ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering - DISI
Two year Master degree in Computer Engineering

NAD-APP: Analysis and Development of a Telemedicine System for Patients Suffering from Intestinal Failure

Supervisor:
Prof. Paolo Bellavista
Correlator:
Prof. Loris Pironi

Author:
Federico Terzi

Session IV
Academic year 2019-2020

Abstract

The recent technological advancements are bringing substantial improvements in several different fields, with an important one being medicine. Specifically, the use of telemedicine systems enables doctors to perform a wide variety of procedures remotely, reducing the number of hospital visits for patients, and therefore, increasing the overall efficiency. Remote diagnosis and triage became even more important during the recent COVID-19 global pandemic, in which limiting the chances of exposure has been a top priority. In our work, we present a novel telemedicine system designed to support patients suffering from intestinal failure, a condition that requires constant monitoring to guarantee the well-being of patients. In particular, the goal is to replace the traditional, paper-based monitoring techniques with an efficient digital system, which allows patients to conveniently record the monitoring data on their smartphones, and doctors to receive it on a dedicated web-application.

Contents

Introduction

The recent technological advancements are bringing substantial improvements in several different fields, with an important one being medicine. Specifically, the use of telemedicine systems enables doctors to perform a wide variety of procedures remotely, reducing the number of hospital visits for patients, and therefore, increasing the overall efficiency. Remote diagnosis and triage became even more important during the recent COVID-19 global pandemic, in which limiting the chances of exposure has been a top priority.

In our work, we focus on a specific subset of telemedicine known as *telemonitoring*, which refers to the practice of sending physiological data, such as heart frequency, and symptom scores directly to care providers through digital means [1]. Specifically, we present a novel telemedicine system designed to support patients suffering from intestinal failure, a condition that requires constant monitoring to guarantee the well-being of patients. The goal is to replace the traditional, paper-based monitoring techniques with an efficient digital system, which allows patients to conveniently record the monitoring data on their smartphones and doctors to receive it on a dedicated web-application. By reducing the friction of the monitoring process, the system allows patients to receive medical consultations and alerts in a more timely matter, eventually improving their quality of life.

From a technical perspective, the main objective is to create a reliable system, with a strong focus on resilience and security. In particular, several cloud-native paradigms are being employed to provide a system with good horizontal scalability and redundancy properties. When appropriate, cryptography is also being used as a mean to protect data confidentiality and to support authentication policies. At the lowest level of our technological stack, we employ Node.js, a popular web server engine, that is particularly well suited for cloud-native systems. As part of the our research towards reliability, we also take into consideration the context in which the mobile application is being used. In particular, patients might want to record monitoring data while offline, requiring a custom synchronization protocol capable of handling the edge cases occurring in such distributed systems. Another

challenge is to support both Android and iOS platforms, a requirement for which the use of a cross-platform mobile development framework, in this case Google's Flutter, is deemed necessary to limit the implementation cost. Finally, due to regulatory constraints, our work needs to support SPID, the Italian's authentication system, whose integration presents a number of challenges, especially in the mobile scenario, which are thoroughly discussed in our work.

The proposed work is structured in four chapters. In the former, a broad overview of telemedicine systems is presented, starting from the ancient methods and leading to some of the latest, state-of-the art techniques. Cloud computing is the subject of a significant portion of the first chapter, with discussions on its advantages, privacy concerns and possible architectures. Thereafter, a brief section is dedicated to state-of-the-art techniques such as edge computing and federated learning. Finally, the ideas and motivations behind NAD-APP are presented, providing the foundations for our work. The second chapter is dedicated to the technologies employed in our work, ranging from the mobile frameworks to the back-end and data storage solutions. For each of them, the motivations and main concepts are presented. In the third chapter, we discuss a number of crucial architectural choices related to our work, such as the database schema, the synchronization protocol used by the mobile app and the authentication strategies. These concepts are presented from a high-level perspective, delegating the actual implementation details to the last chapter. In particular, the fourth chapter is divided into two main sections. The first describes the three main components of our system, that are the server, the mobile app and the web application respectively. The second discusses the load testing process, used to verify whether our work is capable of withstanding the required load, and the achieved results, along with possible future improvements.

Chapter 1

Problem presentation and requirements definition

1.1 Telemedicine Systems

Telemedicine can be broadly defined as the use of telecommunication technologies to provide medical information and services [2]. As such, it covers several medical activities including prevention of diseases, treatment, diagnosis, patient education, and monitoring [3], all executed remotely through the use of technology.

The handling of the recent Covid-19 pandemic required a number of paradigm shifts in healthcare systems, particularly in areas such as early diagnosis and prevention. Previous work already highlighted the potential of telemedicine in response to disasters and public health emergencies [4], with a recent example being *forward triage*, the sorting of patients before they arrive in the emergency department [5], allowing a more efficient screening as well as minimizing the risk of exposure for patients, clinicians and the general community. Furthermore, telemedicine systems are particularly helpful for routine monitoring procedures, a necessary step to guarantee the well-being of patients suffering from long-term diseases, which could often be accomplished remotely to avoid exposures.

1.1.1 History

Despite being mostly considered a byproduct of recent years' advancements in information technologies, telemedicine can be dated to ancient times. The term itself is composed of the prefix *tele*, derived from the Greek term for "at a dis-

tance", and *medicine*. As a result, we can consider telemedicine all transmissions of medical information and practices that do not require the co-presence of patients and doctors in the same place.

A first example could be dated back to the Middle Ages when information about the bubonic plague was transmitted through signal fires, creating a rudimentary public health surveillance network. Thereafter, the development of telegraphy in the 19th century made it possible to better plan medical care and drug supply in public emergencies, such as the American Civil War.

The telephone then established itself as the de facto standard for delivering health services throughout the late 19th and 20th centuries. Besides voice conversations, it was later realized that telephones could also be used to send non-vocal communications, such as the amplified signal of a stethoscope or electrocardiograms (ECGs) [3].

Further development of telemedicine was seen after the discovery and widespread adoption of radio communication. One notable example was the use of telemedicine to provide remote medical assistance to sailors, with the most important center being the International Radio Medical Centre (CIRM), founded in Rome in 1935 and serving over 42,000 patients within its first 60 years of activity [6].

1.1.2 Modern telemedicine

Television marked the beginning of what is considered the advent of modern telemedicine [3]. In particular, by the late 1950s, a number of closed-circuit television and video communications were established to accomplish a variety of medical-related tasks. Notably, in 1964 a two-way closed-circuit television system was set up between the Nebraska Psychiatric Institute and the Norfolk State Mental Hospital, which enabled remote psychotherapy sessions. This link, which connected the two institutions at 180 km farther apart, is by many considered the first telemedicine program in history [7].

As information technologies further developed, telemedicine gained noticeable interest after the introduction and widespread adoption of the internet. The increased number of connected people, combined with the decreasing costs of communication, opened up new opportunities for remote medical applications. In particular, the transferred information could be as simple as a doctor providing consultation to sophisticated data captured from a patient body [8].

Besides the convenience of receiving medical advice from the comfort of pa-

tients' homes instead of hospitals, telemedicine could also be critical in rural areas or emergency situations, where proper medical care is not available. Notable examples include the establishment of a telemedicine clinic at Mt. Everest Base Camp (EBC) in 1998 to assist and track climbers' vital signs and position, providing immediate rescue if needed and avoiding outcomes such as the tragedy of 1996, where 5 climbers died during a storm [9].

Another area currently undergoing intense growth is *mobile telemedicine* [10]. These systems combine the concepts already explored in traditional telemedicine applications with commodity mobile devices, which are ubiquitous in modern societies. Consequently, the resulting systems benefit from the strengths of the mobile paradigm being portable, as the target devices can be easily carried in person or even worn, and cost-effective, as the majority of the population already owns the required hardware.

1.1.3 Types of Telemedicine

As previously mentioned, telemedicine is about transmitting medical information using network and telecommunication technologies. This information can take many shapes and serve a number of different purposes, as shown in Fig 1.1, ranging from a simple telephone call between patient and doctor delivering medical advice to sophisticated real-time remote surgery procedures using robots.

Among them, *telemonitoring* already proved itself effective in a number of real-world applications, as well as being the primary focus of our work. Specifically, home telemonitoring represents a viable way to remotely monitor patients suffering from long-term diseases such as diabetes, hypertension, cardiac and gastrointestinal diseases [11]. These illnesses require constant monitoring to guarantee the patients' well-being. Moreover, these systems might combine patient-generated data with medical sensor readings, providing a detailed overview of a patient's current health status. As a result, the clinical effectiveness increases, optimizing hospital admissions, and emergency visits.

1.1.4 Risks

Despite the numerous benefits introduced, telemedicine is not exempt from ethical and legal concerns, as changing the doctor-patient relationship is a delicate topic. The increasing number of medical-related apps have raised debates regarding the legal implications and security requirements of telemedicine systems, in particular, focusing on identifying the accountable entities. Although developers are generally considered most responsible for the quality and safety of telemedicine applications,

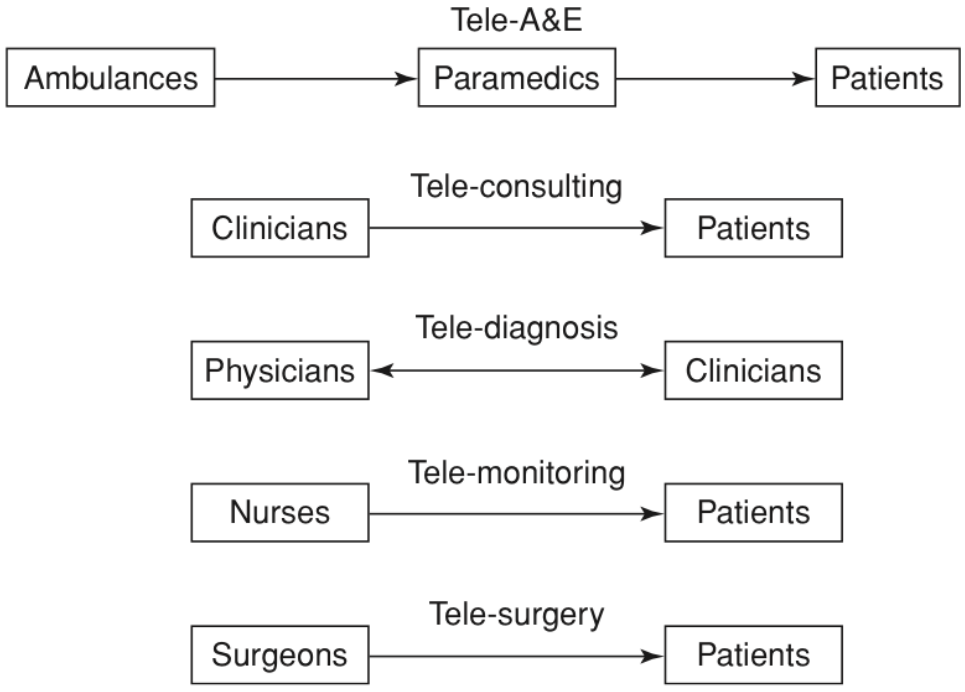


Figure 1.1: Subsets of telemedicine. Source: *Telemedicine Technologies* [8]

also physicians can be held liable for malpractice, even if performing remotely [12].

A major area of discussion has been data protection and confidentiality, especially after the introduction of the General Data Protection Regulation (GDPR), which further classifies privacy requirements and procedures. A great amount of care must be taken when handling patients' data, in particular when transmitting it to others or storing it for later use. Moreover, maximum protection must be provided when dealing with sensitive data regarding minors or genetic material. The lack of a common standard gives rise to a variety of heterogeneous solutions, causing an increased likelihood of misuse, as no procedure is extensively validated [13].

From a purely technical standpoint, telemedicine systems can be classified as IT applications. Consequently, they also inherit all the risks related to computer systems, making them vulnerable to man-made attacks as malware and phishing, as well as natural disasters such as storms, fires, and flooding. Accounting for these events is crucial to build secure telemedicine systems, and generally, it includes both technical and organizational measures to mitigate risks [8].

1.2 Cloud Computing for Telemedicine

The world population is aging, and as a result, medical expenditures are growing across most Western countries [14]. Therefore, providing cost-effective healthcare solutions is now a central topic of discussion, with telemedicine being a key area of research. Traditional systems tend to be expensive to set up and maintain, hence a number of organizations are moving to *Cloud Computing* solutions, drastically reducing costs. With that being said, its adoption should be handled with care, as the new paradigm poses significant concerns about the privacy and security of patients' data [15][16].

1.2.1 Economic impact

Nowadays, healthcare organizations are introducing information technologies into their procedures, with many transitioning from paper-based to electronic health records (EHRs) [17]. EHRs offer a number of advantages over their paper-based counterparts, including improved efficiency and lower costs [15].

Traditional systems are generally implemented following the client-server architecture, requiring the installation of a dedicated server either in the doctor's office or hospital's data center depending on the organization size. These solutions tend to be significantly expensive, both in terms of money and expertise required for maintenance, making them difficult to adopt by small-sized organizations [15]. As a reference, previous work estimated the cost of implementing an EHR system to be around $162,000 for an average five-physician practice [18].

Cloud computing offers several advantages compared to traditional methods in terms of scalability, maintainability, and cost-effectiveness. Consequently, many commercial EHR vendors and start-up companies are experimenting with cloud technologies to build telemedicine services. These cloud-based systems are usually characterized by Web applications that let individual users access and share medical information according to their authorization levels [15]. Most importantly, cloud-based solutions drastically reduce initial investment and maintenance costs [16], making the technology accessible to small healthcare providers.

1.2.2 Applications

Cloud-based telemedicine systems can serve a number of different purposes, as illustrated in Fig. 1.2. Due to the way these applications are designed, the creation of a shared infrastructure is facilitated, allowing all stakeholders to be connected and take advantage of the system in various ways.

Along with the advancements in global infocommunicational infrastructures, cloud computing enable the creation of widely interconnected medical systems, easing interoperability with legacy applications and new technologies such as smart IoT health devices and smartphone-apps [19].

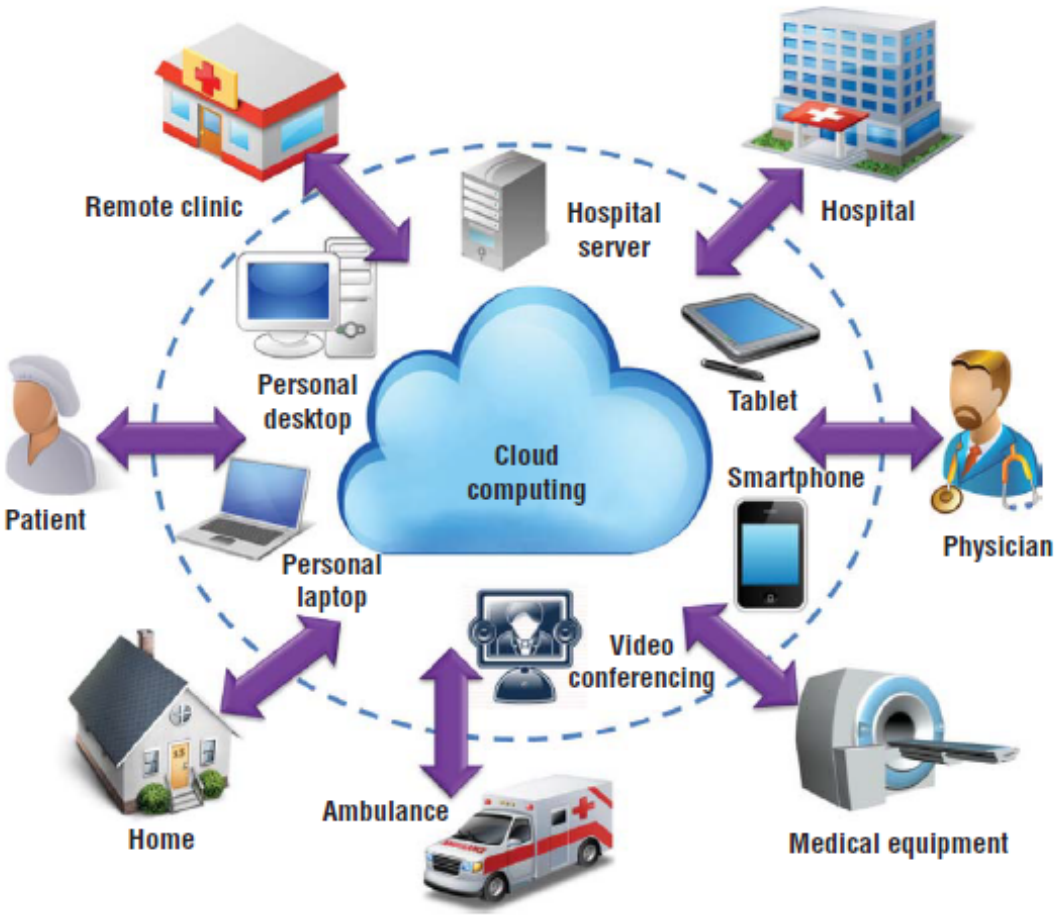


Figure 1.2: Various telemedicine applications based on cloud computing. Source: *Telemedicine in the Cloud Era* [15]

1.2.3 Privacy concerns

Privacy is considered a crucial requirement for eHealth systems [20]. Given the nature of cloud-based solutions, ensuring high privacy levels is more challenging compared to traditional systems as they lack the physical control of servers' hardware. In particular, by storing and processing patient data on cloud services the system becomes more vulnerable to unauthorized entities accessing the data.

Convenience is often considered another important requirement. For instance, letting patients access medical records remotely can be a substantial advantage as hospitals are freed of the additional burden. However, system designers must take into account the chain of platforms and networks personal data will go through before reaching patients, which poses significant risks from a privacy standpoint [15]. More generally, it becomes a problem of balancing usability and privacy, which is known to be inherently difficult [21].

These risks can be partially mitigated with a *Privacy Level Agreement* (PLA), a contract containing the security requirements the cloud provider undertakes to satisfy. For instance, these agreements might describe specific data protection measures, geographical data transfer constraints and data retention policies [16].

Another approach to better comply with privacy regulations is to employ cloud technologies in a hybrid manner, which is later described in 1.2.6.

1.2.4 Security risks

As already discussed, cloud technologies pose a number of risks related to privacy and security [22].

Loss of governance

For organizations, choosing cloud technologies instead of in-house solutions implies giving away some control to the cloud service providers. Consequently, a number of security measures cannot be directly decided by organizations and are delegated to the provider instead. However, these measures are usually impossible to formally verify by the organization [23], which might pose several regulatory problems.

Additionally, loss of governance often comes with a great risk of *vendor lock-in*, that is, the risk of losing the freedom to replace the cloud provider with another, due to the use of proprietary, or non-standard, formats [16].

Management interface compromise

Cloud services are managed by web interfaces that are accessible through the Internet. As a result, these solutions are exposed to a greater risk of attack compared to traditional in-house systems, as the malicious entity doesn't need to be physically located within the servers' buildings [22].

Isolation failure

Cloud service providers are capable of offering competitive prices as they leverage on two principles: economies of scale and global reach [24]. Thanks to the former, providers are able to buy substantial hardware quantities at lower costs. These computing resources are then shared among clients, with each of them having an isolated portion. Consequently, providers can optimize costs by allocating their fixed pool of resources to the varying demand of customers. This isolation is made possible by several virtualization technologies that, even if relatively secure and thoroughly tested, might be vulnerable to a number of attacks, either due to software or hardware bugs, that allow attackers to bypass isolation and access sensitive data [22].

Insecure data deletion

After issuing a deletion request, ensuring the resource has been properly removed, and thus made irrecoverable, is crucial to comply with *data retention* regulations. Unfortunately, this might be challenging due to the very nature of cloud computing, in which *multi-tenancy* and virtualization are employed, increasing the risk of unauthorized users accessing the deleted resources. Furthermore, cloud service providers might not delete customer data due to hidden business interests [25].

1.2.5 Security advantages

Despite the many challenges, cloud computing also offers a number of security advantages compared to traditional solutions. Firstly, given the amount of economic resources available, cloud service providers are able to provide better security measures compared to traditional systems. Substantial investments are made to improve defensive procedures such as filtering, patch management, hardening of virtual machines, redundancy, and strong authentication. Moreover, providers are able to hire a number of specialists to ensure high-security standards, whereas smaller organizations can only afford fewer generalists, thus failing to provide the same level of protection [22]. Cloud architectures are also particularly resilient to natural disasters and denial of service attacks, due to their built-in distributed and redundant nature.

1.2.6 Architectures

Telemedicine systems can adopt a number of different architectures, with varying degrees of reliability, security and convenience.

Traditional architectures

Traditional systems tend to follow the classical client-server architecture, in which a central server is hosted inside the organization, with users connecting from either the local network or, less frequently, the outside using the internet, as illustrated in Fig. 1.3.

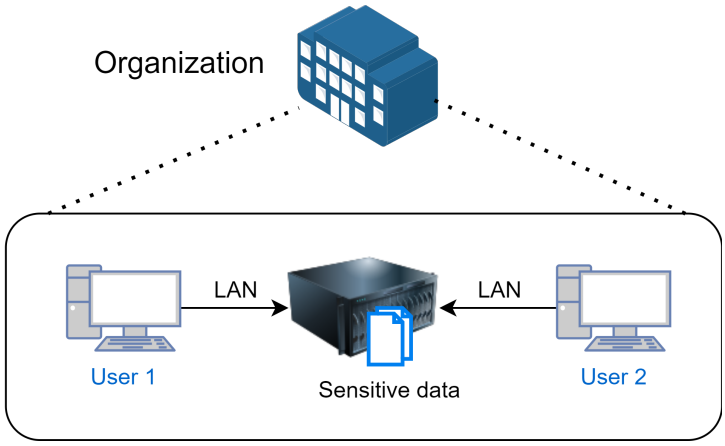


Figure 1.3: A traditional telemedicine architecture.

These solutions offer a number of advantages, such as complete control over sensitive data and physical access control policies. In particular, as data is physically stored on hardware located inside the organization, there is no need to trust an external entity.

On the other hand, these systems tend to be considerably expensive [15], making them difficult to adopt by smaller organizations. Being mission critical systems, they require solid disaster recovery procedures and availability guarantees, usually obtained by regular off-site backups and redundant hardware, which are notoriously costly to obtain. Moreover, if the system needs to be accessed remotely, for example through the internet, other factors must be taken into account, such as the distance from end-users and scalability requirements.

Cloud architectures

Compared to traditional systems, cloud-based architectures offer a completely opposite set of properties. Firstly, applications run on the cloud service provider's datacenters instead of in-house servers, as illustrated in Fig. 1.4. As a result, initial investment costs are drastically reduced, but organizations give away control over data, at least partially. Secondly, these services usually consist of web applications that can only be accessed through the internet, which makes them particularly well suited for distributed use-cases, but also potentially more vulnerable to external malicious attacks. Most importantly, cloud-based solutions greatly facilitate the achievement of higher availability and reliability standards, as geographical and physical redundancy is built-in into the paradigm.

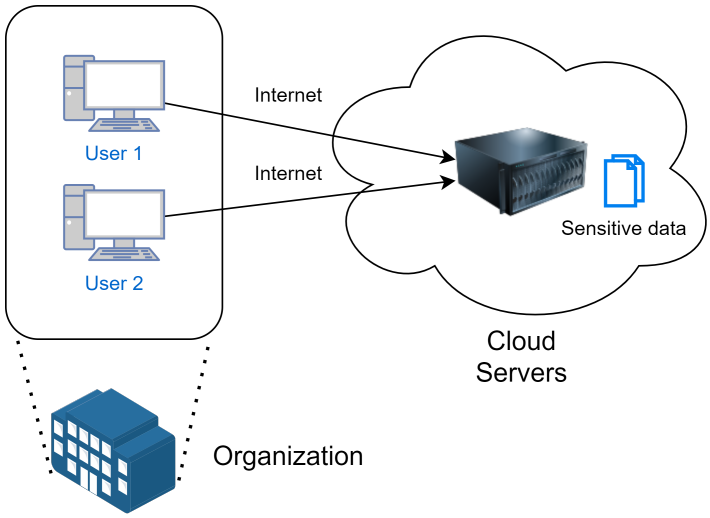


Figure 1.4: A possible cloud architecture.

As control is crucial when dealing with sensitive medical data, additional security measures must be adopted when choosing cloud-based solutions. The first candidate is usually encryption, which enables systems to store data on cloud servers in a way that is not readable by providers. For instance, clients could be given a key by the system administrators and then use it to encrypt sensitive data sent to the server and decrypt data received from them. Under this model, the provider always manipulates opaque blobs of encrypted data, only readable by the clients. Despite offering less control compared to the traditional, in-house hosting, this solution could provide an appropriate level of security for many medical applications.

Hybrid architectures

An alternative approach is to combine cloud technologies with traditional in-house hosting, giving rise to hybrid systems. These are well suited for federated scenarios, in which a number of independent medical organizations need to create a unified system to share sensitive data. The Italian's *Fascicolo Sanitario Elettronico* is an example of such system [16], in which individual AUSLs are responsible for a limited number of local patients and share data between territories through a federated network.

Hybrid systems enable organizations to retain control over sensitive medical data while exploiting cloud-technologies to improve the general performance or capabilities of the system. A possible model is shown in Fig. 1.5, in which medical records are stored by individual organizations, thus retaining control over the most sensitive information, whereas cloud technologies are used to build a registry of the individual resources. This registry is queried by users as part of the record search process, but instead of returning the record itself, it only returns the report location within the parent organization. This information is then used by the user to obtain the actual record from the organization's in-house servers.

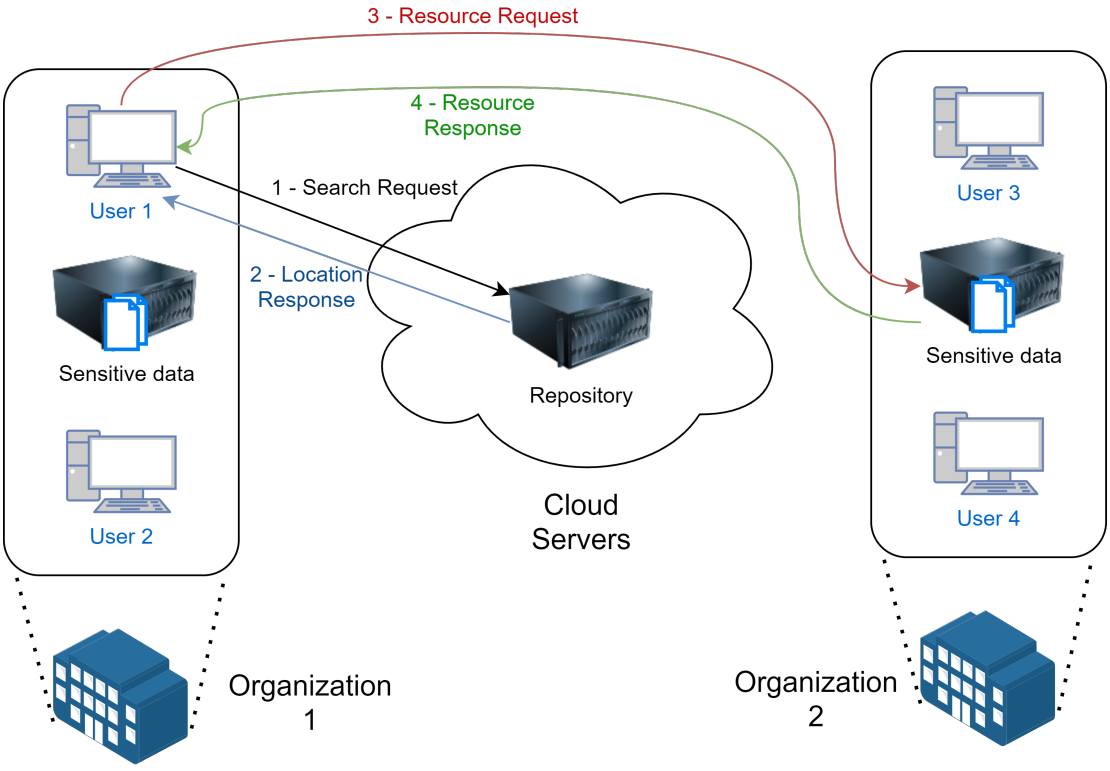


Figure 1.5: A possible hybrid architecture.

A significant advantage offered by this model is the limited amount of sensitive information that needs to be stored on cloud servers, only restricted to the fields on which the search operation is enabled. Organizations retain full control of the sensitive data they are accountable for. With that being said, this model doesn't remove the need for redundancy and reliability on in-house servers, making the system considerably more expensive than a cloud-only solution. Moreover, as sensitive data is stored on the same in-house servers, the solution is likely less resilient to natural disasters than cloud solutions, as geographical redundancy might be overly burdensome to obtain. Finally, the organization's servers are more challenging to scale, becoming a bottleneck as the number of clients increases.

1.3 Emerging Technologies for Telemedicine

In recent years, the increasing concerns regarding data protection and privacy, especially in cloud-based environments, gave rise to a number of technologies that could be significantly beneficial to telemedicine systems. These are subjects of intense research, and in the upcoming years they could completely replace, or at least partially integrate, current solutions to mitigate many of the previously mentioned data protection threats. In the following sections, two of them are presented.

1.3.1 Edge and Fog computing

With the widespread adoption of IoT technologies, huge amounts of data are being produced. Cloud computing is being used to effectively process huge quantities of it, but despite the increasing computational resources available, the internet bandwidth is not growing at the same pace, becoming a bottleneck [26]. Consequently, cloud technologies are proving to be unsuitable for applications in which the computation results should be provided in real-time, such as in autonomous driven vehicles and sensor-based medical diagnosis systems.

Given the nature of cloud computing, in which servers and sensors are intrinsically distant, a possible approach to solve the problem is to move the computation near data sources, a solution known as *edge computing*. In this case, the processor is directly connected to the sensors and must handle all computation related tasks, often including storage and possibly forwarding some data to the cloud [26]. A similar approach is known as *fog computing*, in which the computation is performed in the same local network as the sensors, thus providing an intermediate solution between edge and cloud computing.

Telemedicine systems could greatly benefit from these new paradigms. First

of all, edge computing promotes a local-first philosophy when handling data. As a result, most of the data control problems highlighted in the previous sections when discussing cloud technologies are either solved or at least substantially mitigated [27]. Moreover, the efficacy of telemedicine applications dealing with acute diseases such as heart attacks or pneumonia is directly linked with the diagnosis latency [28]. Consequently, fast edge-computing response times could substantially improve patients' life and health.

With that being said, edge computing presents its own set of challenges. Firstly, compared to cloud environments in which the infrastructure is often transparent to the user, edge nodes are characterized by heterogeneous platforms, thus adding an additional burden to software developers. Secondly, as edge nodes are often resource-constrained, a number of computational-heavy security measures cannot be deployed [29].

1.3.2 Federated learning

Over the past ten years, the use of machine learning in real-world environments has seen a dramatic increase, with strong results in several fields. FDA recently authorized the implementation of an autonomous artificial intelligence diagnostic system, a decision that could have important consequences for the healthcare sector. These systems rely on algorithms to learn how to diagnose diseases from large data sets of medical records, without being explicitly programmed [30].

Besides the numerous advantages in terms of processes, quality and costs, employing machine learning techniques in the healthcare sector raises a number of ethical questions. First and foremost, given the increasing awareness on user privacy and data security, particular care must be taken when dealing with patients' data. Traditional machine learning techniques are based on the concept of a central entity that performs the training on a single, large data set. In order to build these data sets, several different sources must be combined to reach the large volume needed for training, but as a result, the various organizations or individuals lose control over their sensitive, personal data.

A possible approach to mitigate the problem has been recently proposed by Google [31] and takes the name of *federated learning*. The goal is to build machine learning models based on several data sets distributed across multiple devices while preventing data leakage [32]. Federated learning would allow different organizations to jointly train machine learning models without compromising on safety and keeping the sensitive data private.

1.4 NAD-APP

In Italy, an estimated 500/million people are suffering from diseases that require *Home Artificial Nutrition* (HAN) to survive [33]. This condition might persist for weeks, months, or even the whole patient's life [34], requiring constant monitoring either from home or at a specialized clinic.

Located in Bologna, the IICB center serves HAN patients from all over Italy, providing medical support and remote monitoring by means of telephone and email technologies. The COVID-19 pandemic caused all in-person medical visits to being suspended, holding them remotely whenever possible. As a result, medical societies recommended HAN clinics to develop specific telemedicine systems to better support patients during these difficult times.

The work's objective is to create a software application, named NAD-APP after the Italian's translation of HAN, directed at both patients and clinics to improve the remote monitoring process.

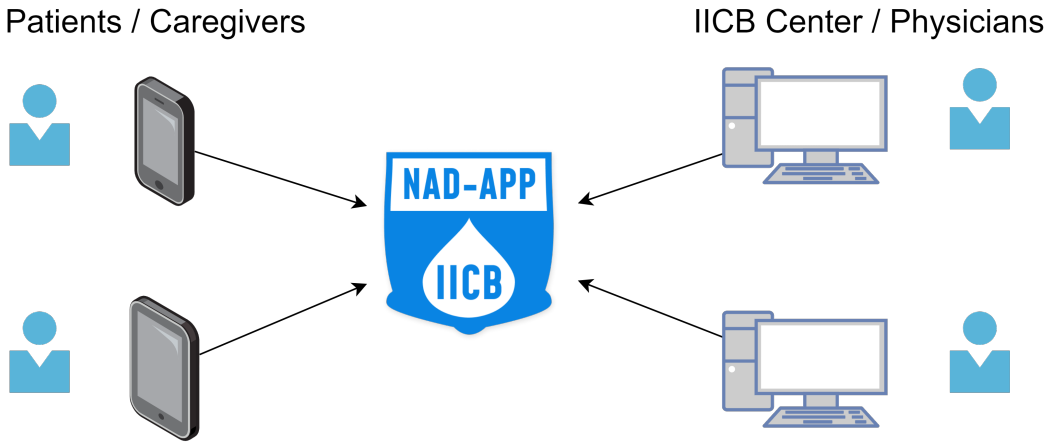


Figure 1.6: NAD-APP high-level overview.

Patients are provided with a mobile application whose aim is to replace the current reporting processes. In particular, patients, or their caregivers, are currently asked to fill paper forms with their monitoring data and then send this information to the IICB center via email. This manual process could be greatly improved by providing digital forms within the app. Then, monitoring data could be automatically sent to the clinic, without the tedious process of scanning the document and then sending it through email. The app could also be used to receive medical reports from the clinic, obtaining information about HAN, and easily access emergency contacts.

From a technical standpoint, the application must be cross-platform and work

on both smartphones and tablets. Moreover, it needs to be resilient to unstable, or even absent, network connections, as patients are not always guaranteed to be connected to the internet. Users must be able to access the services using SPID, the Italian's public digital identity system, according to recent laws [35].

The system also comprises a web application, targeted at clinics and physicians, to navigate monitoring data submitted by patients through the mobile app. Data must be presented in a way that simplifies the analysis conducted by clinics, preferably as tables that resemble traditional paper-based forms.

1.5 Summary

In this chapter, we first presented a general overview of telemedicine systems, starting from the definition and going through their history, from ancient times to modern systems, along with examples and motivations behind them.

Secondly, we introduced the types of telemedicine systems, ranging from simple medical consultation via telephone calls to advanced, remotely-driven robots used to perform surgeries. Then, the main risks of general telemedicine services were discussed.

Thereafter, we presented the limitations of traditional telemedicine systems and the advantages cloud computing can bring to the field, discussing the economic impact and possible applications. We then introduced the potential risks of cloud-based solutions, with a strong focus on privacy and security concerns. Moreover, a number of possible architecture was presented with varying degrees of security and data control levels, discussing the main trade-offs.

Finally, we introduced the work's goal: the creation of NAD-APP, a telemedicine system to support patients under HAN treatments. Firstly, we discussed the medical condition, along with the motivations behind the work. Thereafter, the specific functional and technical requirements were discussed.

Chapter 2

Employed technologies

2.1 Flutter

Nowadays, one of the challenges companies face is the need to create mobile applications to better serve their customers. Most products greatly benefit from the ability to be installed on a smartphone, as the service can be then used on-the-go and also by people without access to a computer.

In order to fulfill this objective, the next challenge becomes choosing the best technology to build such smartphone applications. As of 2020, the leading mobile operating systems are Android and iOS, with 74% and 24% worldwide market share respectively [36], as seen in Fig. 2.1. Given that both operating systems offer official SDKs to develop *native* applications for their system, a first approach would be to build the target application twice, with a separate Android and iOS implementation. This solution offers the best integration with the two platforms and often a better user experience, with the downside of a significantly higher implementation cost. As a result, this approach is generally adopted by corporations having larger development budgets.

Another popular approach is characterized by the use of *cross-platform mobile development frameworks* that allow the developers to build applications for both platforms from a single *code base*, with varying degrees of code sharing based on the chosen framework. These solutions are designed to reduce the cost of multi-platform applications by presenting a uniform abstraction layer, allowing the developer to implement the software while ignoring most of the platform-specific details, delegating them to the framework. Nevertheless, the abstraction layer might pose significant limitations depending on the target application and framework of choice, as some platform-specific features might not be available

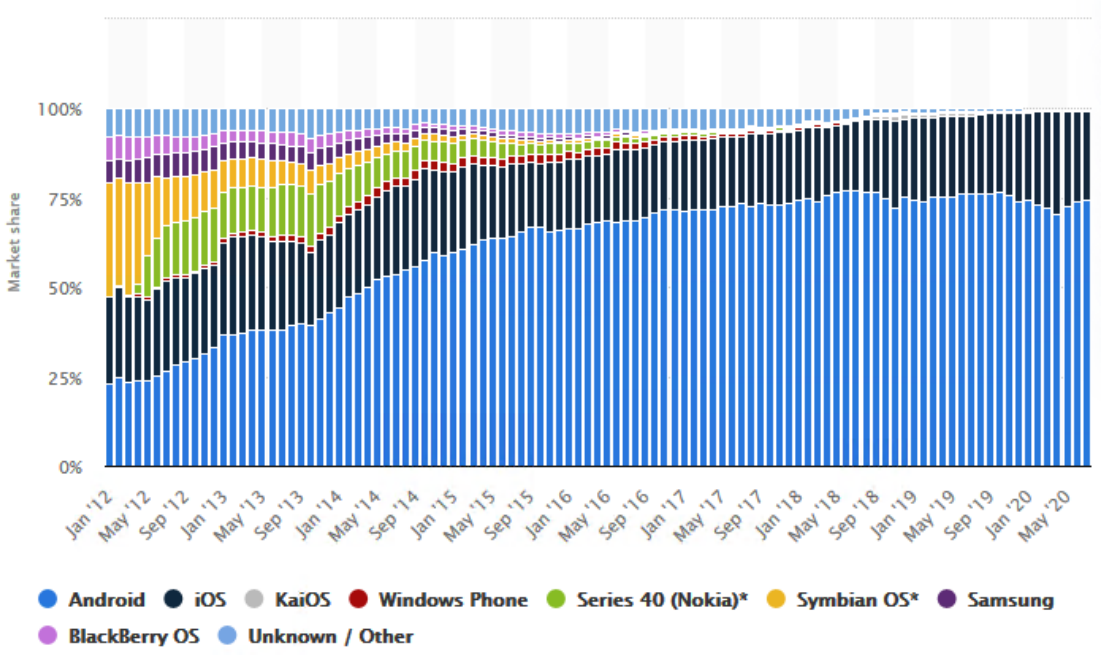


Figure 2.1: Mobile Operating Systems' Market Share. Source: Statista [36]

for the sake of homogeneity. As a result, cross-platform frameworks are generally chosen when the implementation resources are limited, or when the target application is not overly complex.

Given the size and complexity of the project, Google's *Flutter* was selected as the cross-platform application framework of choice. In the following sections, an overview of its main characteristics is presented.

2.1.1 Architecture

Flutter is a cross-platform UI toolkit whose main objective is to maximize code reuse between platforms while embracing the differences when appropriate [37]. Flutter apps are written in *Dart*, a language optimized for client applications [38], that gets later compiled directly to machine code, generally ARM instructions, when targeting mobile devices. As a result, the generated applications can be written in the same language on both platforms, maximizing code sharing, while retaining good performance.

Compared to other frameworks such as Facebook's *React Native*, Flutter doesn't use the platform's native components and instead use its own widgets, rendered by its internal engine [39]. As a result, widgets look consistent between the two

platforms, at the expense of a less native experience.

Flutter applications are structured into layers, as illustrated in Fig. 2.2. At the lowest level, the *Embedder* provides the entry-point on all supported platforms, communicating with the underlying operating system to manage user input, rendering surfaces, and the message loop. Additionally, the Embedder enables Flutter applications to be bundled into existing native apps as modules or run as stand-alone applications, in a way that is transparent to the underlying platform. Finally, there are several different Embedders for the various operating systems, each of them written in a language that fits the given platform, such as Java and C++ for Android and Objective-C for iOS.

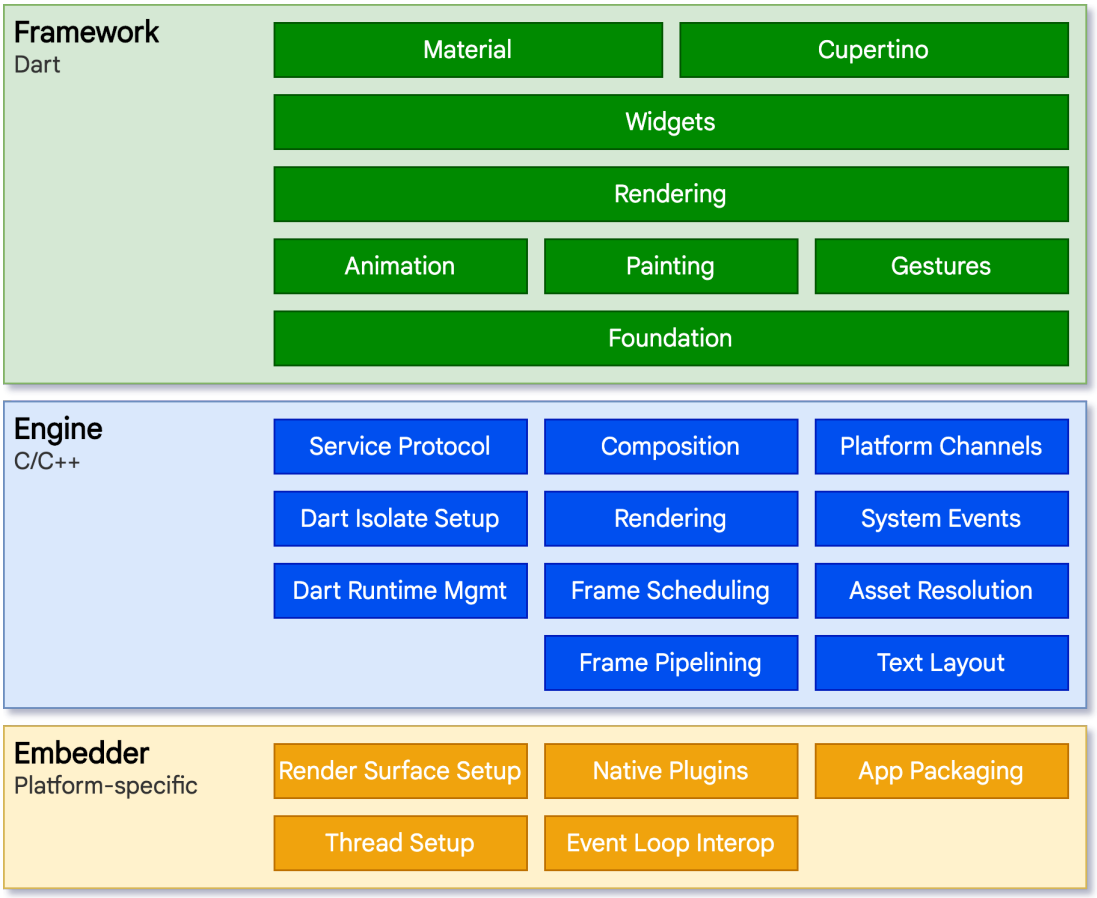


Figure 2.2: Flutter Architecture [37].

Directly above the Embedder layer, the *Engine* provides all the basic primitives necessary to support Flutter applications, such as graphics rendering, text layout, file, and network I/O, and a Dart runtime. Despite being written in C++, the Engine exposes these functions to Flutter applications through the `dart:ui`

package, wrapping them as Dart classes.

Above all, the *Framework* layer provides the high-level methods necessary to build applications, ranging from foundational aspects as animations and gestures to high-level concepts such as widgets. The framework itself is relatively small, with most features, such as camera access and HTTP capabilities, implemented as separate packages.

2.1.2 Reactive user interfaces

Flutter is a reactive, pseudo-declarative UI framework [37]. This model, inspired by Facebook's React framework, is based on the concept of UI as a function of the state:

$$UI = f(state)$$

Given the mapping function, which should be defined by application developers, the framework automatically updates the user interfaces when state changes. This paradigm, in which the UI is decoupled from its state, prevents a number of synchronization bugs that are more difficult to address with traditional approaches. As an example, figure 2.3 illustrates a user interface in which this paradigm is particularly effective. The currently selected color represents the state and could be described using 3 numbers, each encoding one of the RGB channels. Then, according to the paradigm, each UI control needs to define a mapping between the state and its appearance, as well as the way its actions modify the state. Given these definitions, the framework is capable of handling all UI updates coherently as soon as the user interacts with the interface. This seemingly easy task could raise a number of concerns when using traditional approaches, as changing the value in one control has ripple effects on all other controls, requiring some sort of explicit synchronization.

Flutter Widgets are represented by immutable classes and define the state-UI mapping functions described in the previous paragraph. As widgets are immutable, the framework has to create new instances every time the state changes, a task for which Dart is particularly well suited [37].

2.2 Redux

Redux is a pattern and library for managing application state [40], introducing a centralized store and providing methods to update it in a predictable way.

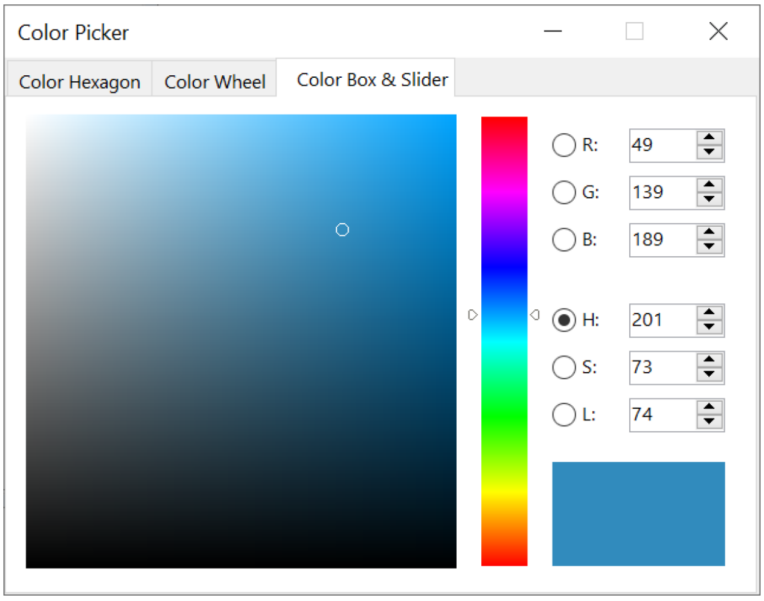


Figure 2.3: A UI example in which the reactive paradigm is particularly beneficial [37].

Although the library was initially conceived for JavaScript applications, it has been ported to many other languages and frameworks, such as Flutter, given its effectiveness in real-world scenarios.

2.2.1 Motivation

In the past decade, the complexity of client-side web applications has grown significantly [41]. Handling multiple server responses, cached results and local non-persisted data became the norm, greatly diverging from the original client-server single-page request and response paradigm. Consequently, managing the state for these applications became increasingly problematic and prone to errors.

Most modern web applications are characterized by a high degree of asynchronicity and mutation, two orthogonal concepts which are notoriously difficult to reason about when considered together [41]. The problem could be partially solved using libraries like React, which removes direct DOM manipulation and asynchronicity, though that only accounts for the view layer, leaving the general application state management to the developer.

2.2.2 Three Principles

Redux aims at simplifying state management, making it more predictable and less prone to errors. In order to achieve this objective, it leverages on three principles:

Single source of truth

The global state of an application is stored into an object tree within a single store [42]. As a result, the application is easier to debug as state is not distributed between modules, and thus there is no need for synchronization. Moreover, as the state is concentrated into a single, serializable object, application persistence potentially becomes easier. Finally, several complex operations such as Undo functionality become trivial to implement under the new paradigm.

For the sake of clarity, an example of Redux counter application is presented. In particular, the following code block represents the initial global state of the application, characterized by the current counter value.

```
1 {
2   value: 0,
3 }
```

Listing 2.1: An example of global state in a counter application.

Read-only State

State is read-only and cannot be mutated directly. Instead, *actions* should be dispatched to change the state. An action is a plain object, whose objective is to describe an event that occurred. Instead of mutating the state directly, they express an intent to transform it.

Being all actions processed serially and in strict order by a central entity there is no risk of race-condition, which is particularly useful when a number of network response and user interactions concur to mutate the state.

Further expanding the previous example, the following section shows three possible actions used to interact with the counter, respectively to increase, decrease, and reset the value. These actions are represented by plain JavaScript objects and might include any arbitrary information. In this case, the first two actions include the number to increment and decrement respectively.

```
1 store.dispatch({
2   type: 'INCREASE_COUNT',
3   n: 2,
4 })
```

```
5
6  store.dispatch({
7    type: 'DECREASE_COUNT',
8    n: 1,
9  })
10
11 store.dispatch({
12   type: 'RESET_COUNT',
13 })
```

Listing 2.2: Actions applied to the counter example.

Only pure functions change the state

The global state is only changed by *reducers*, which are pure functions receiving the current state and an action, and returning a new state. Reducers never mutate the state object directly, but instead, return a copy with the necessary changes. Being pure functions, reducers can be split as the complexity of the app increases, favoring modular designs in which each reducer is only responsible for a single aspect of the application.

Concluding the previous example, in the following section, a possible reducer is presented. When a new action is dispatched, an updated state object is returned by the reducer, as long as the latter is configured to handle the given action type. Otherwise, the previous state is returned.

```
1  function counter(state = { value: 0 }, action) {
2    switch (action.type) {
3      case 'INCREASE_COUNT':
4        return {
5          value: state.value + action.n
6        }
7      case 'DECREASE_COUNT':
8        return {
9          value: state.value - action.n
10       }
11     case 'RESET_COUNT':
12       return {
13         value: 0
14       }
15     default:
16       return state
17   }
18 }
```

Listing 2.3: A possible reducer for the previous counter example.

2.2.3 Middleware

Although actions and reducers are enough to build simple applications like counters, in order to tackle more complex scenarios we must introduce another concept:

middleware. A middleware is a function that get executed after an action is dispatched but before it reaches reducers, enabling them to intercept, block and generate new actions. As a result, they are particularly well suited for handling asynchronous side effects such as network communication. They are composable in chains and are often used to introduce logging, crash reporting, and networking capabilities into an application [43].

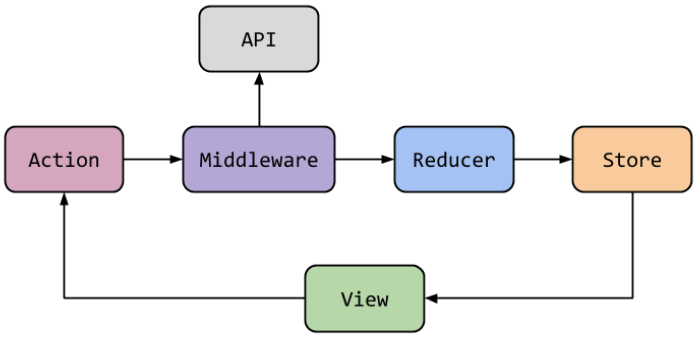


Figure 2.4: Complete Redux architecture [44]

2.3 Node.js

Node.js is an asynchronous, event-driven JavaScript runtime built on top of Google Chrome's V8 engine [45][46]. Both are implemented in C and C++, with a strong focus on low memory consumption and performance, but while V8 is conceived to support JavaScript in the browser, Node.js is designed to build scalable web services.

2.3.1 Event-driven versus Multi-threaded architectures

Traditional web servers rely on the concept of multi-threading to handle concurrent requests from multiple clients. Under this paradigm, each request is handled by a different thread, generally selected from a *thread-pool*, as illustrated in Fig. 2.5. As soon as the handler executes an I/O operation, the operating system switches the processor's execution context to another thread, a process known as *preemption*, to optimize the use of processor's time. Then, when the I/O operation is concluded, the previous thread is marked as ready, and thus might be picked up by the processor again [47].

Multi-threading offers a number of benefits, with the most important one being enabling true parallelism when using modern multi-core processors. With that

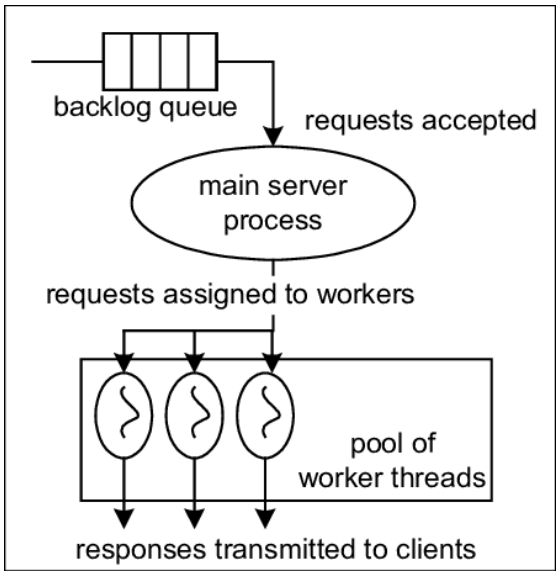


Figure 2.5: A multi-threaded web server architecture [48].

being said, there are also several disadvantages. Firstly, although the operating system is efficient at scheduling threads based on their I/O usage, context switches are expensive and might become a bottleneck as the number of concurrent clients increases. Moreover, multi-threaded architectures are often difficult to reason about, as possible data races and deadlocks must be taken into account when accessing shared resources.

Event-driven architectures offer an opposite set of properties. In the first place, event-driven applications are single-threaded, as illustrated in Fig. 2.6. Although that might raise concerns about efficient exploitation of multi-core processors, the problem is generally solved by running several single-threaded application processes side by side [47]. Secondly, the event-driven runtime relies on event notification APIs to work, which are offered by most operating systems, such as `epoll` on Linux, `kqueue` and `kevent` on BSD, and `IOCP` on Windows. Under this paradigm, applications registers for specific events and are then notified by the operating system when they occur. For instance, an application might want to be notified when there is data ready to be read from a socket, or when a write buffer is full and thus no more data should be sent.

Asynchronous I/O operations are crucial in event-driven applications, as they prevent blocking the main thread and therefore allow other requests to be served concurrently. These asynchronous operations require callbacks to correctly handle completion events, and therefore, choosing the right programming language is important. For instance, languages lacking closures and anonymous functions, such as C, are not well suited for asynchronous operations, as developers would need

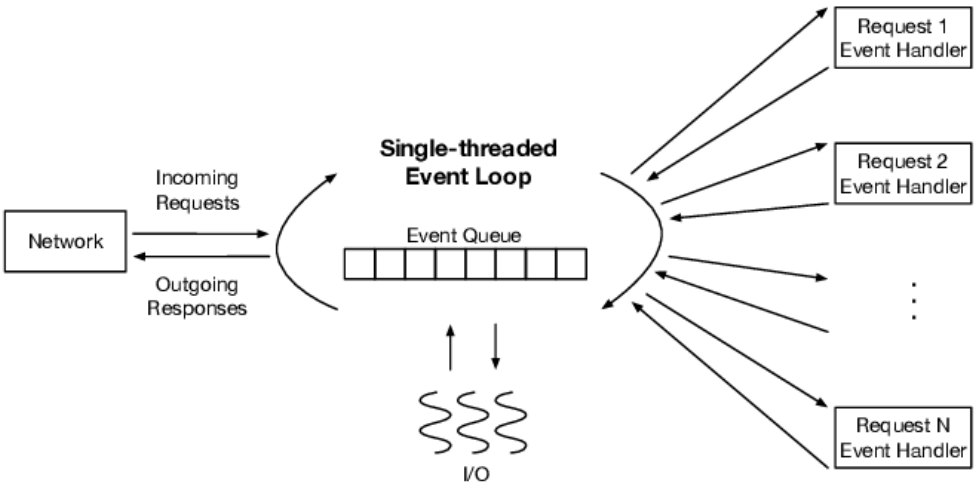


Figure 2.6: An event-driven web server architecture [49].

to manually deal with the different contexts for each callback. Due to these reasons, JavaScript is considered a good fit. Being a functional language, it natively supports closures and anonymous functions. Moreover, high-performance interpreters, such as V8, are available.

2.4 ExpressJS

Express is a fast, unopinionated, minimalist web framework for Node.js [50]. In particular, it builds on top of Node.js's functionality and adds new features, such as middleware and routing, to simplify the web development process [51].

Compared to other frameworks, Express follows a minimalist philosophy and does not enforce a rigid structure. It only provides a number of fundamental utilities that developers can use to build web applications. Consequently, using Express to implement a complete service also requires several other libraries to manage aspects such as persistency and sessions. On one hand, this approach leaves maximum flexibility to the designers, which are free to choose the libraries that better fit their requirements, but on the other, the increased number of possibilities might become an additional burden.

2.4.1 Middleware

In its purest form, Node.js offers a single handler function to manage web requests, taking a request object as an input and returning the response as an output.

Although it might be feasible to implement a complete application around this monolithic function, a better approach is to split the logic among several small handlers, each managing a single responsibility. For this exact reason, Express introduces the concept of middleware.

A middleware is a function that takes the request and response objects as parameters and mutate them according to their business logic. They are organized in a chain, so that each middleware, after executing its logic and potentially mutate the request or response, can either give control to the next middleware or interrupt the chain. They are able to handle tasks ranging from logging requests to user authorization, offering a unique modular approach.

As an example, Fig. 2.7 illustrates a possible chain of middleware in an Express application. The first middleware logs the incoming request and then forwards it to the second one, which verifies whether the user is authorized to access the secret content. If authorized, the control is given to the last middleware which responds with the secret information, otherwise, an error message is returned instead.

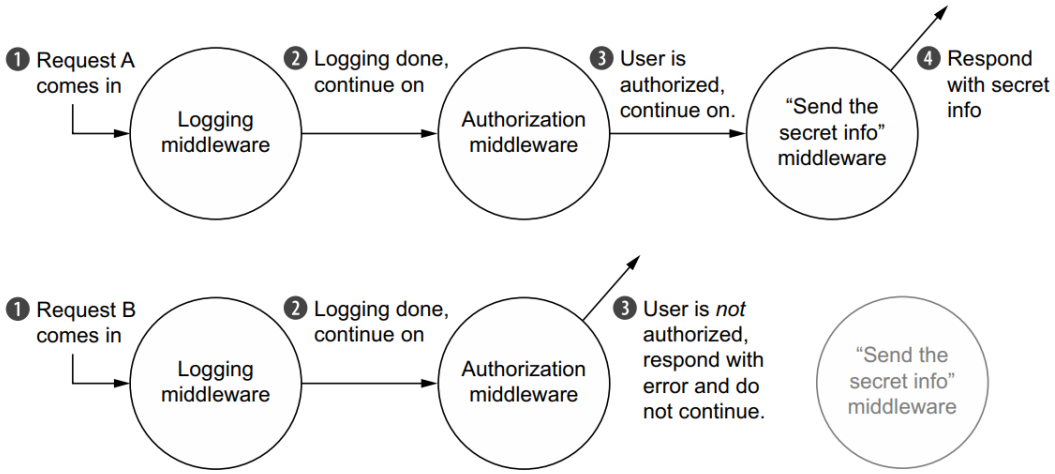


Figure 2.7: An example of Express middleware chain [51].

2.4.2 Routing

While middleware are an important component in modern Express applications, they lack the ability to conditionally execute handlers, which is crucial in most web services. For instance, a GET request to the homepage might need to trigger a different handler compared to a POST request to the login API. In other words, web servers need a way to execute different handlers based on the request path or HTTP method. Express solves this problem by providing methods to register

handlers along with their target request HTTP method and path [51], as shown in the following block.

```
// The handler will return "Hello world" every time a user
// makes a GET request to the /home path.
app.get('/home', (req, res) => res.send('<h1>Hello world</h1>'))
```

Listing 2.4: An example of routing in Express.

2.4.3 Routers

As the complexity increases, dividing the application into multiple files, folders, and *sub-applications* could be an effective way to keep the code manageable. Express solves this problem by introducing the concept of *Routers*, which enable developers to compartmentalize sub-areas of the application. These sub-applications can be implemented and managed in the same way parent applications do but are characterized by a sub-route prefix in the path when accessing them [51]. Moreover, the rules defined in the parent application are inherited by children routes, a useful property when enforcing authorization policies. For example, if the application defines a privileged admin router and requires all users to be logged in before accessing it, all children routes will be automatically protected by this policy. Figure 2.8 illustrates an example diagram of an Express service divided into sub-applications.

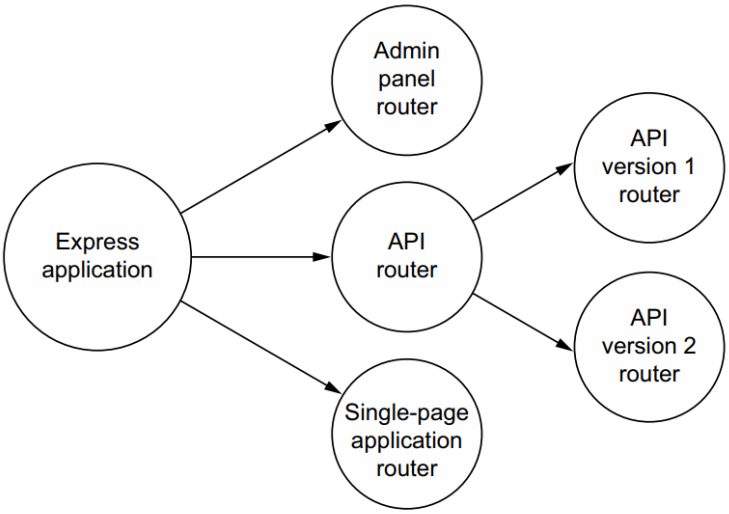


Figure 2.8: An example diagram of Express sub-applications through routers [51].

2.5 Redis

Redis is a fast, in-memory, non-relational database that stores a mapping of keys to different types of values. It also supports disk persistence, as well as replication and sharding to scale read and write performance respectively [52].

Compared to traditional relational databases, Redis does not enforce any schema or relation when dealing with data, nor understands the concept of tables. Redis could be seen as an evolution of in-memory key-value stores like *memcached*, but instead of limiting the value type to strings, it supports also more advanced data structures, such as lists, hash maps, and sets.

In general, when an application needs to reliably persist chunks of data, the recommended approach is to rely on traditional databases, which are designed with strong reliability guarantees in mind. That said, there are a number of cases in which a better performance is needed. If decreasing reliability is an acceptable trade-off, Redis might be an appropriate choice [52]. In particular, Redis offers two persistence modes that could be employed. Firstly, it supports dumping the entire memory to disk either after a condition is met or when specific commands are invoked. This mode is known as *RDB*. Secondly, it supports a mode in which every write operation is logged to an append-only file. When a failure occurs, Redis is restarted and these operations are executed again, returning to a consistent state. This mode is known as *AOF*. These two modes offer incremental reliability guarantees at the cost of a decreased performance [53], thus system designers must enable them only when appropriate, according to the application requirements.

As the number of operations increases, a single Redis instance might become insufficient. Hence, Redis offers a number of features to facilitate scaling based on the workloads. If an application executes numerous reads but relatively few writes, master/slave replication could be an effective choice. In particular, clients would execute the write operations on the master instance, which would then propagate these changes to the slaves periodically. On the other hand, read operations would be executed on slaves, which can be horizontally scaled effectively. On the contrary, if a significant number of writes is executed, sharding allows defining multiple master nodes, responsible only for a subset of the writes. Finally, the most complex application might require a combination of both, which is illustrated in Fig. 2.9.

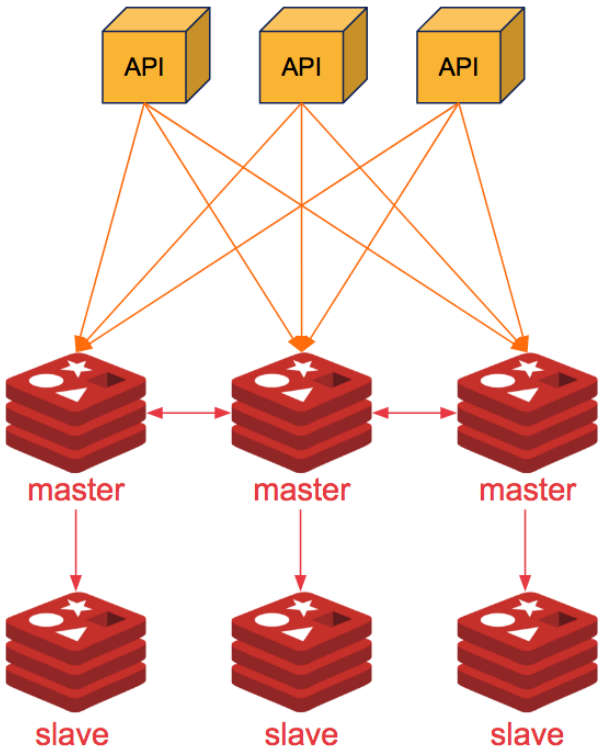


Figure 2.9: An example of Redis cluster with sharding and replication [54].

2.6 SPID

SPID, an acronym for Public Digital Identity System, is a solution that allows Italian citizens to access Public Administration services with a single *Digital Identity* [55]. SPID credentials can be issued by various private companies, known as *Identity Providers*, after verifying the citizen's identity. Then, these credentials can be used to access a number of services, both public and private, known as *Service Providers* [56].

The system offers several advantages. Firstly, citizens can use a single set of credentials, optionally supported by other two-factor methods, to access all public administration websites. Secondly, service providers can reduce costs as the user registration and verification process is already handled by identity providers. Moreover, the user attributes, such as birth date and fiscal number, are guaranteed to be correct by the identity providers, thus no further verification is required.

2.6.1 SAML

SPID is based on the *Security Assertion Markup Language* (SAML) standard, developed and maintained by the OASIS Security Services Technical Committee [57]. The SAML standard defines an XML-based framework for describing and exchanging security information between online business partners, which is based on the concept of SAML assertions [58]. At the abstract level, these could be considered as simple, digitally-signed XML messages sent across trusted organizations.

SAML is mainly employed for two reasons: *Single Sign-On* and *Federated identity* [58]. The former is necessary when organizations want their employees to use the same set of credentials over multiple products, which greatly simplifies account management. As the number of products increases, a standard vendor-independent protocol as SAML becomes crucial to limit interoperability problems. Moreover, as user identities are shared among multiple products, it becomes necessary to define a set of common attributes and unique identifiers. Once they are defined, a user is said to have a federated identity, that is, an agreed-upon set of identity properties on which all products can leverage on.

SAML Participants

A SAML exchange involves two participants: an *asserting party* and a *relying party*. The former, also commonly referred to as *SAML authority*, is the entity that produces SAML assertions. The latter is the entity that receives the assertions and, based on the trust relationship between the two parties, can decide to use or ignore this information.

SAML entities can also assume a variety of *roles*, which define the services, protocols, and assertions they will be dealing with. In the Single Sign-On scenario, the identity provider (IdP) handles the user registration and authentication procedures, whereas the service provider (SP), after querying the IdP to authenticate users, delivers the service. There is also a third role, the *attribute authority*, which produces assertions in response to identity attribute queries [58].

SAML assertions are conceived around the concept of *subject*, an entity that can be authenticated, which could be a person, a company, or a computer. In general, an assertion contains the user identity, composed of information as name and email address, as well as the authentication method used to verify the identity [58]. For instance, the IdP might allow both password-based and two-factor authentication, which offer different levels of security. Given this information, the service provider decides whether granting access to the user is allowed by the organization's access

policies. Hence, the more sensitive the service, the higher the required level of authentication.

2.6.2 Bindings

The different methods used to exchange assertions and request-response protocol messages between SAML systems are defined as *bindings* [58]. The SAML specification defines several types of bindings, with the most commons being the *HTTP Redirect binding* and *HTTP POST binding*.

HTTP Redirect binding

The HTTP Redirect binding specification defines a mechanism by which URL parameters are used to transmit SAML messages. Given the URL length limits imposed by various systems, such as web browsers, this binding can only be used to transmit messages limited in size. Other methods, such as HTTP POST Binding, should be used instead when dealing with significant amounts of data.

This binding is intended for those cases in which the SAML requester and responder need to communicate through an HTTP user agent as intermediary [59]. For instance, this interaction could be used when executing Single Sign-On authentication flows in which the SP and the IdP are unable to communicate with each other. In that case, the authentication information has to pass through the user agent, that acts as an intermediary.

Messages are URL-encoded and transmitted through the HTTP GET method as query parameters, but because this URL must be visited by the user agent, a redirection is employed. In particular, the SAML requester responds to the user with a 302 HTTP Redirect status code pointing to the encoded-URL message. As a result, the client dispatches a request to the SAML responder on behalf of the requester, as illustrated in Fig. 2.10. Finally, another redirection is employed by the responder to transfer the control back to the SAML requester.

HTTP POST binding

The HTTP POST binding specification defines a mechanism by which an HTML form control is employed to transmit messages between SAML entities. As in the HTTP Redirect case, this method can be used when the SAML parties cannot communicate directly, and thus a user agent intermediary is needed, or when the user interaction is necessary to fulfill the request, as in the authentication case.

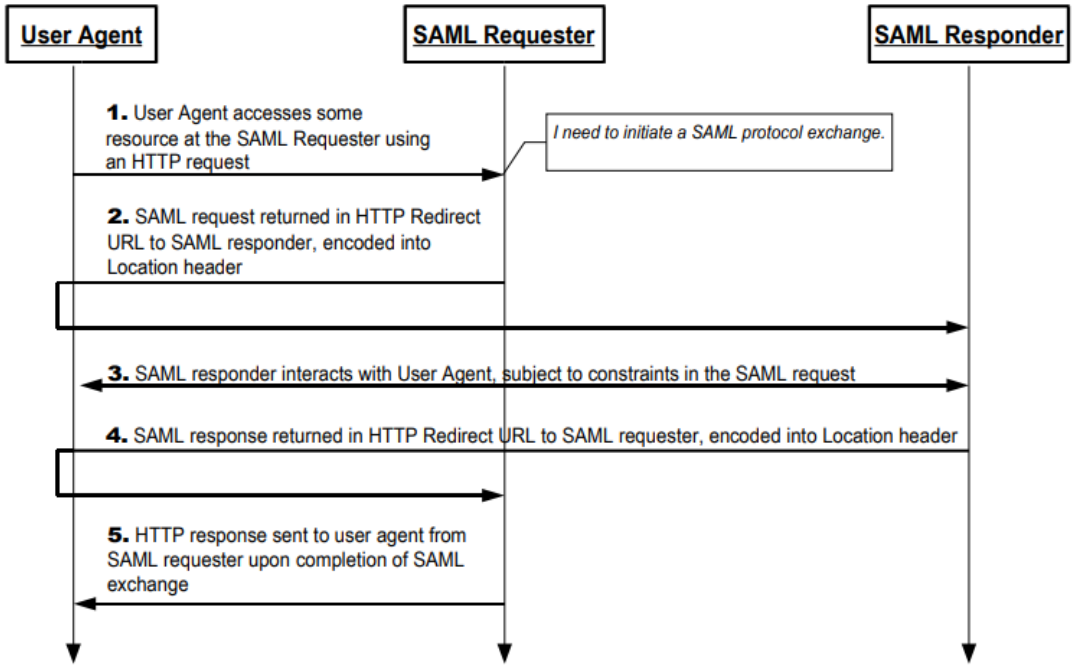


Figure 2.10: A complete HTTP Redirect binding exchange [59].

According to the HTTP POST binding, messages are base64-encoded and inserted inside a hidden form control, whose `action` field is set to be a recipient endpoint. This web page is sent by one of the SAML entities to the user, which then submits the form either manually or automatically using JavaScript, thus dispatching a request to the target entity on behalf of the other party. An example of POST binding web page is presented in listing 4.18, in which the base64-encoded message content is included in the `SAMLRequest` field. If the browser supports JavaScript, it automatically submits the form at page load. Otherwise, it presents the user with a submit button. Finally, Fig. 2.11 illustrates an example of HTTP POST binding exchange.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
5    <body onload="document.forms[0].submit()">
6      <noscript>
7        <p>
8          <strong>Note:</strong> Since your browser does not support JavaScript,
               you must press the Continue button once to proceed.
9        </p>
10     </noscript>
11     <form action="https://ServiceProvider.com/SAML/SLO/Browser" method="post">
12       <div>
13         <input type="hidden" name="RelayState" value="0043
               bfc1bc45110dae17004005b13a2b"/>
```

```
14        <input type="hidden" name="SAMLRequest" value="
              PHNhbWxwOkxvZ291dFJlcXVlc3QgeG1sbnM6c2FtbHA9InVybj
15        ...
16        pvYXNpczpuYW1l=="/>
17      </div>
18      <noscript>
19        <div>
20          <input type="submit" value="Continue"/>
21        </div>
22      </noscript>
23    </form>
24  </body>
25 </html>
```

Listing 2.5: An example of HTTP POST binding form [59]

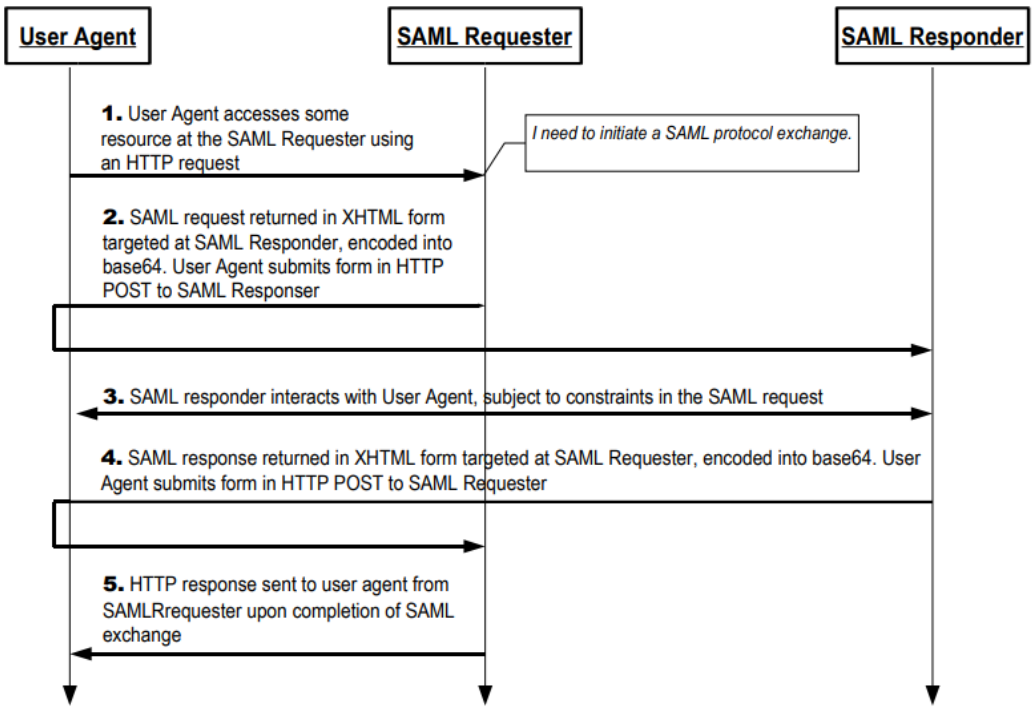


Figure 2.11: A complete HTTP POST binding exchange [59].

2.6.3 Single Sign-On Flow

Although the SAML specification defines several possible authentication flows for Single Sign-On, we focus on a specific variant: the SP-Initiated SSO with Redirect/POST Bindings, which is also the one used by the Italian SPID system.

In a typical scenario, three entities are involved: the user, the service provider,

and the identity provider. In particular, the user wants to access the service at `sp.example.com`, but the resource is sensitive and thus requires authentication. Given that his federated identity is managed by the identity provider `idp.example.org`, the first time he accesses the service provider, he will be redirected to the IdP website, which will then challenge the user and redirect back to the SP if the identity is confirmed. The HTTP Redirect binding is used by the server provider to initially redirect the user to the IdP website, whereas the HTTP POST binding is used by the identity provider to transmit the signed user identity back to the service provider, which in turn establishes a user session and supply the requested resource. The process is schematically illustrated in Fig. 2.12.

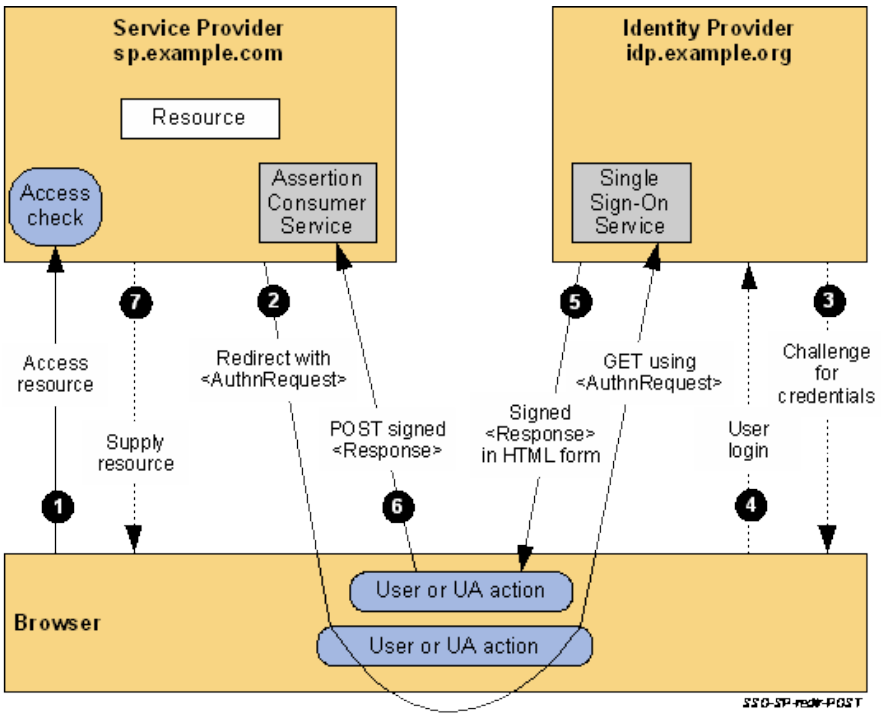


Figure 2.12: An example of SP-initiated SSO with Redirect and POST Bindings [58].

2.6.4 SPID Registry

The SPID Registry is a repository containing all the information about the registered SPID entities. It acts as a proof of the system's *circle of thrust* and it is maintained by AGID, which evaluates all accreditation requests to verify both the technical and legal compliance of SPID entities [60]. The federation registry contains a list of the certified SPID parties, along with their type and metadata. According to the SAML specification, SPID entities can be either service pro-

viders, identity providers, or attribute authorities.

2.7 Summary

In this chapter, we presented an overview of the main software technologies employed to build our work, focusing on the salient features and well-known patterns.

Firstly, we introduced Flutter, a popular cross-platform mobile development framework, along with its motivations, advantages, and trade-offs. Thereafter, an overview of the architecture was discussed, with a focus on its reactive user interface approach.

Subsequently, we presented Redux, an application state-management library. The motivation behind it was presented, along with its three principles, which indicate how a Redux application should be structured, and the concept of middleware.

Thereafter, we introduced Node.js and discussed the differences between traditional multi-threaded web applications and event-driven ones, with the respective advantages and disadvantages. We also presented Express.js, an extensible web framework for Node.js, along with its core concepts and features.

After a brief overview of Redis, one of the databases employed in our work, a particular focus was put on SPID, the Italian's public administration login system, along with a description of the core concepts of SAML, the underlying specification.

Chapter 3

Architecture design

In this chapter, we present a high-level architectural description of our work, NAD-APP, a novel telemedicine system designed to help both patients suffering from severe gastrointestinal diseases and physicians.

The chapter is logically structured following a top-down approach. Firstly, starting from the project requirements, we highlight the main entities involved in the system, along with their relationships. This process, formally known as entity-relationship analysis, is crucial to later define the structure of our database and the application itself. Secondly, a broad overview of the network topology is presented, along with the reasons behind certain architectural choices. Thereafter, a number of more specific topics are discussed, such as the synchronization protocol used by the mobile app to upload the patients' monitoring data and the different authentication protocols supported.

The chapter deliberately omits some specific implementation details, as they are thoroughly discussed in chapter 4, along with the results and possible future improvements.

3.1 ER Analysis

The entity-relationship (ER) model is a crucial tool to design the database structure of an application. Before its introduction, database designers could only rely on their own intuition and experience to convert the organization's real-world requirements into a concrete user schema. This error-prone process had one major flaw: the resulting schema was not a "pure" representation of the real world, but it was limited by database-specific constraints and performance optimizations [61]. As a result, different users would often fail to obtain a consistent and coherent

view of the logical schema, especially when multiple independent systems were involved.

In order to solve the problem, an intermediate stage in the design process was introduced. Starting from the real-world requirements, designers would create a "pure" representation of the world, independent from the specific database implementation and performance consideration, called *enterprise schema* or *entity-relationship model*. This schema describes the entities and the relationships between them, which are only later translated to a specific database schema implementation, as illustrated in Fig. 3.1.

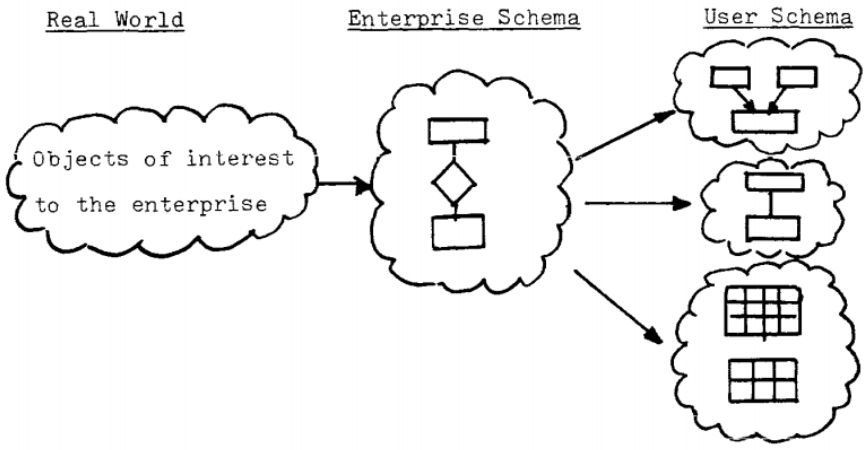


Figure 3.1: The three steps in the database design process [61].

According to this approach, the first step in the database design process is to define the objects of interest, or in other words, the entities involved. In the following section, a description of the relevant ones for our work is presented.

3.1.1 Entities

As previously mentioned, the objective of our work is to create a novel telemedicine system to support both patients and physicians, replacing the current paper-based processes.

These patients suffer from severe gastrointestinal diseases that require constant monitoring to maintain a healthy life. In particular, as part of their treatment, they need to regularly send their meal diary, detailing all the food they ingested throughout the day, and their fluid balance, a series of measurements to record the body intake and production of liquids.

At the same time, doctors continuously track the monitoring data to make sure it remains within healthy ranges, contacting patients if necessary. Following the regular in-person visits, physicians also produce medical reports to keep track of the patient's health status. These reports should be available to both the patient and its doctors for easy consultation.

Finally, doctors can be divided into two categories: general physicians and the IICB center staff. Doctors belonging to the latter are authorized to see the profiles of all the registered patients, whereas the general physicians can only see their subset of authorized patients, after an explicit consent from them.

3.1.2 ER Schema

Starting from the previous requirements, we can define a first, high-level ER diagram for NAD-APP, illustrated in Fig. 3.2.

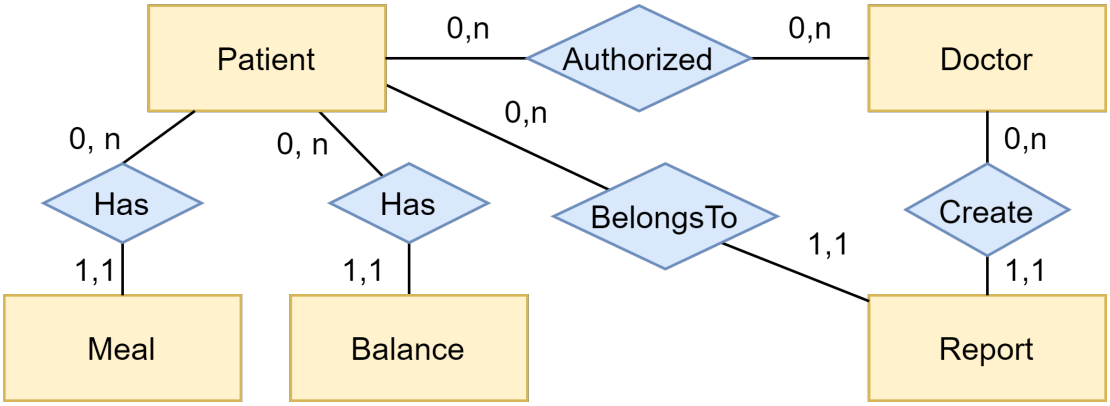


Figure 3.2: A first, high-level ER diagram for NAD-APP.

It must be noted that the proposed diagram deliberately omits the entity attributes, as those are further discussed in the next chapter.

3.2 Network architecture

As previously mentioned, the system is composed of two sub-applications: the mobile app targeted at patients and the web application for physicians. Under the hoods, both applications need to communicate with a common server, which accepts the incoming monitoring data from patients and reports from doctors, and also serves the data when requested, according to the specific authorization policies. The server will be responsible for persisting and retrieving the medical data, as well as managing the user sessions.

From a scalability standpoint, the system is not required to handle high volumes of traffic. In particular, an estimated 500 users are expected to be registered when operational, including both patients and physicians, with peaks of about 50 concurrently connected ones, most likely after launch and dinner times.

A good level of reliability should be guaranteed, especially during day hours, allowing doctors to regularly analyze the monitoring data and past reports of patients. That said, short maintenance downtimes during the night could be accepted, as no activity is expected at that time, being all users located in Italy.

Reasoning about connectivity, doctors are expected to be always online when accessing the system, as a web application is employed. On the other hand, patients use a mobile app that has no such guarantee. In fact, the internet connection on smartphones is often unreliable and, at times, even absent. Therefore the mobile app should include some form of caching mechanism to allow offline usage, uploading the monitoring data as soon as a stable connection is available.

Given these requirements, a possible high-level architecture is proposed in Fig. 3.3. Both patients and doctors interface with a server written in Node.js via REST HTTP requests to the exposed API. The server uses various services to accomplish its tasks, such as Redis to manage user sessions and authentication, PostgreSQL to persist and retrieve data, and an object storage service to store medical records. To achieve a reasonable level of reliability, all services are redundant and regular backups are performed. Moreover, to guarantee an appropriate level of privacy, medical records are encrypted before being uploaded to the object storage service.

3.3 Mobile Synchronization

One of the main features of the patient-targeted mobile app is the ability to digitally upload monitoring data, easing the work of physicians that regularly check it. The app features several forms that the user can use to record different measurements and eventually upload to the NAD-APP servers. Given the potentially unstable internet connection of mobile devices, it is critical to choose an appropriate method that is able to work in most conditions, gracefully failing if necessary. In the next sections, a number of possible solutions are presented, along with their advantages and disadvantages.

3.3.1 Basic approach

A first approach is based on the concept of *thin client*, in which most of the business logic is located on the server side and the client is only an interface to it.

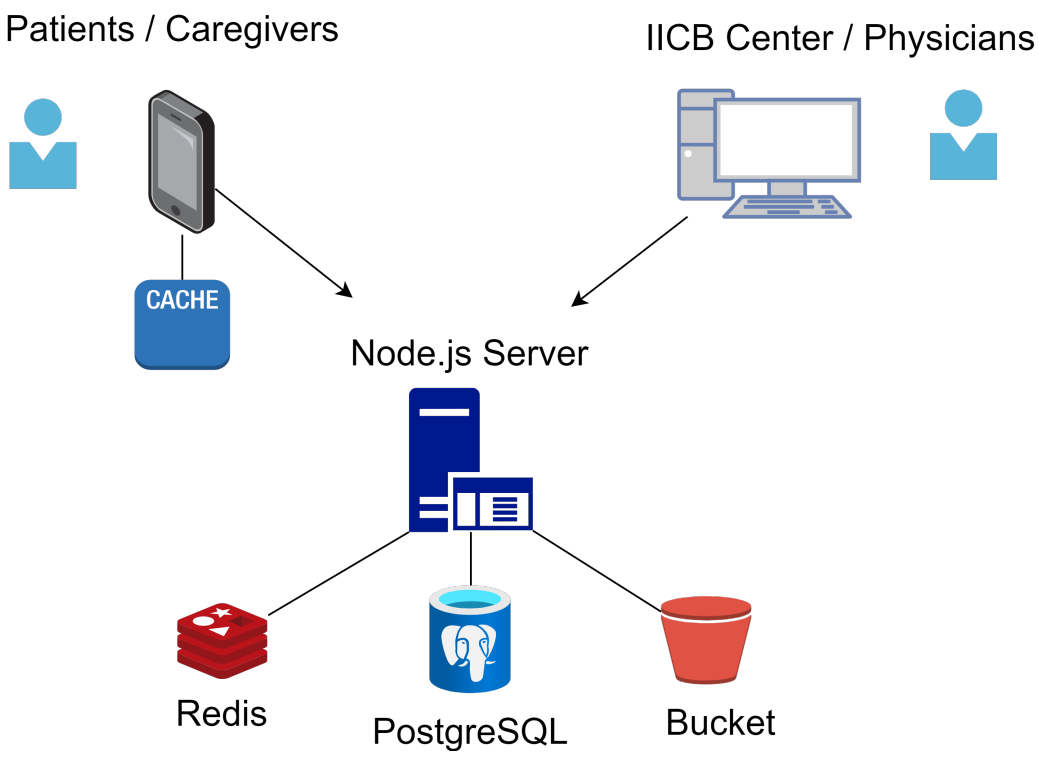


Figure 3.3: High-level network architecture for NAD-APP.

For instance, in the context of web applications, a web page containing an HTML form could be considered a thin client, as the only responsibility of the page is to provide a form that the user can fill in and then submit to the server, which will handle all the related business logic. This approach assumes an online client by default, as without a connection the data cannot be submitted and is therefore lost.

The main advantage of this solution is its simplicity, allowing for a relatively trivial implementation. With that being said, it also presents a number of downsides. Firstly, if the internet connection is absent or the server is temporarily unavailable, the uploading process does not take place, potentially losing the inserted data. Moreover, given the length of some forms, it should be possible for the patient to partially fill it out, completing it throughout the day, a task not easily achievable with this approach. A possible mitigation might consist in saving the current form status on the smartphone's local storage, providing a checkpoint and enabling partial fill-ins.

3.3.2 Two-pass Synchronization

In general, while convenient in some circumstances, the previous approach is characterized by one major downside: the smartphone is assumed online by default. In the context of web applications, this is generally not a problem, as to access the application itself an internet connection is necessary. On the other hand, smartphones are not always guaranteed to have a stable internet connection. In some scenarios, it could be reasonable to prevent the app from working when offline, such as in social networks, as without a connection the app would not provide any value to the user. However, some applications can still be valuable when offline. For instance, a patient might want to insert his monitoring data in rooms where an internet connection is not available. Then, as long as the patient eventually connects to the internet allowing the upload process to take place, the application would still serve its purpose.

Consequently, our mobile application is designed to work offline by default, uploading monitoring data whenever possible. In order to do so, some form of on-device storage must be implemented, so that the application is partially independent from the server. As a result, we need to introduce a synchronization mechanism to propagate changes between the local and server database. Ideally, the synchronization protocol should be reasonably simple to implement and support two-way updates, as sometimes also doctors need to update the patients' monitoring data to fix mistakes and users need to see those changes on their smartphone.

In our work, we propose a synchronization protocol capable of handling data updates from both the mobile and server side of the application, supporting offline usage and resilient to conflicts, based on the combination of timestamps and locally generated UUIDs. In particular, when a patient registers a new record on the mobile app, such as a meal or balance, an entry is generated inside the local database, having a unique UUID and being marked as dirty. The UUID is used to uniquely identify a record in the local database and, paired with the `patientID`, in the central server database. This is needed because the mobile app cannot rely on the server to obtain a globally unique identifier as the smartphone could be offline, but we still need a unique identifier to perform the synchronization process. The dirty property is true when an entry is first created or when modified, and is used during the synchronization to discard the already up-to-date entries from the upload payload, making the process more efficient. After a successful synchronization, all dirty properties are reset to false.

In order to perform the synchronization process, the mobile app keeps track of a value named `lastServerEdit`, a timestamp returned by the server as part of the synchronization response, which represents the last time the patient's data

was modified by the server. After the first synchronization request, the mobile app always send the `lastServerEdit` as part of the request payload, so that the server can determine whether the client has up-to-date entries or it is missing some recent server-side updates. In the latter case, the server sends a dump of all recent patient objects, such as meals and reports, so that the mobile app can replace its local out-of-date data. In Fig. 3.4 and Fig. 3.5 two flow diagrams of the algorithm are presented, from the mobile and server point of view respectively.

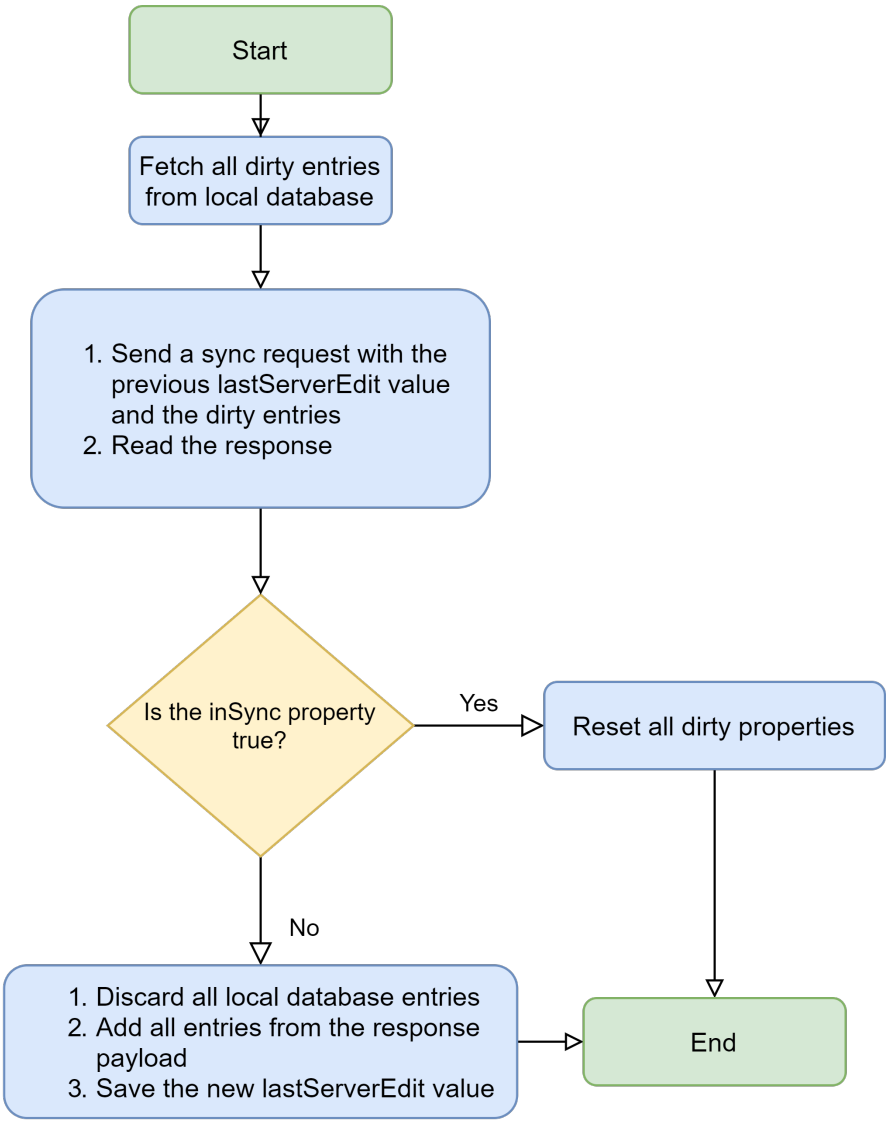


Figure 3.4: Two-pass synchronization: Mobile app flow diagram.

For the sake of clarity, a number of edge cases are discussed to better present the features of the synchronization protocol. As a first example, we consider the

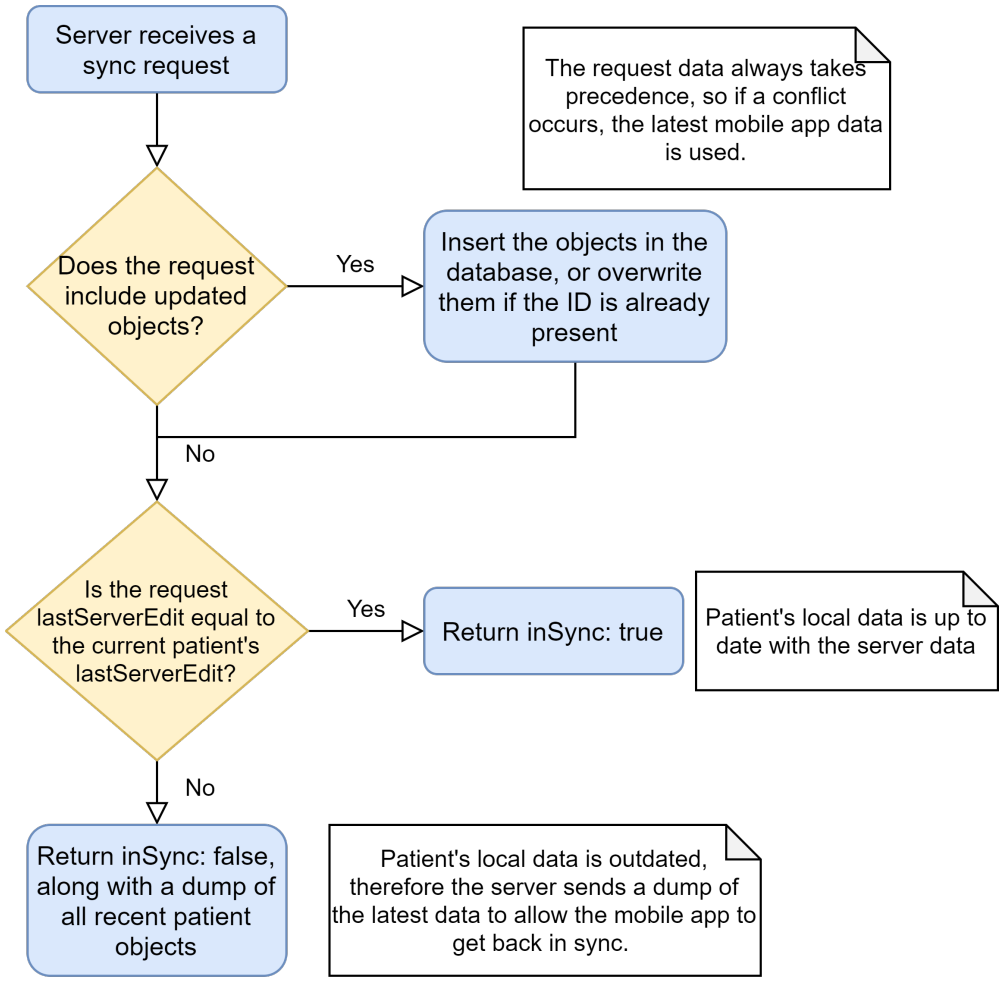


Figure 3.5: Two-pass synchronization: Server's flow diagram.

initial synchronization request made by the mobile app after a successful login, illustrated in Fig. 3.6. Initially, the client does not have any local data and sends a synchronization request having payload with `lastServerEdit = 0`, which is interpreted by the server as an out-of-date request. Consequently, the server responds with a dump of all recent user objects, which will be used by the client to populate the local database, along with the actual `lastServerEdit` value.

The synchronization process is automatically started every time the application is launched, but it can also be dispatched manually when needed. To keep the process efficient, the `lastServerEdit` value is used by the server to determine if a dump is necessary. Figure 3.7 illustrates a case in which the client is already up-to-date with server information, and therefore no object data is returned.

A synchronization request is also dispatched when the patient adds, or changes,

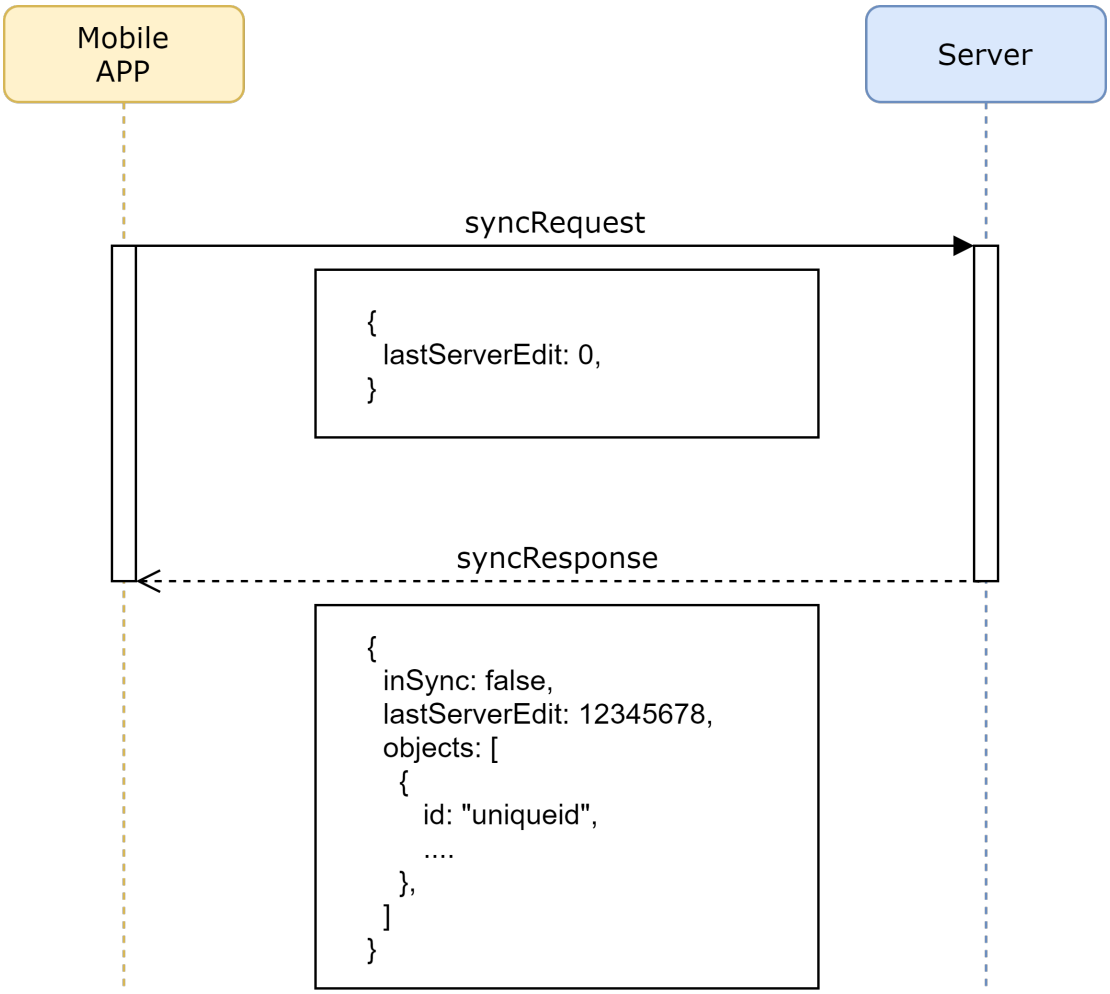


Figure 3.6: Two-pass synchronization: First sync request.

a meal or a balance. As previously said, these records are inserted in the local database and marked as dirty, and thus included in the following synchronization request payload. Figure 3.8 illustrates a common scenario, in which the first patient request includes the newly created object with `id = "uniqueid"`, followed by the server acknowledgment. Then, a doctor changes the patient's object on the server, which in turn also increases the patient's `lastServerEdit` value on server. In the next client request, the server will detect that the client's data is outdated, thus returning a complete dump including the new patient's object value.

A final case of interest is illustrated in Fig. 3.9, which describes what happens if both the server and the mobile app change a patient's object at the same time, therefore generating a conflict. In such circumstances, the patient's value always takes precedence, overwriting the server one. In particular, when the server

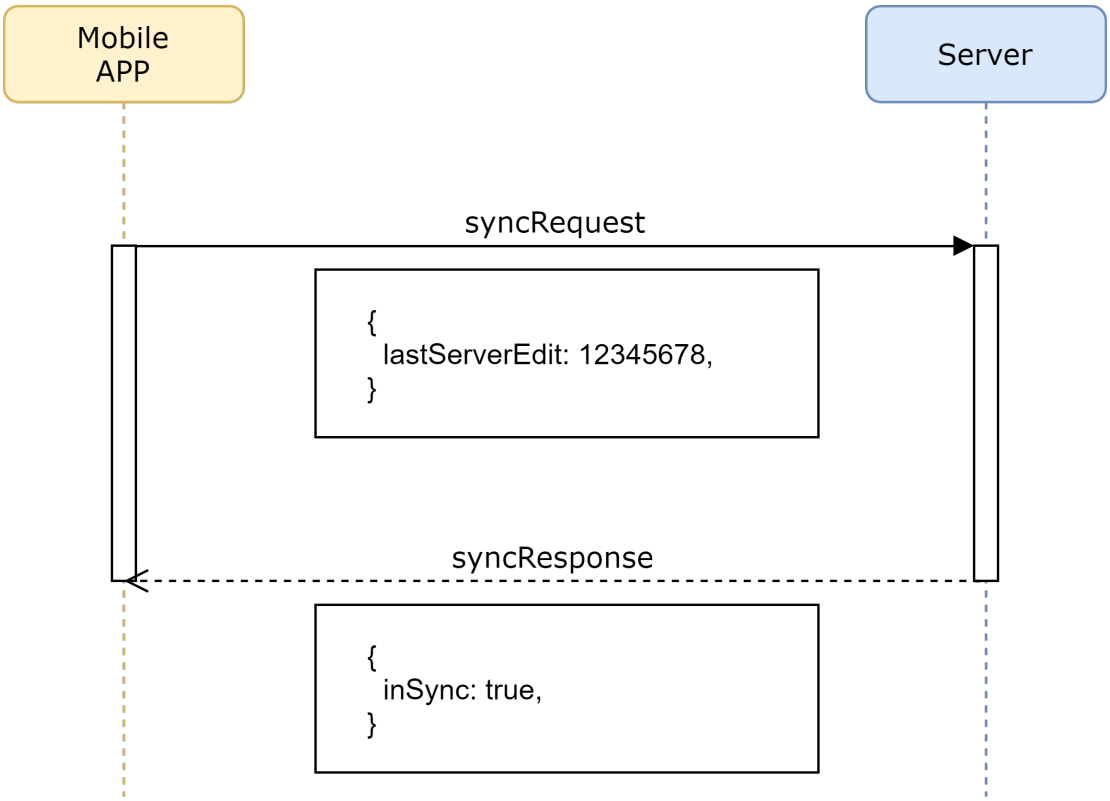


Figure 3.7: Two-pass synchronization: Sync request with up-to-date content.

receives a synchronization request with a changed object, it uses the payload to overwrite the server copy. At the same time, the server detects that the client is out-of-sync, therefore returning a dump of recent patient's objects.

3.3.3 Other approaches

The previously described approach offers a reasonable balance between efficiency and ease of implementation. In particular, if no local synchronization is needed, the server payload is limited to an acknowledgment. However, if the mobile data is outdated, the server returns a complete dump of the recent patient's objects which could be expensive. More efficient solutions are possible, involving elaborate delta-based synchronization algorithms that only transmit the necessary information to keep the two entities in sync. That said, given the relatively rare occurrence of server-side updates, the previous implementation is chosen.

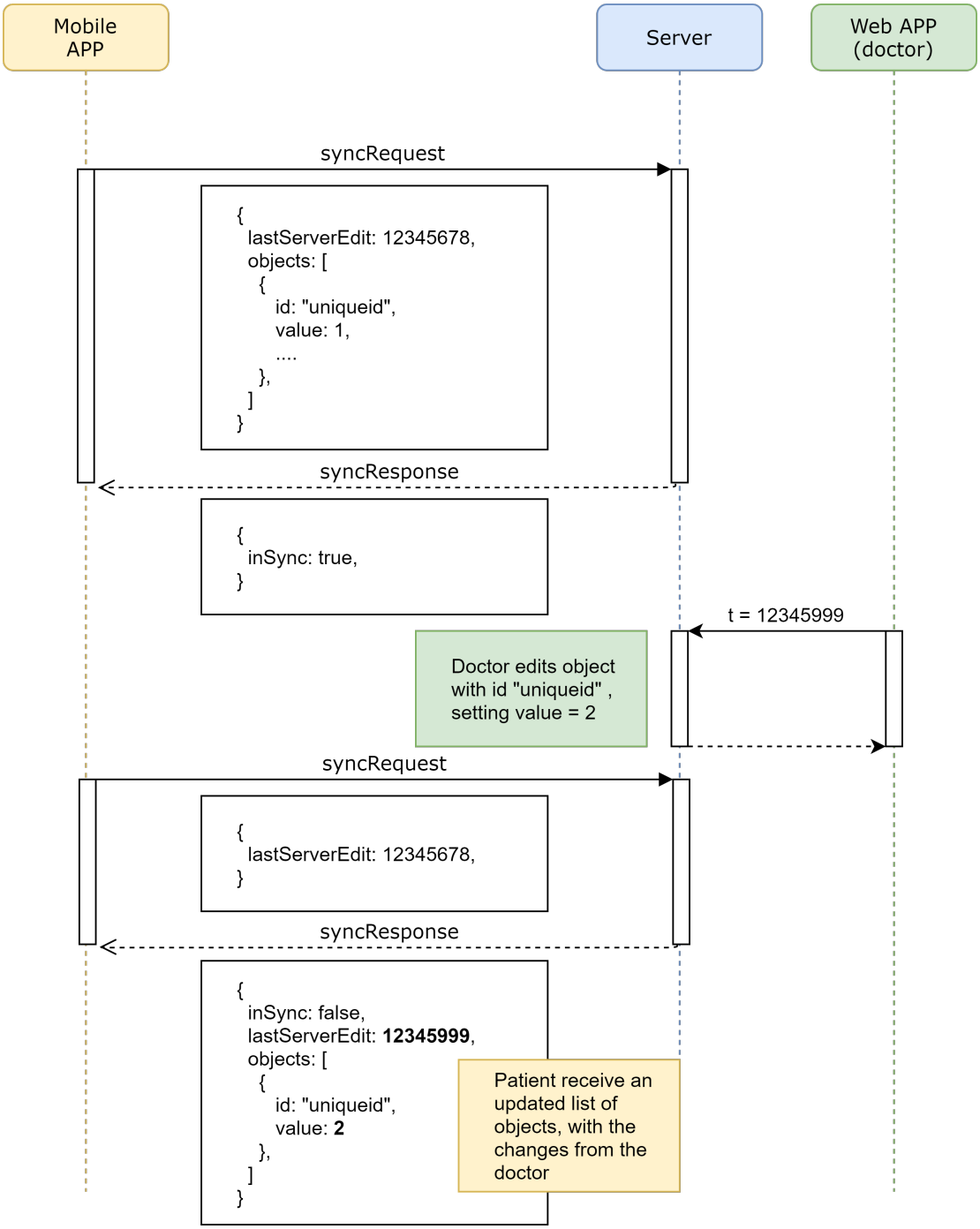


Figure 3.8: Two-pass synchronization: Sync request with patient content and doctor update.

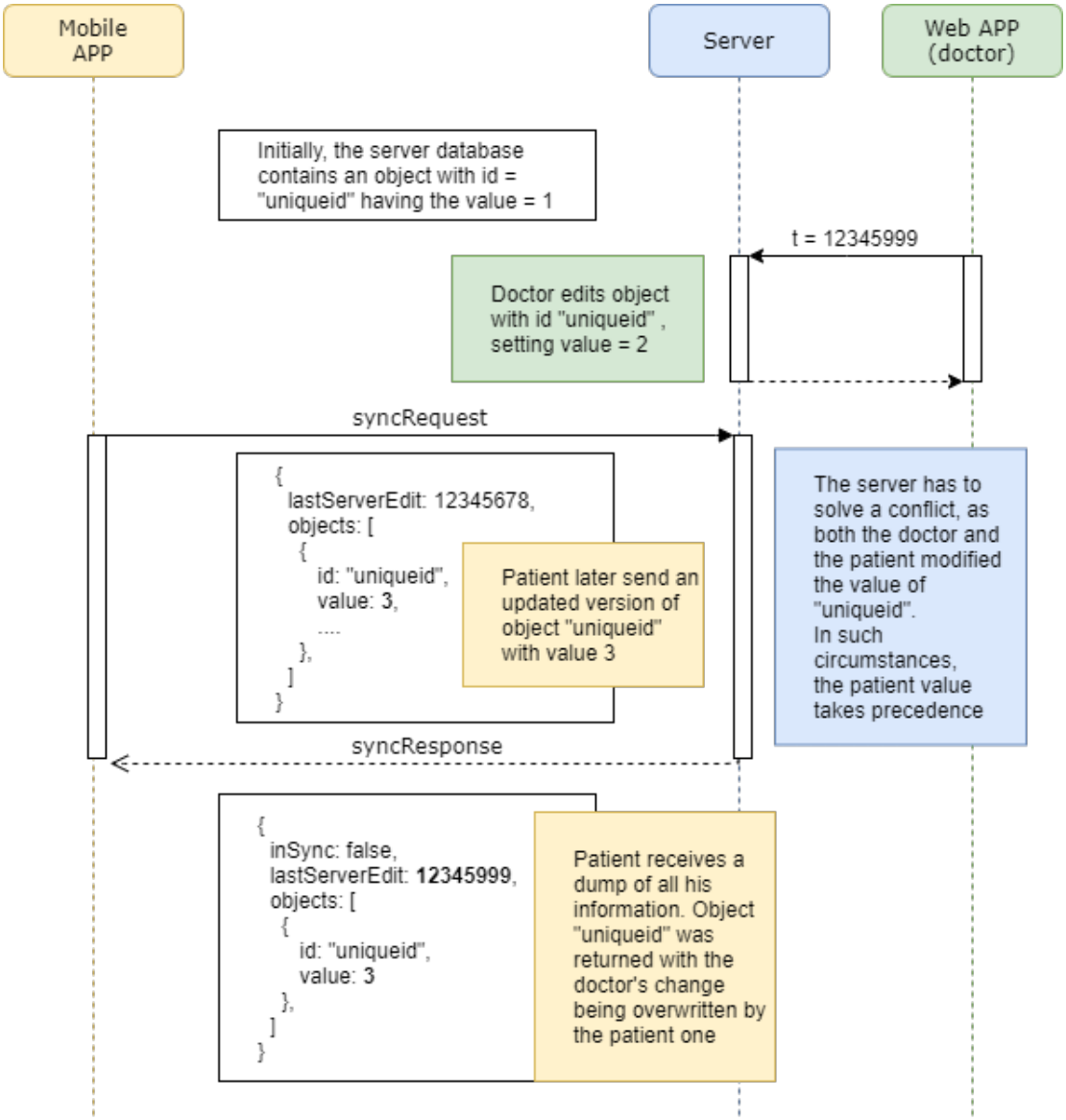


Figure 3.9: Two-pass synchronization: Sync request with conflicting content.

3.4 Authentication

A solid authentication mechanism is crucial in most web applications, especially those dealing with sensitive data. A number of different strategies are possible, ranging from the classical username and password pair to more advanced multi-factor methods, offering increasingly high security protections. The following sections describe the two authentication mechanisms employed in our work.

3.4.1 Two-factor authentication via SMS

Despite being reasonably simple to use, password-based authentication does not offer an high level of security. An attacker could get hold of the password in a variety of ways, for example through brute-forcing or man-in-the-middle attacks, thus compromising the whole user account. To partially mitigate the risk, applications can employ multi-factor authentication, that is, a family of methods in which multiple proofs are used to validate a user identity. For instance, besides the password, users could provide a biometric proof, such as a fingerprint scan, or an ownership proof, such as with hardware-based security keys. On one hand, these additional steps make it harder for an attacker to compromise an account, but on the other, they also make the login process more inconvenient.

A common approach that provides reasonable levels of security while also being convenient for the user is to rely on smartphones to deliver the additional proof. For instance, after entering the password, the user could be asked to insert an additional code, which he received on his previously-trusted smartphone via SMS. Another approach involves the so-called *Time-Based One-Time Password Algorithm* (TOTP), which enables users to generate short-lived OTP values based on time and an initially-agreed secret [62]. The important aspect to consider is that the additional code must be delivered to users through a separate channel.

Although the TOTP-based approach offers a number of advantages, in our work we present an SMS-based two-factor authentication, as it provides the greatest convenience between the two. This is especially important considering that our target users might not be able to use a TOTP code generator effectively.

Figure 3.10 illustrates the two-factor authentication flow designed for our application. Firstly, the user sends its username and password to the server. If valid, the server generates a short random verification code, which is then used by the user to prove its identity, and a longer random verification token, used to uniquely identify the login attempt. The user needs to specify both of these values when performing the second authentication phase. This pair is then saved into the Redis database, which conveniently offers the ability to specify an expiration

date. Then, the server returns the verification token to the client and forwards the verification code to an SMS API, a service that takes an HTTP request and sends an SMS to the user on server's behalf. Eventually, the user receives both the token and the code through different means, and is able to send the phase-two request to the server which, by querying the Redis database, can assert whether the pair is correct. If so, a session is established.

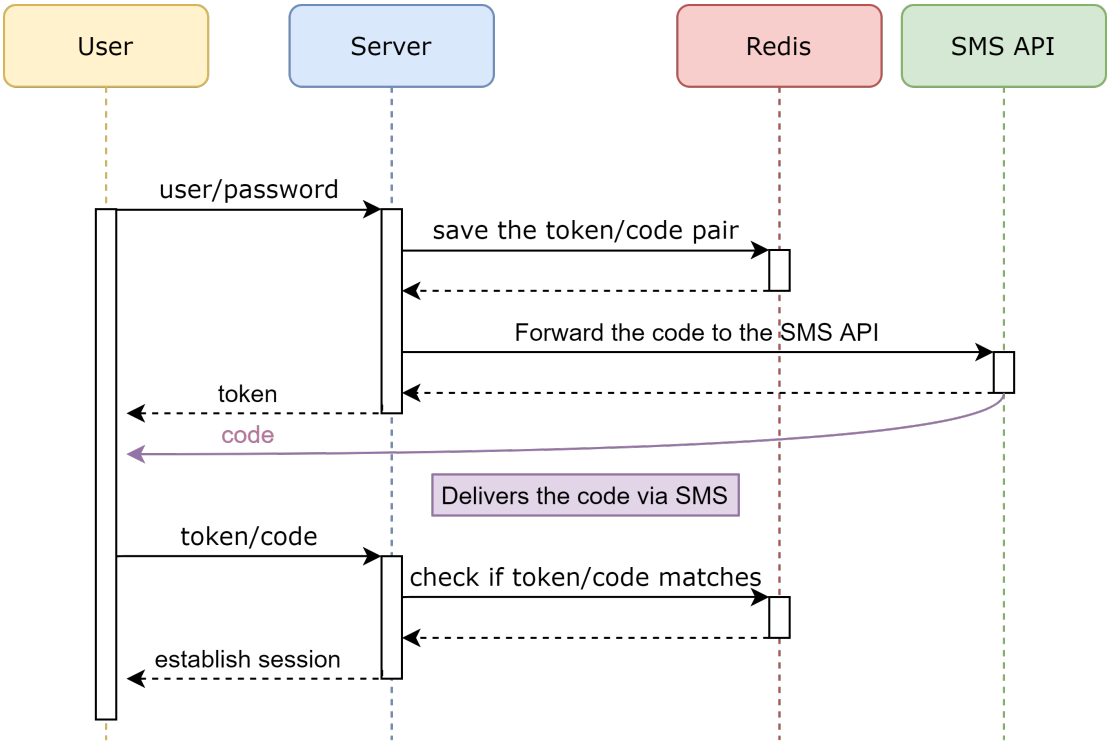


Figure 3.10: SMS-based two-factor authentication flow.

The proposed protocol is designed to correctly handle a number of edge cases, ranging from expired verification codes to replay attacks, and its implementation detail are further discussed in the next chapter.

3.4.2 SPID authentication

Although the SMS-based authentication is a reasonable approach for most users, regional law requirements dictate that all systems belonging to the public administration must support SPID-based logins. Therefore, given that some users might not have SPID credentials yet, especially in the less developed regions of Italy, it is important to design our system to support both authentication strategies.

There are a number of challenges that must be addressed to support SPID-

based logins. First of all, we must define how the SPID federated identity is linked to our internal users' records, as compared to the SMS-based login, credentials are not matched within our database, but are instead evaluated by the IdP. After a successful SPID login, the service provider is provided with a series of verified attributes that can be used to deliver the service to the user. By choosing the right one, the application can associate the federated identity to an internal user profile.

A good candidate is the fiscal number attribute, which uniquely identifies every Italian citizen and is already present in our internal patient records. After a successful login performed on the IdP website, the application is provided with the verified user's fiscal number that can be used to find the correspondent internal user profile, establishing a session. The process is summarized in Fig. 3.11.

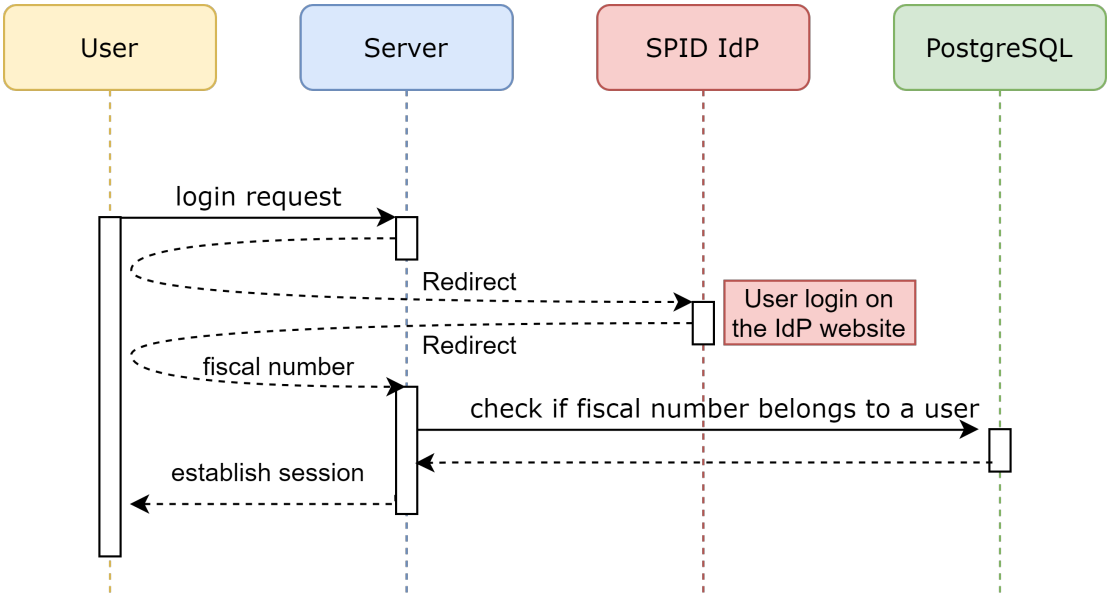


Figure 3.11: SPID-based authentication flow.

Besides the successful login, there are several ways in which the process could fail. Firstly, if the user does not remember the correct credentials, the IdP never sends the fiscal number to the server. Another edge case involves a user with valid SPID credentials but that is not registered within our system. In such case, after completing the login process on the IdP website, our application refuses to establish a session, as no match can be found with the given fiscal number.

3.4.3 SPID login on mobile

Although implementing a SPID-based login is reasonably straightforward on web applications, enabling the same on mobile apps presents a number of challenges. According to the underlying SAML protocol, once the identity is verified, the IdP redirects the user to the service provider website through an HTTP POST binding, along with its verified attributes. This mechanism allows web applications to receive the user's identity information, but it is not directly applicable to mobile apps, as the IdP can only redirect the user to another web page, not an application.

In order to solve the problem, a support back-end server must be used to act as a service provider in the SAML exchange. After the authentication process is concluded, the server generates a random unique token that can be used by the mobile application to establish a session. The specific implementation details are discussed in the next chapter.

3.5 User Experience

As part of a successful product implementation, a crucial step is the *User Experience* (UX) design process, which includes a careful analysis of the user interface's appearance and behavior. In fact, besides the technical problems that must be solved to build a product, the ultimate goal is to provide a solid experience to the user, with usability being a top priority. Moreover, a well-executed design phase positively influences the implementation step, as developers can take the design specification as a starting point to define what to implement. On top of that, the user interfaces can also help to better understand the data entities and relationships involved in the application, as they model the final interaction with the user.

Although a complete UX analysis is outside the scope of our work, a noticeable amount of effort has been spent to delineate the main interfaces and mechanics of our work, as part of a broader requirements definition. Figure 3.12 illustrates an excerpt of the UX analysis process employed in our mobile app, which besides the design, also defines the basic navigation mechanics, highlighted as arrows between components. For the physicians-oriented web-application, a slightly different process is followed. For those users, a solid set of features is more important than the pure design aspect, so when designing the UX experience only the high-level components are taken into account, as shown in Fig. 3.13. The final appearance is then determined by the development framework used during implementation, as described in the next chapter.

3.6 Comparison with existing solutions

As previously discussed, our system relies on cloud technologies to serve both patients and doctors. The use of such approach in the wild is still relatively limited compared to traditional, in-house hosting, for the reasons we already highlighted in the previous chapter, notably data control requirements and legacy systems. With that being said, a number of e-health oriented products are being released and tested in recent years [63][15].

Although several telemedicine and monitoring systems have been released over the last decade, most of them are designed to work either online or offline, making it difficult to support the relatively unstable internet connections on smartphones. On the other hand, our system offers a hybrid approach that enables patients to conveniently use the application when offline, later synchronizing the monitoring data as soon as a stable internet connection is available.

Another important aspect is the integration with existing electronic medical report (EMR) systems, which is critical for an effective diagnosis process [64]. Due to its stand-alone nature, our system is not integrated with the Italian's Fascicolo Sanitario Elettronico, which might limit its effectiveness in some scenarios. An integration between the two would provide the optimal solution, but such effort could only be performed by an organization at the regional or national level.

3.7 Summary

In this chapter, we presented a broad description of the main architectural decisions behind our work. Firstly, after a brief introduction on the advantages and motivations behind the ER analysis, we defined the main entities and relationships involved in the project, which served as a starting point to build the ER schema. Then, an overview of the scalability requirements was presented, enabling a first design of the network architecture.

Thereafter, we introduced the mobile synchronization problem, along with several possible solutions. In particular, we discussed the challenges deriving from the offline capabilities of the mobile app, as well as some protocols to handle them correctly.

Subsequently, the authentication problem was presented, that is, the challenge of creating a system supporting multiple methods for user logins. We described two strategies to support both SMS-based two-factor authentication and SPID-based logins.

Then, a short description of the user experience design process was presented,

along with the motivations behind it and some examples. Finally, a comparison between the existing solutions and NAD-APP was presented.

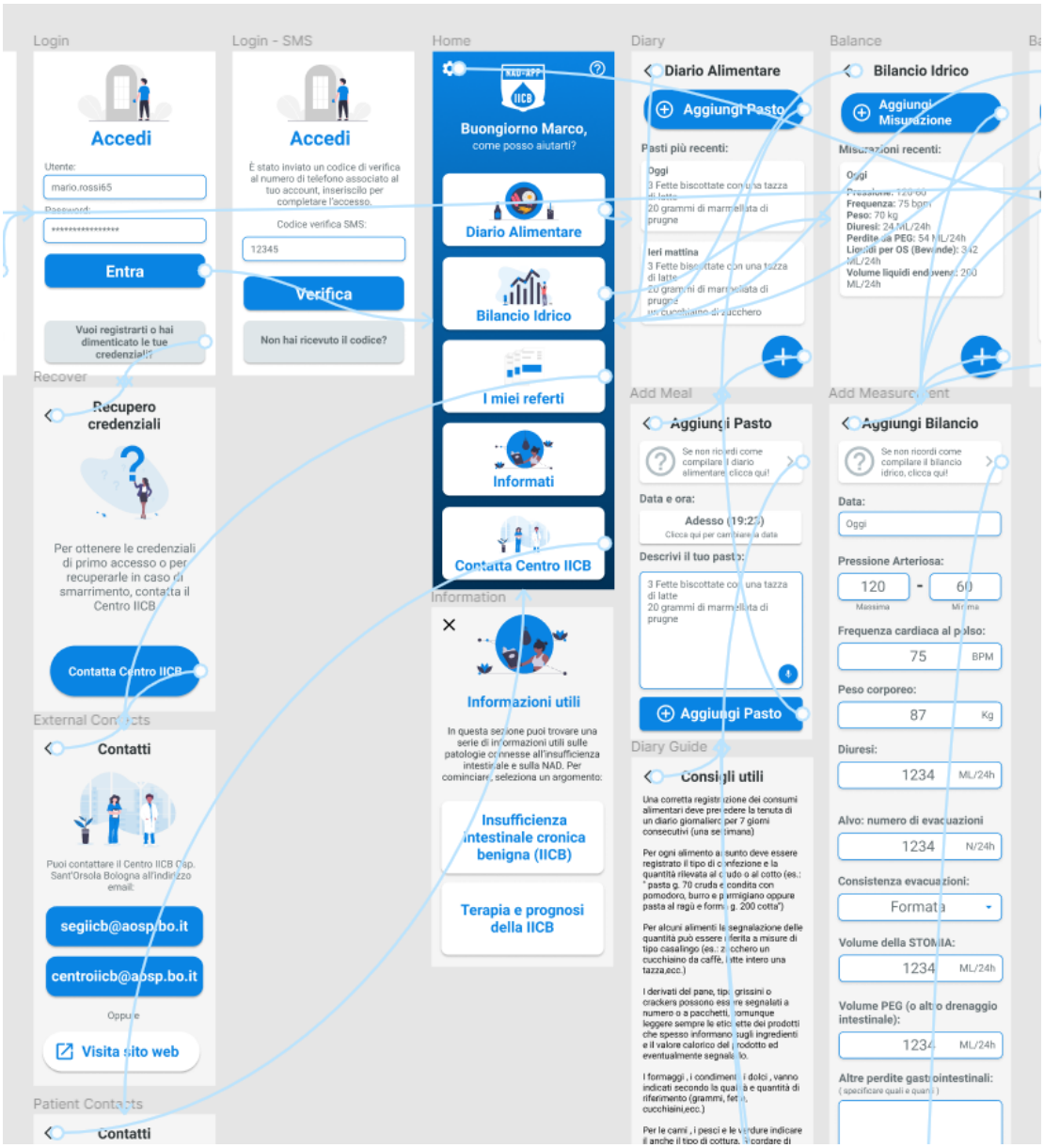


Figure 3.12: A portion of the mobile app UI/UX design, along with the basic navigation mechanics.

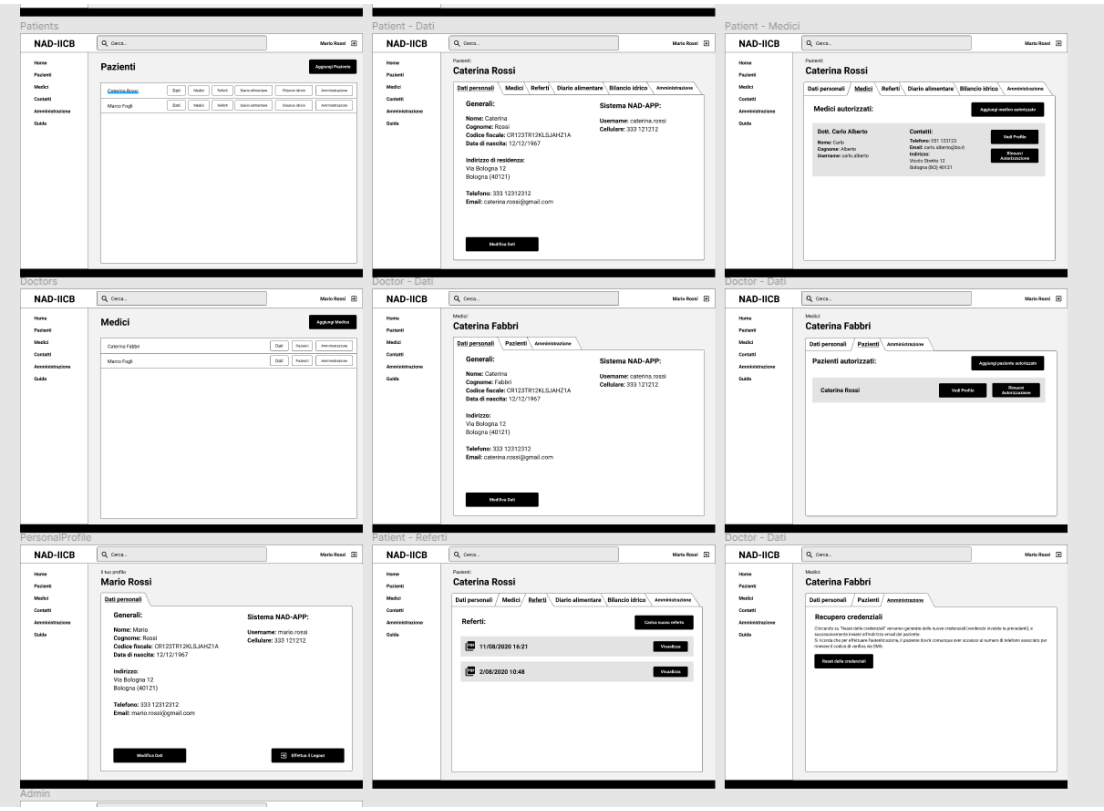


Figure 3.13: A portion of the web application UX design process.

Chapter 4

Implementation and results

In this chapter, we present the NAD-APP implementation process. In particular, the first three sections discuss the development of the server, mobile app, and web application respectively, focusing on the main choices and the motivations behind them. Thereafter, a section is dedicated to load testing, that is, measuring how the application handles an increasingly high volume of traffic. Finally, we discuss the results achieved with our work, along with its possible future improvements.

4.1 Server implementation

Being the entity that receives data from both patients and doctors, the server plays a crucial role, acting as the bridge between the two. As previously mentioned, the server is implemented in Node.js and rely on several storage services as Redis and PostgreSQL. Besides describing the way these components interact with each other, the section also discuss several important topics related to building production-ready applications, such as database migrations, input validation, authentication and testing.

4.1.1 Database

Most data-oriented applications rely on some form of database, or combination of them, to work. These are responsible for storing and retrieving data, and according to modern cloud paradigms, they are also often responsible for maintaining the application state which no longer resides in the web application's memory, enabling easier replication and failover.

Most databases are structured following a client-server architecture in which

applications communicate with the database over a network connection, acting as the client. Generally, these applications do not implement the communication protocol themselves, but rely on *database drivers*, which are libraries that enable applications to control the database using the current programming language constructs in a convenient way.

Figure 4.1: The application communicates with the database through the database driver.

In order to determine the optimal driver, the target database and framework must be taken into account. Given the use of PostgreSQL and Node.js, we choose the `node-postgres` library, as it enables an easy integration between the two.

The `node-postgres` driver provides full control over the database and it is thus sufficient to build most applications. That said, using it directly is not always a convenient choice for developers. The driver accepts raw SQL queries as input, which are necessary to exploit many advanced features, but can be unnecessarily verbose for simple operations. For instance, a substantial portion of the database queries might be limited to simple `SELECT` operations with one or two filters. In such cases, using raw queries is inconvenient as a significant amount of time must be invested to write the objects' fields' marshalling and unmarshalling logic.

A number of higher-level abstractions can be used to improve the situation. Firstly, a *query builder* can help to reduce the verbosity of simple queries by providing a way to specify the desired fields using the current programming language, automatically handling marshalling and preventing SQL injections. To further increase abstraction, another technique known as *Object-Relational Mapping* (ORM) can be used. This approach consists in mapping the data persisted in the relational database to objects in the current programming language, abstracting away the querying aspect entirely. On one hand, this technique makes certain queries extremely convenient, handling all aspects from building the query to unmarshalling the fields inside the object instance, but on the other, it might become limiting in more complex scenarios. The problem arises from the so-called *object–relational impedance mismatch*, that is, the set of conceptual difficulties encountered when mapping data from the relational paradigm to an object-oriented one. Some ORMs attempt to manage the problem by creating a very rich and complex set of methods, but as a result, some queries that would have been reasonably simple when expressed in raw SQL become complex and difficult to reason about.

In our work, we choose an intermediate approach by using a lightweight ORM called `Objection.js`, which provides the basic features of an ORM while facilitating the use of raw SQL queries when needed. An important advantage of this approach is that it does not force developers to use a specific level of abstraction. Sometimes raw queries are necessary to accomplish advanced tasks, but often the ORM-provided queries are enough. Supporting both options is the ideal solution, as it provides the maximum convenience without limitations.

From an architectural standpoint, Objection.js is built on top of Knex.js, an SQL query builder for Node.js, which in turn provides several additional features such as migrations and seeding. The application can use these two layers to access different levels of abstraction, as shown in Fig. 4.2. For instance, Objection.js could be used to conveniently fetch an object with a given id, whereas Knex.js could be used to compute the aggregates of a multi-join query.

Migrations

An important aspect of modern web applications is migration management, that is, the way changes in the database structure are performed while keeping the pre-existing data intact. For example, at some point, it might be necessary to add columns to a table or delete an existing one. These changes are handled by the *schema migration tool*, which facilitates the management through incremental changes that can be easily saved into a version control system. Moreover, these tools allow to rollback to previous schema versions if necessary, although the process is not guaranteed to be loss-less depending on the past operations.

The migration tools are invoked with a specific schema version, which is used to determine what operations must be carried out to bring the database to the desired schema. There are several tools available to perform the migrations with varying degrees of automation, but in our work we choose the built-in migration feature of Knex.js, which offers a good control over the updates while being easy to use.

In Knex.js, migrations are defined as a series of timestamped files, containing the migrate operations. In particular, each file specifies the "up" and "down" methods, which are used by the incremental update and rollback respectively. Knex.js, starting from the timestamp information and desired version, navigates the migration sequence forward or backward to bring the database to the desired schema. An example of migration file is listed in 4.1.

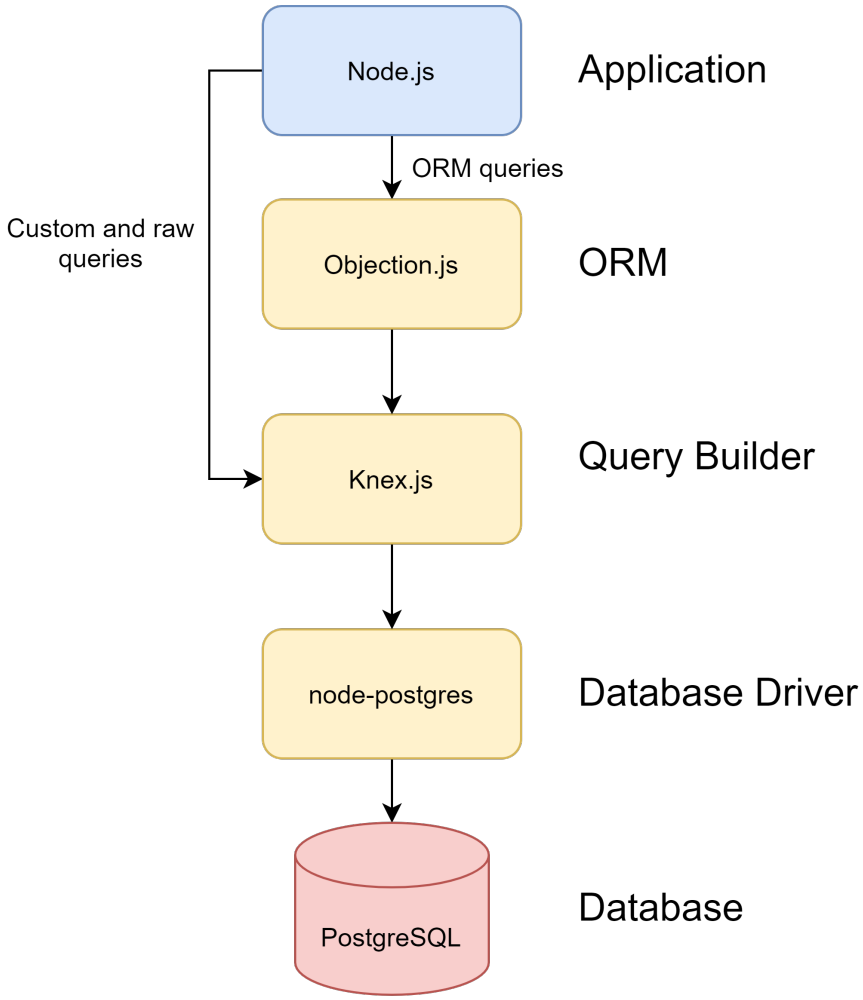


Figure 4.2: The Database stack of our application.

```
import * as Knex from "knex";

export async function up(knex: Knex): Promise<void> {
  return knex.schema.createTable("meals", table => {
    table.increments('id').primary()
    table.string("uuid").notNullable()
    table.integer("patientId").unsigned()
         .references("id").inTable("patients").onDelete("CASCADE").index()
    table.timestamp("date").notNullable()
    table.text("meal").notNullable()

    table.unique(["uuid", "patientId"])
  })
}

export async function down(knex: Knex): Promise<void> {
  return knex.schema.dropTableIfExists("meals")
}
```

Listing 4.1: An example of Knex.js migration file.

4.1.2 API input validation

In general, any application accepting input from the user must include some form of data validation. That can be implemented at different levels of the stack, from low-level database constraints to validation checks on the client web application. That said, although client-side validation could provide the user with instant feedback, its presence should be considered an addition to a solid server-side validation and not a replacement. That is necessary as data from clients should never be trusted, as it can be easily tampered by attackers either on client-side or during transfer. Moreover, server-side validation is even more important in the case of REST APIs, as there is no control over the client.

There are a number of ways to implement such validation. A possible first approach involves manually checking the submitted data to make sure it satisfies the business logic constraints. For instance, a regex could be used to verify if a given email address is well-formatted.

With that being said, manual checks start to become a burden as the API surface and complexity increases. In such cases, being able to specify the expected format and constraints in a declarative way while delegating the actual validation to another entity can speed up the implementation significantly.

A technique that is being heavily used in production is known as *JSON Schema*, which allows to describe the expected format of incoming JSON data and provides a series of validators to automatically enforce those formats. An example is shown in listing 4.2, which is an excerpt of the validation schema used by the mobile app synchronization endpoint. Starting from this definition, we have access to a simple and solid method to validate the user input, as shown in listing 4.3.

```
export const syncValidator = new Ajv().compile({
  type: 'object',
  required: ["lastServerEdit"],

  properties: {
    lastServerEdit: { type: "integer" },
    meals: {
      type: "array", items: {
        type: "object",
        required: ["uuid", "meal", "date"],
        properties: {
          uuid: { type: "string", minLength: 1, maxLength: 50 },
          meal: { type: "string", minLength: 1, maxLength: 1024 },
          date: { type: "string" }
        }
      }
    },
  }
})
```

Listing 4.2: An excerpt of the JSON schema used by the mobile synchronization endpoint

```
if (!syncValidator(req.body)) {
  return res.status(400).json({ message: syncValidator.errors })
}
```

Listing 4.3: Using the previously defined JSON schema to validate user input and providing detailed errors to the user if necessary.

4.1.3 Basic Authentication

As already mentioned in the previous chapter, we dedicate a good deal of effort to implement a solid authentication mechanism for those users without access to SPID credentials. In particular, the system relies on an SMS-based two-factor authentication to provide a convenient access strategy while guaranteeing a good level of security. As the name implies, the process is divided into two phases. Firstly, the user enters its username and password credentials. If they are correct, the user then receives a verification code on his phone via SMS to complete the procedure. In the following paragraphs, a throughout description of the two phases' implementation is presented.

Security

The first phase involves a traditional login method based on username and password pairs. As a result, the server needs a way to verify the submitted password, proceeding to phase two if correct. Technically, there are several methods to implement such verification, but given the sensitivity of the task, security must be a critical priority. State-of-the-art systems do not store passwords, but instead, they only keep track of their *salted hashes*.

Hashes are generated by *cryptographic hash functions* (CHF), one-way functions that map input data of arbitrary size to a fixed size array. In particular, these functions are fed with the user's password and produce a fixed-length string as an output, which is saved inside the database. When a password verification is needed, the submitted attempt is hashed and the result is compared with the previously saved hash. If they match, access is granted.

The strength of this approach is that passwords are not matched directly, but instead, only their hashes are compared. The important aspect to consider is that the resulting hashes must be practically infeasible to revert, as otherwise, an attacker could potentially discover the original password. There are several techniques employed to significantly decrease the likelihood of such events. Firstly, using slow hashing functions makes *brute-force attacks* more difficult to carry out, as they decrease the number of attempts that could be performed in a given time

frame, exponentially increasing the time required to guess the password. Secondly, a random "salt" should be added to the password before the hashing operation, making the process resistant to *rainbow table attacks*. These attacks exploit large datasets of hashes to retrieve the original password. If the hash is present in the list, the plain text password is found without brute-forcing, making the attack particularly dangerous. For this reason, adding a random "salt" string to the password before the hashing process can protect from rainbow table attacks, as pre-computing a dataset of such long passwords would be infeasible. The salt is then stored in the database alongside the password hash, as it is needed by the verification process.

Among the possible hashing algorithms available, *Bcrypt* is a popular choice across the industry due to its properties. Firstly, it incorporates a salt to protect against the previously mentioned rainbow table attacks. Secondly, it is an *adaptive function*, that is, a function that accepts a parameter to increase the iteration count, making it slower and therefore easily adapting it to the increasing computational power of modern computers [65]. For these reasons, we employ bcrypt as the hashing algorithm for our work.

Once the hashing algorithm is defined, the following steps involve integrating such strategy in the framework of choice. A popular library in the Node.js ecosystem is Passport.js, a flexible authentication middleware for Express.js applications. Passport provides a series of methods to ease the authentication process while delegating many non-core aspects to the application, such as database persistence.

Passport leverages on the Node.js APIs to manage user sessions, which in turn support many implementation strategies. For instance, sessions could be saved to a database or maintained in memory. In our case, the Redis integration proved to be an ideal solution, providing an easy way to persist and retrieve user sessions from a Redis store. Once the server determines that a session should be established, an entry is created on the Redis store containing the user identifier and a `Set-Cookie` header is sent to the client. When Node.js receives the subsequent requests, it uses the session id included in the `Cookie` header to retrieve the session information from Redis.

Phase one

As previously mentioned, the result of a successful phase one is a random verification code being sent to the user, which is then used to complete the second login step. An excerpt of the phase one Express handler is shown in listing 4.4. In particular, after extracting the username and password from the request body

and validating their format, the handler queries the PostgreSQL database to obtain the user information. If the user is found, the password is validated using the Bcrypt algorithm, otherwise, an error is returned. Thereafter, a random 6 digits code is generated, along with a verification token, that is, a long random string used to identify the login attempt. The token-code pair is then saved in the Redis store with a default expiration time of 5 minutes. If the user does not complete the login procedure within that time-frame, the entry is deleted, and therefore phase one must be repeated again. Then, the 6 digits code is sent to the user via a POST request to the SMS API, and finally, the token is returned as the response. The user needs to provide both the token and the code to the phase-two endpoint, acting as a strong proof of his identity.

```
router.post(
  '/login',
  async (req, res, next) => {
    try {
      // ... extract the username and password from the body and validate the
          format

      // Query the corresponding patient or doctor
      if (isPatient) {
        user = await Patient.query().select().whereRaw("LOWER(username) = ?", [
            username.toLowerCase()]).first()
      }
      if (isMed) {
        user = await Doctor.query().select().whereRaw("LOWER(username) = ?", [
            username.toLowerCase()]).first()
      }

      if (!user) {
        throw new HttpError("user not found", 401)
      }

      // Validate the password
      const isValidPassword = await bcrypt.compare(password, user.hash)
      if (!isValidPassword) {
        throw new HttpError("invalid credentials", 401)
      }

      // Create the intermediate payload
      let userData = null
      if (isPatient) {
        userData = {
          patientId: user.id
        }
      } else if (isMed) {
        userData = {
          doctorId: user.id
        }
      }

      if (!userData) {
        throw new HttpError("bad user role", 403)
      }

      // Generate a 6 digit random code for the 2 factor authentication
      const code = randomDigitCode(TWO_FACTOR_CODE_LENGTH)
      // Generate a random verification token
```

```
      const verificationToken = randomString(100)

      // Now save the verification data on Redis
      const redisPayload: RedisVerificationPayload = {
        code: code,
        user: userData,
      }
      const redisKey = `${REDIS_VERIFICATION_PREFIX}${verificationToken}`

      await set(redisKey, JSON.stringify(redisPayload), "EX",
          TWO_FACTOR_CODE_EXPIRATION)

      // Send the code to the user via SMS
      await axios.post("https://SMS API URL/", {
        message: "NAD-APP verification code: "+code,
        telephone: user.telephone,
      })

      // Return the attempt token
      res.json({
        verify: verificationToken
      })
    } catch (err) {
      next(err)
    }
  }
);
```

Listing 4.4: An excerpt of the login phase one REST handler

To guarantee an appropriate level of security, the random code and token generation must be performed using a *Cryptographically secure pseudorandom number generator* (CSPRNG), that is, a pseudorandom number generator that guarantees high-levels of entropy and is therefore suited for use in cryptography.

Phase two

A successful phase two starts from the token-code pair obtained at the end of phase one and concludes by establishing a user session. As with phase one, an excerpt of the Express handler is shown in listing 4.5. In particular, after extracting the validation token and code from the request object, the handler queries Redis to obtain the information attached to the given login attempt. If an entry is found, the code is compared and, if it matches, a session is established using Passport's `logIn` method, which automatically creates a session entry on the Redis store.

```
router.post(
  '/verify',
  async (req, res, next) => {
    try {
      const verificationToken = req.body.token
      const code = req.body.code
      if (!verificationToken || !code) {
        throw new HttpError("missing verify params", 400)
      }
```

```
10
11        const redisKey = `${REDIS_VERIFICATION_PREFIX}${verificationToken}`
12
13        // Get the available data, if present
14        const redisPayloadJSON = await get(redisKey)
15        if (!redisPayloadJSON) {
16          throw new HttpError("invalid token", 401)
17        }
18
19        // Delete used key from store
20        await del(redisKey)
21
22        const redisPayload = JSON.parse(redisPayloadJSON) as
              RedisVerificationPayload
23        if (code !== redisPayload.code) {
24          throw new HttpError("invalid code", 401)
25        }
26
27        req.logIn(redisPayload.user, err => {
28          if (err) { return next(err) }
29          res.json({result: "ok"})
30        })
31      } catch (err) {
32        next(err)
33      }
34    }
35  );
```

Listing 4.5: An excerpt of the login phase two REST handler

As it can be seen from the listing, the phase-one entry is always deleted from the Redis store after one attempt, mitigating the risk of replay attacks.

4.1.4 SPID Authentication

In the previous chapters, a broad overview on the reasons behind SPID login support is presented, along with the main characteristics of the protocol. Being a specification aimed at supporting federated authentication and authorization, its implementation requires a few external tools to simulate a complete environment. In particular, an identity provider must be present in order to perform a successful SAML authentication. The SPID ecosystem offers a number of tools to simplify the development process, namely a test identity provider and a Passport.js integration.

Test Identity Provider

The test identity provider is available as a Docker image and thus can be executed with ease on the development machine. The environment's configuration starts by generating a key pair for the service provider, that is, the public and private keys

used to sign its assertions. Being a federated network, the test identity provider needs to be aware of the existence of our application, along with the list of required attributes. Moreover, being the system based on a circle of trust, the IdP needs to know the public key of each service provider to verify the authenticity of any assertion it receives. In a production environment, this information is populated following an official procedure to guarantee the organization's compliance, but in the development environment, this information can be customized by editing the *sp_metadata.xml* file. An excerpt of the important parts is presented in listing 4.6.

```
1  ...
2  <md:EntityDescriptor ...
3      entityID="http://nadappserver:8000/">
4
5      <md:SPSSODescriptor ... >
6          <md:KeyDescriptor use="signing">
7              <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
8                  <ds:X509Data>
9                      <ds:X509Certificate>
10             SERVICE PROVIDER PUBLIC KEY
11                     </ds:X509Certificate>
12                 </ds:X509Data>
13             </ds:KeyInfo>
14         </md:KeyDescriptor>
15 ...
16
17         <md:AssertionConsumerService
18             Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
19             Location="http://nadappserver:8000/spid/acs"
20             index="0"
21             isDefault="true" />
22
23         <md:AttributeConsumingService index="0">
24             <md:ServiceName xml:lang="it">Login NAD-APP</md:ServiceName>
25             <md:ServiceDescription xml:lang="it">Login su NAD-APP tramite SPID</
                   md:ServiceDescription>
26             <md:RequestedAttribute FriendlyName="Codice fiscale" Name="
                   fiscalNumber" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-
                   format:basic"/>
27         </md:AttributeConsumingService>
28 ...
29 </md:EntityDescriptor>
```

Listing 4.6: An excerpt of the IdP configuration file

The previous listing highlights the main information that the IdP needs to know about the other parties. In particular, besides the service provider public certificate, the configuration file also specifies the endpoint and binding used to receive the assertions, and the list of attributes the service provider needs to operate, respectively at line 19 and 26. In our case, only the fiscal number is needed to identify users, therefore no other attribute is requested. Finally, the list of test users must be specified in another configuration file, along with their attributes. At this point, the test identity provider is ready to accept login requests our application.

Node.js integration

Although a Passport.js strategy for SPID is already available on the official website, its use presents a number of challenges, mainly due to its unmaintained state and lack of proper Typescript type definitions. Consequently, our work is based on a slight adaptation of `io-spid-commons`, a library build for the Italian's administration's app "IO", which is open source and provides an up-to-date SPID strategy implementation, along with proper typing definitions.

Once initialized, the library automatically fetches the metadata information from all the registered identity providers, including their public keys and login endpoints. Each party must know the public keys of the entities involved, creating the circle of trust around which SPID is built.

The library leverages on a Redis store to maintain authentication states, as to successfully conclude the login process multiple requests are needed. In our case, the Redis instance is the same used in the basic login, as the expected volume of traffic is not high. If necessary, a new store could be dedicated to this task, increasing the load capacity of the system.

Finally, the library provides two callbacks to define the application-specific business logic to handle both successful and failed login attempts. An excerpt of the callback is shown in listing 4.7 with all the error handling code being removed for the sake of brevity.

```
export const spidAssertionConsumerServiceCallback: AssertionConsumerServiceT =
    async payload => {
  // Obtain the user fiscal number
  const fiscalNumber: string | undefined = (payload as any).fiscalNumber
  ...
  // Obtain the patient info
  const patient = await Patient.query().where("CF", fiscalNumber).first()
  if (patient) {
    userTokenPayload = {
      patientId: patient.id
    }
  }
  ...
  // Generate a random verification token
  const verificationToken = randomString(100)
  // Save the token to redis
  await set(verificationToken, JSON.stringify(userTokenPayload), "EX", 60)
  ...
  return ResponsePermanentRedirect({ href: "/spid/success?token="+
      encodeURIComponent(verificationToken)} as UrlFromString);
};
```

Listing 4.7: An excerpt of the SPID login callback without the error handling code.

In particular, the callback receives a payload with all the verified attributes from the IdP, and after extracting the fiscal number, it queries the database to

find a matching user. If a patient is found, its ID is saved inside Redis, along with a random one-time verification token. The user is then redirected to a success page having the token as the URL parameter, which is then extracted by the client app to perform the final request, establishing the session. As with the verification code in the SMS-based authentication, the Redis entry has a short expiration date, mitigating the risks of misuse.

This double step is needed mainly for two reasons. Firstly, the callback handler does not have access to the request object, and therefore cannot establish a session on its own. Secondly, for reasons that are further explained in section 4.2.4, the mobile application requires a special procedure to support SPID logins.

4.1.5 API structure and authorization

As part of the API design process, an important task is defining the endpoint structure. A popular paradigm, also adopted by Google, is known as *resource oriented design*. This approach promotes the use of URLs to represent resources and HTTP methods to represent actions, respectively known as *nouns* and *verbs*.

The API is modeled as a resource hierarchy, in which each resource can be either a single entity or a collection and can contain sub-entities. For instance, in our case, both doctors and patients are modeled as resources, with the latter also having meals, balances, and reports as sub-resources. Under this paradigm, the APIs tend to have a large number of resources and few methods, with the latter belonging to the standard ones such as `List`, `Get`, `Create`, `Update` and `Delete`, or being a custom one. Designers should prefer standard verbs over custom ones, as they map particularly well to HTTP methods. In listing 4.8, a few examples of resource-oriented design applied to our domain are shown.

```
1 // List all patients
2 GET /patients
3
4 // List all the meals of a particular patient
5 GET /patients/1/meals
6
7 // Create a new report for a particular patient
8 POST /patients/1/reports
```

Listing 4.8: A few examples of resource-oriented design applied to our domain

In our work, we employ a slightly different paradigm based on resource-oriented design that also leverages on Express.js middleware to implement authorization effectively. In particular, we build on the concept of sub-routes to compartmentalize the API surface based on the users' roles.

In our domain, three categories of users might use the application: patients,

doctors, and administrators, with the latter being the doctors of the IICB center that have full control over the system. Starting from the idea that administrators can do everything doctors can do, and patients being the only entity capable of submitting monitoring data, we can define a sub-routes structure to incrementally verify the current user's permissions based on the target endpoint, as shown in Fig. 4.3.

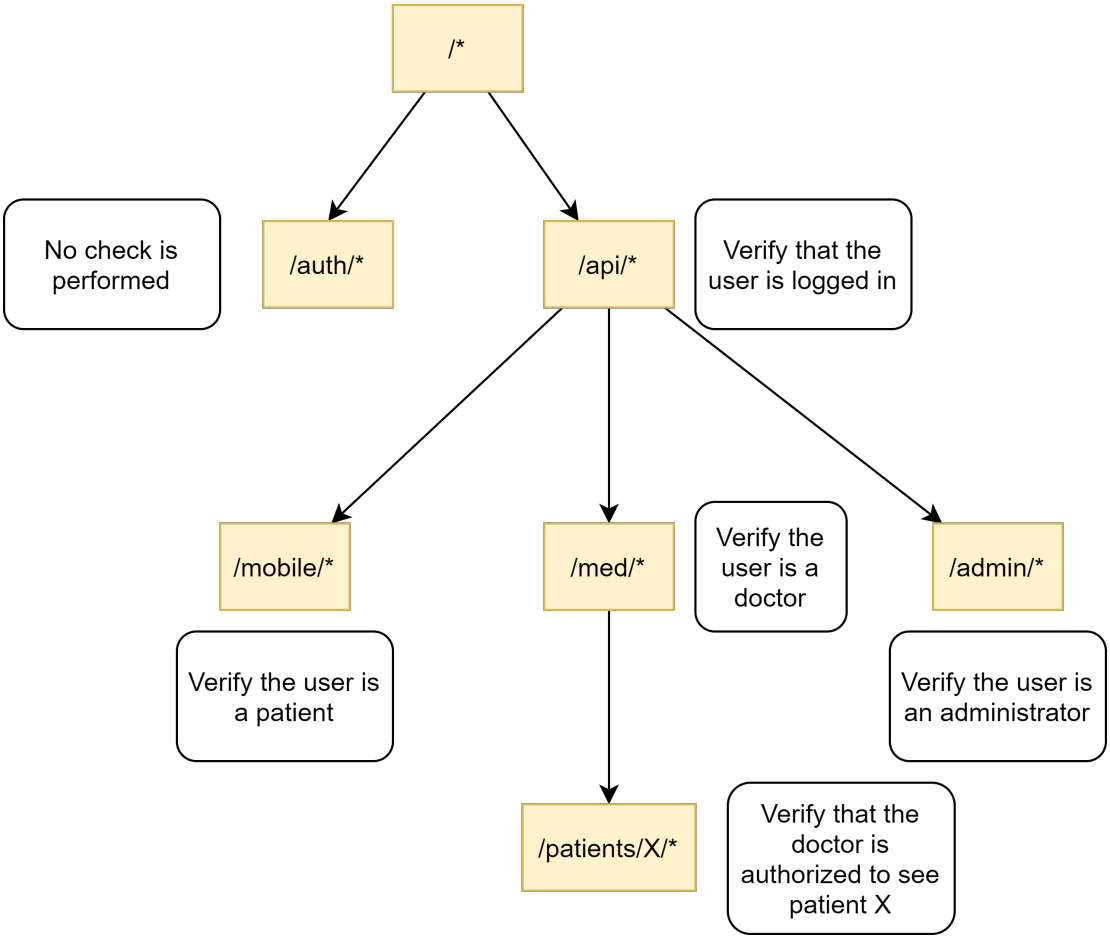


Figure 4.3: Sub-routes structure with authorization routes.

Following the single responsibility principle, each sub-route performs one check only and ensures that all children's endpoints are protected by that policy. For instance, listing 4.9 shows the complete URL to access all reports for a patient with `id=2`. Thanks to the hierarchical policy structure, the system can guarantee that only authorized doctors can access that resource. In particular, the `/api/*` sub-route verifies that the current user is logged in, returning an error otherwise. Then, the `/med/*` sub-route verifies that the currently logged-in user is a doctor. Finally, the `/patients/X/*` sub-route verifies the current doctor is authorized to

see the patient with `id=2`.

```
GET /api/med/patients/2/reports
```

Listing 4.9: API endpoint to access all reports for a particular patient

In practice, this approach is based on a combination of Express.js sub-routes and middleware, as shown in listing 4.10 and 4.11 respectively. In particular, the middleware obtains the user information from the session through the request object and performs a series of checks. In the following example, it verifies whether the user is a doctor and, if so, it obtains its information from the database, saving the result in the `res.locals.doctor` variable. This variable allows subsequent request handlers to obtain the current doctor information without executing another database query. Finally, if all checks are successful, the middleware calls the next handler, continuing the chain. On the other hand, if any of the checks fail, the middleware raises an error, interrupting the chain and preventing the unauthorized user to access the children endpoints.

```
const router = Router()

router.use("/mobile", mobileRoutes)
router.use("/admin", adminRoutes)
router.use("/med", medRoutes)
```

Listing 4.10: Sub-route definition in Express.js

```
// Middleware to check the current user is a doctor
router.use(async (req, res, next) => {
  try {
    if (!req.user) {
      throw new HttpError("missing user information", 403)
    }
    const userInfo = req.user as LoginDoctorInfo
    if (!userInfo.doctorId) {
      throw new HttpError("missing doctor information", 403)
    }

    const doctorInfo = await Doctor.query().findById(userInfo.doctorId)
    if (!doctorInfo) {
      throw new HttpError("doctor not found", 404)
    }

    res.locals.doctor = doctorInfo

    next()
  } catch (err) {
    next(err)
  }
})
```

Listing 4.11: An example of middleware to enforce doctor-only access to the routes

This modular design facilitates authorization enforcement, as once the sub-route access policy is defined, all children endpoint are automatically protected by it without further action.

4.1.6 Report encryption

Patients' reports' are one of the most important and sensitive entities considered in our work, and as such, a substantial amount of effort is dedicated to ensure the correct management of them. According to the project requirements, reports are created and viewed by doctors as PDF files. Therefore, a secure process to upload, store and download them from the app servers while preserving privacy must be defined.

As mentioned in the previous chapters, encryption is often adopted as a way to mitigate risks of data leakage following an attack. In particular, if the stored data is encrypted and the key does not reside in the same place, an attacker needs to compromise multiple systems to read it, complicating the process and thus mitigating the risk. This is especially important when reports are hosted on services like Amazon's S3 buckets, preventing a data leakage if an attacker gains access to them.

In our work, we propose an architecture based on symmetric encryption and hash functions to guarantee a good level of security, privacy, and integrity for patients' reports, as illustrated in Fig. 4.4. For each report, a random key is generated, along with an initialization vector (IV). These are fed to the AES-CBC algorithm to encrypt the file, which is then uploaded to a storage service. The key is also used to generate a signature using the HMAC algorithm. Finally, the key, the initialization vector, the signature, and storage location are saved into the database.

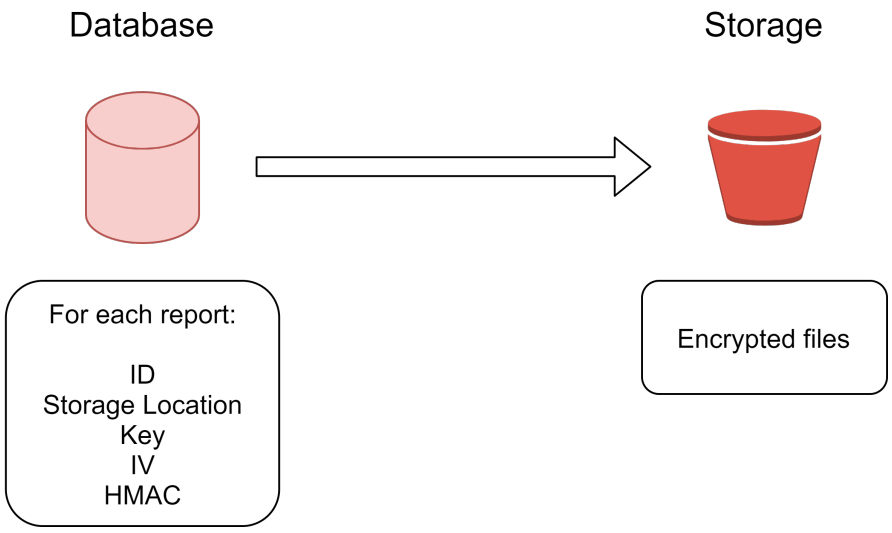


Figure 4.4: Architecture of report management.

Given that reports are stored in an encrypted form on the storage service, when a user needs to download it, the inverse procedure must be followed. The request starts with the report id that is used to retrieve the report information from the database. The server then proceeds to retrieve the encrypted file from the storage service, decrypts it, and finally verifies its integrity using the key. If successful, the server then returns the decrypted report to the user.

Node.js offers several methods to facilitate the use of cryptography in handlers with the `crypto` package. Listing 4.12 shows an excerpt of the report upload handler. In particular, after parsing the input file, the handler generates a random IV-key pair, encrypts the file, and generates the HMAC signature. Then, it stores the files into the storage service, which in this case is a local folder for testing purposes. Finally, the report information is saved into the database and the patient's `lastServerEdit` is updated. The last step is important as it triggers a synchronization for the patient as we described in the previous chapter, making the new report visible on the mobile app.

```
...
// Generate a pair of random IV and key to encrypt the file
const [iv, key] = generateKeyIV()

// Encrypt the file
const encryptedFileData = encryptData(iv, key, uploadedFile.data)

// Generate the HMAC signature
const hmac = crypto.createHmac("sha256", key).update(uploadedFile.data).digest(
    "base64")

// Save the encrypted file to the final destination
await writeFile(finalDestination, encryptedFileData)

// Then finally add the report inside the db, saving the key and IV vector
const report = await Report.query().insert({
  date: new Date(),
  patientId: patientId,
  iv: iv.toString("base64"),
  key: key.toString("base64"),
  hmac: hmac,
  location: finalFilename,
})

await Patient.query().findById(patientId).patch({
  lastServerEdit: new Date()
})
...
```

Listing 4.12: An excerpt of the report upload handler

To conclude this section, a couple of considerations should be made. As previously discussed, the system uses a different key to encrypt each record. This limits the attack surface, as if an attacker manages to find the key of one record through brute-forcing, the other ones are not compromised. Secondly, given the relatively small size of report files, the system performs in-memory encryption and

decryption to limit the implementation complexity. If due to new requirements the expected report size increases, a new buffering mechanism might be necessary.

4.1.7 Mobile synchronization

As discussed in the previous chapter, the mobile synchronization algorithm must deal with data updates. More specifically, given an entry sent by the mobile app, such as a meal or balance, the backend must be able to add the entry if its UUID is not already present or update it otherwise. Semantically, this operation could be implemented using the two-step approach described in the following block:

Algorithm 1: Insert or update an entry from the mobile app

count ← SELECT * FROM entries WHERE *uuid* =?
if *count* > 0 **then**
| UPDATE entries SET *data* =? WHERE *uuid* =?
else
| INSERT INTO entries (data, ...) VALUES (?, ...)
end

In other words, a possible approach is to first check whether the entry exists in the database, and then update it or insert a new one based on the result. A transaction should be used to guarantee the atomicity of the operation, as otherwise, two concurrent operations on the same entries might produce inconsistent results.

This two-step operation is so common that most databases include a specific instruction to handle it, the `UPSERT`. PostgreSQL offers this operation as an extension of `INSERT`, called `INSERT ON CONFLICT DO UPDATE`, which guarantees an atomic `INSERT` or `UPDATE` outcome, even under high concurrency [66]. An example of usage is presented in listing 4.13, which is an excerpt of the mobile synchronization handler. In particular, given the patient id and UUID of the meal, the instruction inserts a new entry if there is no conflict on the patientId-UUID tuple. Otherwise, it updates the date and meal information of the existing entry.

```
1  await trx.raw(`INSERT INTO meals ("patientId", "uuid", "date", "meal")
2    VALUES (:patientId, :uuid, :date, :meal)
3    ON CONFLICT ("patientId", "uuid")
4    DO UPDATE SET ("date", "meal") = (EXCLUDED.date, EXCLUDED.meal)`,
5      {
6        patientId: patient.id,
7        uuid: meal.uuid,
8        date: meal.date,
9        meal: meal.meal,
10     }
11   )
```

Listing 4.13: Using the UPSERT operation as part of the meal synchronization process

In our case, the operation must be executed as part of a transaction, as multiple entries could be updated for each synchronization request, which in turn should be executed as an atomic operation.

4.1.8 Testing

Automatic testing is a crucial part of any robust application development process. Modern software pipelines include several different kinds of tests, ranging from the narrow *unit-tests*, used to verify the behavior of isolated methods or modules, to high-level *functional tests*, exploiting browser automation techniques to simulate user interactions with the application. As a middle ground between the two previous ones, *integration testing* provides a way to test multiple parts of the systems together, also verifying their interactions. For instance, tests connecting to a database belong to this type, as besides the business logic, they also test the integration between the application and the database. In our work, several units and integration tests are used to verify the correct behavior of the application.

Tests are most useful when the environment in which they execute closely resembles the production one, and a particularly important aspect to consider is the choice of database used for testing. Many frameworks openly promote the use of lightweight databases like SQLite during development, as they do not require any pre-existing setup and thus are substantially easier to use. That said, although SQLite supports a reasonable subset of the SQL standard, its feature set is rather limited compared to PostgreSQL and this has several important consequences. Firstly, testing an application on SQLite and deploying it on PostgreSQL force the developer to choose a limited subset of features available on both databases, preventing the use of powerful features only available on PostgreSQL. Secondly, although both databases support a subset of the SQL standard, their behavior in response to the same query might be different, making the application more prone to production bugs.

Because of the above concerns, and the fact that other components, such as Redis, are required to run the application, we need a way to generate a suitable local environment. A popular choice in these scenarios is *Docker compose*, which greatly facilitates the process of defining and running such environments locally, leveraging on containers to execute the different services. In particular, after defining the `docker-compose.yml` file, docker compose automatically takes care of spawning the PostgreSQL and Redis containers. Our testing suite can then connect to these local services to simulate a complete interaction with the REST API.

Another important aspect to consider is that a test's side-effects should not be

visible to the others, or phrased differently, tests should run on independent and clean environments. If this condition is not met, then the test outcomes might become non-deterministic, as each of them might be dealing with spurious database entries created by previous tests. The problem is even more significant when tests are run in parallel, as no ordering constraint is available. Moreover, tests often require mocked data to check their business logic. Both of these requirements are solved using the Knex.js migration and seeding tools. In particular, when the testing phase starts, the PostgreSQL database schema is migrated to the latest version. Then, at the beginning of each test, the database is seeded, that is, all the data is deleted and then repopulated with a clean version of the test data, and the same goes for the Redis store. As a result, the environment in which each test runs is predictable, making the testing suite more robust.

Finally, at the start of the testing phase, two checks are executed to improve the security of the process. In particular, given the potentially disruptive effects of a test suite run on the production database, both the PostgreSQL user and environment are checked, stopping the process if these do not match the testing environment, as shown in listing 4.14.

```
1 // Make sure we are in the testing environment before destroying the database
2 if (config.util.getEnv("NODE_ENV") !== "test") {
3   throw new Error("Attempting to run tests in a non-test environment. Currently
        in " + config.util.getEnv("NODE_ENV"))
4 }
5
6 if ((config.get("DBConfig") as any).connection.user !== "testpostgres") {
7   throw new Error("Attempt to run tests with a non test user. Are you sure you
        are connecting to the right database?")
8 }
```

Listing 4.14: Safety checks executed before the testing suite

4.1.9 Configuration management

Modern applications are characterized by increasingly complex architectures, which in turn require advanced configuration solutions. For instance, most servers need to communicate with multiple services, such as databases or message queues, and in order to do so, they require their location and access information. Moreover, several parameters might be defined, such as the number of database connections or the port on which the server should listen to. More often than not, this information cannot be hard-coded into the application, but instead, it should be determined based on the situation. For example, an application deployed to the test environment will most likely need a different set of parameters compared to the production one.

Solving the above problems requires a *configuration manager*. In particular,

instead of fixing the parameters in the source code, applications rely on the configuration manager to obtain values suitable for the current environment. In other terms, applications delegate the task of choosing parameters to the managers, simplifying the implementation complexity significantly. These managers can take into account a number of sources when choosing the values, ranging from local configuration files to environment variables and command line parameters.

In this context, having a flexible configuration manager is important, as different parameters require different management techniques. For instance, reading parameters from a local file might the best choice in the local development environment, but that approach might raise a number of concerns when applied to production environments. In those cases, a better choice might be to read the parameters from environment variables, which are easy to inject under most orchestration and deployment solutions.

In our work, we choose the `node-config` library to handle configuration management, which is based on the concept of *hierarchical configurations*. In particular, developers first create a JSON configuration file containing the parameters' default values, and then define a new one for each executing environment, such as test and production. All the values that are not explicitly specified for an environment are inherited from the default configuration, which is useful to avoid unnecessary duplicates. Given that these files are included into the project's version control system, the values that are deemed sensitive are omitted, exploiting environment variables to securely serve them to the application instead.

4.2 Mobile implementation

In this section, we present the main implementation details for our mobile app, NAD-APP, which is directly used by patients and caregivers to interact with our telemedicine system. As previously mentioned, the purpose of the application is to support patients in the routine monitoring tasks, as well as helping to deliver medical reports and useful information about the condition.

From a technical standpoint, the application is built using Google's Flutter framework to support both Android and iOS smartphones. Moreover, the Redux paradigm is employed for state management, simplifying the implementation of robust complex dynamics, such as login and synchronization flows. The application also includes a local SQLite database to enable offline usage, as described in the previous chapter. A high-level architectural overview is shown in Fig. 4.5. In particular, the user interacts with Flutter widgets, which in turn could dispatch Redux actions. These go through the various middleware, and based on the situation, they can trigger side-effects, such as a database operation or a net-

work request, and dispatch other actions. Thereafter, reducers update the state according to the received actions, and finally, the state is sent back to Flutter, which updates the views accordingly.

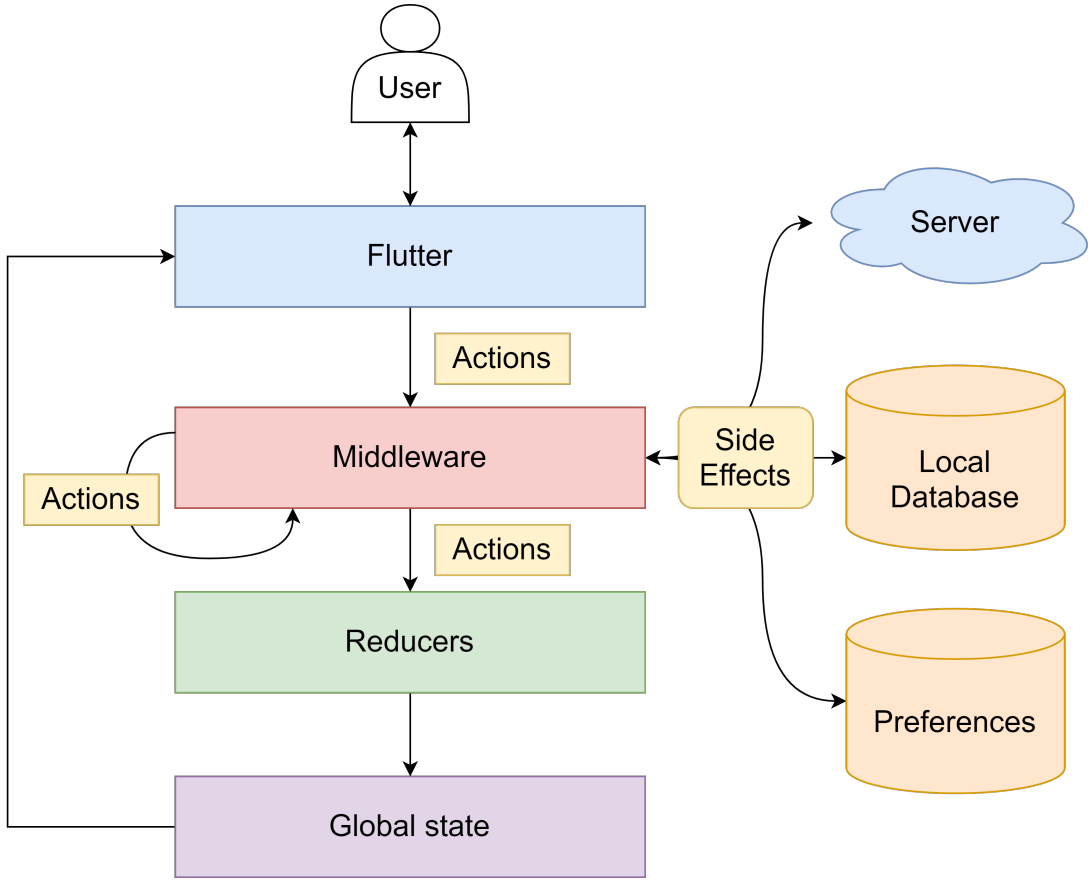


Figure 4.5: NAD-APP mobile application's architecture.

4.2.1 Database

As mentioned in the previous chapter, the application must support offline usage, as patients might not be always connected to the internet when using it. Specifically, the main task carried out with the application is recording a variety of monitoring data, such as meals and fluid balance, to later send it to the IICB center for analysis. Some sort of persistence must be used to satisfy the above requirement, and given the highly structured nature of such data, a database is the ideal solution.

A popular choice in these situations is SQLite, a C-language library that implements a fast and lightweight SQL database engine. Applications that need to

persist and query structured data can use the library to create local databases, and leverage on the SQL language to execute complex queries with ease.

As with the server database, the first step is to define the schema, which in this case is limited to the two tables containing meals and balances respectively, as shown in Fig. 4.6. For the sake of brevity, only the fields of interest are presented. Firstly, although both tables have an integer ID as primary key which locally identifies an entry, it is the unique UUID that globally identifies them when interacting with the server, as we discussed in the previous chapter. In particular, if an entry is deleted and recreated as part of a synchronization, the ID changes, but the UUID remains the same. Secondly, the `dirty` field is used to identifies which entries have been modified since the previous synchronization, and thus must be uploaded. No relation or foreign key is present between the tables.

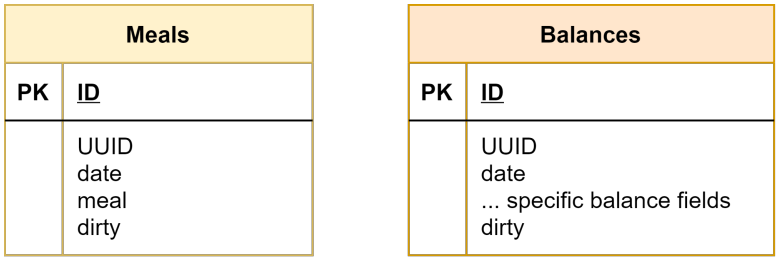


Figure 4.6: NAD-APP mobile application's db schema.

Migrations

As in the server case, when dealing with databases an important aspect to consider is migration management, as if not properly handled, an application update might have disruptive consequences on local data. Flutter's SQLite library offers a simple, albeit effective solution to the problem, based on a similar concept to the one used by the server-side. In particular, migrations are arranged as a list of SQL commands. When the application first connects to the database, the current migration index is confronted with the list, executing all the new migrations, if necessary.

4.2.2 Global state

When designing Redux-based applications, a crucial step is to define the global state representation, or said in other terms, the state model. In our case, it should comprise a number of different aspects, ranging from the current login status to the uploaded diary entries. Instead of designing a single, monolithic

object containing all the required fields for the different areas, we can leverage composition to divide the state into sub-areas, as shown in Fig. 4.7.

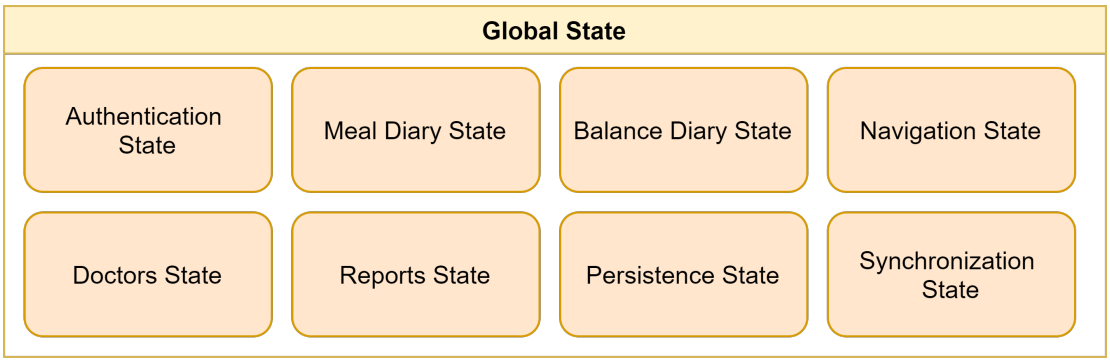


Figure 4.7: NAD-APP global state structure

Each sub-state only manages the fields directly related to its responsibility area and, combined with the others, characterize the global state. In particular, the authentication sub-state stores all the required information necessary for the two-step login process and the user session. The meal and balances sub-states are responsible to keep the recent entries in memory, whereas the persistence sub-state stores the information relative to ongoing saving operations. The navigation sub-state is used to better integrate Redux with Flutter's navigation management. Authorized doctors and medical records are handled by the doctor and report sub-states respectively. Finally, the synchronization sub-state stores the information relative to an ongoing synchronization process, such as the current status and error messages, if present.

According to the Redux paradigm, the global state is never mutated directly, but instead, reducers update it in response to actions, which are either emitted after a user interaction or a middleware side-effect. In the following sections, the main Redux flows are presented, giving a high-level overview of the central app dynamics.

4.2.3 Basic login flow

As previously mentioned in chapter 2, the Redux paradigm is based on the concept of actions, which are first dispatched and then sequentially go through middleware and reducers to perform side-effects and mutate the global state respectively. Consequently, a possible approach to better understand the dynamics of the application is to analyze them in terms of action flows, or in other terms, diagrams presenting how the different actions are dispatched and the resulting effects.

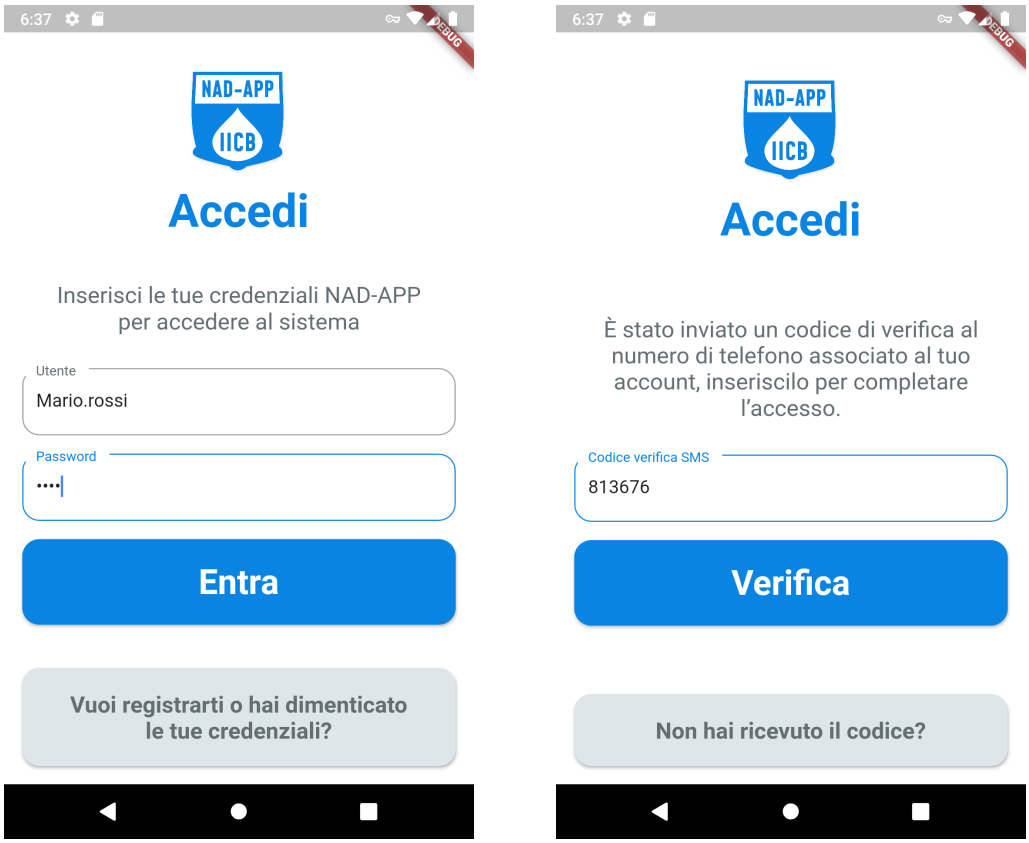
The first complex flow performed by the mobile app is the initial login process, of which a high-level overview is presented in Fig. 4.9. The process starts with the user entering the credentials into the login form, as shown in Fig. 4.8a. Submitting the form dispatches a `LoginRequest` action containing the user credentials. This action first goes through the login middleware, causing a POST request being made to the server API, and a state update when reaching the authorization reducer. If the credentials are correct, the login middleware dispatches a `PhaseOneSuccess` action containing the verificationToken, as discussed in the previous chapter when presenting the two-step SMS authentication process, and causing another state update when reaching the authorization reducer.

Given that state updates are propagated to the view layer, after the previous two actions the user is presented with another form prompting for the SMS verification code, as shown in Fig. 4.8b. In a similar way to the previous phase, submitting the form causes a `VerifyRequest` action being dispatched, which first goes through the login middleware and then reaches the authentication reducer. The middleware aggregates the code from the action payload and the verification token from the previous global state, and sends another POST request to the server API to conclude the two-step protocol. If successful, the login middleware dispatches a `PhaseTwoSuccess` action containing the session token. As soon as this action reaches the preferences middleware, the session token is saved to the device local storage, so that the session is persisted after the application is closed, without requiring another login at the next application launch. As part of a successful result, the login middleware also dispatches the `NavigateTo` and `RequestSync` actions, which bring the user to the main application screen and start the synchronization process.

If an error occurs during the process, for example, due to an invalid combination of credentials and verification code, the login middleware dispatches an appropriate failure action, which in turn updates the global state, and eventually, displays an error message to the user.

4.2.4 SPID login flow

As previously discussed, supporting SPID-based logins is needed to comply with local regulations. Being based on the SAML specification, its implementation is straightforward in the context of web applications, but more complex when dealing with mobile apps. In particular, the protocol is based on a series of HTTP redirects to transfer the identity data between parties, which is not directly applicable to mobile applications, but only to web pages. A possible solution to the above problem is to employ a supporting web service to handle the SAML authentication process and act as a service provider, eventually providing a session



(a) (b)

Figure 4.8: NAD-APP basic login flow screenshots.

token to the mobile application.

Embedded web-views are a common approach to implement such mechanics. In particular, when a SAML-based login is requested, the mobile application creates an embedded web-view in which the supporting web service is loaded, as shown in Fig. 4.10a. Then, the user completes the SPID login in its identity provider of choice, as shown in Fig. 4.10b, which eventually redirects the user back to the supporting web service. The reason behind the use of embedded web-views rather than the default device's web-browser is that in the former case, mobile applications can intercept the complete URLs of the visited pages, whereas that is not possible when using a separate web-browser. This feature is critical to enable communication between the supporting web service and the mobile application. In particular, at the end of a successful login process, the supporting web service generates a one-time token and exposes it as a GET parameter, as shown in listing 4.15.

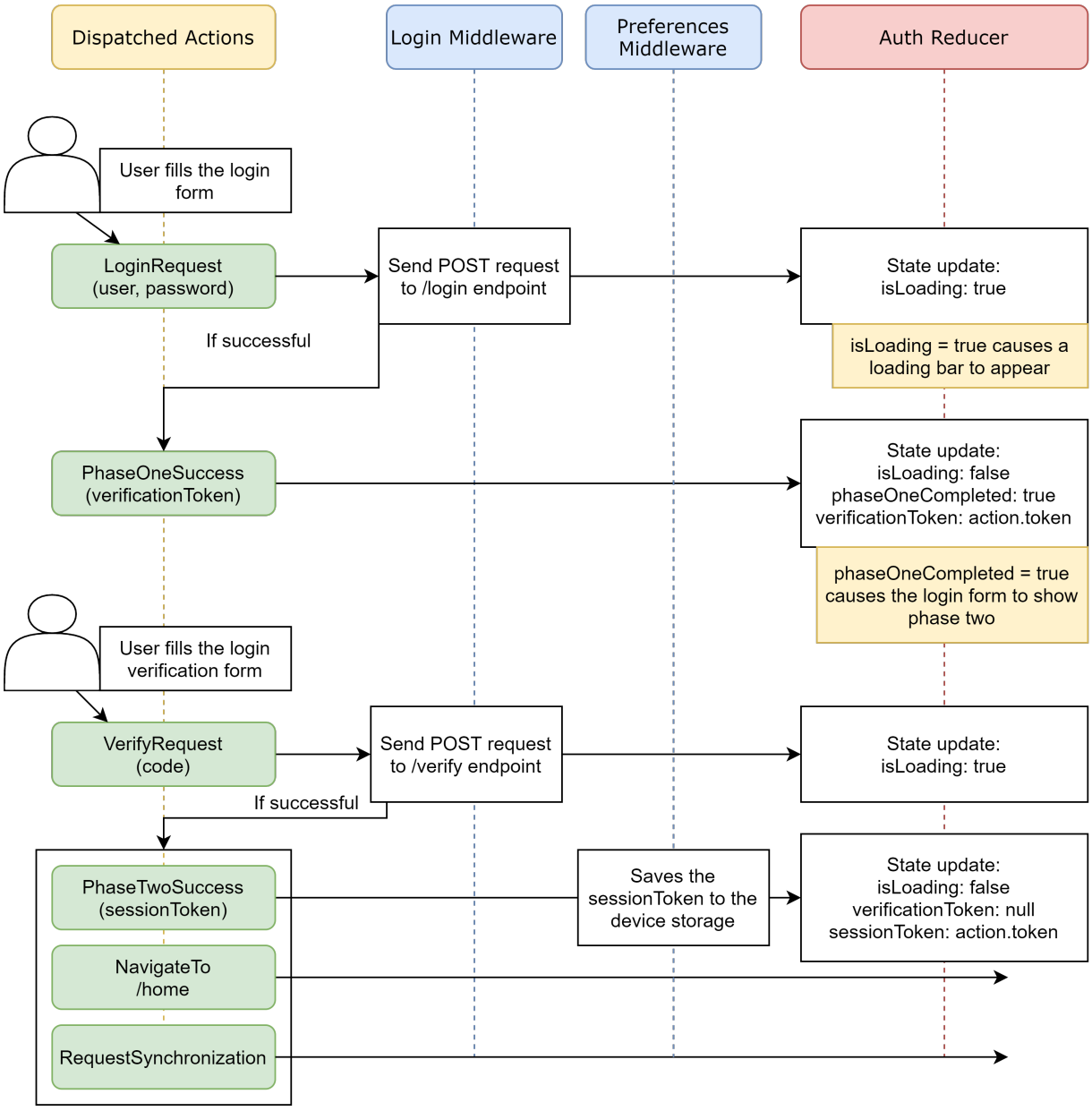


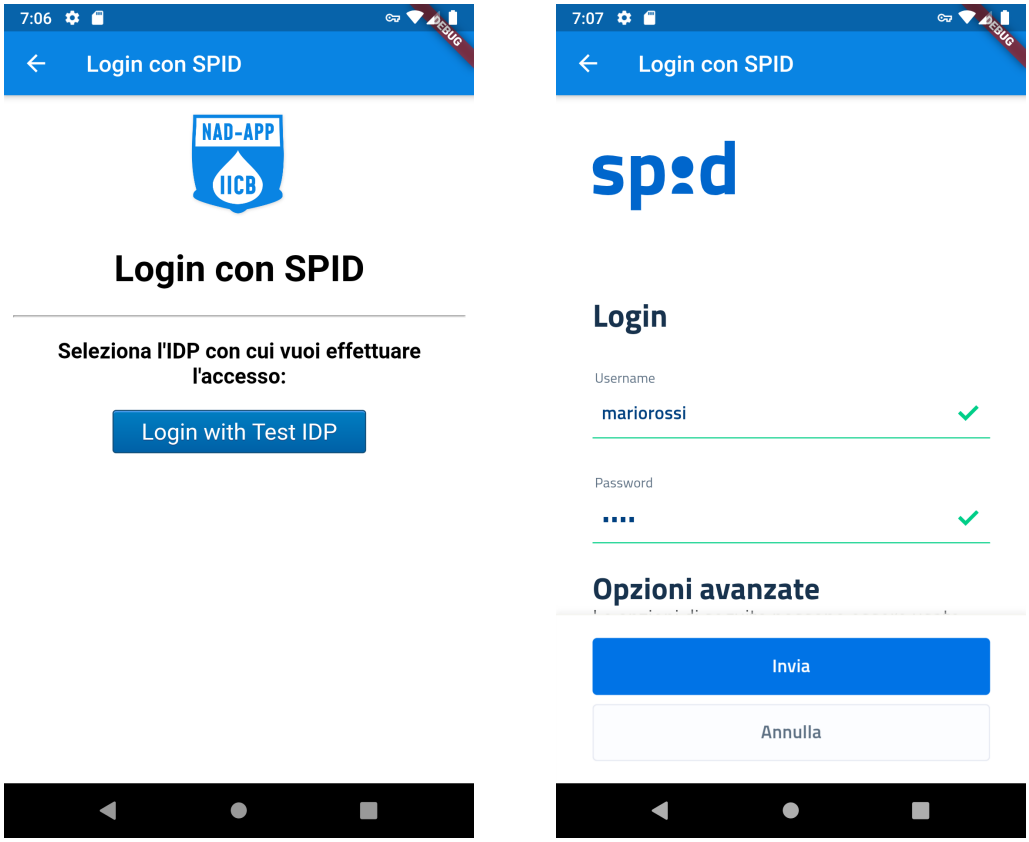
Figure 4.9: High-level overview of the basic login flow in Redux.

```
1 http://nadappserver:8000/spid/success?token=ONE_TIME_TOKEN
```

Listing 4.15: An example of success URL used by the supporting web service

The application can intercept this token by analyzing the visited URLs against a known pattern. Once the token is known, the mobile application executes the Redux flow presented in Fig. 4.11. The flow is relatively similar to the previous

one, although it starts with the `ConvertToken` action, which is used to convert the one-time token into a session token.



(a) (b)

Figure 4.10: NAD-APP SPID login flow screenshots.

4.2.5 New entry flow

Another important Redux flow that is worth discussing is relative to the insertion of a new entry, such as a meal or balance reading, summarized in Fig. 4.12. In this flow, the user dispatches a first action after completing or updating the insertion form, which in turn triggers the saving operation on the SQLite database and updates the state to show a loading bar. After the database operation is completed, the success action is dispatched and causes the state to be updated in multiple areas. In particular, the loading bar is hidden and the new entry is appended to the in-memory list. Finally, a synchronization request is dispatched with the goal of uploading the newly produced entry to the servers.

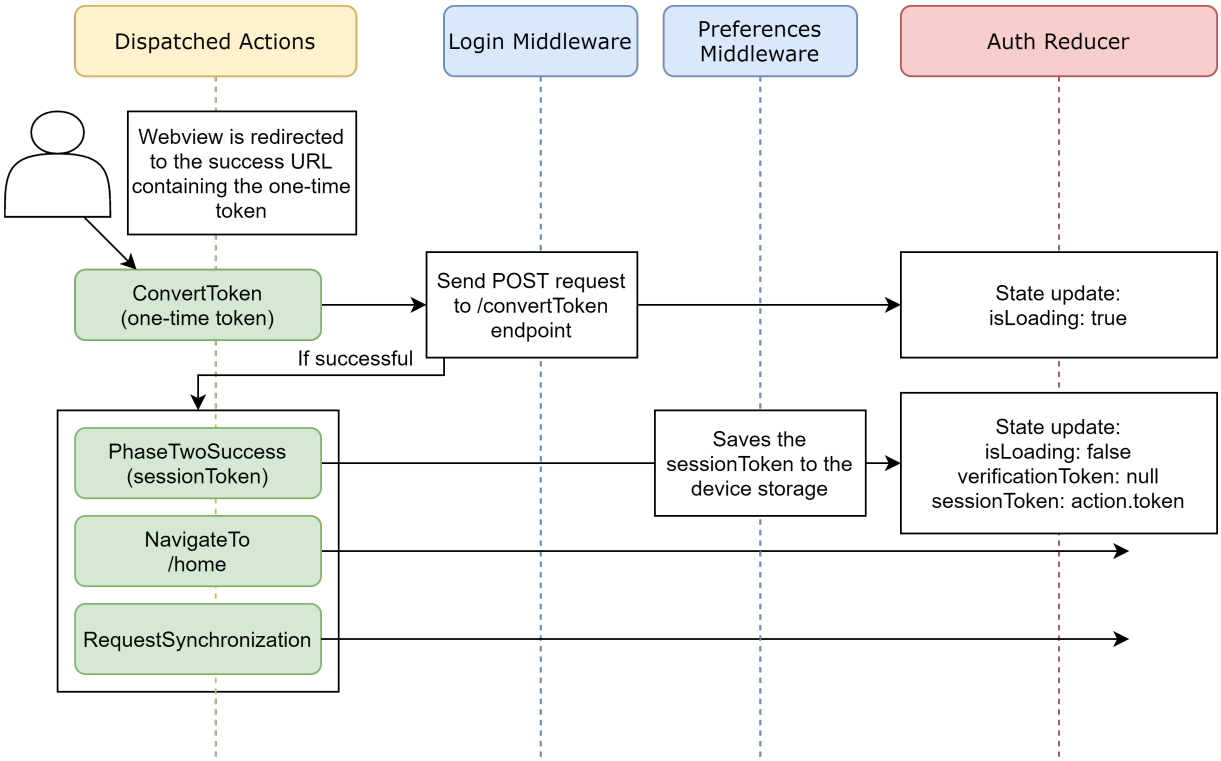


Figure 4.11: High-level overview of the SPID-based login flow in Redux.

4.2.6 Synchronization flow

As discussed in the previous chapters, synchronization plays a crucial role in the functioning of the mobile application. Its Redux flow involves several different actions, middleware, and reducers as most components of the application are involved. A high-level schema is presented in Fig. 4.13, which for the sake of brevity omits some reducers, namely the ones responsible for balances, reports, and doctors.

The process starts with the `RequestSync` action being dispatched, which causes the network middleware to execute a POST request to verify the synchronization status. If already up-to-date, the middleware dispatches a `SyncSuccess` action and the flow ends. Otherwise, a `SyncResponseReceived` action containing the server's response is dispatched, which causes several side effects. In particular, the preferences middleware persists the relevant entries, such as reports and doctors, to the local preferences storage, whereas the SQL middleware updates both the meals and balances on the local database. Eventually, after a series of actions indicating the partial completion of each sub-task, the `SyncSuccess` action is dispatched, terminating the synchronization process.

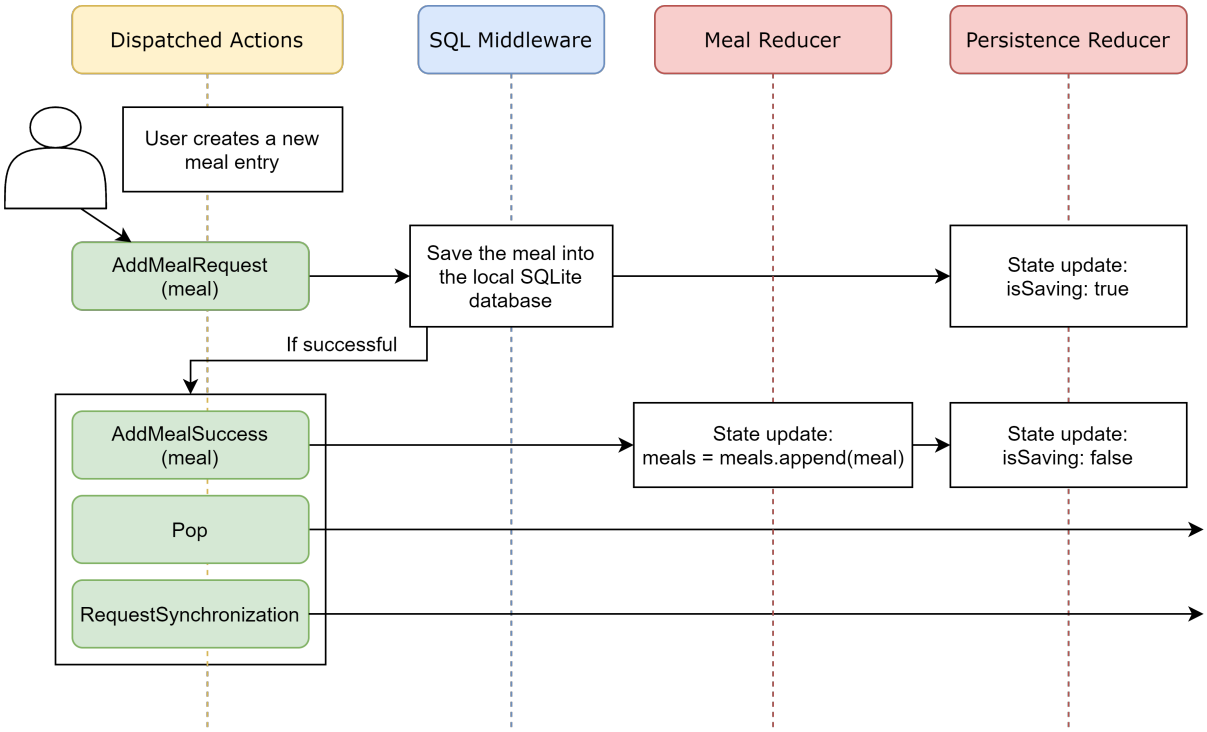


Figure 4.12: High-level overview of the new meal flow in Redux.

If an error occurs during the synchronization process, an appropriate action is dispatched to notify the user. An edge case worth discussing is the 401 status response from the server, which indicates an unauthorized request. In such case, the response is used as a signal for expired user sessions, and as a result, a `LogoutRequest` action is dispatched, prompting the user to authenticate again.

4.2.7 Logout flow

The final Redux flow worth mentioning is the logout flow, which is responsible for clearing the local user data and resetting the application state, if necessary. A brief overview is presented in Fig. 4.14. The logout flow starts with the `LogoutRequest` action, which is either explicitly dispatched by the user or by middleware as a result of an unauthorized synchronization request. This distinction is important, as the choice of clearing the local data should be based on the original intent. In particular, the local database should be cleared if the user explicitly requested a logout, whereas if the action was dispatched as a result of an unauthorized synchronization request, the local data should remain, as otherwise, some entries might not be posted after the new login.

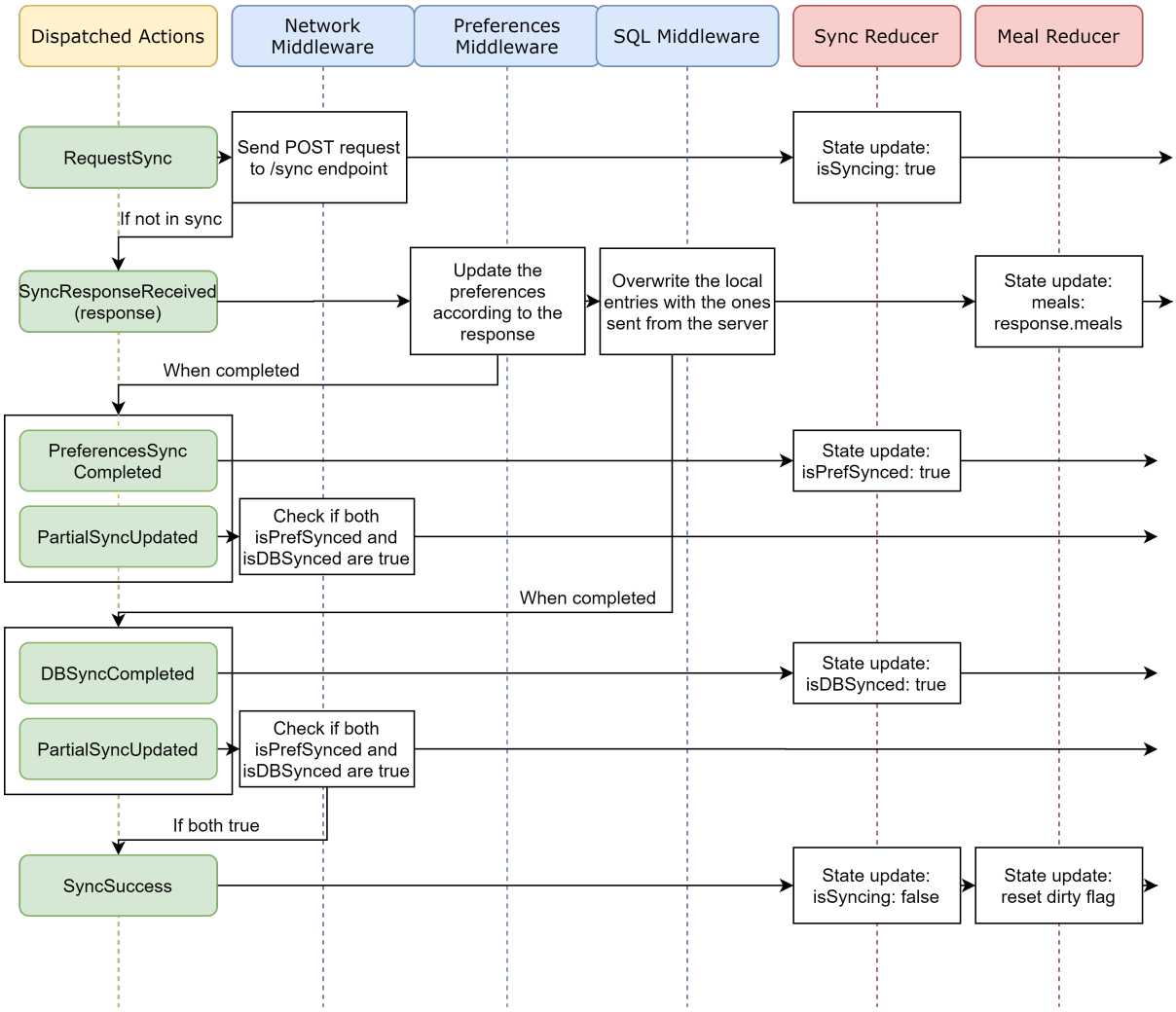


Figure 4.13: High-level excerpt of a successful synchronization flow in Redux. Some reducers are omitted for brevity.

In a similar way to the synchronization flow, the final success action is only dispatched after the involved middleware signal their completion. Notably, the `LogoutSuccess` action causes the root reducer to reset the state to its initial value, as well as navigate the Flutter app to the intro screen.

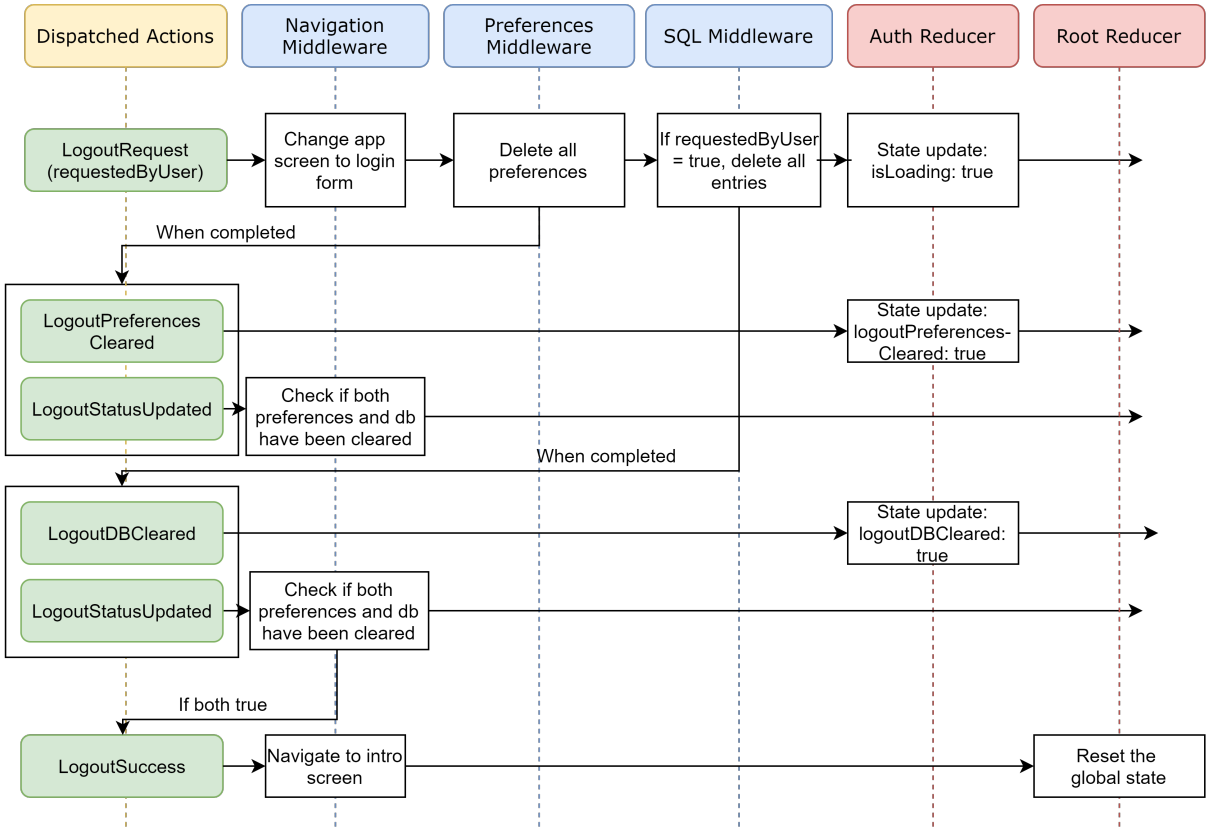


Figure 4.14: High-level overview of the logout flow in Redux.

4.3 Front-end implementation

The doctor-targeted web application is the third and last component of the NAD-APP system left to discuss. As mentioned in the previous chapters, the objective of the web application is to support doctors in the regular monitoring activities, as well as providing an easy way to manage patients' reports. The former can be achieved by conveniently presenting the patients' monitoring data as tables, which resemble the previous paper-based approach, whereas the latter is a simple interface backed by the server's report API discussed in section 4.1.6.

From a technical standpoint, the web application is built using Facebook's React framework for the view layer and Redux as the state management library. Being based on a declarative view pattern combined with the Redux paradigm, the architecture shares some similarities with the mobile application, at least for what concerns the login process. That said, its complexity is significantly lower, as the web application is always considered online, and thus, no offline behavior has to be supported. To further simplify the implementation process, the *Material UI* library, which includes a collection of ready-to-use components for most use-cases, is being used.

4.3.1 Routing

An important topic to discuss when dealing with modern web applications is routing. Traditional solutions are usually characterized by a server handling most of the business logic, which in turn returns HTML pages based on the requested URL and parameters. Being the client-side logic often minimal or non-existent, the only way to request different or additional content is to navigate to another URL, reloading a new page entirely.

Modern web applications greatly diverge from this paradigm, as they are often structured as *Single Page Applications* (SPA). Following this approach, a "fat" client is downloaded during the first request and cached for the successive ones. Most content changes happen without navigating to another page by exploiting asynchronous techniques as AJAX to request new content. This approach enables the creation of arbitrarily complex client-side applications, but it also requires some special techniques to better fit the web paradigm. A common pattern among web applications is to use the URL to locate a specific resource, for instance `example.com/book/10` to access the information page for a book with id=10. Following the traditional approach, this pattern is easily implementable, as a new page must be visited for each request. On the other hand, in SPAs this is not trivial, as the client application is served once and then all the requests happen asynchronously without changing URL.

To solve the above problem, modern web browsers ship with a feature known as *History API*, which provides Javascript methods to manipulate the URL without changing the page. Frameworks can then leverage on this API to facilitate the use of URL into SPAs. For instance, in React these methods are exposed as `Route` nodes, which conditionally render their sub-nodes if the route matches, also accepting parameters if necessary. An example is shown in listing 4.16, in which the different React components are rendered based on the URL. These routes also accept parameters, which are parsed by the library and then provided to the sub-components to customize the logic. For instance, navigating to the `/patients/12` endpoint shows the information page for the patient with id=12.

```
<Switch>
  <Route exact path="/">
    <HomePanel />
  </Route>
  <Route exact path="/patients">
    <PatientsPanel />
  </Route>
  <Route path="/patients/:patientId">
    <PatientPanel />
  </Route>
  <Route exact path="/doctors">
    <DoctorsPanel />
  </Route>
</Switch>
```

Listing 4.16: An excerpt from the NAD-APP web application

An important aspect to consider when dealing with SPAs is that the server must be configured to always serve the same resource regardless of the URL. In other words, if a client first requests a page to the `example.com/` URL and then makes another request to `example.com/home`, the server should return the same resource. The content difference between the two URLs is handled by the client-side application.

4.3.2 Global state and flows

As in the mobile app, the choice of Redux to manage the application state brings a number of consequences. Firstly, the global state structure must be defined and divided into sub-states for each area of responsibility, as shown in Fig 4.15. Compared to the mobile application, the state is significantly simpler due to the lack of synchronization and persistence mechanics.

Also, the Redux flows are easier to handle thanks to the more mature Javascript ecosystem and simpler semantics. In particular, the web application side effects mostly consist of asynchronous HTTP requests used to fetch and upload data. A particularly effective way to handle such effects is the use of *thunks*, an extension

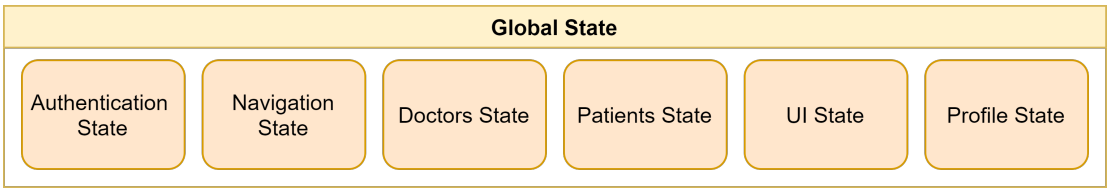


Figure 4.15: Global state structure of the web application.

to the concept of plain Redux action designed for asynchronous tasks. As an example, listing 4.17 illustrates an excerpt of the thunk used to fetch the list of patients.

```
export const fetchPatients = createAsyncThunk(
  "patients/fetchPatients",
  async (_, { dispatch }) => {
    try {
      const response = await axios.get(PATIENTS_ENDPOINT)
      return {
        shortPatientsList: response.data.patients
      } as FetchPatientsListSuccess
    } catch (err) {
      console.log("fetchPatients error:", err.response.data)
      handleRequestFailure(dispatch, err)
      throw err
    }
  })
```

Listing 4.17: An excerpt of the thunk used to fetch the patients list

The above method executes an asynchronous HTTP request and, if successful, returns an object containing the list of patients. If the request fails, an error is thrown. The advantage of this approach is that it automatically dispatches the `pending`, `fulfilled`, and `rejected` sub-actions according to the thunk status, as shown in Fig. 4.16.

4.3.3 Authentication and failure handling

Compared to the mobile app that manually stores the session token on the device storage, the web application exploits cookies to automate the process. In particular, when a session is established, either after an SMS-based login or a SPID one, the server sends a `Set-Cookie` header to persist the token on the client-side. At each subsequent request, the client automatically includes the token as part of the cookie headers, thus not requiring any custom persistence logic from the client.

If an invalid operation is requested to the server, an HTTP error code is returned. The web application can leverage these codes to detect the common edge cases, such as an expired session token or a request with insufficient permissions,

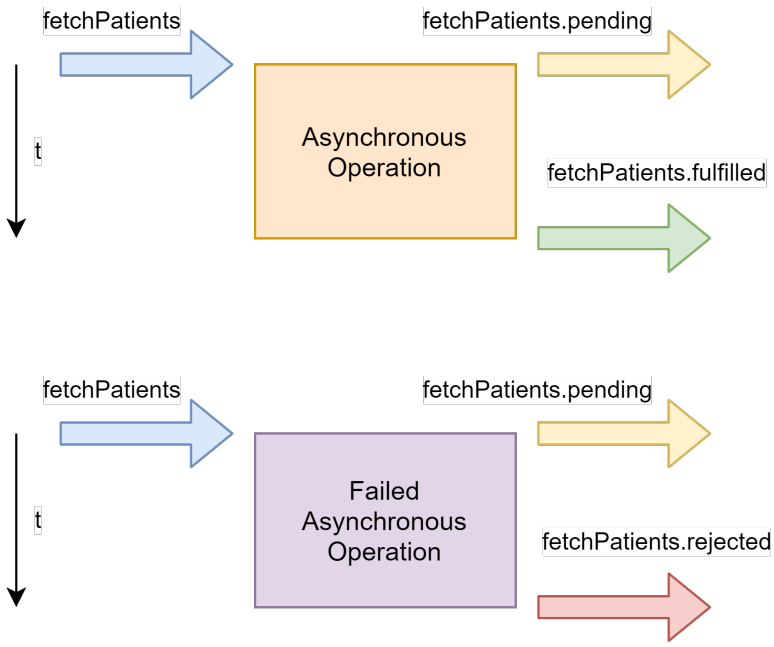


Figure 4.16: Action diagram of a successful thunk (top) and a failed one (bottom)

and inform the user accordingly. In the listing 4.17, the `handleRequestFailure` method is called inside the `catch` clause. This function automatically dispatches the appropriate Redux actions based on the received HTTP error code, and is shown in listing 4.18. In particular, receiving a 401 code means that the session is expired, and therefore a new login is needed. Consequently, the `logoutRequest` action is dispatched, which in turn redirects the user into the login page. The other errors are handled by showing a notification to the user, as shown in Fig. 4.17.

```
export const handleRequestFailure = (dispatch: RootDispatch, error: RequestError)
    => {
  if (error.response) {
    if (error.response.status === 401) {
      dispatch(showSnackbar("Sessione scaduta"))
      dispatch(logoutRequest())
    } else if (error.response.status === 403) {
      dispatch(showSnackbar("Non possiedi i permessi necessari per completare l'
          operazione"))
    } else if (error.response.status === 404) {
      dispatch(showSnackbar("Oggetto non trovato"))
    }
  }
}
```

Listing 4.18: The HTTP code failure handler

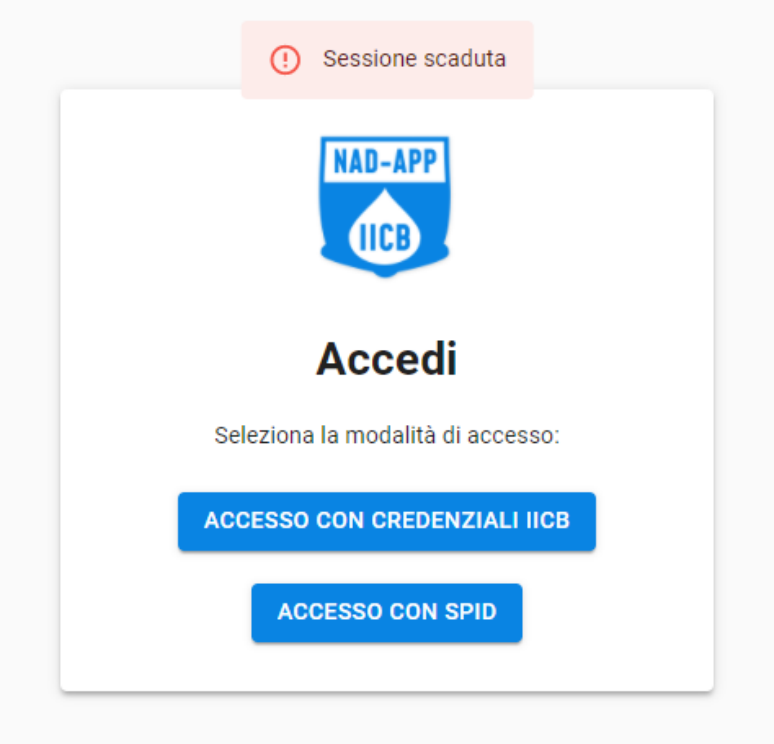


Figure 4.17: When the session expires, the user is prompted to login again.

4.4 Load testing

Load testing is an important phase in the development of web services, and consists in putting the system under a simulated load and measuring its response. In particular, a number of expected user interactions with the system are modeled and then simulated to see how well the web service is capable of handling them. The objective of load testing is not to verify the correct behavior of the services, which is a responsibility of unit and functional tests, but instead to measure the *Quality of Service* under varying workloads. Moreover, it is a valuable tool to identify the main system bottlenecks.

The quality of the simulated user interactions is crucial to obtain meaningful reports. Specifically, the artificial load should resemble the real one as much as possible, as otherwise, the identified bottlenecks might not be relevant in the production environment.

This section describes the load testing setup for our work and focuses on the critical endpoints of the API. In particular, the objective is to determine whether our system is capable of withstanding the expected number of users, which eventually should be around two thousand. For the sake of brevity, only the login and

synchronization endpoints are tested, as they perform the most complex operations within our system.

The load testing is performed on a computer with 16gb of RAM and a 6-core Intel i5 processor. Moreover, the server runs as a single-threaded Node.js process, with both PostgreSQL and Redis running on the same machine. Finally, *Artillery.io*, a library specifically designed for load testing, is used to define and simulate the user interactions.

4.4.1 Login

Our testing process starts with the login endpoints. As we discussed in the previous chapters, the system support two different strategies to authenticate users: the two-factor login and the SPID-based one. Given that most users are expected to use the former, that is where our testing is focused.

The tests are structured in three phases. The first is a warming up phase, characterized by an arrival rate of 5 users per second and lasting for 60 seconds. The second is a ramping up phase, lasting for 120 seconds and linearly increasing the arrival rate to 50 users per second. Finally, a third phase lasting for 120 seconds is used to test a sustained load of 50 new users per second.

The interaction is modeled after the expected login workload, which as we previously discussed, is divided into two phases. The simulated users first send a pair of credentials to the `/login` endpoint, receiving the verification token. They wait for a couple of seconds and then send the verification code to the `/verify` endpoint, concluding the login flow.

The test is then run through Artillery to generate a report, which is presented in Fig. 4.18. Before discussing the results, it is important to explain the meaning of the different graphs. The first shows the requests' latency in milliseconds as time passes, whereas the second graph indicates the number of concurrent users connected to the system at any given time. Finally, the third graph is a plot of the average number of requests per second as time passes.

Analyzing the graphs, we can conclude that the current system configuration is capable of handling 10 requests per second with a latency under 200 milliseconds, but fails to scale to 80 requests per second, at which point the latency reaches over 60 seconds. Although handling 10 RPS is more than enough for our use-case, it is interesting to discuss the reasons behind these scalability issues. As we presented in chapter 3, the login endpoint uses the Bcrypt algorithm to verify the password hashes. This algorithm is specifically designed to be slow to compute, making it

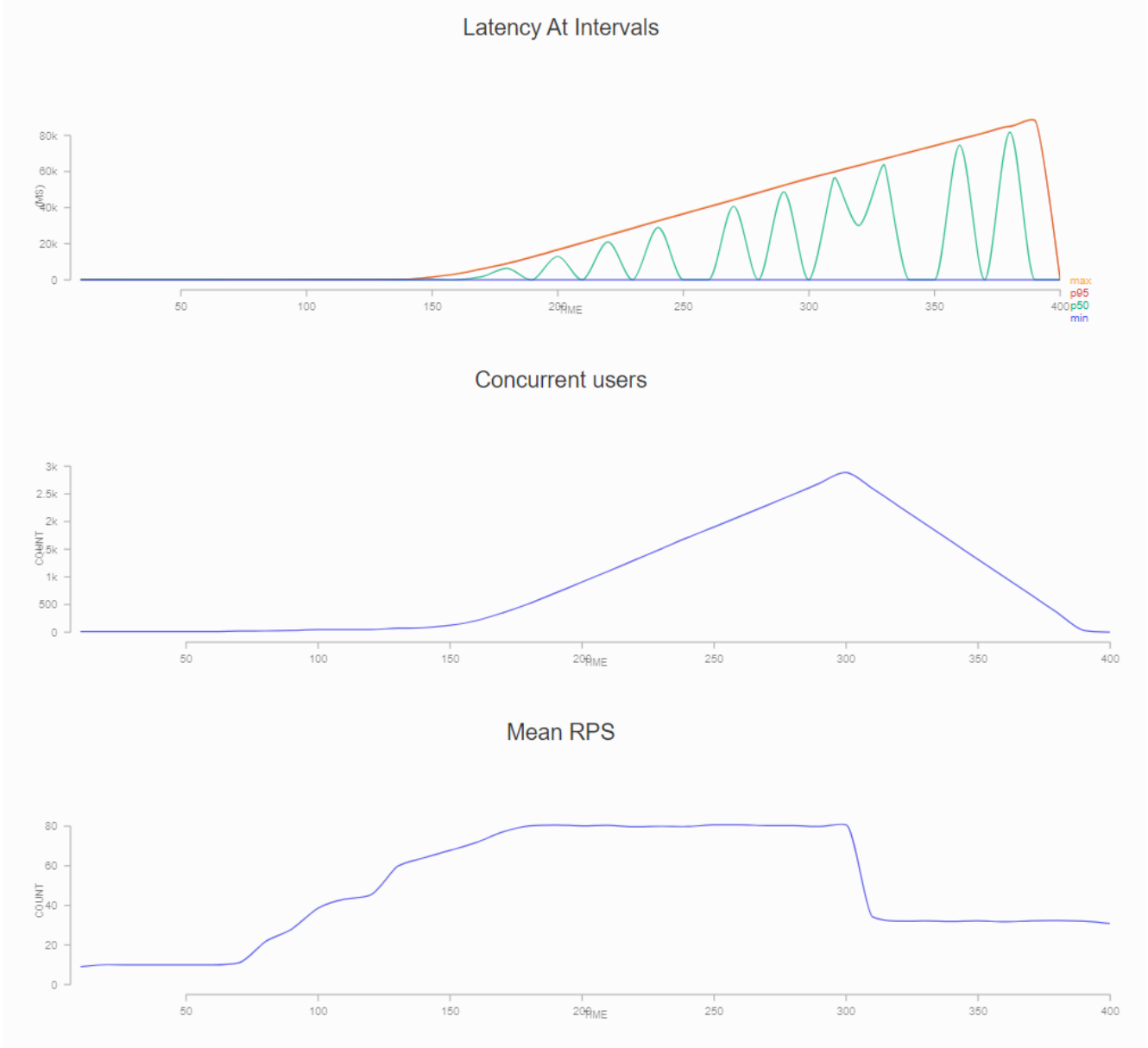


Figure 4.18: Load testing results for the login endpoint.

more resistant to brute-force attacks. As a result, the server's responsiveness is reduced, as each login attempt consumes about 100 milliseconds of processing time. If the number of requests per second exceeds 10, it means that the server cannot keep up with the demand, and requests start being queued, linearly increasing the response times. In our tests, when the load reaches 80 RPS, the server is well over its capabilities, therefore the requests cannot be served in a timely manner anymore.

Another non-obvious behavior can be seen in the green line on the first graph. That line represents the median response time, and as it can be seen, it oscillates between low and high values. That seemingly strange behavior can be explained by considering the two-phase nature of the login process. The first phase is characterized by a heavy CPU load due to the Bcrypt computation, or in other words, it is CPU-bound. On the other hand, the second verification phase is IO-bound, as its task is to retrieve the verification code and token from the Redis database and verify whether it matches with the ones provided in the request, making it significantly faster to process. Consequently, the median response time oscillates between the high values returned from the first login phase and the low values returned by the second one.

As previously said, being able to handle 10 RPS is enough given our current use-case. That said, if the requirements change and the expected number of users increases, the easiest solution is to rely on horizontal scaling to distribute the CPU load across several Node.js instances. In particular, given that the web server does not hold any state, it can be easily replicated across a number of servers and placed behind a load balancer. In such scenario, the number of login requests per second could increase linearly with the number of servers, as the computation time is constant and distributed into multiple machines. The previous assumption should hold until the Redis and load balancer limitations are reached.

4.4.2 Synchronization

The second endpoint worth considering is the synchronization one used by the mobile application, given its critical role and significant processing load. The test is structured in three phases as the previous one, but given the different workloads that this endpoint might serve, both the read-only and read-write scenarios are tested. The results are shown in Fig. 4.19 and Fig. 4.20.

Analyzing the graphs, we can see how the service is capable of handling more than 50 requests per second with a maximum latency of 80 ms. As expected, the read-only scenario is slightly faster than the one including writes, although the difference is negligible in practice.

From our testing, we can conclude that the current synchronization endpoint is capable of withstanding a load well above the required one, thus no particular optimization is needed.

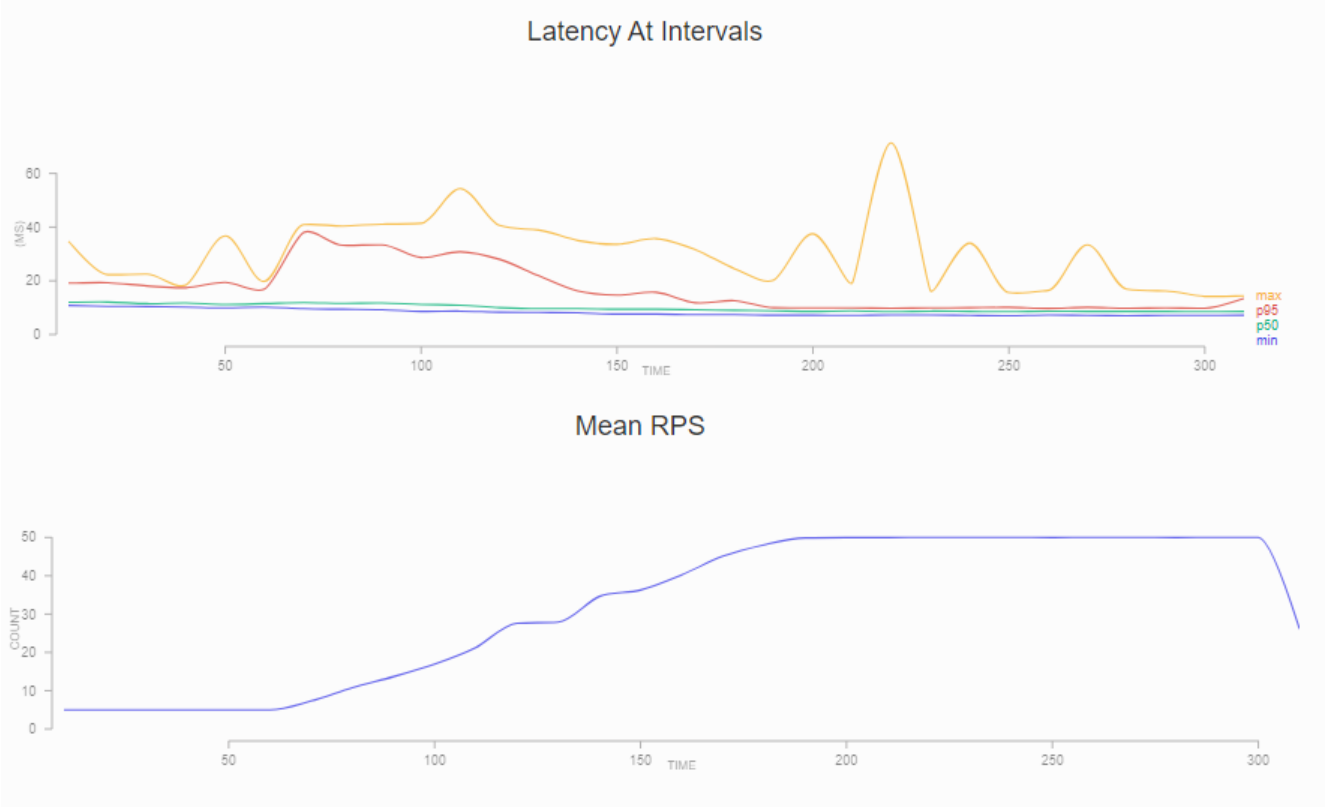


Figure 4.19: Load testing results for the synchronization endpoint with read-only workload

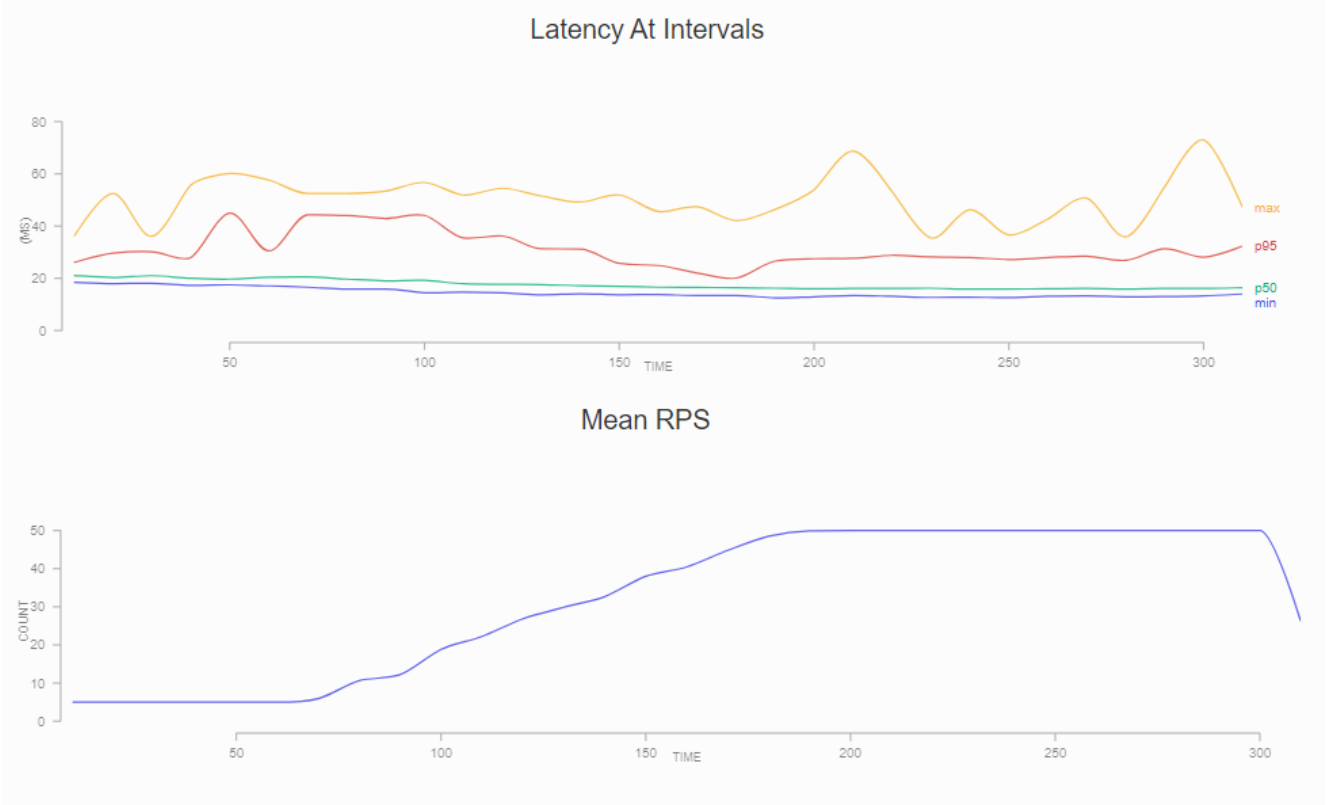


Figure 4.20: Load testing results for the synchronization endpoint with mixed workload

4.5 Achieved results

The proposed system enables patients to conveniently share monitoring data and reports with doctors, facilitating the routine diagnosis and control process required to guarantee the well-being of those suffering from intestinal failure.

Patients are provided with a mobile application, NAD-APP, which can be used to record the monitoring data, as shown in Fig. 4.21. According to the requirements, the application stays functional also when the device is offline and synchronizes with the server as soon as a connection is available. After a successful synchronization, the monitoring data can be analyzed by authorized doctors in the correspondent web application panel, as shown in Fig. 4.22.

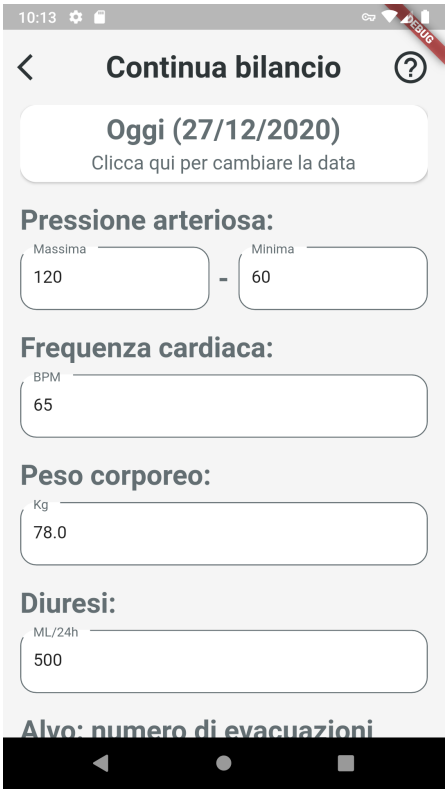


Figure 4.21: New balance form on the NAD-APP mobile application

Doctors are provided with a web application that conveniently displays patient information, monitoring data, and reports in a unified place, facilitating doctor-patient communication, and thus, the patients' quality of life.

Finally, the system supports two secure authentication methods to provide the optimal balance between convenience and security. Moreover, the load testing process provided a strong indication that the system is capable of handling the

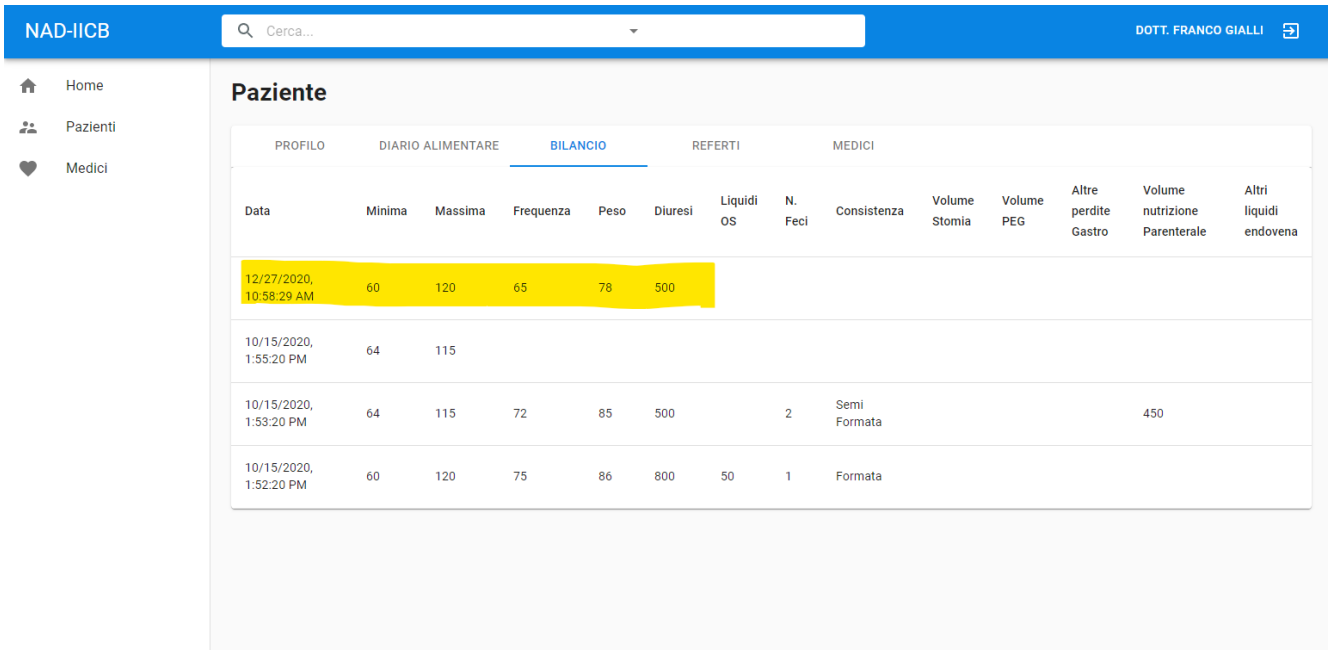


Figure 4.22: New balance appearing on the doctors' web application

expected load without problems.

Conclusion

In this work, we presented a novel telemedicine system aimed at supporting patients suffering from intestinal failure. We first discussed telemedicine in general, starting from the traditional methods and leading to some of the state-of-the-art techniques. Secondly, we discussed how cloud technologies could be used in a medical setting, covering the economical and technical advantages, the privacy concerns and a number of possible architectures. In the second chapter, we discussed the various technologies employed in our work, focusing on their main characteristics, as well as the motivations behind their use. The third chapter was dedicated to the main architectural choices, such as the database schema, the mobile's synchronization protocol and the authentication strategies. These concepts were presented from a high-level perspective, deferring the implementation details to the fourth and last chapter. Specifically, chapter four is about the actual implementation process of our system, composed by the server, the mobile app and the web application. After a throughout discussion of the three components, a load testing process was performed to verify whether our system is capable of withstanding the expected load. Our experiments highlighted how the proposed system is capable of satisfying the project requirements, delivering an application that can effectively improve the doctor-patient communication, and thus, the patient's quality of life.

From a technical standpoint, the proposed mobile application is capable of remaining functional even in challenging situations, for example, when an internet connection is not immediately available. This is possible thanks to the proposed synchronization protocol, which automatically uploads the monitoring data as soon as a connection becomes available. To fulfill this goal, the protocol has to deal with various challenges, such as generating globally unique identifiers while offline and resolving conflicts that might arise when two concurrent and distributed updates are performed, all while limiting the amount of bandwidth consumed. Moreover, we also proposed a possible approach to solve the SAML authentication problem on mobile applications, which required several custom components to be handled correctly. On the server side, our work focuses on providing a scalable

service that is also capable of handling sensitive data securely. An example is presented when discussing the report encryption mechanism, which makes the system less vulnerable to data leaks.

Given the number of possible improvements, the following sections are dedicated to discussing the most promising extensions to our system.

The IICB center routinely asks patients to fill a quality of life questionnaire both for diagnostics and statistical purposes. Given that the current questionnaire is paper-based, an integration with the proposed system could offer several benefits, both in terms of usability and convenience. In particular, patients could easily fill the form using their smartphones, whereas doctors could analyze the results as a unified dataset.

Given the kind of data that patients are asked to record, a possible improvement could be to integrate the mobile application with wearable sensors, which could automatically record useful information, such as the heart frequency. Besides the usability advantages, this approach would enable the system to track critical values multiple times throughout the day, which would be impractical if done manually.

As part of the ongoing research to improve the health conditions of those suffering from intestinal failure, gathering medical data is critical to support new studies. A possible extension would be to automatically extract the data submitted by patients through NAD-APP, remove any personal information and merge it into a single dataset. Such effort could help to expand the IICB center's dataset, already considered the biggest in this area, which proved important to support the previous studies on the condition.

As part of the treatment for intestinal failure, patients are given a list of symptoms and value thresholds that must be carefully tracked to guarantee their well-being. If a listed symptom appears, or a value falls outside the safe range, the patient should contact the IICB center, either immediately or in a few days based on the specific case. Given that a good portion of these warning conditions are relative to the values inserted into NAD-APP, a possible improvement could be to design a system that automatically notifies the patients when these conditions are met, suggesting to call the IICB center for further assistance. This system could also be paired with modern machine-learning techniques to automatically figure out dangerous patterns in values, perhaps combining the readings with the ingested food.

Bibliography

[1] ScienceDirect. Tele-monitoring. https://www.sciencedirect.com/topics/nursing-and-health-professions/telemonitoring.

[2] Douglas Perednia. Telemedicine technology and clinical applications. *JAMA: The Journal of the American Medical Association*, 273:483, 02 1995. doi: 10.1001/jama.1995.03520300057037.

[3] Patterson V. Wootton R., Craig J. *Introduction to Telemedicine, second edition*. London: CRC Press.

[4] Nicole Lurie and Brendan G. Carr. The Role of Telehealth in the Medical Response to Disasters. *JAMA Internal Medicine*, 178(6):745–746, 06 2018. ISSN 2168-6106. doi: 10.1001/jamainternmed.2018.1314. URL https://doi.org/10.1001/jamainternmed.2018.1314.

[5] Judd E. Hollander and Brendan G. Carr. Virtually perfect? telemedicine for covid-19. *New England Journal of Medicine*, 382(18):1679–1681, 2020. doi: 10.1056/NEJMp2003539. URL https://doi.org/10.1056/NEJMp2003539.

[6] F Amenta, A Dauri, and N Rizzo. Organization and activities of the international radio medical centre (cirm). *Journal of Telemedicine and Telecare*, 2(3):125–131, 1996. doi: 10.1258/1357633961929907. URL https://doi.org/10.1258/1357633961929907. PMID: 9375045.

[7] Jaclyn Gaydos. The audio-visual connection: A brief history of telemedicine. *Today's Wound Clinic*, 13(4):26–29, 2019. URL https://www.todayswoundclinic.com/articles/audio-visual-connection-brief-history-telemedicine.

[8] *Telemedicine Technologies*. John Wiley and Sons, Ltd, 2010. ISBN 9780470972151. doi: 10.1002/9780470972151.fmatter. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470972151.fmatter.

[9] Richard Satava. 4.9. telemedicine and real-time monitoring of climbers. *Current problems in dermatology*, 32:141–7, 02 2003. doi: 10.1159/000067366.

[10] Ronald S. Weinstein, Ana Maria Lopez, Bellal A. Joseph, Kristine A. Erps, Michael Holcomb, Gail P. Barker, and Elizabeth A. Krupinski. Telemedicine, telehealth, and mobile health applications that work: Opportunities and barriers. *The American Journal of Medicine*, 127(3):183 – 187, 2014. ISSN 0002-9343. doi: https://doi.org/10.1016/j.amjmed.2013.09.032. URL `http://www.sciencedirect.com/science/article/pii/S0002934313009194`.

[11] Guy Paré, Mirou Jaana, and Claude Sicotte. Systematic Review of Home Telemonitoring for Chronic Diseases: The Evidence Base. *Journal of the American Medical Informatics Association*, 14(3):269–277, 05 2007. ISSN 1067-5027. doi: 10.1197/jamia.M2270. URL `https://doi.org/10.1197/jamia.M2270`.

[12] Enea Parimbelli, Barbara Bottalico, E. Losiouk, Marta Tomasi, Amedeo Santosuosso, G. Lanzola, Silvana Quaglini, and R. Bellazzi. Trusting telemedicine: A discussion on risks, safety, legal implications and liability of involved stakeholders. *International Journal of Medical Informatics*, 112, 04 2018. doi: 10.1016/j.ijmedinf.2018.01.012.

[13] Giulio Nittari, Ravjyot Khuman, Simone Baldoni, Graziano Pallotta, Gopi Battineni, Ascanio Sirignano, Francesco Amenta, and Giovanna Ricci. Telemedicine practice: Review of the current ethical and legal challenges. *Telemedicine and e-Health*, 02 2020. doi: 10.1089/tmj.2019.0158.

[14] Claudine Meijer, Bram Wouterse, Johan Polder, and Marc Koopmanschap. The effect of population aging on health expenditure growth: A critical review. *European Journal of Ageing*, 10, 12 2013. doi: 10.1007/s10433-013-0280-x.

[15] Z. Jin and Y. Chen. Telemedicine in the cloud era: Prospects and challenges. *IEEE Pervasive Computing*, 14(1):54–61, 2015. doi: 10.1109/MPRV.2015.19.

[16] A. Esposito. Analisi valutativa dell'adozione del cloud per il fascicolo sanitario elettronico. 2013. doi: RT-ICAR-NA-2013-1.

[17] Ofir Ben-Assuli. Electronic health records, adoption, quality of care, legal and privacy issues and their implementation in emergency departments. *Health policy (Amsterdam, Netherlands)*, 119, 11 2014. doi: 10.1016/j.healthpol.2014.11.014.

[18] Neil Fleming, Steven Culler, Russell McCorkle, Edmund Becker, and David Ballard. The financial and nonfinancial costs of implementing electronic health records in primary care practices. *Health affairs (Project Hope)*, 30: 481–9, 03 2011. doi: 10.1377/hlthaff.2010.0768.

[19] A. Attila, Á. Garai, and I. Péntek. Common open telemedicine hub and infrastructure with interface recommendation. In *2016 IEEE 11th International*

Symposium on Applied Computational Intelligence and Informatics (SACI), pages 385–390, 2016. doi: 10.1109/SACI.2016.7507407.

[20] Naipeng Dong, Hugo Jonker, and Jun Pang. Challenges in ehealth: From enabling to enforcing privacy. 09 2018.

[21] Zhifeng Xiao and Yang Xiao. Security and privacy in cloud computing. *Communications Surveys and Tutorials, IEEE*, 15:843–859, 01 2013. doi: 10.1109/SURV.2012.060912.00182.

[22] ENISA. Cloud computing security risk assessment. 2009.

[23] K. Lee. Security threats in cloud computing environments. *International Journal of Security and its Applications*, 6:25–32, 01 2012.

[24] VMWare. Cloud economics. `https://www.vmware.com/topics/glossary/content/cloud-economics`.

[25] Liang Xue, Yong Yu, Yannan Li, Man Ho Au, Xiaojiang Du, and Bo Yang. Efficient attribute-based encryption with attribute revocation for assured data deletion. *Information Sciences*, 479, 02 2018. doi: 10.1016/j.ins.2018.02.015.

[26] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5): 78–81, 2016. doi: 10.1109/MC.2016.145.

[27] Ammar Awad Mutlag, Mohd Khanapi Abd Ghani, N. Arunkumar, Mazin Abed Mohammed, and Othman Mohd. Enabling technologies for fog computing in healthcare iot systems. *Future Generation Computer Systems*, 90:62 – 78, 2019. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.07.049. URL `http://www.sciencedirect.com/science/article/pii/S0167739X18314006`.

[28] Z. Ma, J. Ma, Y. Miao, X. Liu, K. R. Choo, R. Yang, and X. Wang. Lightweight privacy-preserving medical diagnosis in edge computing. *IEEE Transactions on Services Computing*, pages 1–1, 2020. doi: 10.1109/TSC.2020.3004627.

[29] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016. doi: 10.1109/JIOT.2016.2579198.

[30] Danton S. Char, Michael D. Abràmoff, and Chris Feudtner. Identifying ethical considerations for machine learning healthcare applications. *The American Journal of Bioethics*, 20(11):7–17, 2020. doi: 10.1080/15265161.2020.1819469. URL `https://doi.org/10.1080/15265161.2020.1819469`. PMID: 33103967.

[31] Jakub Konecný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *CoRR*, abs/1610.02527, 2016. URL http://arxiv.org/abs/1610.02527.

[32] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol.*, 10 (2), January 2019. ISSN 2157-6904. doi: 10.1145/3298981. URL https://doi.org/10.1145/3298981.

[33] L. Pironi, M. Candusso, A. Biondo, A. Bosco, P. Castaldi, F. Contaldo, E. Finocchiaro, A. Giannoni, S. Mazzuoli, P. Orlandoni, A. Palozzo, C. Panella, S. Pastò, E. Ruggeri, G. Sandri, E. Stella, and G. Toigo. Prevalence of home artificial nutrition in italy in 2005: A survey by the italian society for parenteral and enteral nutrition (sinpe). *Clinical Nutrition*, 26(1):123 – 132, 2007. ISSN 0261-5614. doi: https://doi.org/10.1016/j.clnu.2006.07.004. URL http://www.sciencedirect.com/science/article/pii/S0261561406001397.

[34] Loris Pironi, Jann Arends, Janet Baxter, Federico Bozzetti, Rosa Peláez, Cristina Compés, Alastair Forbes, Simon Gabe, Lyn Gillanders, M. Holst, Palle Jeppesen, Francisca Joly, Darlene Kelly, Stanislaw Klek, Øivind Irtun, Steven WM Olde Damink, Marina Panisić, Henrik Højgaard Rasmussen, Michael Staun, and Jon Shaffer. Espen endorsed recommendations. definition and classification of intestinal failure in adults. *Clinical Nutrition*, 34, 09 2014. doi: 10.1016/j.clnu.2014.08.017.

[35] Gazzetta Ufficiale delle Repubblica Italiana. Decreto legge 16 luglio 2020 n. 76 - misure urgenti per la semplificazione e l'innovazione digitale. https://www.gazzettaufficiale.it/eli/gu/2020/07/16/178/so/24/sg/pdf.

[36] Statista. Mobile operating system market share. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, 2020.

[37] Flutter. Flutter architecture. https://flutter.dev/docs/resources/architectural-overview.

[38] Dart. Dart homepage. https://dart.dev/.

[39] React Native. Core components and native components. https://reactnative.dev/docs/intro-react-native-components.

[40] Dan Abramov and the Redux documentation authors. Redux fundamentals. https://redux.js.org/tutorials/fundamentals/part-1-overview, .

[41] Dan Abramov and the Redux documentation authors. Redux motivation. https://redux.js.org/understanding/thinking-in-redux/motivation, .

[42] Dan Abramov and the Redux documentation authors. Redux principles. https://redux.js.org/understanding/thinking-in-redux/three-principles, .

[43] Dan Abramov and the Redux documentation authors. Redux middleware. https://redux.js.org/understanding/history-and-design/middleware, .

[44] Cristopher López Santana. Middleware on redux. https://dev.to/ramclen/understanding-middlewares-on-redux-1912.

[45] OpenJS Foundation. Nodejs homepage. https://nodejs.org/en/, .

[46] OpenJS Foundation. Nodejs about. https://nodejs.org/en/about/, .

[47] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010. doi: 10.1109/MIC.2010.145.

[48] Stavros Papastavrou, George Samaras, Paraskevas Evripidou, and Panos Chrysanthis. Fine-grained parallelism in dynamic web content generation: The parse and dispatch approach. volume 2888, pages 573–588, 11 2003. ISBN 978-3-540-20498-5. doi: 10.1007/978-3-540-39964-3_35.

[49] Yuhao Zhu, Daniel Richins, Matthew Halpern, and V. Reddi. Microarchitectural implications of event-driven server-side web applications. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 762–774, 2015.

[50] OpenJS Foundation. Expressjs homepage. https://expressjs.com/, .

[51] Evan Hahn. *Express in Action: Node Applications with Express and Its Companion Tools*. Manning Publications Co., USA, 1st edition, 2015. ISBN 1617292427.

[52] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., USA, 2013. ISBN 1617290858.

[53] Redis. Redis documentation - persistence. https://redis.io/topics/persistence.

[54] Arseny Zinchenko. Rtfm. https://rtfm.co.ua/en/redis-replication-part-1-overview-replication-vs-sharding-sentinel-vs-cluster-r

[55] Agenzia per l'Italia Digitale. Spid homepage. https://www.spid.gov.it/.

[56] Dipartimento per la Trasformazione Digitale. Spid. https://developers.italia.it/it/spid/.

[57] AGID. Spid - regole tecniche - introduzione. https://docs.italia.it/italia/spid/spid-regole-tecniche/it/stabile/introduzione.html, .

[58] OASIS. Security assertion markup language (saml) v2.0 technical overview. http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html, .

[59] OASIS. Bindings for the oasis security assertion markup language (saml) v2.0. https://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf, .

[60] AGID. Spid - regole tecniche - registro. https://docs.italia.it/italia/spid/spid-regole-tecniche/it/stabile/registro.html, .

[61] P. Chen. The entity-relationship model - a basis for the enterprise view of data. In *Managing Requirements Knowledge, International Workshop on*, page 77, Los Alamitos, CA, USA, jun 1977. IEEE Computer Society. doi: 10.1109/AFIPS.1977.117. URL https://doi.ieeecomputersociety.org/10.1109/AFIPS.1977.117.

[62] Internet Engineering Task Force (IETF). Totp: Time-based one-time password algorithm. https://tools.ietf.org/html/rfc6238.

[63] Lauren George and Raymond Cross. Remote monitoring and telemedicine in ibd: Are we there yet? *Current Gastroenterology Reports*, 22, 02 2020. doi: 10.1007/s11894-020-0751-0.

[64] Mohamed Estai, Yogesan Kanagasingam, Di Xiao, Janardhan Vignarajan, Boyen Huang, Estie Kruger, and Marc Tennant. A proof-of-concept evaluation of a cloud-based store-and-forward telemedicine app for screening for oral diseases. *Journal of Telemedicine and Telecare*, 22:319–325, 09 2016. doi: 10.1177/1357633X15604554.

[65] Niels Provos and David Mazieres. A future-adaptable password scheme. 03 2001.

[66] PostgreSQL. Postgresql insert. https://www.postgresql.org/docs/9.5/sql-insert.html.