Masaryk University
Faculty of Informatics

Design and implementation of an iOS notification system

Bachelor's Thesis

David Alexander Bielik

Brno, Fall 2018

Masaryk University
Faculty of Informatics

Design and implementation of an iOS notification system

Bachelor's Thesis

David Alexander Bielik

Brno, Fall 2018

MASARYKOVA UNIVERZITA
Fakulta informatiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: David Alexander Bielik

Program: Aplikovaná informatika

Obor: Aplikovaná informatika

Specializace: Bez specializace

Garant oboru: prof. RNDr. Jiří Barnat, Ph.D. (BcAP)

Vedoucí práce: doc. Ing. RNDr. Barbora Bühnová, Ph.D.

Katedra: Katedra počítačových systémů a komunikací

Název práce: Design and implementation of an iOS notification system

Název práce anglicky: Design and implementation of an iOS notification system

Zadání: Aim of this bachelor's thesis is to implement a notification system that features an iOS application, a web server and a middleware library that should be able to notify the web server (via a REST API call) and ultimately the iOS application itself.

The web server should include some basic user managed capabilities such as service registration and notification settings. The users of the iOS application should be able to create multiple services (which correspond to their personal projects) so the recieved notifications can be distinguished from each other.

The library and the iOS application are meant to be used by any software developer that needs to recieve immediate information about any situation that might occur in his project of choice. For instance, some developers might want to get notified if some unexpected exception occurs or when someone unauthorized attempts to access a private endpoint.

Literatura: [1] Developer.Apple.com. APNs Overview. URL https://developer.apple.com/library/archive/documentat
CH8-SW1

Zadání bylo schváleno prostřednictvím IS MU.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

David Alexander Bielik

Advisor: doc. Ing. RNDr. Barbora Bühnová, Ph.D.

Acknowledgements

I want to thank my supervisor doc. Ing. RNDr. Barbora Bühnová, Ph.D. for the given opportunity and trust to work on my topic and also for the provided guidance.

Abstract

The aim of this thesis is to describe the development process of implementing a notification system. The system consists of a RESTful web service written in Go, an iOS application written in Swift and a middleware. Besides communicating with the Apple notification server the back-end also includes a user management API that allows the users to create any number of so-called services they want to get notified from. Various notification and service settings can be changed directly from the supplied iOS application. Finally the provided middleware initiates the notification sequence and acts as a direct bridge between the user's iOS device and the user's service.

Keywords

iOS, Swift, Push notifications, Golang, Docker, Microservices, Design patterns

Contents

List of Figures

Introduction

In the current technological trends, almost every software developer is relying on some monitoring tool. These tools inform the outside world about any events that might occur in the application they are embedded in. There are many options to choose from when it comes to logging software. One of the setbacks of many tools is the fact that the logs are usually persisted locally (on the machine that is running the application code). Moreover, they might require additional setup by the developer before being able to retrieve the critical data. Consequently, when the application stores the information about its events and operations locally, any individual that is interested in this data must be able to access this machine by either utilizing some remote access service or directly accessing the log files stored on the computer. Remotely accessing the data adds another functional requirement to the implementation of the system itself by having to choose and implement the right software for accessing the data.

Nevertheless, many third-party web-based and cloud-based monitoring solutions have been developed over the years that solve the mentioned issues and are frequently used in real-world applications. Indisputably, the growth of the mobile hardware market has affected this branch of software. Hand in hand with remote capabilities of persisting and sending messages to a server comes the ability to forward these messages to the mobile devices that belong to the developers themselves. For instance, these logs could inform the receiver of some unprecedented situation taking place in their software, or just periodically inform the person in charge that everything is working flawlessly.

Most hardware manufacturers and mobile operating system providers are also aware of this trend and have already adopted features that allow the functionality of push notifications. This thesis focuses on just one of the hardware and OS notification ecosystems, namely the Apple notification system, which works with devices that run Apple's mobile operating system, iOS. The core concepts for sending push notifications are very similar to other notification systems. As a result, the practices and implementations this thesis describes

can rather easily be extended to other notification services, such as Google's Firebase Cloud Messaging[1].

In conclusion, this thesis will describe the design and implementation of each component that is used in the final product. Namely the server, iOS application and a client middleware. A high level system overview will be provided in the first chapter as well as the internal and external interactions. The three main subsystems or components will be briefly described and a set of functional dependencies will be introduced. The second chapter will focus on describing the server component and its functionality. It will also provide some information on the system's data layer and communication. The third chapter will start off with a description of modern iOS design patterns. The chapter continues with defining the overview of the application screen flow as well as the use of protocols. Brief descriptions of various screens will be presented. The fourth chapter will focus on the client middleware, describing its use. Lastly, the final chapter discusses the future development.

1. https://firebase.google.com/docs/cloud-messaging/.

1 System overview

At the beginning of any development of a larger system, the usual approach consists of mapping the core functional requirements into some textual or visual representation. These early drafts or sketches are meant to be very clear and easy to interpret by the development team. Requirements that shape the first visualizations are collected prior to the modelling of the system as a part of a process called **requirements engineering**. In our notification system, the main functional requirements can be represented by the following bullet points:

- A user should be able to send customizable messages from their applications into the system via a client interface.
- The system handles these messages and forwards them into the *Apple Push Notifications service*, so the notification reaches the appropriate iOS devices.

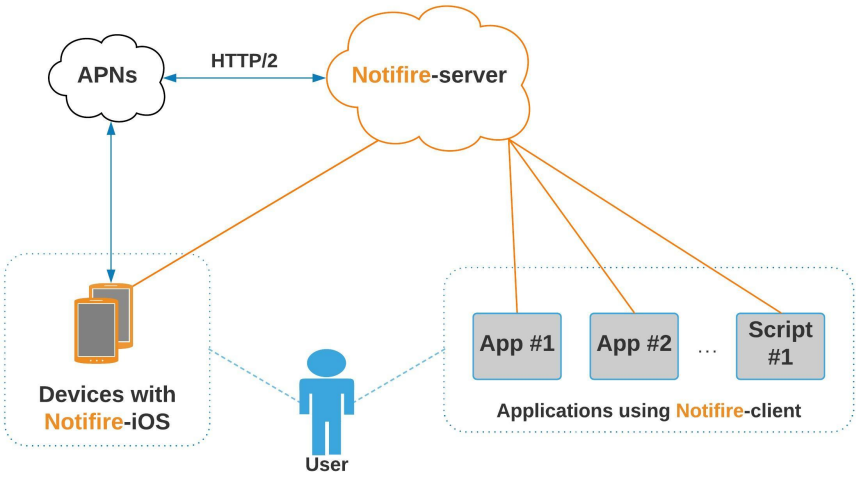


Figure 1.1: Overview of the system.

From these requirements we can already obtain the three main components the system will comprise. The extended from figure 1.1

showcases these components and their associations (both highlighted in orange). The chart suggests that they are self-contained and communicate over some defined link. Regions bounded by the dashed blue line represent the user space, or the components that the user directly interacts with. A detailed description of communication is discussed further in section 1.1.

In addition to the two essential functional requirements, here is the rest that will be implemented and discussed later on:

- A user should be able to easily **distinguish** between notifications coming from their applications.
- A user can **customize** three notification priorities for each application separately.
- The received notifications should be persisted on the user's iOS devices in their appropriate sections (branched by the applications that sent them).
- A user has to validate an email address on his account before being able to use any features.

In order to establish a common ground, the distinction between various notifications is going to be achieved by defining the so-called *Notifire Services*. They will represent some application or a software that the user is using outside of the system's boundary. The notification priorities will somewhat resemble logging levels from some programming languages, except they will only have three stages and will not be hierarchical.

1.1 Component interactions

The interactions between the system components could be categorized into internal and external. **Internal** interactions are limited to the components residing in our system, thus the communication is not constrained by the protocols of some other third party. Therefore most of the internal communication in the Notifire components is done over HTTPS. Besides HTTPS, AMQP is used as the communication

protocol in our server component and will be discussed in section 2.1.3.

The only **external** interactions performed by our system are with the APNs. In the context of APNs [2], any server that interacts with it is referred to as the **provider**. Our current model implies that the server component of our system will have to conform to the communication specification of APNs, thus, become the provider. Besides the provider sending some data to the APNs, we have to make sure that the iOS devices are also ready to receive these notifications. As stated on the apple documentation website [**Security Architecture**, 2], APNs enforces end-to-end, cryptographic validation and authentication using two levels of trust.

1.1.1 Provider-to-APNs connection trust

When it comes to establishing a secure connection with APNs from the provider, there are two authentication standards that are supported.

1. **Token-based provider connection trust**

    - This specification requires the provider to include a JWT [3] token in each notification request sent to the APNS.
    - This method has been introduced recently (2016) and is the preferred way of authenticating with APNs. There is no need to create new authentication keys after a certain period of time as opposed to the certificate authentication.
    - The private key used to sign the data can be obtained from the Apple developer website after creating a developer account. This key can only be downloaded once.
    - Sending notifications to multiple apps from a single token is supported.

2. **Certificate-based provider connection trust**

    - The slightly older method for establishing trust with APNs, the provider is required to obtain a provider certificate from the Apple developer website.

- The certificate is tied to a specific app identifier (bundle ID) which implies that the notifications can only be sent to one app per certificate.
- After establishing the secure connection, the requests don't require any data except the raw notification data. In contrast, the JWT approach requires a token in each APNs request.
- Certificates have to be renewed when they expire. (1 year validity period for each certificate issued by Apple)

After trying out both authentication methods, the token-based approach was used in the APNs connector microservice. The following subsection will describe the other external communication link we had to implement.

1.1.2 APNs-to-Device Connection trust

The APNs to iOS device connection trust is necessary for the retrieval of a unique identifier that is also known as the **device token**. This token is unique for each application installed on the iOS device and has to be manually requested on each app launch by calling `UIApplication.shared.registerForRemoteNotifications()`. This method has a callback that gets invoked when the device registration with Apple servers succeeds, from which we obtain a fresh device token.

One inconvenience that many iOS developers experience is that these device tokens change over time. One of the causes that might trigger the change of a device token is an application update or a reinstall from the App Store. However, these actions are not guaranteed to change the device tokens and the only way to guarantee a device token update is by restoring the iOS device from a backup or reinstalling the OS. Fortunately our server database and APNs connector implementations are ready to deal with these possible inconsistencies.

2 Notifire Server

From the figure 1.1 we can conclude that this subsystem is an essential part of our notification system and has many responsibilities. **Service oriented architecture** (or SOA in short) is in some way the architecture of choice in this component due to the separation of concerns aspect. Sommerville states that the SOA approach structures a software system as a set of separate, stateless services [1]. In addition to this definition, the services usually interact over one *Enterprice Service Bus* (ESB), which is a middleware that is usually responsible for:

- routing messages between the services
- controlling the deployment of active services (load balancing)
- transforming the data between the services connected to ESB (each service can have their own data representation format)

Microservices is an architecture pattern (or style) that evolved from SOA. There are some key differences that are best described in detail. One of them being the granularity of services and as the prefix "micro" suggests, microservices are more fine-grained than services in SOA. Ideally, the microservices should have a single *minimal* purpose in the system. Component sharing is another attribute that distinguishes SOA from microservices. Furthermore, service functionality sharing is often necessary in SOA. On the other hand the data is the only thing that needs to be shared in microservices. They are also bounded by some context that represents their scope of data manipulation and action. All of this should be done with minimal dependencies on other microservices.

Furthermore the containerization software (Docker) that we will discuss in section 2.3 is designed to work in a similar fashion and encourages the microservices architecture.

A comment on dependencies. The decision process for choosing frameworks and libraries was very trivial, more precisely the original idea was to *implement as many reasonable components from the ground up as possible*. This approach is usually not preferred, as the solutions for many problems in this domain already exist and it would be sufficient

to use and adapt a problem-solving library or framework to the system's needs. Also any third party software that is commonly used is very likely to be more stable and polished than any in-house short-term solutions. It is pretty apparent that if many popular applications are dependent on some tool, it most likely stood the test of time and is considered to be reliable.

One good example of a tool that would significantly help in the implementation phase would be the **Firebase** Cloud Messaging solution made by Google. It could potentially be a direct replacement for all of the microservices used in our system except for the frontend service, that is used for Universal Links (explained in section 2.1.5). The most notable advantages of Firebase include:

- Single Sign On via Firebase Authentication SDK[5]
- Real time synchronization via websockets.
- Cross-platform notifications (our implementation would not be limited to Apple devices)

Despite these advantages, this dependency was omitted on purpose.

2.1 Notifire microservices

Each service mentioned in this section is deployed in it's own Docker container. With the microservices concept in mind, most of these services are horizontally scaleable. This means that after a minor modification of the `docker-compose.yml` file, e.g. adding a proxy such as Traefik or Nginx, we could spawn more worker containers for the API microservice. In that case the proxy would act as the load balancer, or the master in a **master-slave** pattern.

Also note that the APNs connector (section 2.1.4) microservice is scaleable by default, because it complies with the competing consumers pattern [6]. Another option would be to use a tool that is commonly used in production in many big companies, Apache Kafka [7]. However due to its complexity this tool was left out.

In the next subsections we will discuss the responsibilities and setup of each of the microservices in use.

2.1.1 Database

The database microservice serves its purpose as the persistent data layer for our system. It stores the data that is required by other microservices, namely the API and APNs Connector. The container for this docker service is using the official postgres docker image [1], thus the SQL implementation in use is PostgreSQL.

This list provides a simple overview of the data that is persisted in our SQL database:

- User data (such as emails, usernames, passwords as salted **hashes**...)
- Service data (service settings, which user do they belong to, creation date...)
- Device data (for APNs device tokens - section 1.1.2)

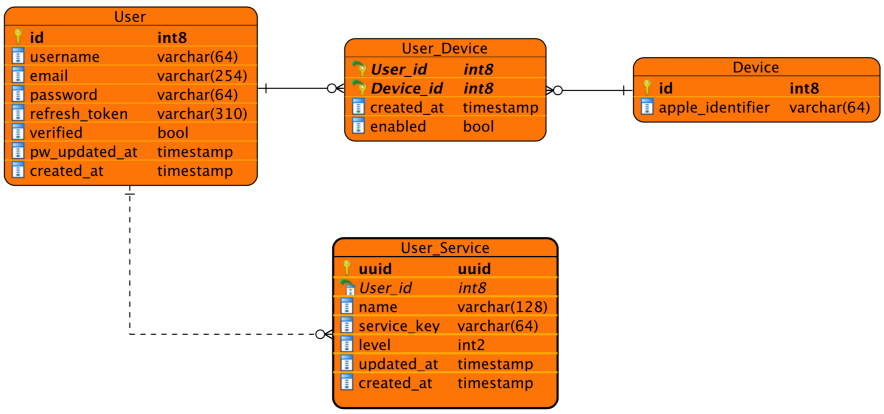


Figure 2.1: The Entity-Relationship diagram.

1. https://hub.docker.com/_/postgres/

Devices The `User_Device` table is an associative (or junction) table for the many-to-many relationship between the user and his devices. To provide a reasoning behind this implementation let's consider the following scenario:

1. User **A** logs in to the iOS application. The device token is registered with the Notifire server and is inserted into the `Device` table as well as the `User_Device` table with the appropriate foreign keys.

2. User **A** logs out from the app. The `enabled` attribute in the `User_Device` table is set to false (as a result of calling the appropriate endpoint in the API after logging the user out).

3. User **B** logs in on the same device. The device token redeemed from the `UIApplication` callback is the same as in step `1`. We only have to insert a new row into the `User_Device` table.

This example covers one half of the many-to-many relationship, more precisely the one-to-many from `Device` to the `User_Device` table.

The second one-to-many relationship that requires the use of a junction table is easily described by a simple fact that one user can possess multiple iOS devices.

The `enabled` attribute is used when an attempt to send notifications is made, namely the APNs connector checks and sends a notification if the `apple_identifier` (device token) associated with the user is enabled.

Levels Our implementation of per service notification levels leverages the use of bitfields (or flags). This approach is best described by figure 2.2 where each notification level is represented by a number that is a power of `2` (counting from the 0-th power). The computation of the combined notification level for a single service is only a matter of summing up the enabled levels. After inserting the combined level into the `User_Service` table we can check for the active flags with some simple bitwise operations. All of the required data manipulation and transformation is performed by the API microservice.

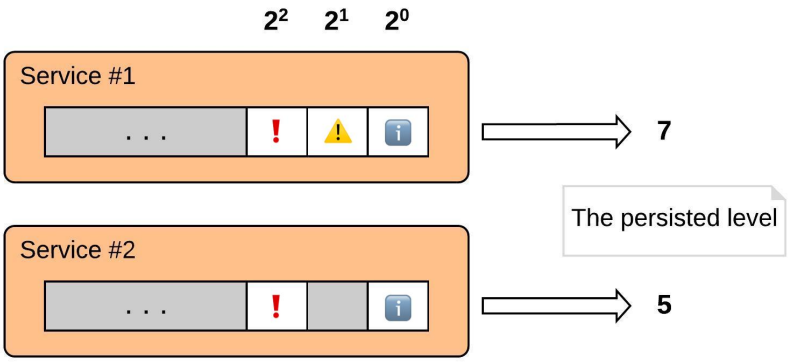


Figure 2.2: The format of notification levels

Room for improvements Note that in our current setup, the database service is missing some features that might be very convenient. One of them being that the notifications sent via the notifire-client are not persisted in the database. This could be fairly easily accomplished by adding a new table to our database and saving each notification request in the APNs connector service. Implementing this would also require additional notification endpoints (e.g. `GET /notifications`) with pagination to the notifire API and subsequently rework the `NotificationsViewController`(3.5.2) in the iOS app.

The other missing feature is sharing services between users. Sharing would require a junction table between the table `User` and `User_Service` tables (instead of the current one-to-many relationship). This would also significantly stagger the development and UI design process of the iOS application, and was left out because of the time constraints. It is however, one of the priorities for any upcoming releases.

A remark on SQL. Lately the SQL based database management systems have been caught up by some **NoSQL** implementations. The pros and cons of using a NoSQL database have been discussed countless times and the decision usually boils down to the data. MongoDB[2] was considered as an alternative to Postgres in this project, but as an exercise for SQL queries, the SQL variant was chosen. After all,

2. `https://www.mongodb.com`

NoSQL databases are preferred in the microservice architecture and their advantages usually outweigh the disadvantages. Their benefits include high scalability and performance.

2.1.2 API

The Notifire API is a RESTful web service written in the Go language. Its dependencies are minimal as Go comes with its own HTTP client and server implementation in the `net/http` package. Yet one of the dependencies (go/chi[3]) we have used is a lightweight router built on top of the `net/http` package. It encourages modular design composed of middlewares that transform the HTTP request handling into a chain of handler functions.

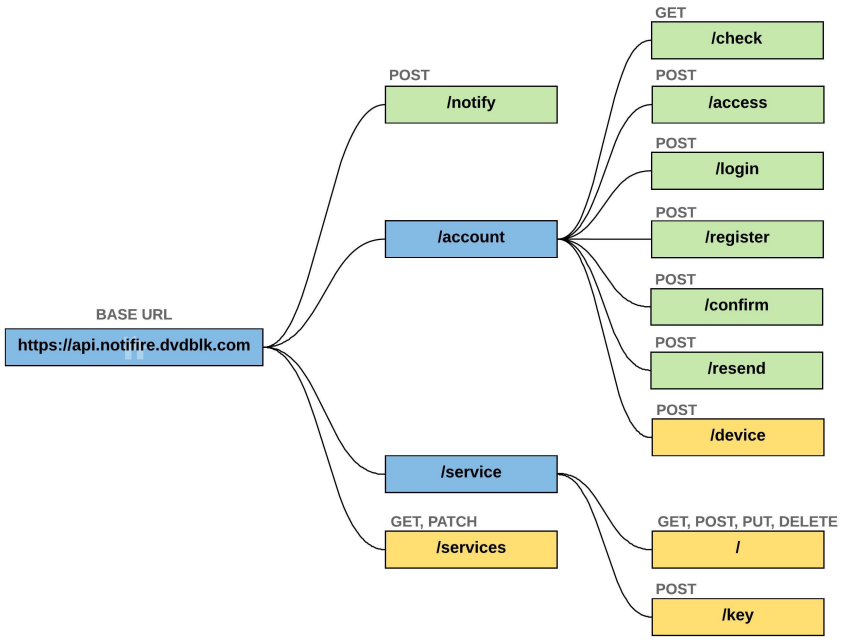


Figure 2.3: The overview of the Notifire API.

3. https://github.com/go-chi/chi

Design The API infrastructure consists of 14 endpoints that are divided into two main categories:

- Unprotected endpoints - these don't require any authorization headers.
- Protected endpoints - these require an authorization header. Used for account and service management.

Figure 2.3 is a visual representation of every endpoint that is defined in our API. Any other route returns a `404` status code with an empty response body. Furthermore, when the request's body doesn't meet the defined format, the returned status code is always `400`. For instance this situation might occur when the request is constructed manually outside of the client applications by a random user or some bots attempting to retrieve some information. The iOS application respects the defined body format (for each request) so this status code will never be returned into the client app.

Table 2.1 mentions all the possible status codes and the situations they occur in. Before diving into the description of each set of endpoints, it is reasonable to mention that each request body is validated before any data processing. For instance, username and email fields from the `/register` endpoint are matched against specific regular expression patterns to verify their format. In case this format is not satisfied, the server returns a `400` status code.

Table 2.1: Status codes returned by the Notifire Server

Status code	Meaning	Use case
200	Success/OK	Most of the endpoints that manage resources and return some data use this status code
204	Success/No content	Endpoints that don't return any data, e.g. `/notify` or `/account/device`
400	Bad Request	Whenever the request format is malformed (client error, should not happen in production environment)
401	Unauthorized	Returned on protected endpoints when the access token expires.
404	Not found	Whenever the resource / endpoint doesn't exist.
500	Internal server error	Each request can potentially result in this status code, returned when an unexpected exception occurs on the server. (e.g. database connection is lost)

Listing 2.1: The Authorized middleware function used for protected endpoints.

```
func Authorized(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter,
  ↪ r *http.Request) {
    // get the authorization header
    ah, err := getAuthorizationHeader(r)
    if err != nil {
      render.Render(w, r, ErrBadRequest())
      return
    }
    // validate the access token
    token, err := validateJwt(ah, accessTokenSecret)
    if err != nil {
      render.Render(w, r, ErrUnauthorized())
      return
    }
    // get username from claims
    if claims, ok := token.Claims.(jwt.MapClaims); ok
  ↪ && token.Valid {
      if username, ok := claims[usernameKey].(string);
  ↪  ok {
        // add the username to a new request context
        ctx := context.WithValue(r.Context(),
  ↪ usernameKey, username)
        // pass the context with the username
        // to the next HTTP handler
        next.ServeHTTP(w, r.WithContext(ctx))
      } else {
        // no username found in the claims field
        render.Render(w, r, ErrBadRequest())
        return
      }
    } else {
      // catch-all
      render.Render(w, r, ErrUnauthorized())
      return
    }
  })
}
```

Each of the protected endpoints from figure 2.3 is routed with the `Authorized` middleware function. Its purpose is to ensure that only authorized entities are allowed to access the data they are requesting. This function is shown and described in listing 2.1.

Account endpoints Endpoints described in this paragraph share a common prefix that denotes their intention. All of them are somehow related to the user's account.

The very simple `/check` endpoint is used for verifying the available username and emails while registering a new account. On each request, the user's input is validated at first. When the validation is successful the database is queried for the existing usernames or emails. Query parameters are used for the desired attribute e.g. `/check?username=test123` or `/check?email=test123`.

The `/register` endpoint begins the flow of registering a user's account. It requires three parameters in the request's body, namely the username, email and a password. These are stored in the database as well as the is_verified attribute set to `false`. Afterwards an email with a freshly generated email token is sent to the supplied email address.

The `/login` endpoint returns the user's refresh and access tokens if the supplied fields in the request's body are valid. The required fields are username (or email) and a password. However, the tokens are not returned if the user didn't confirm his account via email beforehand.

`/resend` endpoint sends an email to the specified email address if it exists in the database and has not been verified yet. It was meant to be used for situations when the email token would expire, so the user has to request a new one Token expirations are mentioned by the table 2.2.

Unsurprisingly, the `/confirm` endpoint links the user's account with an email address. It also executes a query in the database which sets the user's is_verified attribute to `true`. Note that the email token is a required field in the request's body.

Finally the `/access` endpoint is used to obtain a new access token, which grants access to protected endpoints. A valid refresh token is required for this request to succeed with a `200` status code.

The only protected endpoint in the `/account` route is `/device`. Naturally, only the owner of the account can request this endpoint. A

device token of the iOS device must be supplied and a boolean value for the enabled field as well. This boolean value indicates if the device token should be enabled or disabled for the user. If the token is disabled, the user receives no notifications. As soon as the iOS application receives the device token from the APNs, this endpoint is requested with `"enabled": true`. In contrast, this request is also called when the user logs out of the application while thevalue of the `enabled` field is `false`.

Service endpoints These set of endpoints are related to the service data. The HTTP methods of the `/service/` endpoints are quite self explanatory in terms of their functionality. However, to identify the requested service resource we have opted for the less usual approach of providing the uuid field in the body of the request. The common approach requires the uuid to be provided in the endpoint's url path, `/service/<uuid>/` like so. All methods except `POST`, which creates the resource, require this field.

The `/service/key` endpoint generates a new service key for the service specified by the uuid field. Besides the uuid field it also requires a valid password to be specified in the request's body, as another layer of protection against accidental or malicious attempts that would result in the invalidation of the active service key.

Finally, the `/services` endpoint has two methods that it responds to. The `GET` method simply returns an array of services meanwhile the `PATCH` was meant to be used to update multiple services in one request. However this functionality is not implemented in the current version of the API.

Notify This endpoint's role is crucial for the entire notification system. Simply said, it is used for *forwarding* notifications to the APNs connector microservice. An example request body is provided in listing 2.2. Note that the levels specified here are sequential numbers (1,2,3) as opposed to the powers of `2` while operating with the database.

Listing 2.2: `/notify` request body

```
{
  "serviceKey": "", // obtained from the iOS app
  "level": 1,       // notification level
  "body": "",       // the notification body
  "text": "",       // optional additional text
  "url": ""         // optional url
}
```

A status code 204 is returned on a successful request. On the other hand, a status code 200 with a verbose error message is returned in case the service key doesn't match any service stored in our system or when the notification is not sent due to the user notification settings. An example of the latter is provided in section 4.2.

Responding to user errors It is quite common for web services to use only a small subset of all of the available response codes and handle most of the user errors within the response body with status code 200. However if there is any user input involved, e.g. a login form, the server has two additional responsibilities. **Input validation** and the necessity to respond with a description of the user error. Lets take a look at the `/account/login` endpoint in which we have defined three possible user errors:

1. The username or email are invalid (no account is associated with the provided credentials).
2. The password for the specified account was invalid.
3. The account is not verified yet.

User errors like these also arise in other endpoints, for example a user can try to confirm the same account twice, or try to use an expired confirmation link (both user errors are related to the `/account/confirm` endpoint). To wrap these user error requests under some common interface, so the application and the server can handle these errors effortlessly, we have decided to use this format in any JSON response that could be impacted by the user.

Listing 2.3 showcases a situation where the user error didn't occur. Therefore the `success` attribute is `true` and the payload data is present in the response. Note that this payload data is request specific. On the other hand if the value of `success` is `false`, the `error` attribute is not `null` and contains some object with an error code and a message (listing 2.4).

Listing 2.3: Successful user error response

```
{
    "success": true,
    "payload": { ... }
}
```

Listing 2.4: Unsuccessful user error response

```
{
    "success": false,
    "error": {
        "code": x,
        "message": "..."
    }
}
```

An architecture variation In our implementation, the emphasis on microservice architecture could have been further embraced by splitting the API microservice into multiple microservices, each serving only a part of the entire notifire public API. This is described in figure 2.4.

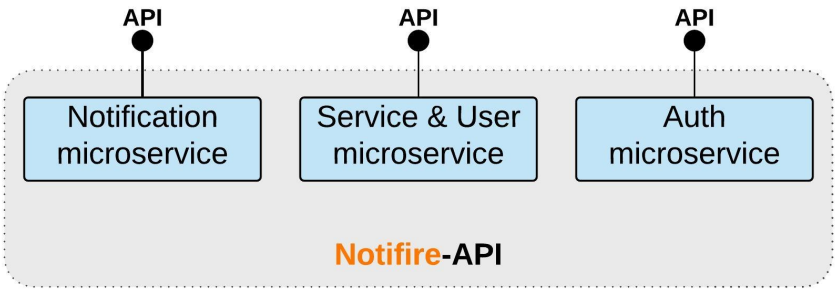


Figure 2.4: Possbile API architecture.

2.1.3 Notification queue

The notification queue component is responsible for providing a communication link between the API and the APNs Connector services. It is based on the **Advanced Message Queuing Protocol** (AMQP in short). AMQP is an application layer protocol that is designed to provide an efficient way of communicating between applications. It defines a standard for sending, routing and delivering asynchronous messages between applications and in conclusion, allows them to share their resources.

However, AMQP is just a protocol that needs to be implemented first. Thankfully there are many solutions that implement AMQP that will allow us to use the proposed functionality. One of the most popular middlewares that implements AMQP is called **RabbitMQ**.

To describe the basic model of our RabbitMQ notification queue component we have to get familiar with these terms:

- **Message** is a block of binary data which consists of a *header* and a *bare message* part. The *header* stores some metadata which could be further analyzed by the broker or the receiving application (priority, time to live, ...). The bare message is created by the sending application (the producer) and is therefore immutable.
- **Routing key** is specified with each message, the default exchange type that we are using is matching this against the queue name (they have to be equal for the message to be delivered).
- **Exchange** is a message routing agent that receives messages sent by the producers. After receiving a message the exchange sends a copy to an appropriate queue based on the routing rules (exchange type).
- **Producer** is any application that produces messages by sending them to an exchange through a broker. Applications that receive and handle these messages are the **consumers**.
- **Queue** is an entity that stores the messages received by the exchange. Brokers either deliver these enqueued messages to the consumers or let the consumers handle the queue popping by themselves.

- **Broker** is a middleware that implements AMQP.

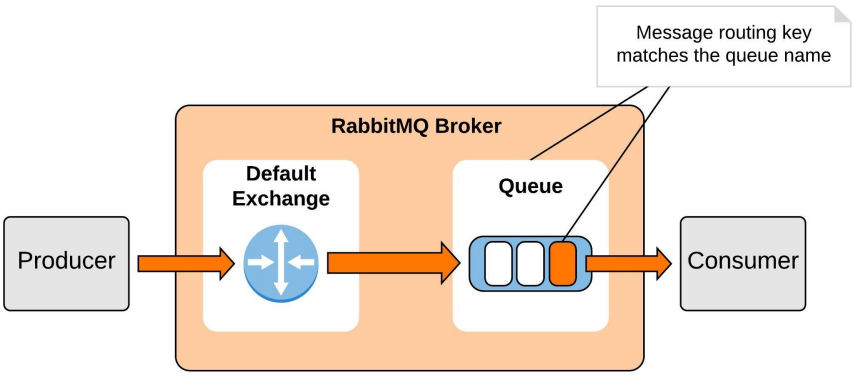


Figure 2.5: A detail of the notification queue service.

The RabbitMQ broker from figure 2.5 is actually our notification queue component. In context of our system the producer is the API service that sends messages. These messages contain the notification data that needs to be sent to the iOS application via APNs connector (consumer). Format of the message is described in section 2.1.4.

RabbitMQ allows us to use multiple routing mechanisms which are also known as the **exchange types**. For our implementation we have chosen the default (*nameless*) exchange which forwards the messages based on the queue name. In detail, after we declare a queue with some name, the broker will bind this name to the default nameless exchange. Any subsequent messages that have the same routing key as the queue name, are sent to the queue. This is the simplest option that is perfectly viable for our use case, in which we need to handle only a single type of events, namely the sending of notifications (described in 2.1.4).

Another benefit of using RabbitMQ is the notion of *message acknowledgements*. Whenever a message is delivered to a consumer, the consumer can automatically notify the broker that he has received the message. The consumer can also opt for manual acknowledgement after some processing is done. This is especially useful if a failure from the consumer side is possible. The message can be rejected by

the consumer and returned to the queue by the broker. Afterwards it can be handled and acknowledged by other consumers.

2.1.4 APNs Connector

This component was created to distribute the tasks from the API microservice. Its main responsibility is to consume messages coming from the notification queue and handle them appropriately. This message handling is best described by pointing out its multiple stages:

1. First of all, we have to persist a secure connection with APNs as creating a new connection on each attempt to send a notification would result in APNs supposing that some kind of a DDoS attack is taking place. Additionaly, this would result in a temporary restriction of use for our service.

2. Use the user identifier that arrives in the message body to get all devices that the user might be actively using.

3. Send a POST request to the APNs for each device with the appropriate data.

4. Wait for the response and update the users devices accordingly. APNs might return a status code which signifies that the `device_token` is no longer valid and should not be used anymore.

The implementation was done in the language Go. The APNs connection persistence was achieved via apns2[4]. However the last step of the provided list was omitted from the project due to time constraints. Listing shows a sample of the message that is consumed from the queue.

4. https://github.com/sideshow/apns2

Listing 2.5: The bare message

```
{
  "uid": "",                // id of the user
  "datetime": "",           // ISO date of the /notify
   request
  "sid": "",                // service uuid
  "request": {
      // the request body from /notify
  }
}
```

2.1.5 Front-end

The front-end component is exceptionally simple in terms of functionality. Its only purpose is to serve a single resource, namely the `apple-app-site-association` file. Initially, this microservice was not planned to be included in the system. The main reason we had to incorporate it was because of the restrictions that many email providers have in their email clients.

To fulfill our functional requirements the system had to verify all accounts before they were able to retrieve their access tokens. Strictly speaking account verification via email had to be implemented. The plan was to send an email after the `/register/account` endpoint was successfully requested. The email would include a URL that would begin with a custom URL scheme (`notifire://`) that could be opened by our iOS application. This turned out to be a dead end, because most of the modern email clients remove custom URL schemes from the email HTML as a security measure. The only possible solution was to use Apple's Universal Links which would require the previously mentioned `apple-app-site-association` file to be served by our own web service. Note that this file could also be served by the API microservice, but as there was some intention to provide a front-end UI in the near future, this was a nice starting point and would have to be implemented anyway.

Universal links Apple fortunately provides another approach to opening an app from a URL. In comparison to the older deep links, or custom url schemes, universal links are intended to be a seamless transition from browsing the web pages to the iOS application. As stated on the apple developer website, universal links create a two-way association between our app and our website and specify the URLs that our app handles [10]. Furthermore the context of the web page should be persisted in the iOS application. Deep links on the other hand are perfectly viable for inter-app communication, and are still a feasible option.

In order to enable the Universal link functionality, some non trivial setup is required. First, we have to make sure that our iOS application settings include our associated domain that we wish to open in our app. However, this domain now has one additional constraint, it has to be accessible over HTTPS. The SSL certificate for web server is provided by the Let's encrypt authority and also for the API, which is serving from another subdomain.

The aforementioned *AASA* file must be reachable and a reference to the application identifier as well as some other settings must be included. Whenever the app with an associated domain setting is installed, iOS proceeds to verify if this file is reachable on the supplied domain. When the file is present and the `appID` matches the bundle identifier, functionality of universal links is allowed and the app is further tied with that webpage. Listing 2.6 shows the JSON file that the front-end microservice is serving.

This JSON file includes a `paths` field that specifies which paths are included or excluded from the application-website association. The way the field is defined in our *AASA* file allows URLs with the path `/account/confirm/*` to be opened by our application. Particularly, the URL that is sent to new users via email is of this format `https://notifire.dvdblk.com/account/confirm/<email_token>`. Naturally, the URL is a part of a HTML button that the user can tap on.

Listing 2.6: The publicly accessible apple-app-site-association file.

```
{
    "applinks": {
        "apps": [],
        "details": [
            {
                "appID": "6QH7E4QW2D.com.dvdblk.
   Notifire",
                "paths": [ "/account/confirm/*"]
            }
        ]
    }
}
```

2.2 Authentication

While facing the issue of authorizing user actions and in fact authenticating the users, we have decided to implement our own, greatly simplified version of the OAuth2 [11] standard. The OAuth2 authorization process involves some entities, namely:

- the **resource owner** or user, that has some data which he grants access to, usually through the client.
- **Client** is the application that makes the protected requests against the resource server after the user's permission has been granted. (e.g. iOS application, web UI)
- **Resource server** responds to the protected requests by providing or updating some data. The requests are accompanied with the access tokens provided by the authorization server.
- **Authorization server** issues access tokens to the client after a successful user authentication and authorization.

User's permission must be granted mainly because of the fact that the authorization servers are an external entity in relation to

the resource servers. Our implementation streamlines this process by merging the authorization and resource server into one entity, the API microservice. As soon as we omit the external authorization server and let our API function as the authorization and resource server, we don't even need the users explicit permission. The next subsection describes the authentication tokens that we have used in our system.

2.2.1 JSON Web token

Or **JWT** in short, is a JSON-based open standard (RFC 7519) for creating access tokens that assert some claims. These claims are used to pass identity information of authenticated users so the resource server (notifire API) can identify the client that initiated the protected request. The claims can also include some other metadata. For instance, the metadata can be used to deny access to the resource when an expired access token would be supplied. This is used fairly frequently in our implementation as most of the tokens are expiring after some time.

The format of JWT is a base64URL encoded string of characters concatenated with periods. The three parts of the token are the following:

- A **header** part that identifies which algorithm was used for generating a signature. In our case this was the usual `HS256` (HMAC-SHA256).
- A **payload** part where the claims are located. The standard fields include the date of issue (`iat`) or the expiration date (`exp`). Every field is optional and anything data can be inserted into this part. However they usually contain data that represents the user in some way e.g. the username or user ID.
- A **signature** part that validates the origin of the token. The signature is calculated from the header and the payload as well as a secret key.

Notifire tokens The JWTs that were used in the implementation could be categorized by their time of expiration and the contents of their payload. Table 2.2 provides a simple overview of each JWT that was used. Also, for each of the described tokens a different secret key

Table 2.2: The JWTs in use

Type	Expiration	Custom claims
Access token	1 hour	`username`
Email token	1 day	`username`
Refresh token	Never	`username`, `userstring`

for the signature part was opted for. If a single secret key was used for all the tokens, they could be used interchangably. E.g. the email token could have been used as an access token. Assuming a single key would have been used, to differentiate between various tokens, a new custom claim field could have been added. For instance, `"type": "email"` and so on.

Access tokens expire after 1 hour and have to be reissued by making a request to the `/account/access` endpoint. They are used for accessing resources on the protected endpoints where the `username` field identifies the resource owner.

Email tokens have a longer expiration time, and they can only be used once (for account confirmation). The `/account/confirm` endpoint simply reads the username claim from a valid email jwt.

However the *refresh tokens* have no expiration. To provide a way of invalidating the token, two possible approaches could have been chosen.

One of them includes a special `userstring` field that is a digest of some values associated with a user. Namely, `userstring = hash(username, pw_updated_at)`. This means that on each access token generation attempt we have to compute and compare the digest with the userstring field. If these values are not matching, the access token is not generated because the pw_updated_at field was changed. Thus the refresh token has no more value for the user.

The other approach is storing the refresh token inside the database and comparing it on each access token request. The latter was opted for however the userstring is still present in the refresh token.

2.2.2 Service key

This key is used for authentication in the `/notify` request. Simply said, it is an API key that points to a single service. API keys are a go to choice in many web services that need to track the usage or simply authenticate an entity that is requesting some data. There are no exact rules for generating these keys nor there is a standard for them, but we can deduce the obvious.

1. the generated key must be unique
2. it can be revoked / invalidated
3. the key must be hard to guess by an unauthenticated entity / user

To guarantee **uniqueness**, uuid of the service can be used, as the probability of them (UUID v4) not being unique is negligable. However, to accomplish key invalidation, we have to take some salt, or random data. In this case it was the date of key creation. The last of the three properties is achieved with the use of an HMAC generating function, more precisely the HMAC-SHA256 algorithm was used.

One drawback that this key format has is that on each attempt to send a notification to the iOS devices, a query for a service belonging to this key has to be executed in the database.

Note that the user can invalidate the currently active service key by generating a new one via the iOS application.

2.3 Containerization via Docker

In section 2.1 we've discussed how the Notifire server is partitioned into multiple microservices. However, some of them need to be connected and share the same data store. All of this is relatively easily achieved and provided by **Docker**[5] which is a popular operating-system-level virtualization software. The main concept of Docker is **containerization**, which encapsulates each software (microservice) package into a separate container that is isolated from the rest of the system it runs on. These containers can be configured and packaged

5. `https://www.docker.com`.

with the appropriate files or executables, which after configuration act as a microservice.

Docker comes with many command line interface (or CLI) commands that are essential for the container configuration and management (such as `docker run` which runs the initial command in a new container).

Docker-compose While developing a system consisting of multiple Docker containers the `docker-compose` tool can be handy. It is a wrapper for the `docker run` command and its arguments in the form of a single YAML file (its name defaults to `docker-compose.yml`). After creating this file with the proper settings, the developer can start all his services in a single command, `docker-compose up` (invoked from the directory where the YAML file is located).

Even though the containers are meant to be separated from the external environment, the systems built with Docker often require some form of communication. Docker provides a neat solution for establishing network links between containers so that they can access each other. These links are managed with the `docker network` command (or the network configuration option in a YAML compose file). One benefit of the compose file over the `docker run` command is that a default network for all containers in a compose file is created automatically.

External networks can be joined by multiple containers outside of the scope of a compose file. The server that Notifire is deployed on has one external network created before the deployment of each container. The reason behind this is that in a production environment, the compose file is omitted, and each container is started with the `docker run` command. This is frequently achieved by the use of some orchestration software (such as Kubernetes [6]). However due to the hardware constraints of our server, this software has to be omitted.

In the listing 2.3 we can observe the network configuration used on the production server (some of the JSON fields were omitted for clarity):

6. `https://kubernetes.io`

```
dvdblk@srv ~$ docker network ls
NAME                      NETWORK ID
...
notifire-net              <network_id>
...
dvdblk@srv ~$ docker network inspect notifire-net
[
    {
        "Name": "notifire-net",
        "Containers": {
            "<container_id>": {
                "Name": "notifire-apns-connector",
                ...
            },
            "<container_id2>": {
                "Name": "notifire-db",
                ...
            },
            "<container_id3>": {
                "Name": "notifire-notification-queue
   ",
                ...
            },
            "<container_id4>": {
                "Name": "notifire-api", ...
            }
        },
        ...
    }
]
```

The containers included in the `"Containers"` node are able to access each other through their open ports with their container name as the domain name. For instance, the `notifire-api` container has a TCP port 8080 open, thus any of the containers on the same network are able to access this container via `http://notifire-api:8080`. Note that in a larger system the networks would be divided from a single network to

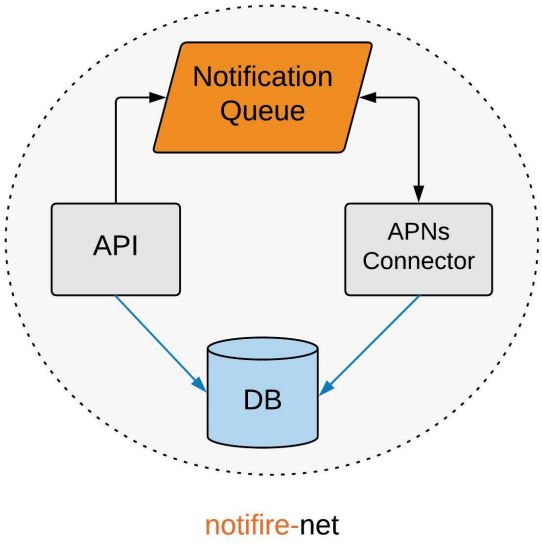


Figure 2.6: The notifire docker network and its containers.

a multitude of networks as a security precaution. Figure 2.6 is a visual representation of the Notifire network.

3 Notifire iOS

The iOS application was developed in Apple's latest language for the iOS platform, Swift. The popularity of Swift is undeniable, though Objective-C is still an option. The latter does have some drawbacks especially its not very beginner-friendly syntax, stricter memory management and also the language is considered to be fairly outdated.

Cocoapods were used as the dependency manager and this is a list of the very few required dependencies:

- **SwiftLint**[1] - for codestyle checking
- **KeychainAccess**[2] - wrapper for the Apple Keychain API
- **RealmSwift**[3] - database

Many developers tend to use some auto layout DSL dependency such as SnapKit or Cartography. DSL stands for domain specific language and in the context of auto layout it represents a wrapper for its verbose syntax. However for this project this dependency was omitted as the auto layout API is quite concise since the introduction of safe area layout guides [15] in iOS 11 and layout anchors [16].

3.1 Design pattern comparison

One of the first tasks while starting the development of an iOS application is to choose a well-suited design pattern. The choice of a design pattern influences the software in many ways. For instance, **coupling** [17] describes the degree of interaction and dependency between classes. If you were to choose a pattern with very strict decoupling of classes, you might end up with a fairly complex hierarchy that might not be easily picked up by a new member of your team. On the other hand, if you stick to classes that break the single responsibility principle[4], the classes will be hard to unit test.

1. https://github.com/realm/SwiftLint
2. https://github.com/kishikawakatsumi/KeychainAccess
3. https://realm.io/docs/swift/latest/
4. https://en.wikipedia.org/wiki/Single_responsibility_principle

However in any software, there is no rule of thumb which states one single pattern that is the best for every situation. It all comes down to the development team preferences and the specific needs for an application. Besides the data and business domains the most common factors that influence the choice of a design pattern include the application size, view re-usability or the necessity to write unit tests. As of now, Apple continues to promote and recommend their slightly modified version of a pattern popular across many branches of software development, the Model-View-Controller.

One thing worth mentioning is that neither of these patterns is a strict description of how the software will look like. In reality, only the concepts of these patterns should be obeyed and extended for any specific needs. However, each of these derivations from the original MVC concept enhances and extends the base in some way. Explicitly stating what pattern an application is using gives the developers an assumption of what to expect in certain parts of the code. This is useful for long-term bigger projects that many people work on simultaneously as well as providing an extensible code-base.

3.1.1 MVC

Conforming to Apple's Model-View-Controller [18] pattern (MVC in short) is a good starting point for any iOS developer. Each of the three types of objects are separated in an abstract manner and communicate with others usually through some interface (protocol[5] in the context of Swift). By properly following this pattern, the objects tend to be fairly reusable and extensible. Figure 3.1 is a visual representation of this pattern.

As stated in the Apple MVC article[6], **Model** objects should encapsulate data specific to some context and define the logic and computation for processing this data. As per the definition, the model objects should *not* have an explicit connection with the view objects; it is the controller's responsibility to update the view based on the model changes. To provide an example from our application, we have

5. https://docs.swift.org/swift-book/LanguageGuide/Protocols.html
6. See footnote 4.

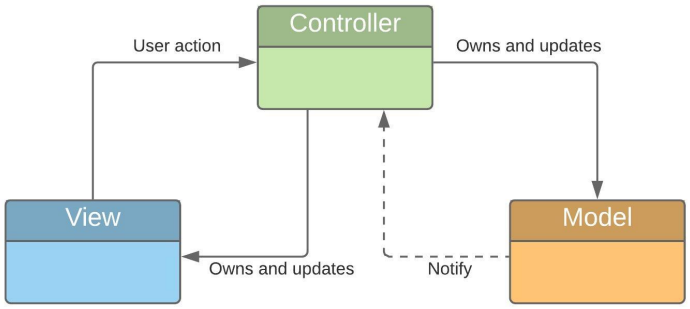


Figure 3.1: Model-View-Controller pattern

a model object that represents the data of a notification. It's only responsibility is to serve as a data structure and provide some very basic functions that modify the data.

The **View** object has the responsibility of receiving gestures and displaying the data from the Model objects. These are the objects that the user can see, and they also know how to draw themselves on the screen. It is fair to say that the view objects are one of the most reused objects in an application. In our case, some buttons were reused frequently, so the theme of the application is consistent across different screens.

The last object that acts as an intermediary between multiple view and model objects is the **Controller**. Controllers are the objects that hold the reference to their views and models while also updating them as needed.

Despite this wishful thinking of how the implementation of the pattern should look like, the correct adaptation in most MVC based iOS applications is often not executed properly. This is denoted by figure 3.2. It is relatively easy to make the mistake of having too much code in the controller classes, especially for inexperienced developers. This comes down to the fact that the controller classes are also the views (or `UIViewControllers`[7] in UIKit), and they have to manage some view layout and presentation code as well as some business logic. This

7. `https://developer.apple.com/documentation/uikit/uiviewcontroller`

can easily make the controller class too complicated and therefore the MVC abbreviation rather frequently represents a Massive View Controller.

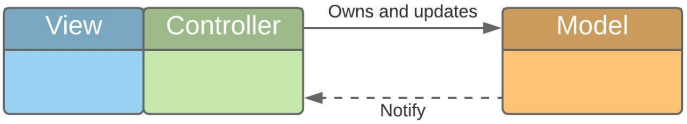


Figure 3.2: The undesired Massive View Controller

3.1.2 MVVM

Model-View-ViewModel is a pattern that evolved from MVC and is very valuable in the context of mobile applications, both iOS and Android. The fundamental distinction is that the controller also acts as the view and the business logic operations are transferred to another object, namely the **ViewModel**. Rather than the controller object owning the model, this becomes the responsibility of the ViewModel.

The controller is still somehow different from the pure view objects, particularly it owns and updates the ViewModel and the rest of the views. It also binds the appropriate views with the ViewModel. However, the data binding feature is not fully supported by UIKit on iOS, even though it is available on macOS in Cocoa. Many workarounds for this missing feature have been established since the popularity of MVVM on iOS has grown.

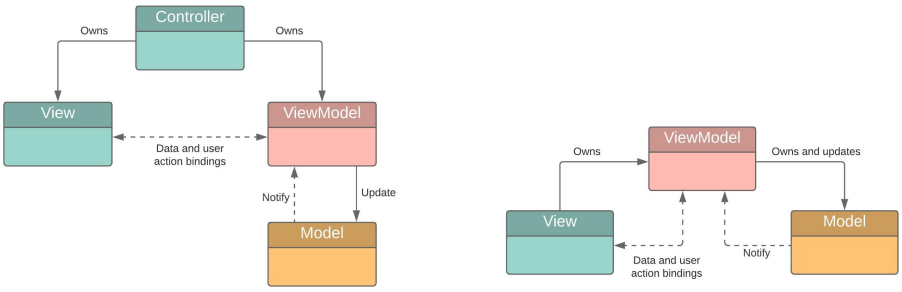
The most flexible and extensible one is a part of RxSwift[8], a third-party functional reactive programming framework. To use the data bindings, the framework must be added as a dependency to the project. However, without using the other features that RxSwift provides, it is usually left out in favour of simpler and less code heavy solutions.

Another possibility is to use the rather unpopular Key-Value-Observing API[9] which is an implementation of the Observer pattern

8. https://github.com/ReactiveX/RxSwift
9. https://developer.apple.com/documentation/swift/cocoa_design_patterns/using_key-value_observing_in_swift

that is provided by the standard library. Due to its various side effects like unpredictable performance, it is often left out as a last resort.

The solution used in the iOS application was to define ViewModel callbacks that are invoked when the model changes. This can also be done via the delegate pattern the decision is up to the preferences of the programmer.



(a) Explicit notation. (b) Simplified notation.

Figure 3.3: Model-View-ViewModel pattern

Figure 3.3(a) showcases the MVVM model with an emphasis on ownership of objects, the view and controller are generally considered to have the same behaviour. In the context of iOS and UIKit the `UIViewController` classes are a superset of `UIViews`. Each `UIViewController` also has a `view` property which is of the type `UIView` and is often the parent of many other `UIViews`. As per the MVVM standard, they are meant to be considered equal with each other, which is described by the simplified figure 3.3(b) (equal objects have the same color scheme).

Lastly, the MVVM pattern is beneficial when it comes to unit testing. With the usage of **dependency injection**[10](DI), which is just a fancy name for a simple concept. To be precise, specifying arguments (usually protocols) in an object's initializer. Hand in hand with DI comes the possibility to write mocks. By creating a mock of some class, we are essentially creating a fake version of its methods, where the implementation usually returns some predefined data. This way we can mock any external dependencies such as an API Service object or database management object and use these mocks in ViewModel initializers while unit testing. This approach was used since the start

10. `https://en.wikipedia.org/wiki/Dependency_injection`

of the development of the iOS application, as the server component wasn't implemented until the iOS application was fully functioning on mocks.

3.1.3 MVVM-C

Another fairly new and very promising design pattern worth mentioning is Model-View-ViewModel-Coordinator [19]. This pattern extends MVVM with the explicit requirement for an object that handles the presentation of controller objects, the **Coordinator**. Following this pattern requires a hierarchy of coordinators that have a root coordinator which should be initialized at the start of the application. This approach proves its value when it comes to viewcontroller reusability, and especially the reusability of only a part of some view hierarchy. Figure 3.4

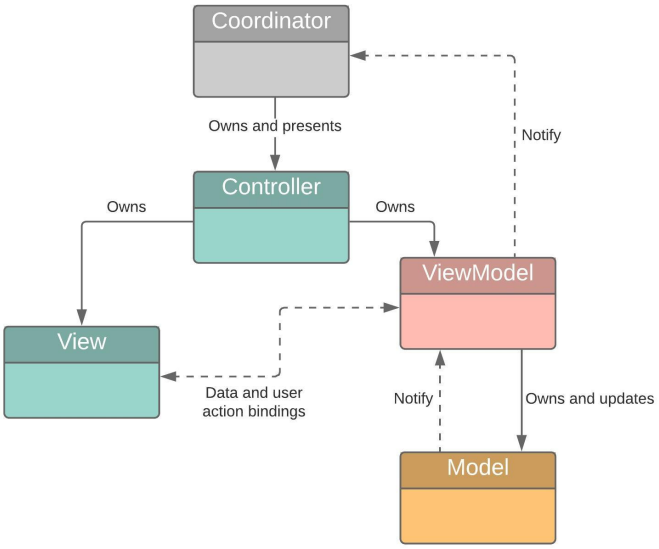


Figure 3.4: Model-View-ViewModel-Coordinator pattern

One of the disadvantages is that we need to define a new set of protocols that are used as the communication interfaces from the

ViewModel to the coordinator. More precisely the coordinator becomes the delegate of the ViewModel object and reacts to the delegate callbacks by changing the view controller hierarchy. There are also some cases where it makes sense to require a delegate directly from the viewcontroller, especially when a tiny `UIViewController` doesn't require a viewmodel (e.g. some alert view controller). As we have previously mentioned in section 3.1, these patterns are adaptable.

3.2 UI Overview

When it comes to prototyping and describing an iOS app screen flow a good tool that comes bundled with Xcode is Apple's **storyboards**. Besides the ability to create functional `UIViewController` layouts they provide a nice overview of the application flow. They are a great WYSIWYG[11] tool, however, they have their drawbacks. For example, storyboards are XML files and their presentation is handled via Xcode. If two or more people work on them at once, it can potentially result in an immense amount of merge conflicts. Also their slow load times for bigger projects are rather unpleasing to experience. They also include segues which are bound to the `UIViewController` classes and are used to denote the view controller flow.

However this defeats the purpose of the MVVM-C architecture, which was chosen for the iOS application. In this architecture the flow should be handled by the *Coordinators*.

Because of the fact that the autolayout for the entire Notifire application was done without the use of storyboards the entire screen flow has to be provided with some visual representation of the Coordinator classes. All coordinators that were used are depicted in figure 3.5. Before more protocols are introduced in this section, it is fair to say that appending the "-ing" or "-able" suffixes behind a type name indicates that it is a protocol.

11. `https://en.wikipedia.org/wiki/WYSIWYG`

Listing 3.1: The Coordinator protocol definition

```
protocol Coordinator {
    func start()
}
```

The initial application flow starts in the `AppCoordinator`. The coordinator is instantiated in the equivalent of a `main()` function in iOS applications, the `application(_:didFinishLaunchingWithOptions)` method of the `AppDelegate`. This method also includes the creation of the root `UIViewController` that is injected as a dependency to the `AppCoordinator` after being added to the main application window.

The protocol that defines coordinators (as shown in listing 3.1), has a single method, `start()`, that must be called after every coordinator initialization. To reiterate, the coordinators are responsible for the presentation of ViewControllers, this implies that the start method somehow presents the next view controller. The listing 3.2 provides a code sample of the `AppCoordinator start()` method which presents the initial view controller.

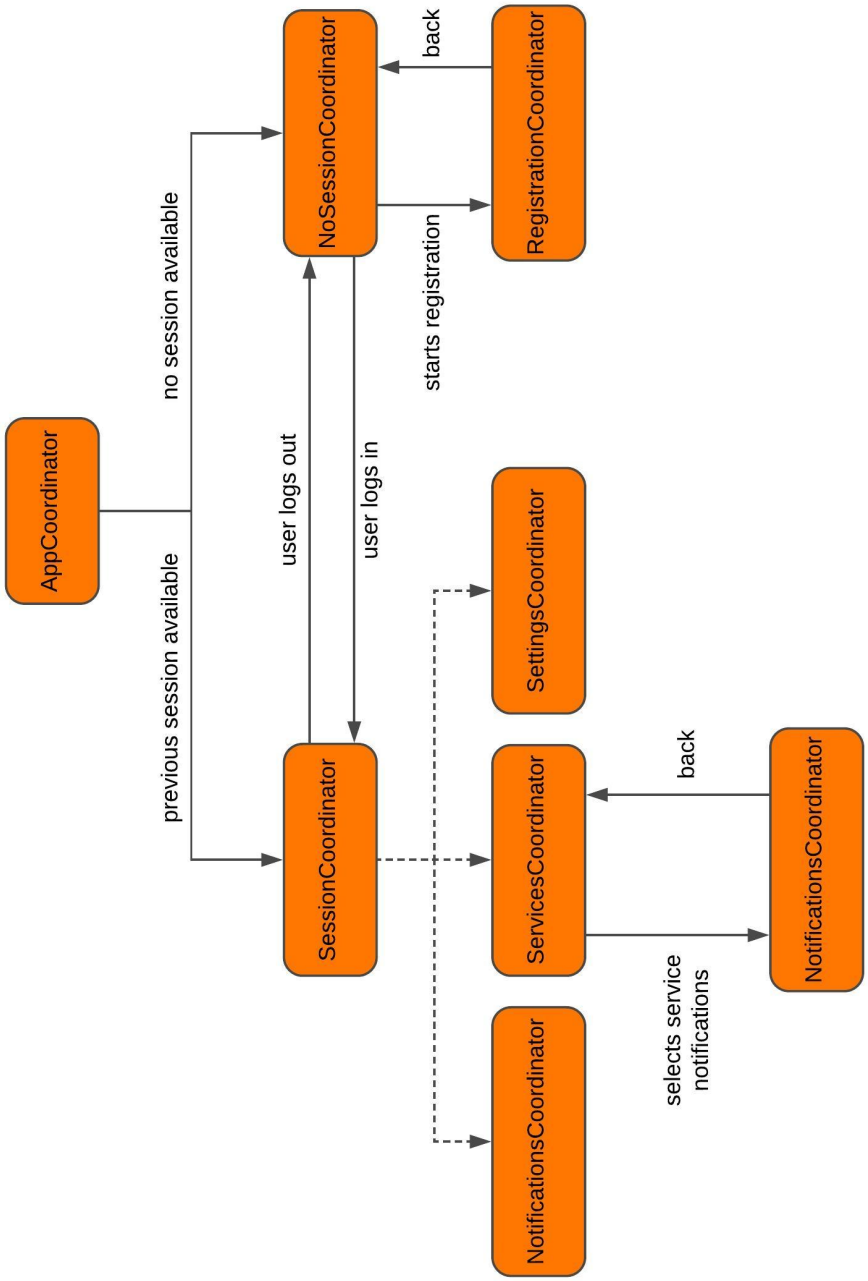


Figure 3.5: The coordinator hierarchy

Listing 3.2: Simplified start() method.

```
// NOTE: AppRevealing is a protocol that defines
    ↪ viewcontrollers that can be presented with the
    ↪ app reveal animation.
func start() {
    let revealingViewController: UIViewController &
    ↪ AppRevealing
    if let session = sessionManager.previousSession()
    ↪ {
        // previous session found
        revealingViewController = createSessionVC(
    ↪ session: session)
    } else {
        // no previous session stored in the Keychain,
    ↪  present the NoSessionCoordinator
        revealingViewController = createNoSessionVC()
    }
    rootViewController.add(childViewController:
    ↪ revealingViewController)
    // start the initial revealing animation
    revealingViewController.revealContent()
}
```

The initial subcoordinator is determined in the start method, more precisely by the `NotifireUserSessionManager`. The session manager checks if any previously created session data is available. This session is represented by a class `NotifireUserSession` and contains the refresh token and the username associated with a specific user. This confidential data is stored securely in the iOS Keychain, which is a secure storage implemented by Apple. If there is no user data stored in the Keychain, the session won't be available and `NoSessionCoordinator` will be the active subcoordinator.

3.3 Local database with Realm

A great part of the iOS application development was influenced by the **Realm** dependency. Essentially it is a NoSQL object store database, with many powerful features such as:

- *query* results are thread-local views, which are automatically updated when a write transaction is committed from any thread.
- every thread-local view returns proxy objects and as soon as their properties are accessed, the underlying data is provided. This is commonly referred to as lazy loading.
- notifications (or callbacks) that can be fired on collection or object changes.

Realm objects The objects that were used in the iOS application were necessary for the persistence of received notifications. Specifically, an object describing a service was used as well as an object for the received notifications. A shortened version of the `LocalService` and `LocalNotification` are provided in listing 3.3.

Listing 3.3: The Realm models

```
import RealmSwift
// Object is a class used to define Realm model
   ↪ objects
class LocalNotifireNotification: Object {

    // @objc dynamic var is needed
    // in order for these properties to become
    // accessors for the underlying data
    // (lazy loading)
    @objc dynamic var body: String? = nil
    @objc dynamic var urlString: String? = nil
    @objc dynamic var date: Date
    @objc dynamic var text: String? = nil
    @objc dynamic var rawLevel: String
    @objc dynamic var isRead: Bool = false
}

class LocalService: Object {
    @objc dynamic var name: String = ""
    @objc dynamic var uuid: String = ""
    @objc dynamic var serviceKey: String = ""
    @objc dynamic var updatedAt: Date? = nil
    @objc dynamic var info: Bool = true
    @objc dynamic var warning: Bool = true
    @objc dynamic var error: Bool = true
    // one-to-many relationship with notifications
    let notifications = List<
   ↪ LocalNotifireNotification>()
}
```

Notification Service Extension As soon as the notifications started arriving from our APNs connector component, their persistence also worked flawlessly. However when the iOS application was forcefully removed from memory, there was no callback that could have been

fired on the notification arrival as the application was in the **suspended** state.

The only solution was to provide a **Notification Service Extension**, which can modify the contents of incoming notifications notwithstanding the main application state. Despite the fact that we could potentially edit the incoming notification's content, it was soon realized that the extension lives in a different sandbox than our iOS App. The extension has a very simple API, basically it is a single class with a method that gets triggered when the notification arrives. There is also one additional constraint, the method has a limited amount of time to perform its task. Luckily this time was measured to be approximately 30 seconds. It is fair to say that iOS performs some heuristics and this time varies on a per user basis, but nonetheless it was acceptable for our use case.

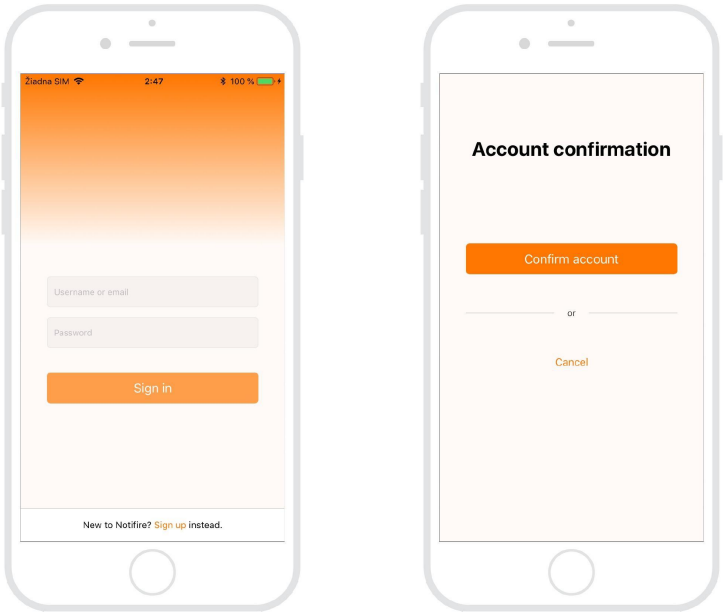
In relation with our realm database this meant one thing. The file where the database locates it's data had to be moved to a shared directory. This App-to-extension (and vice-versa) shared directory requires the setup of application groups via the apple developer portal as well as the project settings. After some trial and error this started to work. One more thing that was necessary was to enable access to the iOS keychain sharing, again between both, the iOS app and the extension. This was needed because of the username property of the current user session. The realm filename was related to the user, to ensure that multiple users can use the app at once and their notification data is persisted between the session changes.

After all of this work, the application was ready to save the notification data whenever the extension was instantiated by iOS. On the bright side, this allows us to seamlessly start developing a *watchOS* extension, because as the name suggests, it also has its own sandbox that behaves much like the Notification Service Extension.

3.4 Registration and login

These flows are a part of the `NoSessionCoordinator` class. The first view controller that is presented by this coordinator is the `LoginViewController` (figure 3.6(a)). Whenever the user decides to register he can do so by tapping the *Sign up* text in the bottom navi-

gator. This action creates a `RegistrationCoordinator` that presents a `RegisterViewController` to the screen. Each of these view controllers has a view model that is a subclass of `InputValidatingViewModel`. This implies that they have to handle some user input validation before their main action can be started.



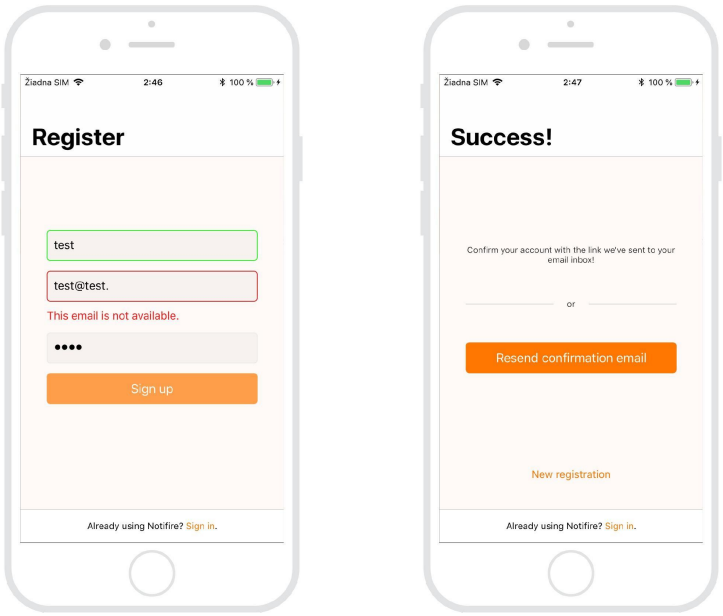
(a) `LoginViewController`

(b) Account confirmation screen, presented over any content.

Figure 3.6: View controllers of `NoSessionCoordinator` and `ConfirmEmailViewController`

3.4.1 Input validation

The user input validation is done each time the user types something into a custom component called `ValidatableTextInput`. Essentially it is just a wrapper that owns the native `UITextField` view. The text input class conforms to the `ValidatableComponent` protocol that requires an array of `ComponentRules` to be defined. With regard to Swift, a type that conforms to a protocol satisfies the protocol's requirements.



(a) Partially filled `RegisterViewController` (b) `RegisterSuccessVC`

Figure 3.7: View controllers of `NoSessionCoordinator` and `ConfirmEmailViewController`

Listing 3.4: Input validation

```
protocol ValidatableComponent: class {
    var rules: [ComponentRule] { get set }
    // ...
}

struct ComponentRule {
    enum Kind {
        case minimum(length: Int)
        case maximum(length: Int)
        case regex(NSRegularExpression)
        case equalToComponent(ValidatableComponent)
        case equalToString(String)
        case validity(CheckValidityOption)
    }

    let kind: Kind
    let showIfBroken: Bool
}
// used in the GET /check API endpoint
enum CheckValidityOption: String {
    case username = "username"
    case email = "email"
}
```

The `showIfBroken` property of the `ComponentRule` determines if the `ValidatableComponent` displays some error label if the specific rule is broken. Finally the input validation is done asynchronously and the rules are verified in sequence. As soon as one rule is broken, the other rules in the array are ignored. Listing 3.5 shows an instantiation of a username text field in the `RegisterViewController` class. The `validity` rule with a `CheckValidityOption` of `.username` is used. When the user starts to type, this triggers a `GET /account/check?username=` request.

Listing 3.5: Initialization of the username input

```
let input = ValidatableTextInput(textField:
    ↪ usernameTextField)
input.rules = [
    ComponentRule(
        kind: .minimum(length: 4),
        showIfBroken: false),
    ComponentRule(
        kind: .maximum(length: 64),
        showIfBroken: true),
    ComponentRule(
        kind: .validity(.username),
        showIfBroken: true)
]
// NOTE: for the sake of brevity the min/max lengths
    ↪  are shown as values
```

The email confirmation So far we have mentioned that the user can create his account via the `RegistrationCoordinator`. But the registration itself doesn't return any refresh tokens that are necessary for the `SessionCoordinator` to be initialized. There are two ways a user can receive the refresh token.

1. Login via the `LoginViewController`

2. Confirm his account via the URL that has been sent to his email address.

Each of these options provides a valid refresh token that can be used immediately. In addition to the refresh token, the `/account/login` and `/account/confirm` methods return an access token (their user error response payloads are the same).

In conclusion, whenever the user confirms a previously unconfirmed account and the current coordinator is the `NoSessionCoordinator`, the `AppCoordinator` switches to the `SessionCoordinator`. The same applies to the login action.

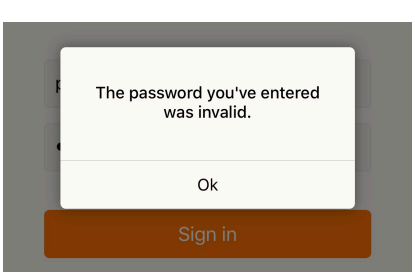
Handling user errors Following the discussion from section 2.1.2, the iOS application interface also has to handle user errors appropriately. Each error that is caused by the user is represented by a protocol `UserErrorRepresenting`.

View models that might produce these user errors are conforming to the `UserErrorFailable` protocol. And finally the views / viewcontrollers that can present the alerts are implementing the `UserErrorFailableResponding` protocol. Actually, there is no need to implement anything in the conforming view controllers. The default implementation that can be provided for any protocol in Swift is doing all the work. However, with a one place definition in mind, figure shows an alert that has one more action available besides the default *OK* button. Listing shows an example of an optional method in the `UserErrorFailableResponding` protocol that be provided when a custom action is needed. In conclusion, we achieve the desired functionality with a simple protocol, that has an extensible definiton and can be reused by any viewcontroller that wants to display user errors.

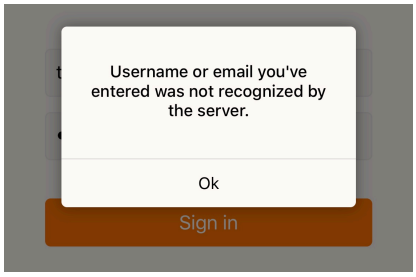
Listing 3.6: The `alertActions` method of `LoginViewController`

```
extension LoginVC: UserErrorFailableResponding {
    func alertActions(for error: LoginUserError,
   ↪ callback: ...) -> [NotifireAlertAction]? {
        guard viewModel.shouldHandleManually(
            userError: error
        ) else {
            return nil
        }
        let action = NotifireAlertAction(
            title: "Resend confirmation email",
            style: .positive,
            handler: { [unowned self] _ in
                self.viewModel.resendEmail()
                dismissCallback()
        })
        return [action]
    }
}
```

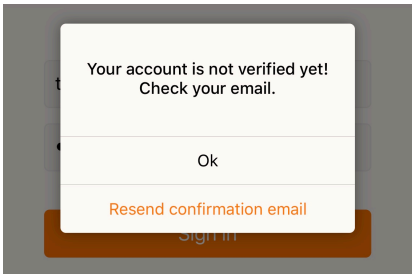
Listing 3.6 shows that `LoginViewController` conforms to the `UserErrorFailableResponding` protocol. In addition to inheriting the default protocol implementation for presenting user errors, the view controller provides its own definition for the custom alert actions. The default protocol implementation handles alert like shown in figures 3.8(a) and 3.8(b). Figure 3.8(c)



(a) Invalid password alert



(b) Invalid username alert



(c) Unverified account alert with a custom action from listing 3.6

Figure 3.8: Custom alert views handled by protocols

Protocols in Swift So far, a fair amount of protocols was described, in fact, the protocol oriented approach is strongly recommended by the Apple developer guidelines and should be used as frequently as possible. In some sense it provides a composable code-base that is not tightly coupled as opposed to using class inheritance.

3.5 The session screens

When the `SessionCoordinator` is instantiated a custom `TabBarViewController` is presented to the screen. This initial presentation also triggers the renewal of the access token via the `/account/register` endpoint. Furthermore the device token is also registered in the background.

3.5.1 Services

Classes that belong to the service group are all related to service and notification management. The main `UIViewController` that is responsible for displaying a list of services is the `ServicesViewController`. His viewmodel is listening for `LocalService` realm collection changes and notifies the controller accordingly.

Whenever there is no service associated with the user's account, an empty state is presented (figure 3.9(a). As soon as the user wishes to create a new service, the viewcontroller from figure 3.9(b) is presented by the `ServicesCoordinator`.

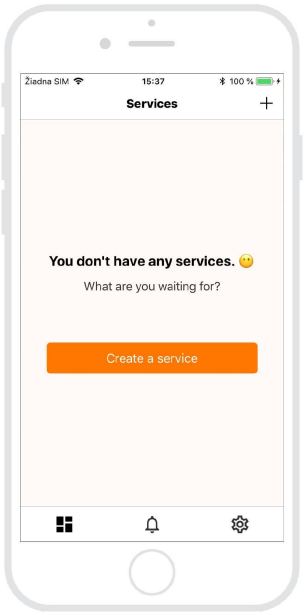
Service settings As shown in figure 3.9(d), the *Chocolate factory* service has each of the three notification levels enabled. They are represented by the three emojis and can be individually turned on or off. When the user changes the value of the notification level `UISwitch`, a `PUT /service` endpoint is requested with the new notification level.

Furthermore the **service key** can be changed by pressing the appropriate tableview row. This action is allowed if the user enters his password into the alert view and is verified via the notifire API by the `POST /service/key` endpoint.

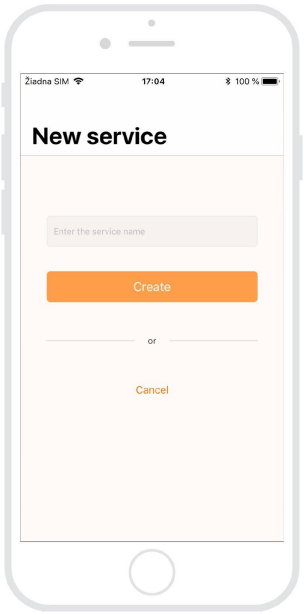
The service can be deleted via the `DELETE /service` endpoint, this also deletes the notification data associated on a device. The notifications for each service can be deleted as well.

3.5.2 Notifications

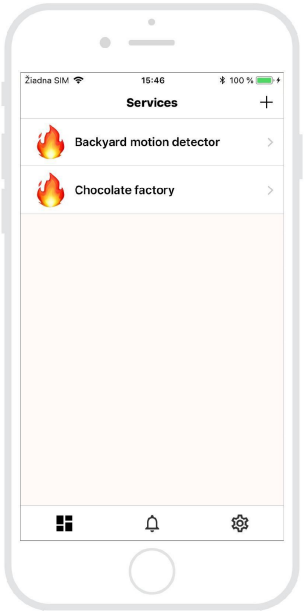
The notification view controllers are handling the displaying of the notifications. Namely, the `NotificationsViewController` with a `NotificationsViewModel` is presenting the middle tab, list of all notifications in the application. When a `NotificationsViewController` is initialized from the service



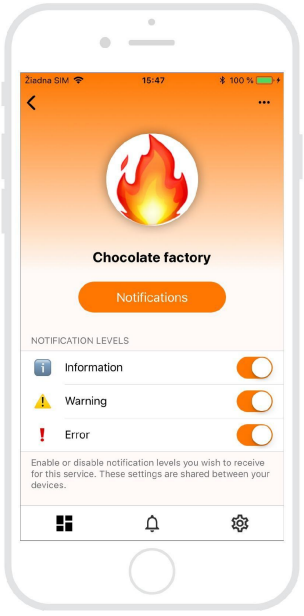
(a) `ServicesVC` in an empty state



(b) `ServiceCreationVC`



(c) `ServicesVC` with a populated tableview

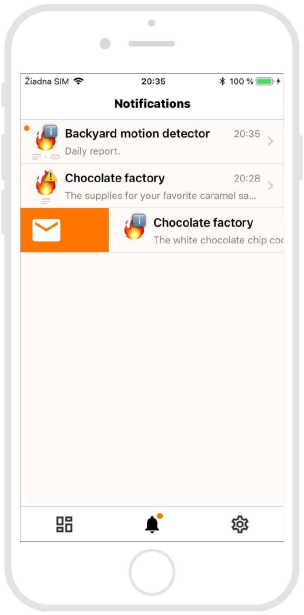


(d) `ServiceViewController`

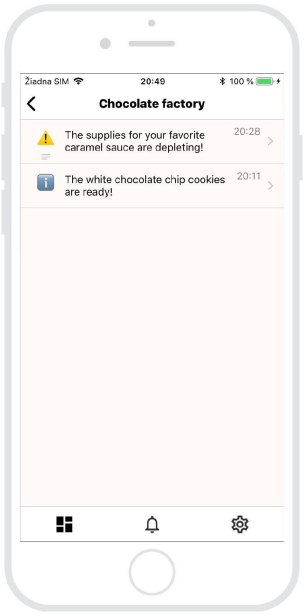
Figure 3.9: View controllers of `ServicesCoordinator`

view controller, indicating that only the notifications for this specific service should be displayed a `ServiceNotificationsViewModel` is used instead. This can be observed in figures

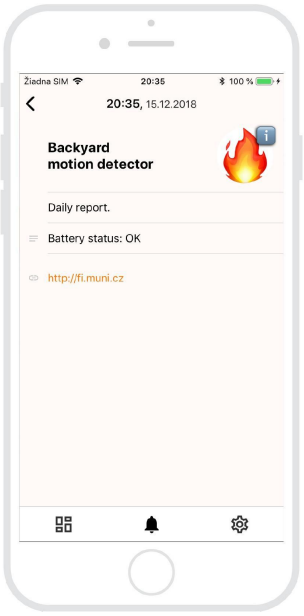
Each notification can be marked as read, and the proper realm notifications are delivered to every viewmodel that is listening.



(a) `NotificationsVC` with `NotificationsVM`



(b) `NotificationsVC` with `ServiceNotificationsVM`



(c) `NotificationsDetailVC`

Figure 3.10: View controllers related to notifications

4 Notifire CLI

The last component that is provided but not necessarily needed for our system is the Notifire command line interface (or the client). Also referred to as the client middleware, or Notifire-client. Originally, this component was meant to be provided in the form of a library or a module that could be added as a dependency into any project implemented in the most popular programming languages (such as Python, Javscript and so on). However, the simple command line interface is a viable alternative as the actual intent was to provide a rather simple tool for testing purposes.

Moreover the entire code for the Notifire client is just a wrapper for a single `POST /notify` request that is available in the Notifire API. Simply put, everyone that is willing to send a notification can access the API directly without using any additional dependencies.

4.1 Command line arguments

The CLI would not be of much use if we could not specify the notification parameters. For this purpose the executable has two required arguments and also a few optional ones. Two **required** arguments:

- `--key` - The service key will determine which service will receive the notification (must match the service key for the service that wants to notify something)
- `--body` - The main message of the notification, will be displayed on a lock screen, if the app is not in the foreground.

Besides the required ones, we can specify these **optional** arguments:

- `--level` - the sender can specify the notification level by supplying this argument, defaults to 1 which is the information level.
- `--text` - specifies some additional text even though this argument doesn't appear in the default iOS notification UI. Nevertheless, it is used in the notifire iOS application. It might be

useful in such systems where Notifire-CLI is used as a logger in catch-all exceptions handlers, i.e. the stacktrace could be specified as a `--text` argument.

- `--url` - much like the `--text` argument, the `url` is not displayed by iOS. The sender can specify some URL that is associated with the notification.

- `--apiurl` - this argument is only useful for debugging purposes, specifies the url of the Notifire server (e.g. `--apiurl=http://localhost:8080`)

Note that the `url` and `text` arguments are handled and displayed by the Notifire app, as shown in figure 4.1(b). A small URL icon right below the service image indicates that the notification has an URL associated with it. This URL can be opened in the `NotificationDetailViewController` screen which is presented after tapping the notification from the list.

4.2 Example of use

First, we need to select a service and then copy its service key, as shown in the figure 4.1(a). We have chosen the `Backyard motion detector` service whose intent is to notify the user whenever some movement is detected in the backyard while the sensors are turned on. The next step is to call the Notifire-CLI executable with the appropriate arguments. Listing 4.1 showcases a successful send operation via the CLI executable.

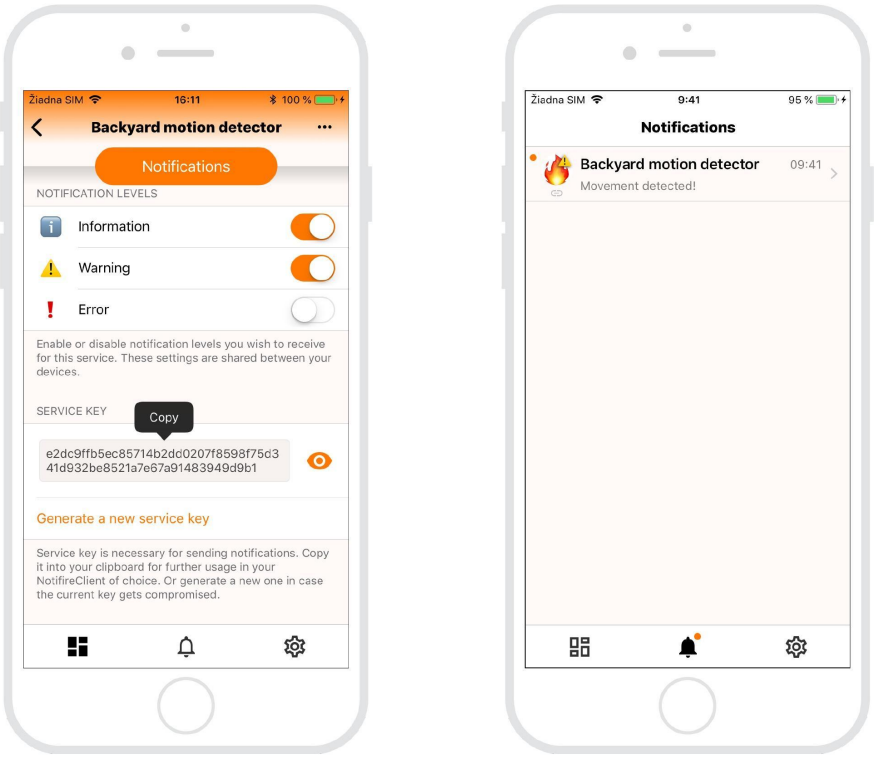
Listing 4.1: Example of a successful notification request

```
dvdblk@Artemis $ ./notifire-cli \
>     --key=e2dc9ff... \
>     --body=Movement\ detected! \
>     --level=2 \
>     --url=https://imgur.com/somepicture
Sending notification.
Success!
```

The notification is received and persisted on all iOS devices that are associated with the user and have their level for warning notifications (2) enabled (figure 4.1(b)). However if an attempt to send a level 3 notification is made, the executable gracefully fails while providing an appropriate error message. This is denoted by listing 4.2.

Listing 4.2: Example of an unsuccessful notification request

```
dvdblk@Artemis $ ./notifire-cli \
>     --key=e2dc9ff... \
>     --body=The\ camera\ battery\ has\ died! \
>     --level=3 \
Sending notification.
Error: notification not sent due to user
   notification level settings.
```



(a) Service key retrieval (b) The received notification

Figure 4.1: Example of usage in the iOS app.

5 Future plans and challenges

5.1 Ensuring consistency

One of the exciting features that didn't quite make it into this version of the iOS app was a queue for service requests. Its purpose is best described by this following example:

- Imagine that a user with a very bad internet connection successfully creates two services inside the app.
- His internet connection drops for a few seconds and he proceeds to change the notification settings of both services while also deleting one of them in this time frame.
- A possibility to update a non-existent service exists in the current implementation due to the nature of TCP. Namely, the service *PUT* requests might reach the server after the *DELETE* request.

This scenario would have been eliminated with the use of a proper serial `OperationQueue` for all service requests coming from the iOS application. Each request would be enqueued and executed in a FIFO manner.

5.2 Upcoming features

The very next destination this project will have is the submission of the iOS app to the App store. However while developing this system many new features that would be challenging and fun to implement were discovered and noted for further releases. These are the ones that were noteworthy of being placed on this list:

- Besides the queue for service requests, the iOS app strives for service image changing functionality. This will be done before the app is submitted for App Store review [1].

1. `https://developer.apple.com/app-store/review/`

- *Webhooks* were one of the original ideas. Each service would have it's own webhook settings that will be triggered after every `POST /notify` request. This idea came up because of the Phillips Hue[2] light system in my office. It has an API that allows changing the colors and light intensity. A simple middleware would have to be created between the webhooks and the Hue API. The lights would most likely change colors for a brief moment on notification delivery.
- A feature that was considered from the beginning was a web UI for the notification settings / a proper front-end.
- Extending and replacing the service REST API with websockets would be a nice touch that would play well with the web front-end.
- watchOS extension for the Apple Watch.

2. `https://developers.meethue.com`

Bibliography

1. SOMMERVILLE, Ian. *Software Engineering*. 9th ed. Harlow, England: Addison-Wesley, 2010. ISBN 978-0-13-703515-1.
2. *APNs overview* [online]. Apple, 2018 [visited on 2018-12-07]. Available from: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html.
3. *Introduction to JSON Web Tokens* [online]. jwt.io [visited on 2018-12-13]. Available from: https://jwt.io/introduction/.
4. WATTS, Stephen. *Microservices vs SOA: What's the difference?* [online]. 2017 [visited on 2018-12-12]. Available from: https://www.bmc.com/blogs/microservices-vs-soa-whats-difference/.
5. *Firebase Authentication* [online]. Google [visited on 2018-12-14]. Available from: https://firebase.google.com/docs/auth/.
6. *Work Queues* [online]. RabbitMQ, 2017 [visited on 2018-12-12]. Available from: https://www.rabbitmq.com/tutorials/tutorial-two-swift.html.
7. *Kafka* [online]. Apache [visited on 2018-12-14]. Available from: https://kafka.apache.org.
8. *RABBITMQ: UNDERSTANDING MESSAGE BROKER* [online]. 3pillarglobal [visited on 2018-12-15]. Available from: https://www.3pillarglobal.com/insights/rabbitmq-understanding-message-broker.
9. *Part 4: RabbitMQ Exchanges, routing keys and bindings* [online]. cloudamqp, 2015 [visited on 2018-12-15]. Available from: https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html.
10. *Allowing Apps and Websites to Link to Your Content* [online]. Apple, 2018 [visited on 2018-12-15]. Available from: https://developer.apple.com/documentation/uikit/core_app/allowing_apps_and_websites_to_link_to_your_content.

11. BILBIE, Alex. *A Guide To OAuth 2.0 Grants* [online]. 2018 [visited on 2018-12-15]. Available from: https://alexbilbie.com/guide-to-oauth-2-grants/.

12. SAJADI, Khash. *8 Components You Need to Run Containers in Production* [online]. Cloud66, 2017 [visited on 2018-12-13]. Available from: https://blog.cloud66.com/8-components-you-need-to-run-containers-in-production/.

13. *Docker (software)* [online]. Wikipedia, 2018 [visited on 2018-12-08]. Available from: https://en.wikipedia.org/wiki/Docker_(software).

14. *Overview | Docker network* [online]. Docker, 2018 [visited on 2018-12-08]. Available from: https://docs.docker.com/network/.

15. *safeAreaLayoutGuide* [online]. Apple, 2018 [visited on 2018-12-12]. Available from: https://developer.apple.com/documentation/uikit/uiview/2891102-safearealayoutguide.

16. *NSLayoutAnchor* [online]. Apple, 2018 [visited on 2018-12-13]. Available from: https://developer.apple.com/documentation/uikit/nslayoutanchor.

17. *What is decoupling and what development areas can it apply to?* [online]. StackExchange, 2014 [visited on 2018-12-08]. Available from: https://softwareengineering.stackexchange.com/questions/244476/what-is-decoupling-and-what-development-areas-can-it-apply-to.

18. *Model-View-Controller* [online]. Apple, 2018 [visited on 2018-12-08]. Available from: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html.

19. *Coordinators Redux* [online]. Khanlou, Sourosh, 2015 [visited on 2018-12-09]. Available from: http://khanlou.com/2015/10/coordinators-redux/.