DESIGN AND IMPLEMENTATION OF THE NOTIFICATION SYSTEM: *Project YASMIN*

TECHNICAL REPORT

Petro Liashchynskyi[*]
Department of Computer Engineering
West Ukrainian National University

ABSTRACT

In this document, I design and implement YASMIN (Yet Another System to Manage Instant Notifications) that supports sending notifications using any channel (e.g. SMS, Email, Sockets). The software includes abstract classes and entities that allow the building of further notification systems depending on the concrete project. YASMIN also provides capabilities to write as many channel implementations as you want.

Keywords OOP · Notification system · Design and Implementation · Software Engineering

1 Introduction

How many times have software developers solved the problem of sending notifications? Quite a lot, I think. And I bet that implementation differs from project to project. First, you start sending only email notifications, but as the project grows, you realise that now your customer wants to integrate different notification channels. They could include Slack messages, SMS, Push, and many more. To deal with that, there should be a simple unified system that could handle all those channels easily.

During my research, I encountered a few notification systems that could help to solve this problem. But some of them are not what I was looking for, some are too complicated to integrate into a small project [1, 2].

What I propose is to design a high-level, abstract system that can be easily extended to meet your project needs. It is important to understand that YASMIN is not a full implementation, but only the idea. It only gives you basic "flow" of how to extend your system further. You still need to write your channel providers (e.g. Sendgrid integration for Email channel, Twilio for SMS, etc.).

I have implemented notifications using this approach on several projects and it works just fine!

If for some reason YASMIN basic implementation is not suitable for your project (notification entity or provider should have additional fields and methods, etc.), you can always rewrite it or create your own implementation following the architecture and main concept (which is very easy as you will see).

2 Design

The main requirement of the system is to handle several notification channels at the same time and be easily extendable. For now, we can think of the architecture that might look like Figure 1. We have an email notifier that sends a message to a bunch of email addresses. To extend receiving channels we create a new class that extends the base Email Notifier. It is quite simple, isn't it?

Of course it is simple and this implementation will work. You just create a class for each notification channel and use it. But it brings problems if we want to send notifications through different channels at the same time. With this

[*]https://liashchynskyi.net

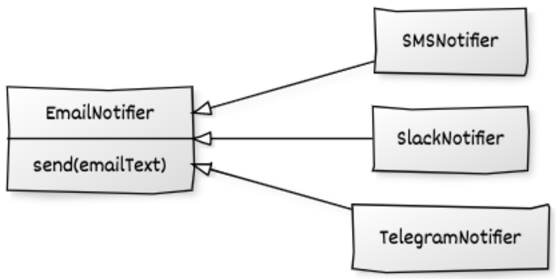

Figure 1: The simplest design

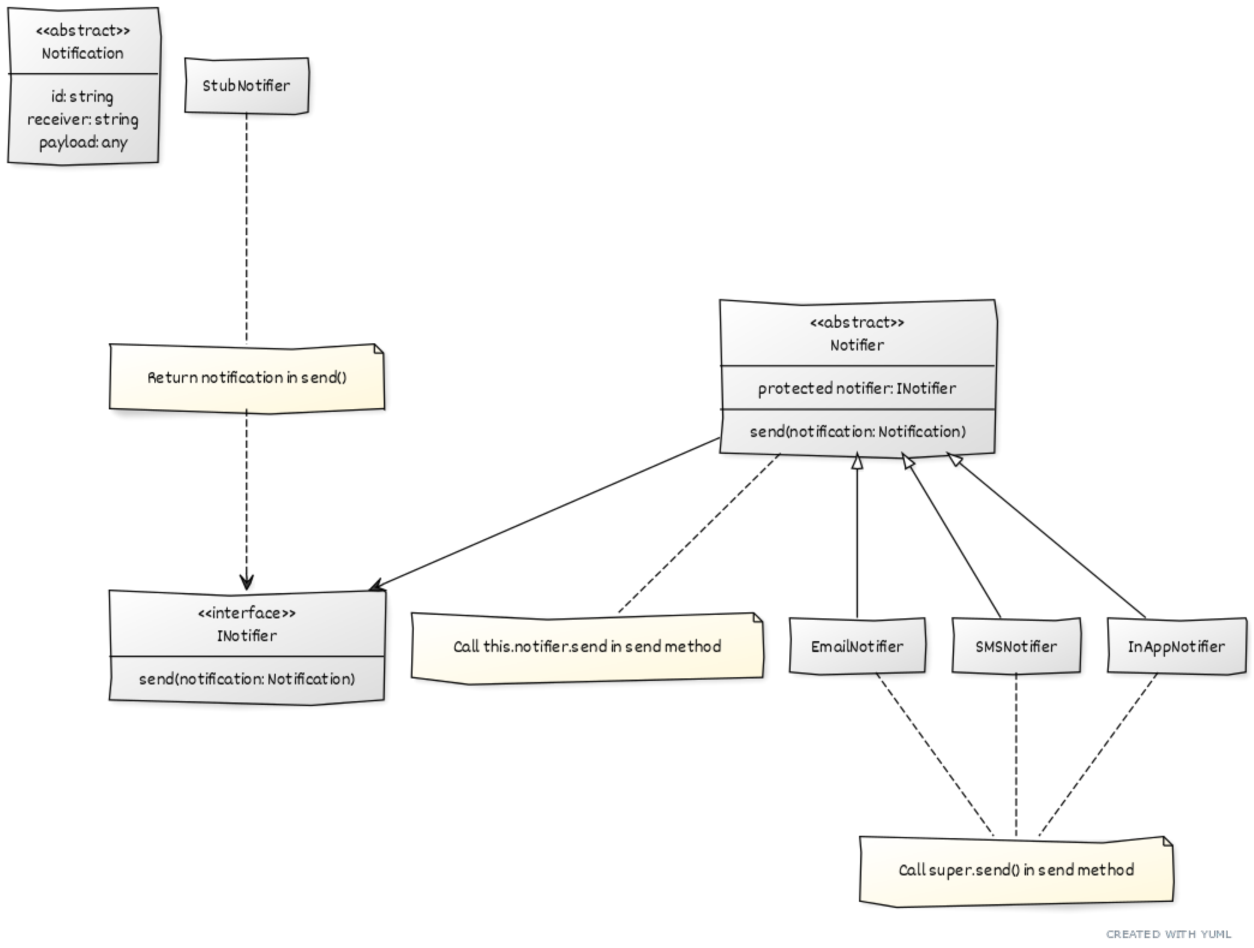
<<abstract>>
Notification
id: string
receiver: string
payload: any
StubNotifier
Return notification in send()
<<abstract>>
Notifier
protected notifier: INotifier
send(notification: Notification)
<<interface>>
INotifier
send(notification: Notification)
Call this.notifier.send in send method
EmailNotifier
SMSNotifier
InAppNotifier
Call super.send() in send method
CREATED WITH YUML

Figure 2: New structure

implementation, we would need to somehow combine existing notifiers. Something like this: we must have Email + SMS Notifier, Email + Slack Notifier, Email + SMS + Slack, etc. You got the idea: we must create a combined type with each of the existing ones. That is not a very clean and good approach. Because the number of total channel combinations will grow alongside channel numbers that need to be implemented.

We can choose another and more elegant way. We could decorate our providers. With this approach we'd have a base Provider with only one method `send()` and a bunch of other providers (email, sms) that will implement the Provider bahaviour. This approach gives something like chain method execution. For example, we first send Email notification, then SMS, then Slack, etc. The best thing about it is that we can configure chain of providers for each user in the system.

As can be seen from Figure 2, the notification system consists of a few classes. Let's discuss the code implementation.

3 Implementation

Firstly, let's create some interfaces and abstract classes.

```
export interface Notification {
   id: string;
   receiverId: string;
   payload: any;
}

export class TestNotification implements Notification {
   id = "random-uui";
   receiverId = "user-uuid";
   payload = { text: "hello" };
}

export interface INotifier {
   send(notification: Notification): Promise<Notification>;
}

export abstract class Notifier implements INotifier {
   protected constructor(private readonly notifier: INotifier) {}

   async send(notification: Notification): Promise<Notification> {
      return this.notifier.send(notification);
   }
}
```

`Notification` interface is used as the basic notification that other notifications will inherit from. There are a lot of properties that your project might need here, but you can start with mandatory ones such as ID of the notification, ID of the notification receiver (usually id of the user in database), and notification payload (this is the data, you want to transmit).

`Notifier` is the abstract class that other providers for each of the receiving channels must extend. Let's define some providers (aka notifiers).

```
export class StubNotifier implements INotifier {
   async send(notification: Notification): Promise<Notification> {
      return notification;
   }
}
```

The next notifier is Email. Remember that you can pass needed dependencies in constructor.

```
export class EmailNotifier extends Notifier {
   constructor(notifier: INotifier, private readonly emailService: any) {
      super(notifier);
   }

   async send(notification: Notification): Promise<Notification> {
      // Add logic to send via Email here and call super
      // Your code goes here ->>>

      return super.send(notification);
   }
}
```

SMS notifier goes below.

```
export class SmsNotifier extends Notifier {
   constructor(notifier: INotifier) {
      super(notifier);
   }

   async send(notification: Notification, private readonly smsService: any): Promise<Notification>
       {
      // Add logic to send via SMS here and call super
      // Your code goes here ->>>

      return super.send(notification);
   }
}
```

This is the example how to use it.

```
let notifier = new DefaultNotifier();

// Check if receiver has Email channel enabled
const emailService; // sendgrid, mailchimp, etc.
notifier = new EmailNotifier(notifier, emailService)

// Check if receiver has SMS channel enabled
const smsService; // AWS SNS, Twilio, etc.
notifier = new SmsNotifier(notifier, smsService)

const notification = notifier.send(new TestNotification())

// Do further processing, save to database, etc.
```

4 Discussion

With the YASMIN approach, you can configure as many providers as possible. Just extend the base class, put the logic, call the send from the superclass and that's it. The main advantage is that you can enable/disable some of the providers based on the receiver's notification preferences.

Of course, you can (and you should) move the above code to a `send` method of service, let's say `NotificationService`, and that's when it gets beautiful.

When you need to send a notification, you create a class with all the required info (see `DefaultNotification` example) and call `NotificationService.send(notification)`. The OOP will do the rest for you. And here comes the YASMIN: a notification system idea with countless abilities.

References

[1] Amazon sns. `https://aws.amazon.com/sns/`. Accessed: 2022-11-11.

[2] Google cloud messaging. `https://firebase.google.com/docs/cloud-messaging`. Accessed: 2022-11-11.