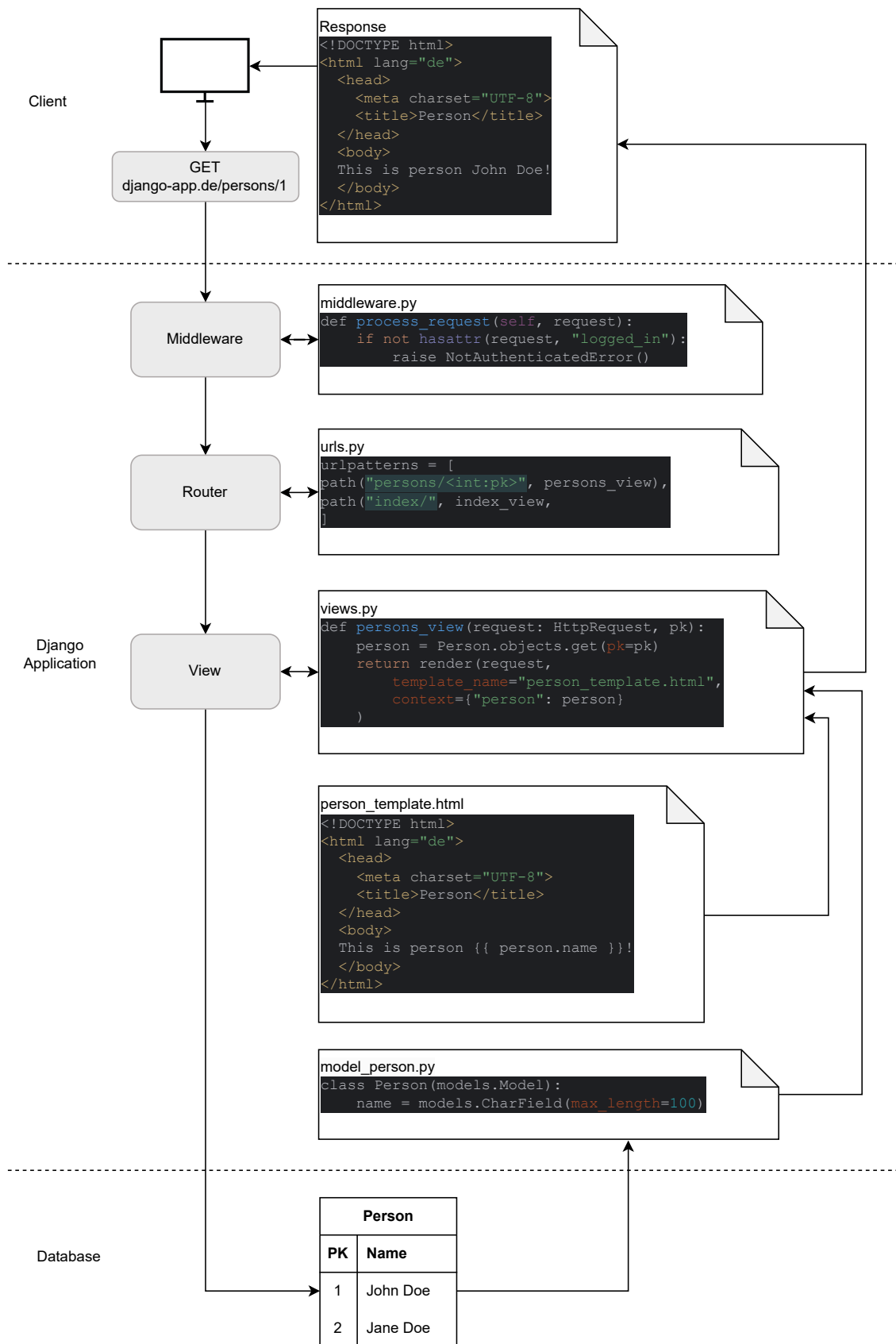


Figure 3 Request/Response Processing in a *Django* Application

3.3 Celery

The task queue *Celery* allows to define tasks by simply annotating regular Python functions with a decorator. A task then is the command to execute such a function with specific arguments. An example can be found in Listing 1: The function *increment_employment_duration* is annotated with the decorator *app.task* to tell *Celery* that this function should be executable as a task. Such a task can then be queued for execution in either an event-based manner by just calling e.g., *increment_employment_duration.delay(1)* to increment the employment duration of the employee with a PK of 1 by one week, or in a time-based manner by configuring the function as a periodic task (see lines 7-12 in Listing 1 for an example of how to queue a task for execution on every first day of the week).

In the context of *Celery*, queuing tasks for execution is performed by so-called producers, where a producer could be the mood diary application calling *task.delay(x,y)* when some event has occurred or the *Celery* scheduler triggering such a call at a specific point in time. A function executable as a task will serialize all necessary information about itself in a JSON message and send that message to a broker when called by a producer. A broker is a service allowing different applications or processes to communicate asynchronously with one another by providing the capability to send messages to as well as to receive messages from the broker. As an example, *Redis*¹⁷, a simple key-value storage, can serve as a broker. It has got one or more queues available to store messages in that were sent to it by producers. On the other side, so-called consumers, i.e., workers that run in separate processes, can be subscribed to a queue and will pick up messages stored in it. A consumer will then deserialize the picked up message into a function call with specific arguments again and actually execute it. For an illustration of this process, please refer to Fig. 4.

Taken together, *Celery* enables the asynchronous execution of tasks in an event-based as well as in a time-based manner by providing the possibility to define tasks and manage their execution via an architecture of producers, a broker and consumers.

¹⁷ <https://redis.io/>

```

1 from celery import Celery
2 from celery.schedules import crontab
3 from models import Employee
4
5 app = Celery("tasks", broker="redis://localhost")
6
7 @app.on_after_configure.connect
8 def setup_periodic_tasks(sender, **kwargs):
9     sender.add_periodic_task(
10         crontab(hour=0, minute=0, day_of_week=1),
11         increment_employment_duration.s(1),
12     )
13
14 @app.task
15 def increment_employment_duration(employee_pk):
16     employee = Employee.objects.get(pk=employee)
17     employee.weeks_employed += 1
18     employee.save()

```

Listing 1 Example *Celery* Task

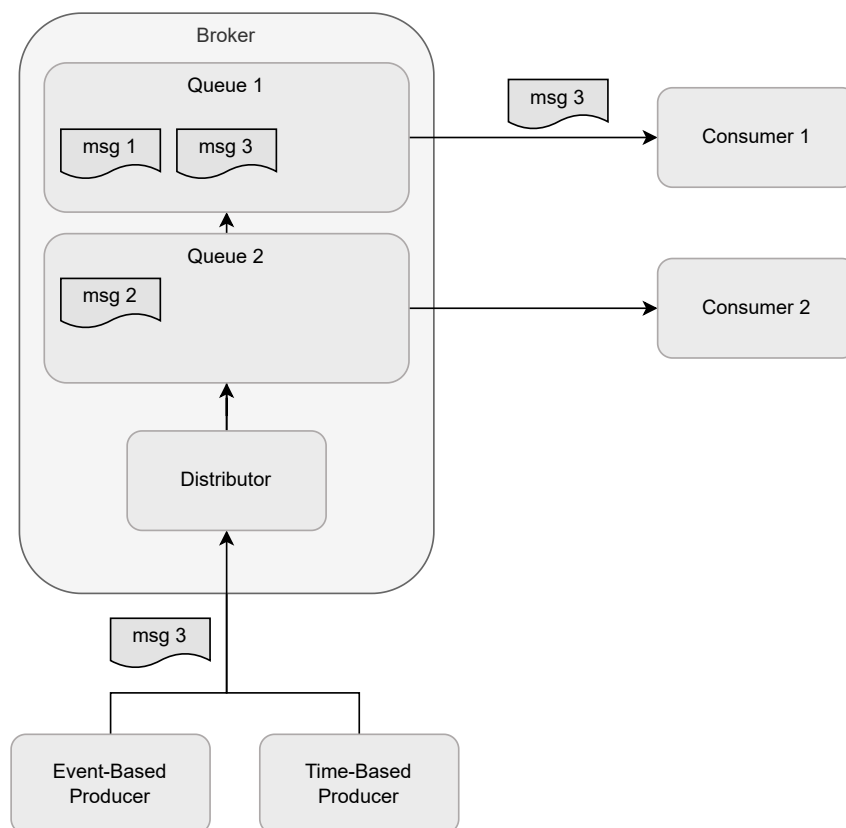


Figure 4 *Celery* Task Processing

3.4 Docker

The virtualization tool *Docker* allows to package an application along with all necessary dependencies and configurations into a so-called container that can then be run in many places. The important concepts here are those of images and containers [MS19]: An image can be thought of as serving as a blueprint for a container, just like a class in object-oriented programming serves as a blueprint for an instance or object of that class. An image can be defined in a *Dockerfile*, which is a collection of copying and installation commands comprising everything that is needed to run a specific application. From the *Dockerfile*, the actual image can be built by executing the steps defined in the file. The resulting image can then be run as a container.

An example for a *Dockerfile* can be found in Listing 2: Here, an already existing image is used as a basis (line 1), extending it as needed. Sources for existing images can be other images that are either self-built or offered by other persons or companies via an registry such as *Docker Hub*¹⁸ for hosting images. The base image already contains a Python distribution. Then, the *Django* application directory is copied into the image (lines 2-4) and all Python dependencies listed in the *requirements.txt* file are installed (line 5). Lastly, the command to be executed when running the image as a container is defined, which in this case is the starting of the *Django* application (line 6). These steps can be executed by calling *docker build -t my_image path_to_build_context* from the command line, where the image is tagged with the name *my_image* for easier reference and a build context, i.e., the files located at the provided path, is provided. Per default, the *Dockerfile* is expected to be located in the root directory of the build context. Afterwards, the image is ready and can be run as a container with the command *docker run my_image*. This command could also be executed on another machine than the image was built on - the image is fully portable.

```
1 FROM python:3.11-slim-bullseye
2 RUN mkdir -p /app
3 WORKDIR /app
4 COPY ./my_django_app/. ./
5 RUN pip install -r requirements.txt
6 CMD ["python manage.py runserver"]
```

Listing 2 Example *Dockerfile*

Oftentimes, the architecture of an application contains more than one *Docker* container, and the processes running inside the different containers need to communicate

¹⁸ <https://hub.docker.com/>

with each other. In that case, it is useful to employ *Docker Compose*¹⁹ to manage multiple containers. With *Docker Compose*, all containers that are needed can be specified in a special *docker-compose.yml* file that can then be used to e.g., start or stop all of them at once. More importantly, all containers specified in one such file are able to communicate with each other e.g., by accessing ports they are exposing. Listing 3 shows an example of a *docker-compose* file: Under the keyword *services*, two containers are defined that can be managed with this file - one for the *Django* application mentioned above (lines 4-5) and one for a *PostgreSQL* database (lines 6-12). Note that for the database, a user and password are set via environment variables. Also, the database container is given selective access to the host system it is running on, in this case to the host's file system. More specifically, in line 12, the directory *postgres_data* from the host machine is mounted into the container at a path where the database stores its data per default. Like that, the data will be persisted on the host system when the database container is stopped.

```

1 version: '3.8'
2
3 services:
4   my_django_app:
5     image: my_image
6   database:
7     image: postgres:15.3
8     environment:
9       - POSTGRES_USER=${POSTGRES_USER}
10      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
11   volumes:
12     - ../.docker/postgres_data:/var/lib/postgresql/data

```

Listing 3 Example *Docker Compose* File

All in all, with the the concepts of images and containers, *Docker* is able to make an application along with all necessary dependencies and configurations portable across host systems. In the event of dealing with multiple containers at once, *Docker Compose* can help to manage the containers, enabling the processes running inside them to communicate with each other.

3.5 GitLab Continuous Integration/Continuous Delivery

The main concept needed to define a CI/CD process in *GitLab* is a so-called pipeline. A pipeline consists of one or multiple stages and one or multiple jobs within each stage. Jobs within each stage are executed in parallel by default, while stages are executed sequentially. A pipeline is defined in a *.gitlab-ci.yml* file that is located at

¹⁹ <https://docs.docker.com/compose/>

the root of the project. A typical pipeline consists of the three stages build, test, and deploy. In the build stage, a Docker image could be created containing the application along with all dependencies and configurations. In the test stage, the image from the build stage could be started as a container to run all tests defined in the project. Finally, in the deploy stage, the image from the build stage could be started as a container in a production environment.

Each job has got an associated script containing its execution steps. To execute the jobs defined in a pipeline, or their scripts, respectively, a *GitLab* runner is used. A *GitLab* runner is an application that is capable of processing these jobs and sending the results back to *GitLab* to be displayed. *GitLab* runners can be installed on a various number of OSs and with different execution environments, for example one that comes preconfigured with *Docker*. A *GitLab* project can have multiple *GitLab* runners registered that can be located on other machines than the project itself. Every time the project's code base changes, e.g., through a commit, *GitLab* runners automatically pick up the *.gitlab-ci.yml* file and start running the jobs defined in the pipeline. During job execution, the *GitLab* runner can access the project's files in the version of the code base that triggered the pipeline.

As an example for a pipeline definition, please refer to Listing 4: The pipeline consists of the three stages *build*, *test*, and *deploy* (lines 1-4). In the *build* stage, there is one job using the *Dockerfile* from Listing 2 to build a *Docker* image (line 11). The resulting image is then pushed into the *GitLab* Registry, which is a registry for hosting *Docker* images integrated into *GitLab* (line 12). In the *test* stage, the *execute_tests* job uses the image from the previous stage (line 16) and runs all tests defined in the project (line 18). Lastly, in the *deploy* stage, one job connects to a production server via Secure Shell (SSH) and copies over the current *docker-compose.yml* file (line 24). Then, it pulls the updated image from the *build* stage from the registry and runs it as a container, thereby updating an already running container, if one exists (lines 28-29). Consequently, this example pipeline covers both CI via building and testing the project's application and CD via automatically deploying it to a production environment.

Please note that for brevity, only one image is used in Listing 4 for both the testing and deployment stages and the deploy stage runs in every pipeline. In a real-life example, the image used in production would differ from that used for testing in most cases, as the production image usually requires different configurations, like no installation of testing tools. Further, the pipeline would presumably be configured in a way that not every change to the code base would trigger a deployment. Also

note that the login to the *GitLab* registry (lines 10 and 27) is performed with some predefined CI variables²⁰.

Recapitulating, *GitLab* implements a pipeline concept defining a series of jobs amounting to a CI/CD process that can be configured via a *.gitlab-ci.yml* file. Each time the project's code base changes, one or multiple *GitLab* runners execute all jobs defined in the pipeline.

```

1 stages:
2   - build
3   - test
4   - deploy
5
6 build_image:
7   stage: build
8   image: some_base_image
9   script:
10    - docker login -u ci-user -p $CI_JOB_TOKEN $CI_REGISTRY
11    - docker build -t my_image .
12    - docker push my_image
13
14 execute_tests:
15   stage: test
16   image: my_image
17   script:
18    - pytest app
19
20 deploy_to_prod:
21   stage: deploy
22   image: image_with_ssh
23   script:
24    - scp docker-compose.yml root@my-domain.de:/home/deployment
25    - ssh -l $DEPLOY_SSH_USER $DEPLOY_SSH_HOST "
26      cd /home/deployment &&
27      docker login -u ci-user -p $CI_JOB_TOKEN $CI_REGISTRY &&
28      docker-compose pull &&
29      docker-compose up -d "
```

Listing 4 Example *GitLab* Pipeline

²⁰ See https://docs.gitlab.com/ee/ci/variables/predefined_variables.html for a full list.

4 The Mood Diary Intervention

Switching from a technical to a content perspective, the following section digs deeper into the principles of the mood diary intervention. At the intervention's core, clients log their (non-)activities together with the mood they felt when executing them over a period of at least two weeks. The idea behind this process is to recognize associations between mood and behavior [Hau21]. Logging activities and moods should help clients identify factors influencing their state of mind [MD11]. Specifically, mood diary entries can reveal triggers deteriorating mood, thereby improving the client's understanding of their own behavior and feelings. This, in turn, can raise clients' self-awareness and lead to better coping with challenges. What is more, keeping a mood diary is a means to make therapeutic and counseling sessions more efficient, as it allows for data collection between sessions [MD11].

This procedure is widely used in psychological counseling and psychotherapy with regard to depressive symptoms [Hor+22; GH09]. Information gathered in a mood diary has been shown to predict depression [Hor+22]. Further, mood diary interventions help identifying triggering and sustaining mechanisms of negative mood. This is crucial for curing depression since subdued mood is one of the key depressive symptoms. Here, it is particularly helpful to identify every activity positively influencing mood, even if only by a small magnitude.

Usually, the mood diary intervention is completed in an analog, i.e., paper-pencil form. This is also the case at the PSB. Here, a paper form is handed out to clients (see Fig. 5 for an example). On this form, clients can enter their mood on an interval scale together with an activity in hourly time slots. Also, for each day, times of waking up and getting up should be recorded. Additionally, there is some extra space to log general notes or particular aspects per day.

Health professionals, like the counselors at the PSB, put the ideas behind the mood diary intervention to use by specifically employing a triad of steps based on the data collected with the paper form. In the first step, the collected data is used to identify patterns and correlations in the moods and behaviors of clients that could be relevant to the counseling. This could for example be any patterns or correlations playing a role in maintaining or reinforcing problem behavior but also ones hinting at resilience or resourcefulness that could help in developing coping strategies. In the second step, counselors teach their clients psychoeducative contents related to the identified patterns or correlations. This is done in order to help establish a self-awareness in the clients and to help them getting a read on their own moods, behaviors, and associations between them. In the third step, approaches for change

A simple example for this triadic procedure of (1) pattern identification, (2) psychoeducation and (3) offering approaches for change or maintenance could be the following: A client's mood diary entries indicate that they regularly spend considerable time intervals browsing social media sites on their phone. The mood associated with this activity is often negative (1). The counselor explains to the client that prolonged time intervals of social media consumption can negatively affect one's mood [MSS17] (2). To counteract this habit, the counselor proposes to limit social media consumption to an overall amount of time per day as well as to an amount of time for each specific event of consumption (3).

Naturally, this procedure works best based on complete and detailed data. However, homework-like interventions in paper-pencil form like the mood diary one often lack compliance [GLN06; GS02; HF04]. For the mood diary intervention in particular, this seems to be especially the case for young persons [MD11]. With homework completion being related to treatment effectiveness [PBP88; BS00; HF04], a lack of compliance endangers this very same effectiveness. In this situation, a digital solution in form of a mood diary application available for smartphones seems promising. Not only does mobile technology improve treatment outcome in general [Lin+15], but there is evidence suggesting that it can specifically increase compliance [TK17; Mat+08; MD11].

Along with an increased compliance and other advantages like a more convenient recording of data via a smartphone than via a paper form that would be inherent features of a mood diary application, more room for improvements would open up. First, with a mood diary application, counselors could gain access to their clients' mood diary entries already before the next counseling session. This would allow them not only to monitor the intervention but also more time to analyze the data and prepare fitting psychoeducative contents and approaches for change or maintenance. Second, with automatic analyses of the mood diary entries, clients could receive timely feedback about their moods and behaviors, with such a temporal contiguity being an important factor in learning experiences [Gin06].

Summarizing, the mood diary intervention is a valuable tool in the context of depressive symptoms, enabling a triad of identifying patterns related to moods and behaviors, providing psychoeducation about them, and offering approaches for change or maintenance. Reproducing this intervention in form of a mood diary application available for smartphones could increase compliance with the intervention. Beyond that, counselors could profit from the opportunity to view mood diary entries prior to the next counseling session and clients from timely automatic feedback.

5 Related Work

In the broader domain of tracking mood and diary-like applications that in some way target mental health, a huge number of applications is available, for example from Android’s or Apple’s app stores. Most of the applications offered here are designed for personal use and not as part of a counseling or therapeutic process. What is more, many of these applications seem to lack scientific foundations. Instead, there are often disadvantages like advertisements or paid features. In accordance with that, a study reviewed 1156 German apps targeting depression, concluding that none of these applications were scientifically investigated and that practically all of them were qualitatively flawed [Ter+18].

Looking at the area of Digitale Gesundheitsanwendungen (DiGA), five officially approved applications indicated for use with depression could be identified²¹. Being approved as DiGA applications, all five of them should be effective tools to battle depression. However, all of them have a wider focus, either aiming to bridge waiting times before beginning a psychotherapy, keeping clients stable subsequent to a psychotherapy, or even substituting one. To this end, these applications are built up more like courses, covering a broad range of modules and contents, not concentrating on a specific intervention.

The scientific community has dealt and actively is dealing with the matter of digital mood diary applications. There are studies gathering mood data in digital ways (e.g., [Hor+22]) or even developing and evaluating specific tools to this end [Bau+05; Bau+06; Mat+08; MD11; Cao+20]. These tools are however mostly employed in research contexts, without any indication of them being put to practical long-term use in counseling or therapeutic settings. Only two studies could be found where publicly available applications were used. The application from the first study [BR18] only tracks mood, not activities and is only available in a paid version. The application from the second study [DEM21] seems to be unavailable for current Android versions.

Taken together, many applications in the broader domain of this project are available in big app stores, as DiGA applications, and through scientific work. Nonetheless, no readily available application specifically realizing the mood diary intervention as part of a counseling or therapeutic process could be identified. Even less, an application allowing counselors to access the data collected by clients and providing timely feedback to them could be made out. Hence, this project will extend previous work by providing an application specifically containing these features.

²¹ <https://diga.bfarm.de/de/verzeichnis>

6 Conceptualization of the Mood Diary Application

After establishing a mutual basis of understanding concerning the selected technology stack and the psychological and practical principles of the mood diary intervention, the aim of this section is to conceptualize the mood diary application based on this technology stack. As established before, the mood diary application should (1) make the mood diary intervention available on a smartphone to ensure client compliance, (2) make mood diary entries accessible to counselors and (3) provide timely feedback about them to clients through automatic analyses. To reflect these goals, this section will cover the following aspects: First, an overall architecture for the application will be developed. Second, a digital version of the mood diary intervention will be conceptualized. The third part will then deal with making the mood diary entries accessible by counselors, and the fourth part will cover the automatic analyses of the entries.

6.1 Architecture

Fig. 6 depicts the architecture of the mood diary application. It is divided into three logical areas - development, staging, and production. In the development area, the application is developed on a local machine, hosting the code base on a *GitLab* server. The *GitLab* server is connected to another server that has a *GitLab* runner installed. The runner picks up CI jobs like executing tests and returns job logs back to the *GitLab* server. Additionally, the runner executes CD jobs, i.e., automatically deploying the mood diary application.

There are two automatic deployments. The first targets a staging server, where each new version of the mood diary application can be tested before deploying it to production. The staging server mirrors all the services from the production server, with the difference that the data used to fill the application with life is randomly seeded here. Apart from that, the staging version of the mood diary application can be accessed like a regular production version by test users.

The second deployment targets a production server. The mood diary application is deployed here divided into five different services, where each service consists of one *Docker* container. First, there is a *Django* application enhanced with PWA capabilities as the core service of the mood diary application. Here, central business logic is located and communication with users from both mobile and desktop devices is handled via the request/response process described in section 3. Also, the *Django* application can send push notifications and emails to external push and email ser-

vices taking care of delivering them to the users. The *Django* application relies on a *PostgreSQL* database as the second service, communicating with it via the *Django* ORM. Tasks can be sent to a *Redis* broker as the third service, both based on events from the *Django* application and based on time from a *Celery* scheduler as the fourth service. The broker manages the tasks in queues and relays them to a *Celery* consumer as the fifth service for execution. The consumer can interact with the database to retrieve data for task execution and to store results.

In summary, the mood diary application consists of five services - a *Django* application, a *PostgreSQL* database, a *Redis* broker, and a *Celery* consumer and scheduler. It can be automatically deployed to both a staging and a production server by a *GitLab* runner triggered in turn by a *GitLab* server.

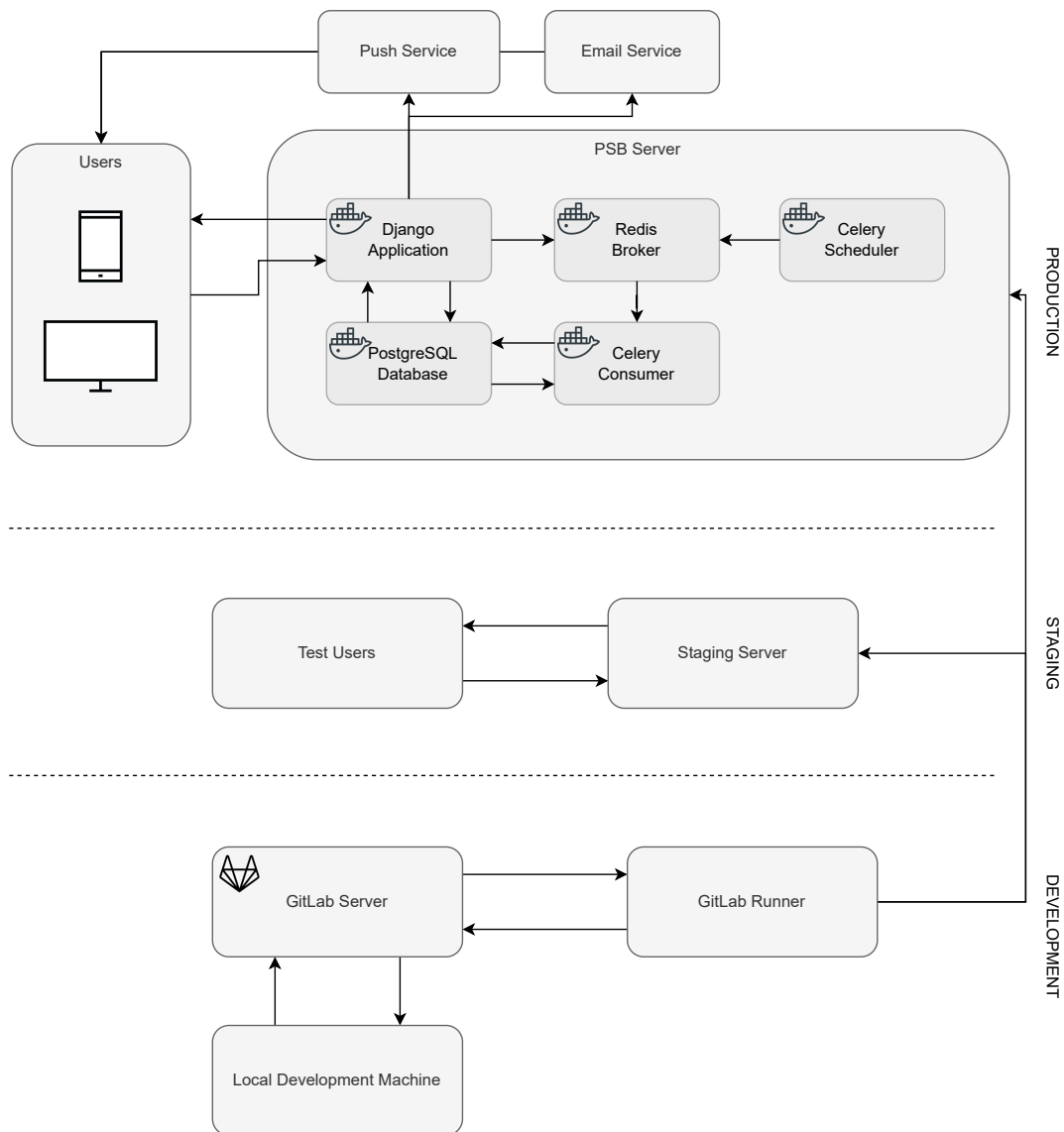


Figure 6 Mood Diary Application Architecture

6.2 Digital Mood Diary Intervention

At the heart of the mood diary intervention is the logging of mood diary entries. Essentially, a mood diary entry assesses the following information: date and time, activity, activity duration, and mood. In the digital version, an activity can be selected from a set of predetermined activities like "meeting friends" or "sleeping", where each activity belongs to a category like "social" (an excerpt of activities and categories, that were developed based on the experience of counselors at the PSB, can be found in Appendix A). This is done to guarantee a standardized data format that can be used as a basis for analyses. It is most useful to assess mood interval-scaled as this kind of numerical scaling helps measuring mood changes precisely. For the digital mood diary intervention, in consultation with the PSB, a seven-point interval scale ranging from -3 to 3 is employed (see also Tab. 2). Together with this required data, the digital mood diary intervention offers an optional input field to document any thoughts or circumstances deemed important by the client.

Value	Label
-3	Very bad
-2	Bad
-1	Rather bad
0	Neutral
1	Rather good
2	Good
3	Very good

Table 2 Mood Assessment Scale

A simple design for an input form capturing the described data can be found in Fig. 7. Similar to the input form, Fig. 8 shows a detail view of a mood diary entry, also containing buttons to edit the viewed entry (directing back to the input form) and to delete it. In addition, as also depicted in Fig. 8, a list of mood diary entries can be displayed together with a button to create a new entry (also directing to the input form). As a result, basic Create, Read, Update, Delete (CRUD) functionality surrounding mood diary entries is covered.

As one goal of the digital mood diary intervention is to ensure high compliance rates, a dashboard view is added. Here, there is a diagram visualizing the average mood per day for the last week as well as a list of the most recent top mood diary entries with respect to the mood scale. For the design of the dashboard, please refer to Fig. 7. This dashboard is aimed to offer an appealing starting point in the digital

The figure consists of two side-by-side panels representing different views of a mood diary application.

Left Panel: Mood Diary Entry Input Form

- Start date:** A text input field.
- End date:** A text input field.
- Start time:** A text input field.
- End time:** A text input field.
- Activity:** A text input field.
- Mood:** A text input field.
- Additional info:** A larger text input field.

Right Panel: Dashboard View

- Mood Overview:** A line graph showing mood fluctuations over time. The graph has a vertical y-axis and a horizontal x-axis. The line starts at the origin, rises to a peak, dips slightly, and then rises again.
- Mood Highlights:** A section containing a text box with the following text:
 - Very Good (3)
 - Eating
 - October 01, 2023
- ...** An ellipsis indicating more highlights.

Figure 7 Mood Diary Entry Input Form and Dashboard View

mood diary intervention providing an overview of the logged data, so that clients are motivated to regularly open and use the application.

Thinking in layers, the digital mood diary intervention consists of three of them. As the first layer, there is the dashboard as the start page. From there, users can reach the list view of mood diary entries as the second layer. The third layer is then composed of the input form to create or edit an entry or the detail view of an entry, respectively. Each layer is enclosed in a uniform frame, containing for example a navigation bar (see Fig. 9).

Note that for brevity, the conceptualization of features independent from the content of the mood diary intervention, such as login or logout processes, user settings like password management, language translations and similar aspects is omitted here. Of course, the application still possesses these features.

To conclude, the digital mood diary intervention offers a dashboard view as a start page. From there, mood diary entries organized in a list view, a detail view and an input form, together covering basic CRUD functionality, can be reached. All views are embedded into a uniform frame.

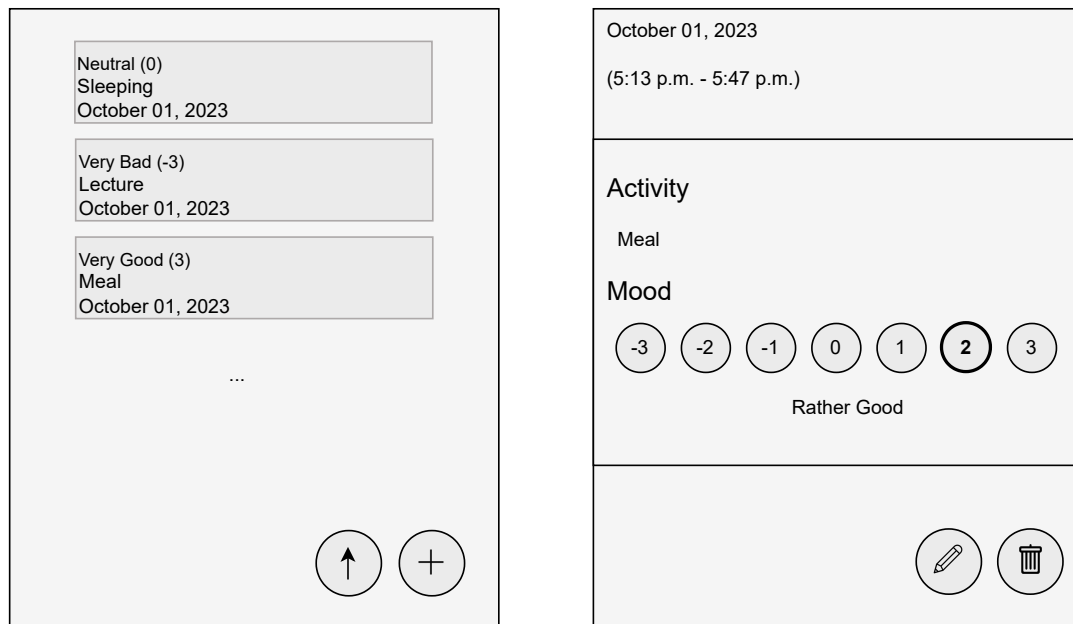


Figure 8 Mood Diary Entry List and Detail Views

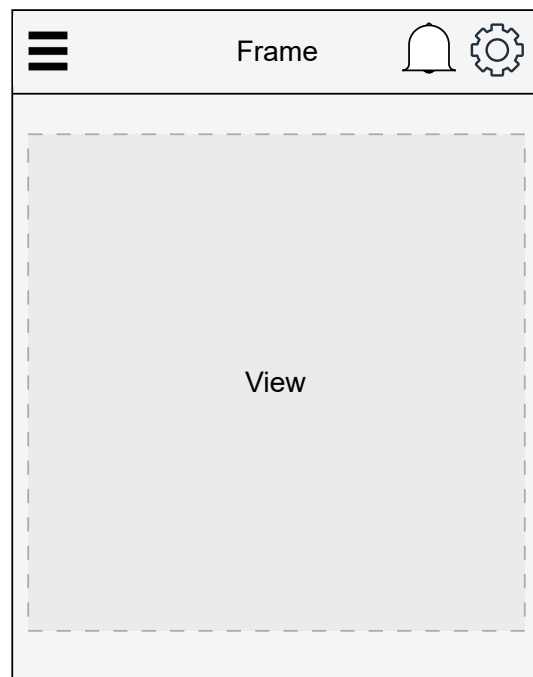
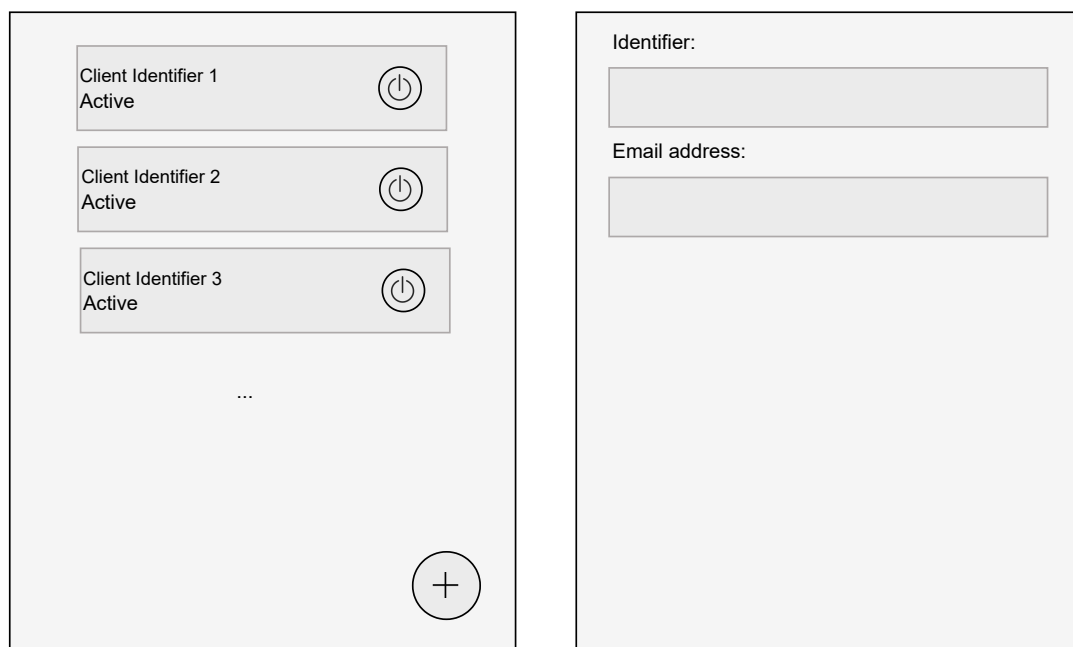


Figure 9 Mood Diary Application Frame for Views and Forms

6.3 Counselor Access

In addition to client users, the mood diary application provides access for counselor users. They can access the mood diary entry list and detail views of clients assigned to them. For this, the views from Fig. 8 are reused, but without the buttons to add, create or delete entries. To give clients full control over their mood diary entries, there is a button to release the entries to their counselor (the bottom with the arrow in the left part of Fig. 8). In that way, entries can only be viewed by the counselor if the client explicitly shared them. Additionally, counselors can access a list view of their clients, as depicted in Fig. 10. This view also contains buttons to set a client's status to "inactive" to filter them out of the list view, and to create a new client.



The figure consists of two side-by-side panels. The left panel, titled 'Client List View', displays a list of three clients. Each client entry is a light gray box containing the text 'Client Identifier 1', 'Client Identifier 2', and 'Client Identifier 3' (all followed by 'Active' on the next line), and a circular power button icon on the right. Below the list is an ellipsis '...'. At the bottom right of the panel is a circular button with a plus sign '+'. The right panel, titled 'Input Form', contains two input fields. The first is labeled 'Identifier:' and the second is labeled 'Email address:'. Both labels are in a small font above their respective text input boxes.

Figure 10 Client List View and Input Form

Counselors accounts are created by a designated system administrator. They can in turn create client accounts, allowing them to fully control client access to the mood diary application. In the context of developing an application for an intervention in psychological counseling, it makes sense that counselors can decide whether and when a client will gain access to the application as opposed to implementing an open registration process. Naturally, a client created by a counselor is assigned to that counselor and can only be viewed by them. In the input form for client creation, counselors provide an email address and an identifier - like that, there is no need for the application to store clients' real names. When submitting the input form, a client user with a randomly set one-time password is created in the database and an email containing this one-time password and a link to the application is sent to

the provided email address. When logging in via the link in the email, the client is directed to a standard password change view prompting them to set their own password. After that, they can use the application in the regular way. The input form for client creation and the process it triggers are depicted in Fig. 10 and Fig. 11, respectively.

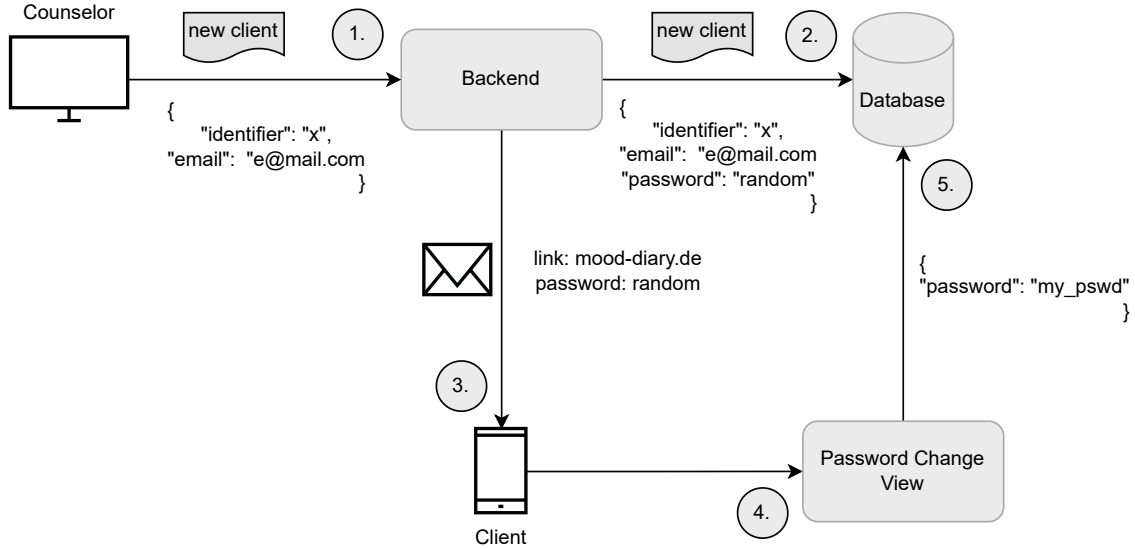


Figure 11 Client Creation Process

In the layer representation, for the counselor, the first layer is the list view with the selection of clients as a start page. The second layer consists of the list view of released mood diary entries for the selected client. An entry's detail view or the input forms to create a client make up the third layer, respectively. Again, all views are enclosed into the uniform frame shown in Fig. 9.

To summarize, counselors can provide clients with access to the mood diary application. They can view a list of their clients and access their clients' mood diary entries released to them in a list and in a detail view. All views are again embedded into a uniform frame.

6.4 Automatic Analyses

The mood diary application analyzes mood diary entries logged by clients to provide timely feedback about them. This process is based on the triad of pattern identification, psychoeducation and offering approaches for change or maintenance as described in section 4. Conveying this triad to an automatic analysis system, the system has to be capable of detecting patterns in recorded data and to deliver

content comprising psychoeducation and approaches for change or maintenance to the user. In effect, the system aims to draw conclusions and act accordingly to them in a way a counselor would do, with the advantage of making feedback available in a timely manner, as the system can analyze the data at any point in time.

Consequently, the automatic analysis system has to implement the key features of pattern detection and content delivery. As the system, generally speaking, tries to mimic the expertise of counselors, an expert system comes to mind. In expert systems, knowledge is encoded in rules [Hay85; GA11; Lig06]. These rules usually consist of preconditions on the left hand side and conclusions on the right hand side [Lig06]. Transferred to this use case, the preconditions on the left hand side would be a pattern of moods or activities and the conclusions on the right hand side would be the content to be delivered when the pattern is detected. Therefore, an expert system based on rules seems like a fitting mechanism for pattern detection and content delivery in this domain and is consequently pursued by the mood diary application.

Bringing both parts of the mechanism together, rules consisting of preconditions describing patterns and conclusions as messages are stored in the database. Each conclusion message comprises the psychoeducative content and approaches for change or maintenance related to the pattern described by the rule's preconditions. When these preconditions are fulfilled, meaning that a pattern has been detected, a notification is created from the corresponding conclusion message. The notification is then again stored in the database, so that it can be displayed to the client at the next opportunity.

The evaluation of rule preconditions depends on their type. There are event- and time-based rule preconditions, depending on the pattern that the preconditions describe. Rules with event-based preconditions are evaluated when new mood diary entries have been logged. Rules with time-based preconditions are evaluated periodically. Specifically, that means that after a mood diary entry has been logged, the *Django* application acts as an event-based producer (cf. Fig. 4), queuing a task with the *Redis* broker to be processed by the *Celery* worker. The task instructs to check whether preconditions for any event-based rule are fulfilled and to create a notification from the corresponding conclusion message if they are. In the same manner, the *Celery* scheduler queues tasks acting as a time-based producer.

Notifications as a means of content delivery are integrated into the mood diary application via the bell icon visible in Fig. 9. Here, new notifications can be indicated by displaying a notification badge next to the icon. When clicking on the bell, clients

reach a notification list view (cf. Fig. 12). In a detail view, the psychoeducative content and the approaches for change or maintenance making up each specific notification are displayed (see also Fig. 12). What is more, the notifications inside the mood diary application are enhanced with push notifications, meaning that PWA functionality, specifically the service worker, is leveraged to retrieve push notifications sent from the *Django* application to clients. Like that, each time a rule is fulfilled, i.e., a pattern is detected, clients can be notified that new feedback has been delivered (and not only when they open the application the next time), assuring temporal contiguity.

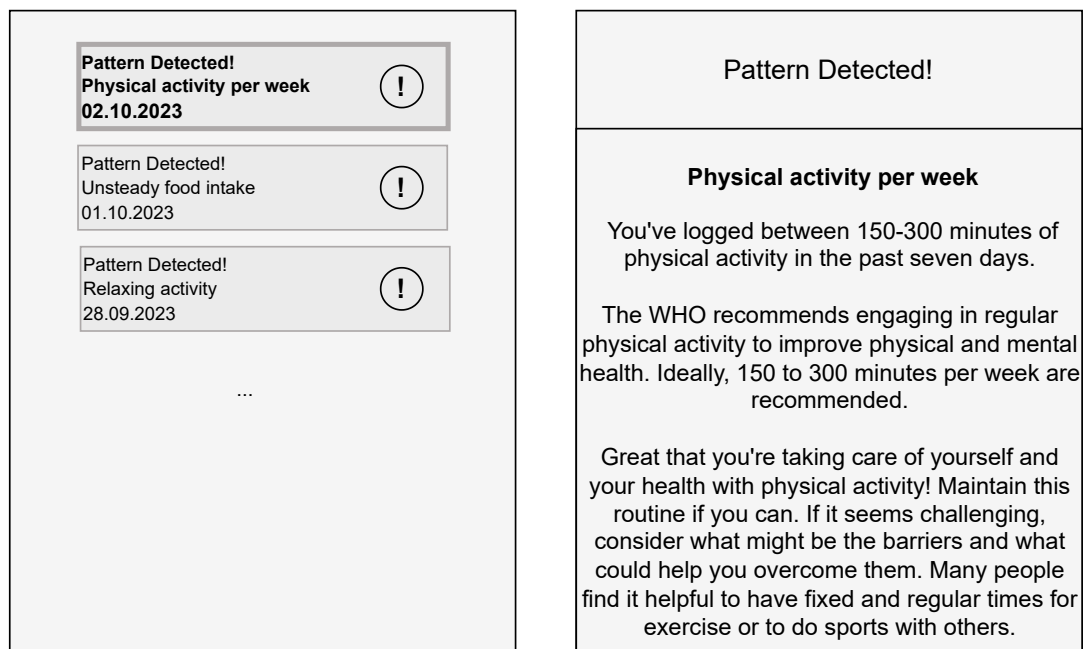


Figure 12 Notification List and Detail Views

Specific rules were developed in consultation with the PSB based on both a scientific foundation and their experience with the mood diary intervention. With these rules, mood diary entries are analyzed either with regard to a fixed threshold criterion or with regard to an intraindividual change criterion. Evaluation of the rules can be triggered event- or time-based. An overview of the developed preconditions per rule, categorized by criterion and evaluation type, can be found in Tab. 3. Note that the preconditions for a rule can target mood, activity, or both, and that where applicable, references for selected thresholds are provided within the table. For an example of a conclusion message for the *Physical activity per week* rule, or, more specifically, the notification created from that, please refer to Fig. 12.

The resulting rule set is of general nature, meaning that it may be the case that not all rules make sense with regard to the personal context of a client. Specific rules

could be problematic for single clients because of the nature of their psychological strain or they could be irrelevant or hindering with regard to their personal goals. To address this potential issue, before starting the mood diary intervention, clients can determine which rules should be turned on or off together with their counselors. In the course of the intervention, they can update this configuration at any time if they experience that specific feedback is not helpful to them. The rule configuration view (see Fig. 13) can be reached by clicking the cogwheel displayed in the upper right corner of the mood diary application (cf. Fig. 9).

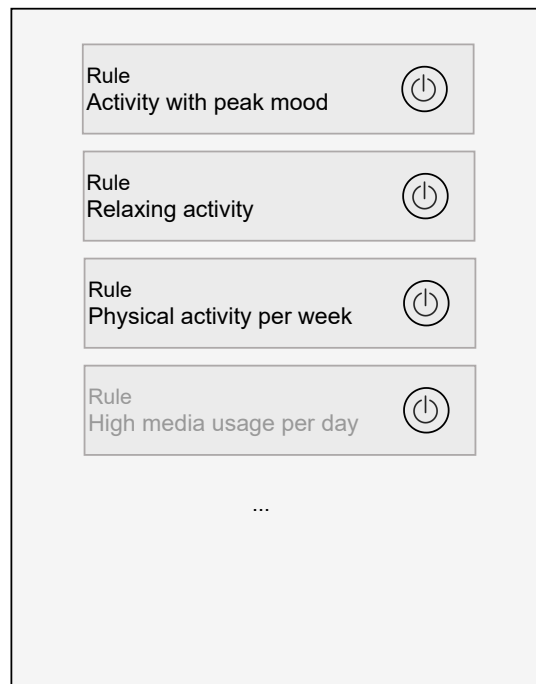


Figure 13 Rule Configuration View

Recapitulating, the mood diary application analyzes mood diary entries via an expert system of rules. Each rule consists of preconditions describing a pattern and a conclusion in form of a message that is used to create a notification for clients when a rule's preconditions are fulfilled, following the triad described in section 4. The application contains twelve such rules with threshold or change criteria that are evaluated in an event- or time-based manner using the asynchronous task processing implemented via *Celery*. All rules can be configured to be active or inactive by clients.

Rule Title	Preconditions	Criterion	Evaluation
Activity with peak mood	Activity with mood value equal to 3	Threshold	Event-based
Relaxing activity	Activity from the category "relaxing"	Threshold	Event-based
Physical activity per week	Sum of 150 min. of physical activity	Threshold [WHO22]	Event-based (max. 1 trigger per week)
High media usage per day	Sum of 6 hours of media usage	Threshold [MSS17]	Event-based (max. 1 trigger per day)
Low media usage per day	Sum of less than 30 min. of media usage	Threshold [Hun+18]	Time-based (1 trigger per day)
14 day mood average	Average mood value of less than 0 for at least 9 out of 14 days	Threshold [Här+10]	Time-based (1 trigger per 14 days)
14 day mood maximum	Maximum mood value out of 14 days is less than 1	Threshold [Här+10]	Time-based (1 trigger per 14 days)
Unsteady food intake	Less than 3 meals per day for the last 3 days	Threshold [Mac+22] [Mor+22]	Time-based (1 trigger per day)
Positive mood change between activities	Two consecutive activities with a mood difference of 3	Change	Event-based
Negative mood change between activities	Two consecutive activities with a mood difference of -3	Change	Event-based
Daily average mood improving	Higher average mood than the day before	Change	Time-based (1 trigger per day)
Physical activity per week increasing	Greater sum of physical activity than the week before (if below 300 min.)	Change [Con10]	Time-based (1 trigger per week)

Table 3 Rule Preconditions Categorized by Criterion and Evaluation Type

7 Implementation of the Mood Diary Application

Following conceptualization, this section will describe the development and implementation of the mood diary application. To this end, first, an overall data model will be presented, followed by a description of the project’s code base structure. Next, an overview of the Mood Diary application’s functionality will be provided, subsequently covering selected aspects in greater detail. The presentation of these aspects will be structured into subsections pertaining to each of the three main goals of the project as well as to the topics of PWA functionality and the CI/CD process.

7.1 Data Model

The full relational data model implemented for the mood diary application is presented in Fig. 14. Apart from a general *user* entity type, there are three logical areas marked with rectangles of different color representing the three main goals of this project. As for providing counselor access, there is not only the above-mentioned *user* type, but a separate *client* type (light gray mark). This allows to comfortably assign users with the role *client* to users with the role *counselor* and to keep client-specific relations to other entity types separate from the general *user* type. Entity types pertaining to making the mood diary intervention digitally available can be found within the dark gray mark. These basically represent diary entries with associated activities and moods. Lastly, there are entity types for rules and notifications implemented with regard to delivering timely feedback to clients through automatic analyses (black mark). These include e.g., an entity type for push subscriptions containing all necessary information to send a push notification to a user device.

Please note that it is not technically necessary to declare a separate model for the mood diary. It would also have been possible to directly link mood diary entries to their clients. However, this design is chosen to be able to define methods on the *Django* model corresponding to the mood diary table that can naturally use all mood diary entries for a client without needing filter operations. An example for this is a method calculating average mood values for the client dashboard. Also note that translation fields are omitted from this depiction. The actual data model contains extra fields like *title_de* for displaying rule titles in German.

Taken together, the data model can be divided into three areas all of which originate from a general user entity type. These areas each cover one of the project’s three main goals, providing entity types for counselor access, a digital mood diary intervention and timely feedback through automatic analyses.

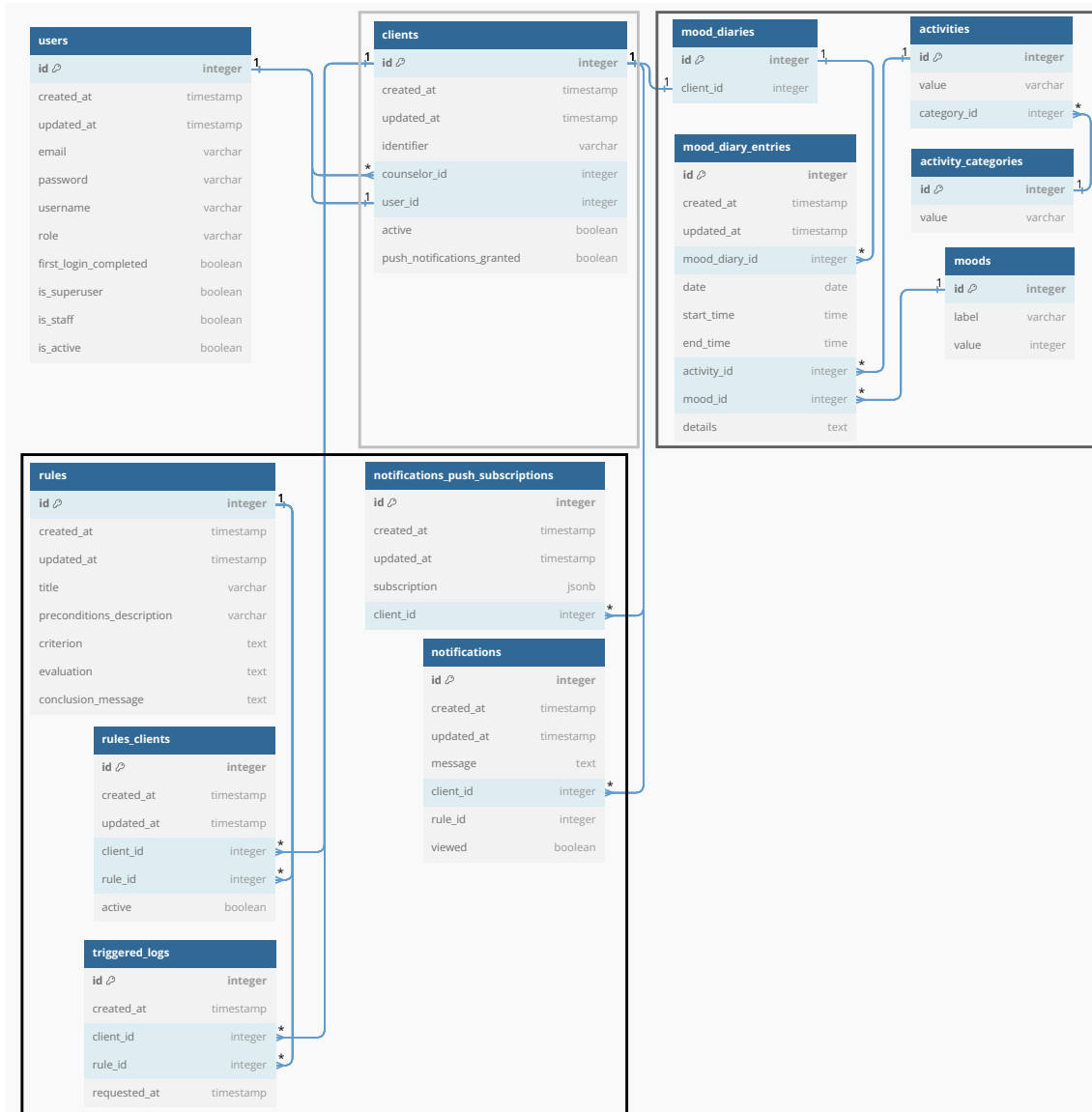


Figure 14 Data Model for the Mood Diary Application

7.2 Project Structure

At its heart, the project's code base contains a *config*, a *mood_diary*, and a *requirements* directory (for an illustration refer to Listing 5). The *requirements* directory is made up of the project's Python dependencies, i.e., specifying libraries like *Django*. These dependencies are structured into three different files, one containing all dependencies necessary to run the project and then two files extending the first one with regard to different use cases. The *local.txt* file extends the base dependencies with libraries used in the context of local development, like tools for ensuring code quality, and the *test.txt* file adds libraries needed for testing. Like that, docker images or local virtual environments can be equipped with precisely the dependencies

required for their use case. For example, a docker image charged with running tests in a CI environment can install base and test dependencies, but omit dependencies aimed at local development.

```

1 project_root/
2 |-- requirements/
3 |   |-- base.txt/
4 |   |-- local.txt/
5 |   |-- test.txt/
6 |-- config/
7 |   |-- settings/
8 |   |   |-- base.py
9 |   |   |-- local.py
10 |   |   |-- test.py
11 |   |   |-- staging.py
12 |   |   |-- production.py
13 |   |-- urls.py
14 |   |-- ...
15 |-- mood_diary/
16 |   |-- templates/
17 |   |-- static/
18 |   |-- locale/
19 |   |-- clients/
20 |   |-- core/
21 |   |-- dashboards/
22 |   |-- diaries/
23 |   |-- notifications/
24 |   |-- rules/
25 |   |-- users/
26 |-- ...

```

Listing (5) General Structure

```

1 project-root/
2 |-- ...
3 |-- mood_diary/
4 |   |-- ...
5 |   |-- diaries/
6 |   |   |-- migrations/
7 |   |   |-- tests/
8 |   |   |   |-- factories.py
9 |   |   |   |-- test_models.py
10 |   |   |   |-- test_views.py
11 |   |   |   |-- test_forms.py
12 |   |   |-- models.py
13 |   |   |-- views.py
14 |   |   |-- forms.py
15 |   |   |-- urls.py
16 |   |   |-- ...
17 |   |-- ...
18 |-- ...

```

Listing (6) Diaries *Django* App

Figure 15 Structure of the Project's Code Base

In a similar pattern, the *config* directory specifies different levels of settings files. There is again a *base.py* file storing general settings. These base settings are then extended or partly overwritten in contexts of local development, testing, staging or production environments. For instance, in an environment for local development, debugging settings might be active that will be disabled in a production environment. Sensitive information like database credentials or API keys are passed to the settings files as environment variables, allowing again to specify different environments such as different database connections for e.g., staging and production contexts. The *config* directory also features a central *urls.py* file, which is the entry point for the *Django* router as here, all available URLs for the project are referenced. These references originate from different *url.py* files in sub-directories of the *mood_diary* directory, where the actual application code is held.

The *mood_diary* directory is made up of different *Django* apps as well as a central *templates* directory holding all HTML templates, a *static* directory holding static assets like images, icons, CSS, and JS files and a *locale* directory dealing with translations. A *Django* app is a module summarizing all code related to a specific aspect of the application, thus helping to organize the application code into logical sections. Typically, a *Django* app consists of Python code for models and migrations, views, forms, URLs and tests. As an example, the diaries app (see Listing 6) holds models storing mood diary entries, their corresponding database migrations, views and input forms to perform CRUD operations on these entries and tests verifying the aforementioned parts are working as expected. In the *tests* sub-directory, next to the actual tests themselves, there exists a factory class for each model, allowing to conveniently create test data in the database for running tests.

In addition to *Django* apps dealing with logical sections of the application, there is the special core app containing functionality shared by all other apps. This shared functionality, by means of example, can be an abstract model class like *TrackCreation* (see Listing 7) equipping all model classes deriving from it with a *created_at* timestamp. As a second example, there is a so-called mixin class *AuthenticatedClientRoleMixin* (see Listing 8) that view classes can derive from. By doing that, they inherit functionality checking if an incoming request both is authenticated and originates from a client user by utilizing the *dispatch* method, which for a request serves as an entry point to the view class it has been routed to.

```

1 class TrackCreation(models.Model):
2     created_at = models.DateTimeField(auto_now_add=True)
3
4     class Meta:
5         abstract = True

```

Listing (7) Abstract Model Class

```

1 class AuthenticatedClientRoleMixin(AccessMixin):
2     def dispatch(self, request, *args, **kwargs):
3         if not (user := request.user).is_authenticated or not user.
            is_client():
4             return self.handle_no_permission()
5         return super().dispatch(request, *args, **kwargs)

```

Listing (8) View Mixin

Figure 16 Examples for Shared Functionality from the Core App

To conclude, the project's code base is organized in a way allowing to easily switch between different contexts as for both dependencies and settings. Regarding the

actual application code, separate *Django* apps each subsume all Python code relevant for each logical section of the application.

7.3 Functionality Overview

Based on the data model and project structure described above, user-facing functionality is implemented via the MVT pattern. The *role* property of the *User* model can take three different values: *client*, *counselor*, and *admin*. All user types share a general set of functionalities and have further functionality specific to their role available to them. An overview of all general user-facing functionality as well as that being specific for user roles can be found in Fig. 17. It also provides an overview of the program flow and the navigation inside the mood diary application. As an example, after logging in, a client is able to view a list of all their mood diary entries, then select a specific entry to view in detail and from there update or delete the entry under consideration. The navigation bar embedded into the uniform frame common to all views introduced in Fig. 9 allows to reach for instance the dashboard from every other page.

In addition to the functionality users can see and experience, there is system-level functionality that may not be directly apparent from using the application. To begin with, *Django*'s internal session framework is leveraged to supply the possibility of logging users in and out. Based on that, authentication ("Who am I?") and authorization ("What am I allowed to see and do?") are implemented via custom view mixins checking whether users are logged in, belong to a certain user type and to restrict displayed data to that a user is allowed to see.

Another implicit feature is the automatic translation of the application's content from English to German based on a user's browser settings. Specifically, if a request indicates German as the preferred language via an *Accept-Language* header, contents are translated automatically application-wide by substituting original English texts from templates and source code wrapped in translation functions with their German equivalents. These equivalents are held and maintained in the *locale* directory of the project. Also based on the language header, information from the database is fetched from fields containing this information either in English or in German.

To be able to send emails to users in case of client creation and forgotten passwords, a transactional email provider is connected to the mood diary application as a third-party service integration. This allows to programmatically trigger the sending of emails from any email account. For client creation, for example, an HTML template is filled with user-specific information and then used as an email message, much like

in the case of rendering a view. In case the client the message is delivered to is unable to display the HTML content, a fallback message in plain text is attached to the email.

A key element of system-level functionality is asynchronous task processing via a *Redis* broker and a *Celery* scheduler and consumer. This is used to evaluate rules in both an event-based and a time-based manner. Every time a client creates or updates a mood diary entry, the corresponding view will trigger an evaluation of all event-based rules, i.e., queue a corresponding job via the broker. Every morning at 6 a.m., a job for the evaluation of time-based rules will be queued per client. In case the preconditions of a rule are fulfilled, the consumer creates a notification for the user and, if supported and enabled, sends a push notification to the client's device(s).

Concerning the mood diary application's PWA enhancement, the application is able to send push notifications. A service worker listens for push events and displays push notifications accordingly (see also Fig. 2). Additionally, the service worker is responsible for caching an application shell in order to provide users with an instant response after opening the mood diary application. Next to the service worker, a web app manifest specifying that the application should be displayed like a native app after installing it as a PWA is implemented and, like the service worker, served as a static asset by the *Django* server.

Far away from user interactions, but viable to a well-developed and portable application that can be instantly updated, a CI/CD process including virtualization techniques is implemented. The CI part of this process builds a *Docker* image of the current application and runs all tests it contains whenever a push is made to the project's code base stored on a *GitLab* server. When pushing to specific branches, the CD part of the process automatically deploys the newest version of the application to a staging or a production environment, replacing the previously running application without any downtime.

Summarizing, the mood diary application implements user-facing functionality shared across all user roles as well as functionality specific to different user types via the presented data model and the MVT pattern. To complete the overview, this user-facing functionality is complemented with system-level functionality amongst other things handling issues like authentication and authorization or asynchronous task processing and a CI/CD process facilitating application development and deployment.

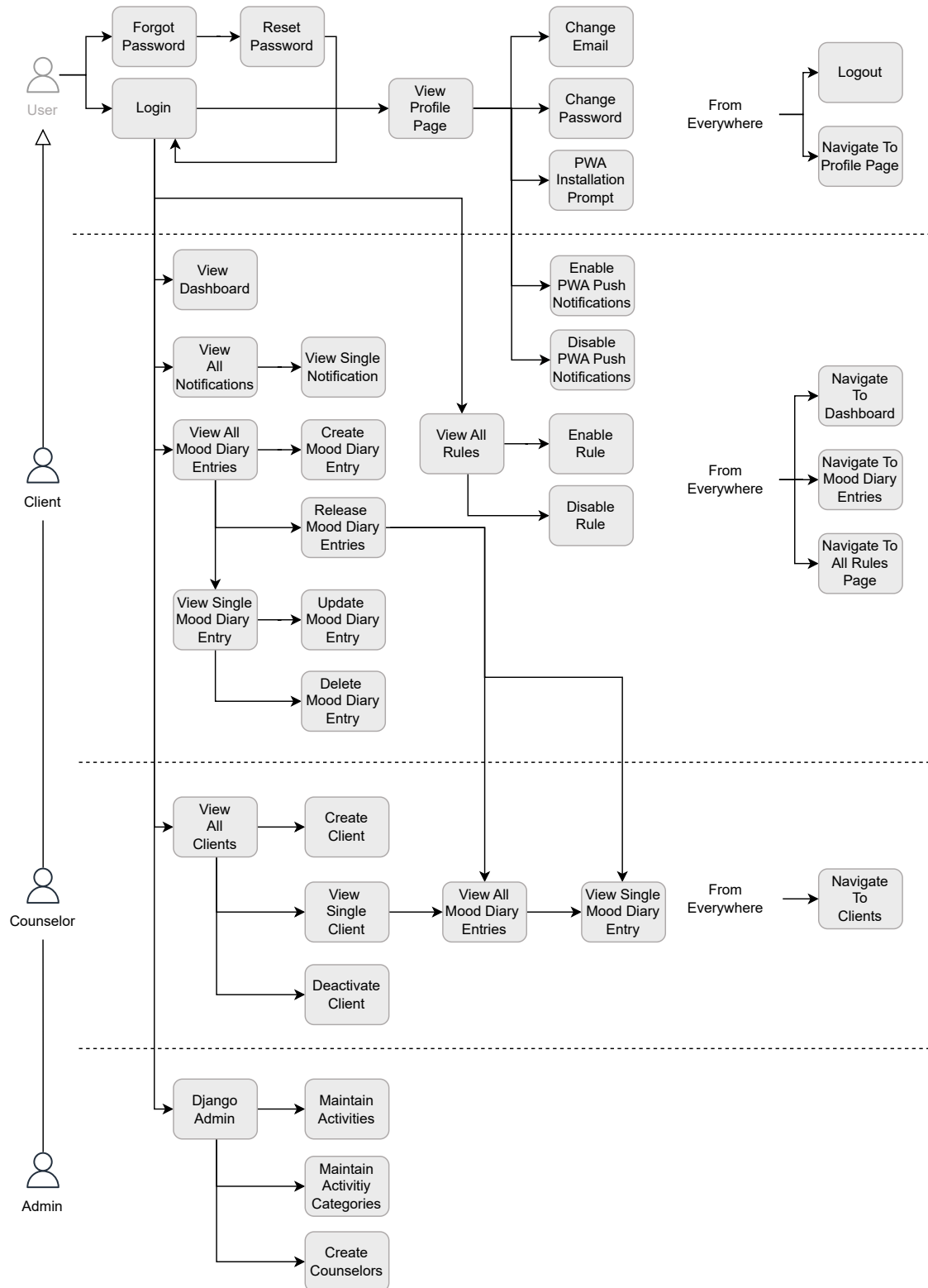


Figure 17 Overview of User-Facing Functionality

7.4 Implementation Details

In the following, some selected functionality comprised in the overview given above will be highlighted in greater detail. This deeper dive will be guided by the project's three main goals in order to provide insights into the realization of each of them. Furthermore, both the implemented PWA functionality and the CI/CD process will be covered.

7.4.1 Digital Mood Diary Intervention

Apart from providing CRUD functionality in the context of mood diary entries, the mood diary application developed in this project features a dashboard containing a chart showing the progression of daily average moods and the most recent mood highlights. While it was established that the intervention in a digital form could inherently benefit from an increased compliance, this feature is implemented to further strengthen this compliance. With the overview provided in the dashboard, clients hopefully experience it as an informative starting point to a regular use of the mood diary application.

As described above, the data model was designed with a separate mood diary model to allow declaring methods on it that can conveniently query mood diary entries for a specific client. Such methods are the source for the data displayed in the dashboard. The mood diary model can aggregate average mood values and return most recent mood highlights (see the methods in Listing 9). To do that, both methods leverage *Django's* ORM to select, aggregate, and order mood diary entries as requested. The key concept here is that of a *Django Queryset* which represents a set of objects from the database that can be manipulated by various selection, filter, and ordering methods. For example the combination of the *values* and *annotate* methods used in lines 6 and 7 correspond to a Structured Query Language (SQL) clause of *GROUP BY*, resulting into an average mood value per date.

```

1 class MoodDiary(models.Model):
2     ...
3     def average_mood_scores_previous_days(self, n_days: int) ->
      QuerySet:
4         return (
5             self.entries.order_by("-date")
6             .values("date")
7             .annotate(average_mood=Avg("mood__value"))[:n_days]
8         )
9
10    def most_recent_mood_highlights(self, n_highlights: int) ->
      QuerySet:

```

```

11         return self.entries.order_by("-mood__value", "-date")[:
12             n_highlights]
    ...

```

Listing 9 Data Transformation Methods on the Mood Diary Model

These data transformation methods are used in the view for the dashboard. Here, when a *GET* request is dispatched to the view's *get* method, average mood scores for the last seven days as well as the three most recent mood highlights are fetched from the database and rendered into the template associated to the view (see Listing 10). As the data transformation methods return *Querysets*, the results can be further trimmed down to e.g., selecting just the average mood values per date from them, as it is done in lines 11 and 12. Note that the method aggregating the average mood scores returns the scores beginning with the most recent date. This ordering is reversed here with the function call in line 8. Also, in addition to the values, the dates are provided to the template which is omitted here for brevity.

```

1 class DashboardClientView(AuthenticatedClientRoleMixin, View):
2     template_name = "dashboards/dashboard_client.html"
3
4     def get(self, request):
5         ...
6         mood_scores = mood_diary.average_mood_scores_previous_days
7             (7)
8         mood_scores_values = list(
9             reversed(
10                 [
11                     round(mood_score_value, 1)
12                     for mood_score_value in mood_scores.values_list
13                         ("average_mood", flat=True)
14                 ]
15             )
16         )
17         ...
18         mood_highlights = mood_diary.most_recent_mood_highlights(3)
19         return render(
20             request,
21             self.template_name,
22             {
23                 "mood_scores_values": mood_scores_values,
24                 "mood_highlights": mood_highlights,
25                 ...
26             },
27         )

```

Listing 10 Data Transformation Methods on the Mood Diary Model

The transformed data the average mood chart is based on needs to be made available to the JS code responsible for displaying the actual chart, as a dynamic display like a chart cannot be accomplished by HTML alone. This is achieved by rendering the data into script tags via the *json_script* utility function, resulting in a script tag

like for example `<script id="chart-data" type="application/json">[0.6, 1.3, 1.5, 2, 1.7, 2.3, 2.0]</script>`. Then, the JS code is able to pick up the data and use it. This process is depicted in Fig. 18.

```
1 ...
2 {{ mood_scores_dates|json_script:"chart-labels" }}
3 {{ mood_scores_values|json_script:"chart-data" }}
4 ...
5 <script src="{% static 'js/average_mood_chart.js' %}"></script>
6 ...
```

Listing (11) Chart Data in the Dashboard Template

```
1 const labels = JSON.parse(document.getElementById('chart-labels').
    textContent);
2 const data = JSON.parse(document.getElementById('chart-data').
    textContent);
3 ...
```

Listing (12) Chart Data in JS Code

Figure 18 Passing Data from a *Django* Template to JavaScript Code

Recapitulating, the mood diary application features a dashboard for clients showing the progression of daily average moods in a chart and the most recent mood highlights to strengthen compliance. Data for the dashboard is transformed via calling methods on the mood diary model from the dashboard view and then, for the chart, rendered into script tags to make it available to the JS code displaying it.

7.4.2 Counselor Access

Counselors can access their clients' mood diary entries. However, this is only possible if the entries have been explicitly released by a client. The process to release mood diary entries implemented in this project can serve as an example of using template inheritance to structure templating code. All templates extend a base template that defines the uniform frame all views are embedded into. What is more, this base template declares different named sections, so-called blocks. Like that, in a template extending the base template, these specific blocks can be targeted to conveniently insert, overwrite or complement code, making templates dynamic and reusable. For example the client list view for mood diary entries contains a button to release entries. When clicked, it opens a modal window where the client can confirm that they want to release entries to their respective counselor. This modal window is inserted into the corresponding block declared by the base template as can be seen in Listing 13: The client list view template specifies that it is extending the

base template, then targets the block for modals declared there and adds a modal for releasing mood diary entries after calling `{{ block.super }}` to ensure that no modals declared in this block by the base template (namely a *Logout* modal) are overwritten. This reminds of Python inheritance, so it is intuitive to use in this context.

```

1 {% extends "base_client_role.html" %}
2 ...
3 {% block modals %}
4     {{ block.super }}
5     <!-- Release Entries Modal-->
6     <div class="modal fade" id="releaseModal" tabindex="-1" role="
        dialog" aria-labelledby="exampleModalLabel" aria-hidden="
        true">
7         ...
8         <form method="post" action="{% url 'diaries:
        release_mood_diary_entries' %}">
9         ...
10    </div>
11 {% endblock %}
12 ...

```

Listing (13) Template Inheritance With Blocks

```

1 class MoodDiaryEntryReleaseView(AuthenticatedClientRoleMixin, View)
2 :
3     def post(self, request):
4         mood_diary = self.request.user.client.mood_diary
5         mood_diary.release_entries()
6         return redirect("diaries:release_mood_diary_entries_done")

```

Listing (14) Mood Diary Entry Release View

```

1 class MoodDiary(models.Model):
2     def release_entries(self):
3         self.entries.filter(released=False).update(released=True)

```

Listing (15) Mood Diary Entry Release Method

Figure 19 Releasing Mood Diary Entries

Clicking the *Release* button in the modal triggers a post request to an endpoint responsible for releasing mood diary entries. Releasing the entries really just means to update the value of the boolean field *released* in the mood diary entry model (see Fig. 19 for the release view and method). This boolean field, in turn, controls which mood diary entries of their clients a counselor can see. This is done by filtering out any unreleased entries in list and detail mood diary entry views for the counselor role.

To conclude, the mood diary application implements counselor access to their clients' mood diary entries in a way that clients can control when new entries are made available to their counselor. This is achieved via a simple boolean field and serves as an example for using template inheritance to structure templating code into dynamic and reusable parts.

7.4.3 Automatic Analyses

Automatic Analyses, i.e., rule evaluations providing timely feedback to clients about their mood diary entries are triggered in two fashions: Event- and time-based. Event-based evaluations are triggered whenever a client creates or updates an entry. In that case, a *RuleMessage* object containing a client's id and a timestamp identifying when exactly the rule evaluation was requested is provided to a task function in an asynchronous manner (cf. Listing 16). This queues a task with the *Redis* broker, where it is stored until it is picked up by the *Celery* consumer. The consumer then iterates through all event-based rules described in Tab. 3, where each rule is represented by a specific rule class. It instantiates an object of each specific rule class, providing it with the client's id and the timestamp identifying when the evaluation was requested, and calls the *evaluate* method common to all rule classes (see Listing 17). In a similar fashion, time-based evaluation is triggered by the *Celery* scheduler every morning at 6 a.m. by queuing an initialization task that iterates through all active clients and in turn queues a task for the evaluation of time-based rules for each of them. These time-based evaluation tasks work like those for event-based rule evaluation, calling the *evaluate* method on instantiated objects for all time-based rule classes.

This *evaluate* method is common to all rule classes as all rule classes derive from the same abstract base class. The base class defines several properties and methods each specific rule class must implement. This ensures that all specific rule classes provide the same API for interacting with them and that during evaluation, they all run through the same process (cf. Listing 18). This process consists of (1) checking whether the client is subscribed to the rule, (2) if the rule is allowed to trigger and (3) if the preconditions for the rule are met, and is implemented via three separate methods all required to return a boolean value. All three conditions are evaluated incrementally, meaning that as soon as one condition is not met, the whole evaluation process is terminated. This could for example be the case if dealing with a rule that is only allowed to be triggered once a day and that has been triggered today already. In that case, the method checking whether the rule is allowed to trigger would resolve to returning *False*. If otherwise all conditions are met, the

```

1 class MoodDiaryEntryListView(AuthenticatedClientRoleMixin,
    AjaxListView):
2     ...
3     def form_valid(self, form):
4         ...
5         msg = RuleMessage(client_id=client.id, timestamp=entry.
            updated_at)
6         task_event_based_rules_evaluation.delay(msg)
7         ...

```

Listing (16) Queuing a Rule Evaluation Task

```

1 @shared_task
2 def task_event_based_rules_evaluation(msg: RuleMessage):
3     logger.info(f"Event-based Rule Evaluation: {msg}")
4     for rule_class in EVENT_BASED_RULES:
5         rule = rule_class(*msg)
6         rule.evaluate()

```

Listing (17) Event-based Rule Evaluation Task

Figure 20 Event-based Rule Evaluation

triggering of the rule is persisted into a log table and a notification is created in the database from the conclusion message associated with the rule to be displayed within the mood diary application. In case the corresponding client has push notifications configured, also push notifications are sent to all devices registered for the client, informing them that a pattern has been detected.

```

1 class BaseRule:
2     ...
3     def triggering_allowed(self) -> bool:
4         raise NotImplementedError
5     ...
6     def evaluate(self):
7         if not self.client_subscribed():
8             return
9         if not self.triggering_allowed():
10            return
11        if not self.evaluate_preconditions():
12            return
13        self.logger.info(f"Rule triggered for client {self.
            client_id}: {self.rule_title}")
14        self.persist_rule_triggering()
15        self.create_notification()
16        self.create_push_notifications()

```

Listing 18 Abstract Base Rule Class

Taken together, event- and time-based rule evaluations are run on a regular basis as tasks executed by the *Celery* consumer, where each specific rule class implements a common API operated by the consumer. If all conditions in the evaluation process

are met, a notification as well as optional push notifications are created informing clients about detected patterns.

7.4.4 Progressive Web App Enhancement

One feature made possible by PWA enhancement is installing a PWA to a device so that it can be used like a native application. To notify users about this possibility, multiple modern browsers support the *BeforeInstallPromptEvent* [Moz23c]. This event is triggered when a browser detects that a website can be installed as a PWA. Depending on the device and the browser, different interfaces prompting the user to install the PWA are displayed in consequence. To give users of the mood diary application the opportunity to manually trigger the installation process at any time, on the profile page, a button is placed allowing users to prompt the installation process. Naturally, this button only has an effect if the utilized browsers supports the *BeforeInstallPromptEvent*. Under the hood, there is an event listener storing the prompt event as soon as it is triggered to display the interface associated with it when the button on the profile page is clicked (see Fig. 21).

```

1 ...
2 <a href="#" class="btn btn-primary" id="installButton">...</a>
3 <script>
4 document.getElementById('installButton').addEventListener('click',
    function () {installApp();});
5 </script>

```

Listing (19) Installation Button

```

1 let deferredPrompt;
2 window.addEventListener('beforeinstallprompt', (e) => {
3     deferredPrompt = e;
4 });
5 function installApp() {
6     if (deferredPrompt) {deferredPrompt.prompt();}
7 }

```

Listing (20) Manual Installation Prompt

Figure 21 Progressive Web App Installation Prompt

A second PWA enhancement allows for sending push notifications to users via the web push API [W3C22a]. To enable this, first, a user must grant permission via a pop-up interface. This interface can again be prompted from the profile page in a similar fashion to the installation prompt, so that users can fully control when and if they want to deal with the pop-up for configuring push notifications. After granting permission, the *subscribeUserToPush* function is called which creates a

subscription object containing, amongst other things, an individual endpoint URL for the push service the respective browser is using. This subscription object is sent to the server and stored in the database so that the endpoint contained in it can be used to send push notifications to a user's device at a later point in time. In this process, a private and public key pair, so-called Voluntary Application Server Identification for Web Push (VAPID) keys, are used to ensure that only the application responsible for creating the push subscription can use it to send push notifications. Clients can at any point disable push notifications by changing the value of the *push_notifications_granted* field on the client model via a button on the profile page.

To summarize, in browsers supporting the functionality, users of the mood diary application can manually prompt an installation process to install the application as a PWA and configure their preferences for receiving push notifications. This is made possible by handling browser events related to PWA installation and push notification procedures.

7.4.5 Continuous Integration/Continuous Delivery Process

The CI/CD process implemented for the mood diary application consists of four stages: build, test, release, and deploy. The first two stages make up the CI part of the process and run every time a commit is pushed to the project's code base. In the build stage, a *GitLab* runner executes a job that creates a docker image from the current status of the code base, pushing it to the *GitLab* container registry. From there, the image is pulled by the job executed in the test stage, that spins it up as a container next to running a *Redis* and a *PostgreSQL* container. Against this infrastructure, all tests are executed, configured to additionally measure test coverage. The test coverage results are displayed in two ways in a merge request associated with such a CI pipeline. For one, a total percentage of test coverage is calculated and displayed. Second, each line of code changes introduced by the merge request is marked as covered or not covered by the existing tests. For a simplified version of the test job omitting the rather detailed post-processing of the recorded test coverage, refer to Listing 21.

```

1 test_mood_diary:
2   stage: test
3   image: $APP_TEST_IMAGE
4   services:
5     - name: postgres:15.2
6       alias: postgres
7     - redis
8     ...
9   script:
```

```

10     - pytest -n auto --dist loadfile --cov --cov-config=pyproject.
      toml --cov-report=xml:coverage.xml --record-mode=none
11     ...
12     ...

```

Listing 21 Continuous Integration Test Job

The release and the deploy stages, making up the CD part of the process, only run when a commit is pushed to specific branches of the code base. When pushing to the *develop* branch, a staging deployment is triggered, using the image from the previous build stage. When pushing to the *main* branch, in the release stage, an image targeted at production use is built that, for example, does not include any test dependencies. This image is then used in the deploy stage to deploy a new version of the mood diary to a production environment. Both the staging and the production deployment jobs work in a similar fashion: They copy the *docker-compose* file to the target machine, log in to the container registry, pull the newest versions of the images specified in the *docker-compose* file, start them up and run any outstanding database migrations. See Listing 22 for the main components of the staging deployment job.

```

1 deploy_to_staging:
2     stage: deploy
3     ...
4     script:
5         - cp infra/staging/docker-compose.yml ${DEPLOY_TARGET_DIR}/
          docker-compose.yml
6         - cd ${DEPLOY_TARGET_DIR}
7         - docker login -u $CI_REGISTRY_USER -p $CI_JOB_TOKEN
          $CI_REGISTRY
8         - docker compose pull && docker compose up -d
9         - docker compose exec django python manage.py migrate
10        ...
11        ...
12        only:
13            - develop

```

Listing 22 Continuous Delivery Deploy Job

Concluding, the project's CI/CD process continuously builds the application in its current version and executes all available tests against it, ensuring the detection of any failed tests and additionally providing helpful information about test coverage. Also, it enables a seamless deployment by automatically updating the staging and production applications when pushing to the branches associated with them.

For an impression of the mood diary application resulting from the described implementation, please refer to the screenshots in Appendix B. It contains screenshots from accessing the application from both a desktop computer (Fig. 22) and a mobile device (Fig. 23).

8 Evaluation of the Mood Diary Application

Assessing the whole development process of the mood diary application, this section will evaluate the fulfillment of the three main goals defined for this project as well as the technical requirements it set out to meet, that were in part derived from those goals. Further, it will illustrate limitations of the mood diary application and propose ideas for future work.

8.1 Main Goals

The first goal was to develop a digital form of the mood diary intervention with the motivation to improve client compliance. The developed mood diary application implements all conceptualized features pertaining to this goal. Specifically, these features subsume CRUD functionality for mood diary entries, aggregation of entry data into a dashboard view, the possibility for admin users to maintain activities and activity categories and push notifications for re-engagement.

With this digital intervention form introducing possibilities beyond what was feasible in a paper-pencil context, counselors should be given the opportunity to access their clients' mood diary entries earlier than only during the next counseling session. In the mood diary application, this second goal was realized by providing counselors with access to list and detail views of their clients' entries. This should hopefully enable counselors to monitor the intervention process and to prepare for reviewing its results. Still, clients can control whether and when their mood diary entries are released to their counselor to counteract any feeling of observation.

As a third goal, clients should receive timely feedback about the mood diary entries they logged through automatic analyses. Conceptualized as a rule-based expert system trying to mimic counselor expertise, the mood diary application is able to detect behavioral patterns and to deliver content based on these patterns to users. This is achieved by evaluating rules in both event- and time-based manners via asynchronous task processing. As a result, notifications with content structured into a triad of pattern identification, psychoeducation, and approaches for change or maintenance are created. What is more, the mood diary application is able to send push notifications informing users that a pattern has been detected.

All in all, the project's three main goals can be considered as fulfilled. During the development process, each goal was conceptualized into specific features. These features were then implemented accordingly and are ready to be put to production use at the PSB.

8.2 Technical Requirements

The mood diary application was developed as a web application enhanced with PWA functionality for an app-like experience, thus providing platform-independence with easy and extensive access for both clients from mobile devices and counselors from desktop computers. This was verified by test users successfully accessing the staging system from multiple desktop computers and mobile devices. To support automatic analyses, asynchronous task processing was successfully implemented via *Celery* with a *Redis* broker, proffering event- and time-based task queuing.

As it is likely that maintenance and advancement of the mood diary application will be transferred to PSB staff, it was aimed to utilize a limited amount of popular, at best easy-to-learn technologies. These mainly consist of Python and *Django*, *Docker*, *GitLab* CI/CD, HTML, and CSS. However, especially in the context of PWA functionality being backed by a service worker written in JS, it is not entirely possible to rely on the Python language alone. Although there are only a few lines of JS code, in retrospective, it would have also been a valid option to use a JS framework. To further keep maintenance simple, a portable application that can be easily deployed should be developed. Fulfillment in that regard is indicated by successfully deploying the virtualized application to a staging system and everything being set up for a production deployment only awaiting some final coordination with the PSB.

The mood diary application manages sensitive data, so nothing should be stored with third parties to provide full data sovereignty. This requirement was matched with the selected architecture as all user-related data only exists within the application on a single server. It was calculated that this single-server application needs to support up to 300 users. Using *JMeter*²², 300 users accessing the application by logging in, viewing the dashboard and retrieving mood diary entries during a 10 second interval were simulated on the staging system. Response times were fast, laying below 150ms for login and below 85ms for the other requests. On top of that, a CI/CD process should contribute to developing a qualitative software in a reasonable amount of time that is easy to deploy. Based on a test coverage of over 97% and a full development process lasting less than 9 weeks featuring an already proven automatic deployment, this process can be considered as successfully implemented.

In summary, the proposed technical requirements are satisfied by the resulting mood diary application. Nonetheless, with opting for JS instead of Python, that was

²² <https://jmeter.apache.org/>

chosen due to its easy-to-learn character, the number of employed technologies could have been reduced by one.

8.3 Limitations and Future Work

PWA enhancement offers great features, providing web apps with the possibility to look and feel like native ones. Browsers have their own implementations of PWA functionality, which unfortunately is not standardized in every aspect. Although by now, all major browsers support service workers [Moz23a], there are different levels of support for e.g., specific web app manifest properties like declaring a background color [Moz23b]. While manually adding the PWA to the homescreen on mobile devices is widely supported, only some browsers implement the installation prompt to conveniently trigger this process [Moz23c]. Being able to use the installation prompt instead of having to manually install the PWA from the browser's context menu arguably improves the user experience when dealing with a PWA, especially for users who are new to the concept. In a similar fashion, enabling push notifications depends on a pop-up interface controlled by the browser. If for example on a mobile device, notifications for the whole browser are turned off, the browser will automatically prevent the pop-up interface from appearing. In that case, the button on the mood diary app's profile page meant to trigger showing that interface would have no effect, which happened for one test user. The mood diary application cannot control the notification settings of the browser it is running in. Consequently, although theoretically enabled to look and feel like a native app, the mood diary application cannot guarantee the full PWA experience to all users. This is because the range of support depends on the browser they are employing to access the application and its configuration. Hopefully, with an increasing adoption of PWA concepts by developers, a higher level of convenience and standardization between browsers in terms of PWA support will be reached in the future.

The rule-based expert system providing timely feedback to clients is currently implemented as a set of hard-coded rules that globally apply for every client. While these rules were developed based both on the experience of the PSB and a scientific foundation, it is still conceivable that for specific individuals, different thresholds or totally other rules could be relevant. What is more, the state of research naturally changes over time, which may also require adjustments to the rules in the future. As of now, clients can deactivate rules if they are not helpful to them. In a next evolutionary step, however, it would be beneficial to allow health professionals to update rules, to fit rule parameters to specific individuals or to create new rules altogether. This functionality could be implemented via a Domain-Specific Language

(DSL), enabling non software developers like health professionals to maintain the expert system. To this end, the Business Rules Management System *Drools*²³ could be a good starting point, as it has been successfully used in other work [Neu+22].

Next to advancing the mood diary application to feature a more sophisticated expert system, future work should comprise an empirical compliance study, which unfortunately lay outside the current project’s scope. The application was developed with the motivation to increase client compliance for the mood diary intervention. While there is a solid scientific foundation backing up this motivation [TK17; Mat+08; MD11], it is mandatory to investigate the hypothesis that using this specific application indeed increases compliance for the mood diary intervention. Such a study could, in a randomized controlled trial, compare compliance rates for clients conducting the traditional paper-pencil version of the mood diary intervention and for those using the mood diary application.

On a further note, there are numerous ways to refine and expand the mood diary application. These include but are not limited to using push notifications also to remind clients about logging their moods and activities, to introduce an onboarding process into the application providing an overview of its functionalities to new users or to extend automatic analyses and push notifications to counselors. What is more, in the long term, the mood diary application could advance to something like a companion application for psychological counseling, digitizing more aspects of the counseling process and adding additional value to it. An example for this could be a digital space to document psychoeducative content and personal notes related to insights gained from counseling sessions.

Recapitulating, there are several limitations as well as still uncharted pathways regarding the mood diary application. Due to varying extents of support and individual configurations in different browsers, it cannot guarantee the same PWA experience to all users. In terms of flexibility and maintenance, the currently implemented rule-based expert system would benefit from being advanced into a DSL and an empirical compliance study could verify whether the mood diary application indeed increases client compliance. Beyond that, there are many possibilities to improve and extend the application into a kind of companion application for psychological counseling.

²³ <https://www.drools.org/>

9 Conclusion

A widespread demand for mental services, for example in the context of depressive symptoms, meets an insufficient supply. It therefore is essential that existing services are as efficient and effective as possible. However, psychological counseling and therapy methods like the mood diary intervention in their original paper-pencil form can lack compliance, thereby endangering treatment outcome. Based on indications that a digital form of the intervention could increase said compliance, this project set out to (1) develop a digital application for the mood diary intervention available on a smartphone to ensure client compliance. Beyond that, exploiting further possibilities introduced by a digital solution, it was aimed to (2) make mood diary entries accessible to counselors and to (3) provide timely feedback about them to clients through automatic analyses.

Based on these goals as well as additional development considerations with regard to the kind of sensitive data at hand and to delivering a portable, qualitative software that is easy to maintain and deploy, technical requirements were derived. These in turn motivated the selection of a suitable technology stack. With this foundation, the mood diary application was conceptualized centered around the project's three main goals and implemented accordingly. The resulting mood diary application is a PWA based on the *Django* web development framework backed by a *PostgreSQL* database allowing clients and counselors easy access from both mobile devices and desktop computers. It possesses the capability to process asynchronous tasks via *Celery* used for automatic analyses and a CI/CD process featuring automatic deployment mechanisms to both a staging and a production environment.

A subsequent evaluation indicated that the developed application meets all three main goals and that all technical requirements could be matched to a satisfying extent. The application demonstrates fast response times also under expected load and exhibits a high test coverage. Nonetheless, there are limitations to the PWA enhancement and the rule-based expert system providing feedback to clients. These limitations as well as the conduction of a compliance study and further ideas for the application's advancement demand future work. In spite of that, the mood diary application in its current form is ready to be put to production use by the PSB, an institution offering psychological counseling for students. There, it should replace the analog form of the mood diary intervention used so far.

Wrapping up, this project motivated, conceptualized and implemented the mood diary application as a digital intervention to help increase client compliance and make psychological counseling at the PSB more effective.

Appendix

A Activities and Categories

Category	Activity
Sleep	Getting Up
	Going to bed
	Sleeping
	Other
Food	Meal
	Snack
	Restaurant
Physical Activity	Sports
	Walking
	Distance by foot/bike
Problematic Behavior	Ruminating
	Worrying
	Self-harm
	Alcohol/Drug consumption
Social	Meeting friends
	Meeting partner
	Telecommunication with friends
	Telecommunication with partner
	Meeting family
	Telecommunication with family
	Going out/Partying
	Supporting friends
	Supporting family
	Supporting partner

Table 4 Excerpt of Activities and Categories for the Mood Diary Application

Each category contains an additional activity "Other" that is omitted here.

B Screenshots

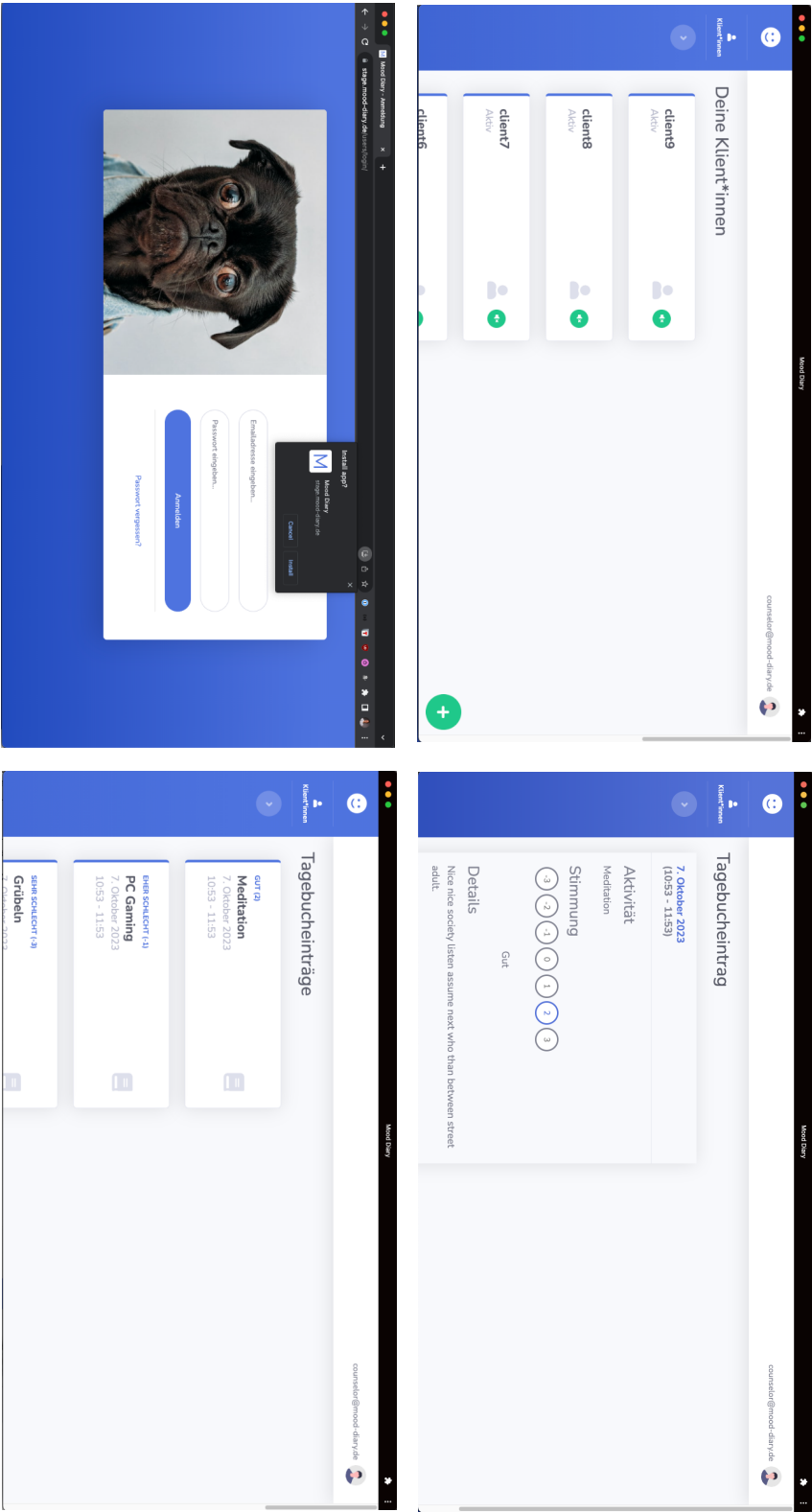


Figure 22 Screenshots Desktop Computer

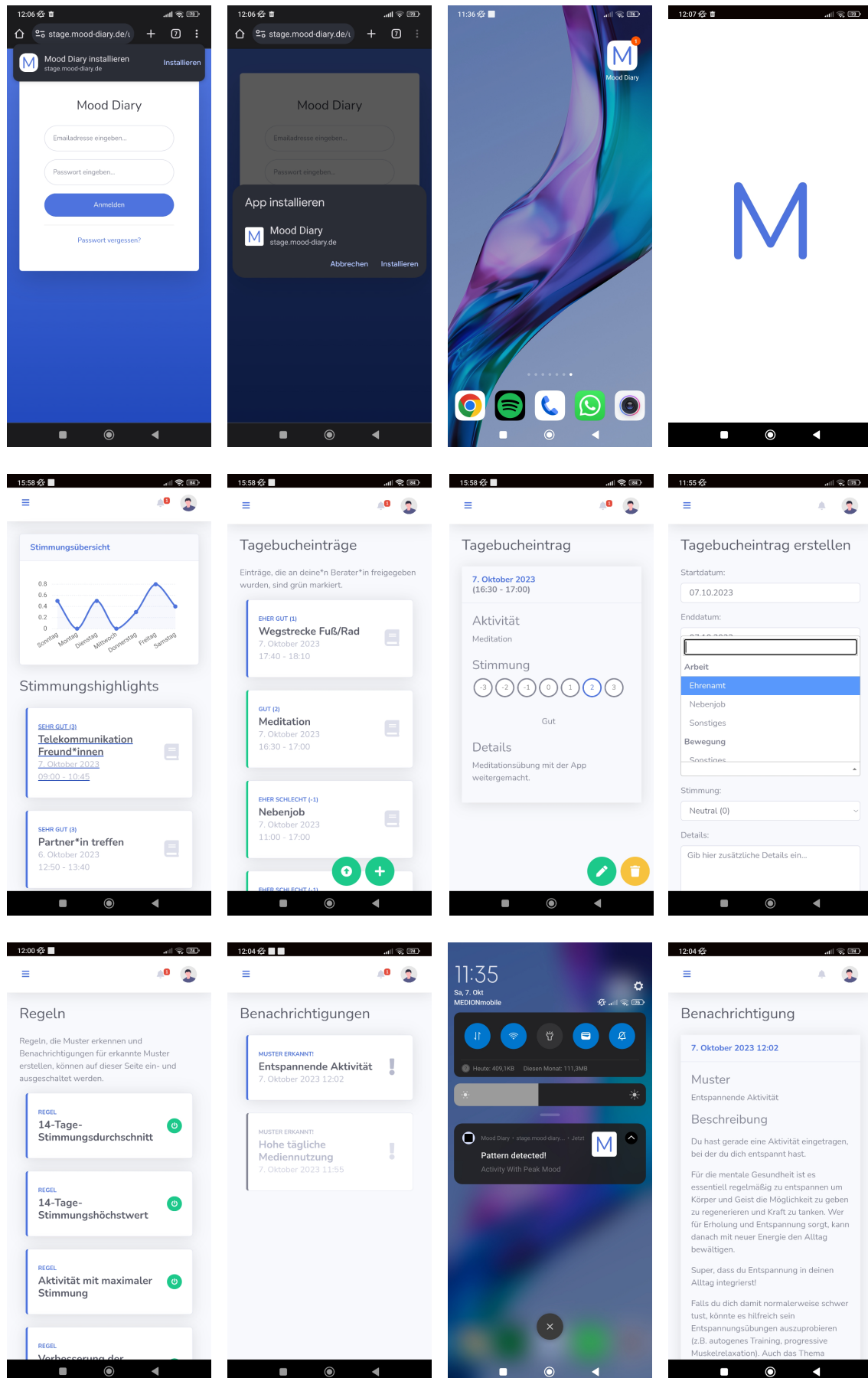


Figure 23 Screenshots Mobile Device

References

- [Ang+23] Angle, Patrick et al. *WebKit Features in Safari 16.4*. 2023. URL: <https://webkit.org/blog/13966/webkit-features-in-safari-16-4/> (visited on 10/01/2023).
- [Aue+18] Randy P. Auerbach et al. “WHO World Mental Health Surveys International College Student Project: Prevalence and distribution of mental disorders”. In: *Journal of abnormal psychology* 127.7 (2018), pp. 623–638. DOI: 10.1037/abn0000362.
- [BR18] David Bakker and Nikki Rickard. “Engagement in mobile phone app for self-monitoring of emotional wellbeing predicts changes in mental health: MoodPrism”. In: *Journal of affective disorders* 227 (2018), pp. 432–442. DOI: 10.1016/j.jad.2017.11.016.
- [Bau+05] Michael Bauer et al. “Does the use of an automated tool for self-reporting mood by patients with bipolar disorder bias the collected data?” In: *MedGenMed : Medscape general medicine* 7.3 (2005), p. 21.
- [Bau+06] Michael Bauer et al. “Mood Charting and Technology: New Approach to Monitoring Patients with Mood Disorders”. In: *Current Psychiatry Reviews* 2.4 (2006), pp. 423–429. DOI: 10.2174/157340006778699747.
- [Bau+19] James W Baurley et al. “A web portal for rice crop improvements”. In: *Biotechnology: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2019, pp. 344–360.
- [BMG18] Andreas Biørn-Hansen, Tim A. Majchrzak, and Tor-Morten Grønli. “Progressive Web Apps for the Unified Development of Mobile Applications”. In: *Web Information Systems and Technologies*. Ed. by Tim A. Majchrzak et al. Cham: Springer International Publishing, 2018, pp. 64–86.
- [BZS13] A. Bogdanchikov, M. Zhaparov, and R. Suliyev. “Python to learn programming”. In: *Journal of Physics: Conference Series* 423 (2013), p. 012027. DOI: 10.1088/1742-6596/423/1/012027.
- [Bun18] BundesPsychotherapeutenKammer. *Studie Wartezeiten 2018*. 2018. URL: https://api.bptk.de/uploads/20180411_bptk_studie_wartezeiten_2018_c0ab16b390.pdf (visited on 10/07/2023).
- [BS00] D. D. Burns and D. L. Spangler. “Does psychotherapy homework lead to improvements in depression in cognitive-behavioral therapy or does improvement lead to increased homework compliance?” In: *Journal of consulting and clinical psychology* 68.1 (2000), pp. 46–56. DOI: 10.1037/0022-006x.68.1.46.
- [Cam+20] Jonathan Champion et al. “Addressing the public mental health challenge of COVID-19”. In: *The lancet. Psychiatry* 7.8 (2020), pp. 657–659. DOI: 10.1016/S2215-0366(20)30240-6.

- [Cao+20] Jian Cao et al. “Tracking and Predicting Depressive Symptoms of Adolescents Using Smartphone-Based Self-Reports, Parental Evaluations, and Passive Phone Sensor Data: Development and Usability Study”. In: *JMIR mental health* 7.1 (2020), e14045. DOI: 10.2196/14045.
- [Con10] Vicki S. Conn. “Depressive symptom outcomes of physical activity interventions: meta-analysis findings”. In: *Annals of behavioral medicine : a publication of the Society of Behavioral Medicine* 39.2 (2010), pp. 128–138. DOI: 10.1007/s12160-010-9172-x.
- [Cur+19] Dasari Hermitha Curie et al. “Analysis on Web Frameworks”. In: *Journal of Physics: Conference Series* 1362.1 (Nov. 2019), p. 012114. DOI: 10.1088/1742-6596/1362/1/012114. URL: <https://dx.doi.org/10.1088/1742-6596/1362/1/012114>.
- [DEM21] Muna Dubad, Farah Elahi, and Steven Marwaha. “The Clinical Impacts of Mobile Mood-Monitoring in Young People With Mental Health Problems: The MeMO Study”. In: *Frontiers in psychiatry* 12 (2021), p. 687270. DOI: 10.3389/fpsy.2021.687270.
- [Fow06] Martin Fowler. *Continuous Integration*. 2006. URL: <https://martinfowler.com/articles/continuousIntegration.html> (visited on 09/27/2023).
- [Fow13] Martin Fowler. *ContinuousDelivery*. 2013. URL: <https://martinfowler.com/bliki/ContinuousDelivery.html> (visited on 09/27/2023).
- [Fun21] Kay Funke-Kaiser. *BPTK-Auswertung: Monatelange Wartezeiten bei Psychotherapeut*innen*. 29.03.2021. URL: <https://bptk.de/pressemitteilungen/bptk-auswertung-monatelange-wartezeiten-bei-psychotherapeutinnen/> (visited on 09/27/2023).
- [GS02] Anne Garland and Jan Scott. “Using homework in therapy for depression”. In: *Journal of Clinical Psychology* 58.5 (2002), pp. 489–498. DOI: 10.1002/jclp.10027.
- [GLN06] Scott T. Gaynor, P. Scott Lawrence, and Rosemary O. Nelson-Gray. “Measuring homework compliance in cognitive-behavioral therapy for adolescent depression: review, preliminary findings, and implications for theory and practice”. In: *Behavior modification* 30.5 (2006), pp. 647–672. DOI: 10.1177/0145445504272979.
- [Gin06] Paul Ginns. “Integrating information: A meta-analysis of the spatial contiguity and temporal contiguity effects”. In: *Learning and Instruction* 16.6 (2006), pp. 511–525. DOI: 10.1016/j.learninstruc.2006.10.001.
- [GH09] Ian H. Gotlib and Constance L. Hammen. *Handbook of depression*. 2nd ed. New York and London: Guilford Press, 2009.
- [GA11] Crina Grosan and Ajith Abraham. “Rule-Based Expert Systems”. In: *Intelligent Systems: A Modern Approach*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 149–185. DOI: 10.1007/978-3-642-21004-4_7. URL: https://doi.org/10.1007/978-3-642-21004-4_7.
- [Här+10] Martin Härter et al. *S3 Praxisleitlinien in Psychiatrie und Psychotherapie: Nationale VersorgungsLeitlinie - Unipolare Depression*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-13103-5.

- [Hau21] Martin Hautzinger. *Kognitive Verhaltenstherapie bei Depressionen: Mit E-Book inside und Arbeitsmaterial*. 8., überarbeitete Auflage. Weinheim: Julius Beltz GmbH & Co. KG, 2021.
- [Hay85] Frederick Hayes-Roth. “Rule-Based Systems”. In: *Commun. ACM* 28.9 (Sept. 1985), pp. 921–932. DOI: 10.1145/4284.4286. URL: <https://doi.org/10.1145/4284.4286>.
- [HF04] Sylvia Helbig and Lydia Fehm. “Problems with Homework in CBT: Rare Exception or Rather Frequent?” In: *Behavioural and Cognitive Psychotherapy* 32.3 (2004), pp. 291–301. DOI: 10.1017/S1352465804001365.
- [Hor+22] Adam Horwitz et al. “Utilizing daily mood diaries and wearable sensor data to predict depression and suicidal ideation among medical interns”. In: *Journal of affective disorders* 313 (2022), pp. 1–7. DOI: 10.1016/j.jad.2022.06.064.
- [Hun+18] Melissa G. Hunt et al. “No More FOMO: Limiting Social Media Decreases Loneliness and Depression”. In: *Journal of Social and Clinical Psychology* 37.10 (2018), pp. 751–768. DOI: 10.1521/jscp.2018.37.10.751.
- [Jam+16] Muhammad Abid Jamil et al. “Software Testing Techniques: A Literature Review”. In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. 2016, pp. 177–182. DOI: 10.1109/ICT4M.2016.045.
- [Job13] William Jobe. “Native Apps Vs. Mobile Web Apps”. In: *International Journal of Interactive Mobile Technologies (iJIM)* 7.4 (2013), pp. 27–32. DOI: 10.3991/ijim.v7i4.3226.
- [Lee+18] Jiyeon Lee et al. “Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1731–1746. DOI: 10.1145/3243734.3243867. URL: <https://doi.org/10.1145/3243734.3243867>.
- [Lig06] Antoni Ligeza. “Basic Structure of Rule-Based Systems”. In: *Logical Foundations for Rule-Based Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 91–96. DOI: 10.1007/3-540-32446-1_6. URL: https://doi.org/10.1007/3-540-32446-1_6.
- [Lin+15] Oliver Lindhiem et al. “Mobile technology boosts the effectiveness of psychotherapy and behavioral interventions: a meta-analysis”. In: *Behavior modification* 39.6 (2015), pp. 785–804. DOI: 10.1177/0145445515595198.
- [Mac+22] Allison H. MacNeil et al. “Food and Mood: Daily Associations Between Missed Meals and Affect Among Early Adolescents”. In: *Journal of clinical child and adolescent psychology : the official journal for the Society of Clinical Child and Adolescent Psychology, American Psychological Association, Division 53* (2022), pp. 1–8. DOI: 10.1080/15374416.2022.2096045.

- [MSS17] K. C. Madhav, Shardulendra Prasad Sherchand, and Samendra Sherchan. “Association between screen time and depression among US adults”. In: *Preventive medicine reports* 8 (2017), pp. 67–71. DOI: 10.1016/j.pmedr.2017.08.005.
- [MBG18] Tim A. Majchrzak, Andreas Biørn-Hansen, and Tor-Morten Grønli. “Progressive Web Apps: the Definite Approach to Cross-Platform Development?” In: *Proceedings of the 51st Hawaii International Conference on System Sciences*. Ed. by Tung Bui. Proceedings of the Annual Hawaii International Conference on System Sciences. Hawaii International Conference on System Sciences, 2018. DOI: 10.24251/HICSS.2018.718.
- [Mal+17] Ivano Malavolta et al. “Assessing the Impact of Service Workers on the Energy Efficiency of Progressive Web Apps”. In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2017, pp. 35–45. DOI: 10.1109/MOBILESoft.2017.7.
- [MD11] Mark Matthews and Gavin Doherty. “In the Mood: Engaging Teenagers in Psychotherapy Using Mobile Phones”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, pp. 2947–2956. DOI: 10.1145/1978942.1979379. URL: <https://doi.org/10.1145/1978942.1979379>.
- [Mat+08] Mark Matthews et al. “Mobile phone mood charting for adolescents”. In: *British Journal of Guidance & Counselling* 36.2 (2008), pp. 113–129. DOI: 10.1080/03069880801926400.
- [Mic18] Microsoft News Center. *Microsoft to acquire GitHub for \$7.5 billion*. 4.06.2018. URL: <https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/> (visited on 09/29/2023).
- [MS19] Ian Miell and Aidan Hobson Sayers. *Docker in practice*. Second edition. Shelter Island, New York: Manning Publications, 2019.
- [Mor+22] Mehrab Bin Morshed et al. “Food, Mood, Context: Examining College Students’ Eating Context and Mental Well-being”. In: *ACM Transactions on Computing for Healthcare* 3.4 (2022), pp. 1–26. DOI: 10.1145/3533390.
- [Mos+21] Isaac Moshe et al. “Digital interventions for the treatment of depression: A meta-analytic review”. In: *Psychological bulletin* 147.8 (2021), pp. 749–786. DOI: 10.1037/bul0000334.
- [Moz23a] Mozilla Foundation. *ServiceWorker*. 2023. URL: <https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker> (visited on 10/04/2023).
- [Moz23b] Mozilla Foundation. *Web app manifests*. 2023. URL: <https://developer.mozilla.org/en-US/docs/Web/Manifest> (visited on 10/04/2023).
- [Moz23c] Mozilla Foundation. *Window: beforeinstallprompt event*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/beforeinstallprompt_event (visited on 10/04/2023).

- [Nat08] National Research Council. *The National Children's Study Research Plan: A Review*. Washington (DC), 2008. DOI: 10.17226/12211.
- [Neu+22] Jonathan Neugebauer et al. "A Medical Information System for Personalized Rehabilitation after Ankle Inversion Trauma". In: *Proceedings of the 17th International Conference on Software Technologies*. Vol. 1. SCITEPRESS - Science and Technology Publications, 7/11/2022 - 7/13/2022, pp. 319–330. DOI: 10.5220/0011295800003266.
- [New+13] RL Newman et al. "Wilber 3: A python-django web application for acquiring large-scale event-oriented seismic data". In: *Agu fall meeting abstracts*. Vol. 2013. 2013, IN51B–1543.
- [OG15] Addy Osmani and Matt Gaunt. *Instant Loading Web Apps with an Application Shell Architecture*. 2015. URL: <https://developer.chrome.com/blog/app-shell/> (visited on 09/30/2023).
- [Oul94] Martyn A. Ould. *Testing in software development*. Repr. with minor corr. British Computer Society monographs in informatics. Cambridge: Cambridge Univ. Press, 1994.
- [Ove23] Stack Overflow. *Stack Overflow Developer Survey 2023*. 2023. URL: <https://survey.stackoverflow.co/2023/#most-popular-technologies-webframe> (visited on 09/30/2023).
- [PBP88] Jacqueline B. Persons, David D. Burns, and Jeffrey M. Perloff. "Predictors of dropout and outcome in cognitive therapy for depression in a private practice setting". In: *Cognitive Therapy and Research* 12.6 (1988), pp. 557–575. DOI: 10.1007/BF01205010.
- [Rus15] Russell, Alex. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. 2015. URL: <https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/> (visited on 10/01/2023).
- [SFV19] AS Saabith, MMM Fareez, and T Vinothraj. "Python current trend applications-an overview". In: *International Journal of Advance Engineering and Research Development* 6.10 (2019).
- [SCD06] D.M. Selfa, M. Carrillo, and M. Del Rocio Boone. "A Database and Web Application Based on MVC Architecture". In: *16th International Conference on Electronics, Communications and Computers (CONIELECOMP'06)*. 2006, pp. 48–48. DOI: 10.1109/CONIELECOMP.2006.6.
- [Sel+13] Kirusnapillai Selvarajah et al. "Native Apps versus Web Apps: Which Is Best for Healthcare Applications?" In: *Human-Computer Interaction. Applications and Services*. Ed. by Masaaki Kurosu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 189–196.
- [SHG13] Nicolas Serrano, Josune Hernantes, and Gorka Gallardo. "Mobile Web Apps". In: *IEEE Software* 30.5 (2013), pp. 22–27. DOI: 10.1109/MS.2013.111.
- [She15] Esther Shein. "Python for beginners". In: *Communications of the ACM* 58.3 (2015), pp. 19–21. DOI: 10.1145/2716560.
- [She17] Dennis Sheppard. *Beginning Progressive Web App Development*. Berkeley, CA: Apress, 2017. DOI: 10.1007/978-1-4842-3090-9.

- [SNW22] Shefaly Shorey, Esperanza Debby Ng, and Celine H. J. Wong. “Global prevalence of depression and elevated depressive symptoms among adolescents: A systematic review and meta-analysis”. In: *The British journal of clinical psychology* 61.2 (2022), pp. 287–305. DOI: 10.1111/bjc.12333.
- [Sri17] KR Srinath. “Python—the fastest growing programming language”. In: *International Research Journal of Engineering and Technology* 4.12 (2017), pp. 354–357.
- [TK17] Wei Tang and David Kreindler. “Supporting Homework Compliance in Cognitive Behavioural Therapy: Essential Features of Mobile Apps”. In: *JMIR mental health* 4.2 (2017), e20. DOI: 10.2196/mental.5283.
- [Ter+18] Yannik Terhorst et al. “«Hilfe aus dem App-Store?»: Eine systematische Übersichtsarbeit und Evaluation von Apps zur Anwendung bei Depressionen”. In: *Verhaltenstherapie* 28.2 (2018), pp. 101–112. DOI: 10.1159/000481692.
- [V+22] Puneet V et al. “A Django Web Application to Promote Local Service Providers”. In: *2022 6th International Conference on Computing Methodologies and Communication (ICCMC)*. 2022, pp. 1517–1521. DOI: 10.1109/ICCMC53470.2022.9754099.
- [Vas+15] Bogdan Vasilescu et al. “Quality and Productivity Outcomes Relating to Continuous Integration in GitHub”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 805–816. DOI: 10.1145/2786805.2786850. URL: <https://doi.org/10.1145/2786805.2786850>.
- [VK08] Iwan Vosloo and Derrick G. Kourie. “Server-centric Web frameworks”. In: *ACM Computing Surveys* 40.2 (2008), pp. 1–33. DOI: 10.1145/1348246.1348247.
- [W3C22a] W3C Groups. *Push API*. 2022. URL: <https://www.w3.org/TR/push-api/> (visited on 10/02/2023).
- [W3C22b] W3C Groups. *Service Workers*. 2022. URL: <https://www.w3.org/TR/service-workers/> (visited on 10/01/2023).
- [W3C23] W3C Groups. *Web Application Manifest*. 2023. URL: <https://www.w3.org/TR/appmanifest/> (visited on 10/01/2023).
- [WHO22] WHO. *Physical activity*. 2022. URL: <https://www.who.int/news-room/fact-sheets/detail/physical-activity> (visited on 10/03/2023).
- [Win+20] P. Winkler et al. “Increase in prevalence of current mental disorders in the context of COVID-19: analysis of repeated nationwide cross-sectional surveys”. In: *Epidemiology and psychiatric sciences* 29 (2020), e173. DOI: 10.1017/S2045796020000888.
- [Wor17] World Health Organization. *Depression and other common mental disorders: global health estimates: Technical documents*. 2017. URL: <https://iris.who.int/bitstream/handle/10665/254610/WHO-MSD-MER-2017.2-eng.pdf?sequence=1> (visited on 10/07/2023).

- [Ziv+09] Kara Zivin et al. “Persistence of mental health problems and needs in a college student population”. In: *Journal of affective disorders* 117.3 (2009), pp. 180–185. doi: 10.1016/j.jad.2009.01.001.

Declaration of Academic Integrity

I hereby confirm that this thesis on the "Developing a Digital Intervention in the Context of Depressive Symptoms: The Mood Diary Application" is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited. I am aware that plagiarism is considered an act of deception which can result in sanctions in accordance with the examination regulations.

(date, signature of student)

I consent to having my thesis cross-checked with other texts to identify possible similarities and to having it stored in a database for this purpose.

I confirm that I have not submitted the following thesis in part or whole as an examination paper before.

(date, signature of student)