Assignment of master's thesis

Title:	Haskell Framework for Webhooks Implementation
Student:	Bc. Vojtěch Balík
Supervisor:	Ing. Marek Suchánek, Ph.D.
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Webhooks provide a flexible way to integrate various applications through automated messages triggered when something happens based on their configuration. This thesis aims to design and implement a framework for Haskell that will support the webhooks specification, triggering and sending messages in web applications. It should provide an intuitive interface to programmers and versatility in configuration.

- Research briefly webhooks and their use in current well-known applications.
- Provide an overview of existing webhook-supporting frameworks and libraries.
- Specify requirements and design your solution that fulfils the requirements.
- Implement the framework based on the design and test it. Justify your selection of used technologies. Provide examples of use as part of the documentation.
- Prepare the project for open-source distribution in the Haskell community.
- Evaluate your framework and compare it to existing solutions. Discuss possible future development.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 22 June 2022 in Prague.

Master’s thesis

Haskell Framework for Webhooks Implementation

Bc. Vojtěch Balík

Department of Software Engineering
Supervisor: Ing. Marek Suchánek, Ph.D.

February 15, 2024

Acknowledgements

I would like to thank my parents, to whom I owe so much. I also would like to thank my supervisor, Ing. Marek Suchánek, Ph.D., for his patience, and encouragement while advising me on this thesis, despite my shortcomings.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on February 15, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Vojtěch Balík. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Balík, Vojtěch. *Haskell Framework for Webhooks Implementation.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Webhooky jsou metoda umožňující vlastní zpětná volání ve webových aplikacích na základě protokolu HTTP. Webhooky jsou se stávají stále populárnějšími ve webových službách a byť se jejich implementace může na první pohled zdát nenáročná, přinášejí s sebou specifické výzvy, zejména pokud se jedná o zabezpečení a spolehlivost komunikace. Tato práce zkoumá webhooky a související nejlepší postupy a následně navrhuje framework, který poskytuje základ pro integraci webhooků do webových aplikací za použití zmíněných nejlepších postupů. Výsledkem práce je volně dostupný a snadno rozšiřitelný softwarový balíček pro programovací jazyk Haskell.

Klíčová slova Webhooks, REST, API, events, Haskell, framework.

Abstract

A webhook is a method of enabling custom callbacks in web applications over the HTTP protocol. Webhooks are becoming increasingly popular in web services. Although webhooks may seem easy to implement on the surface, they come with a specific set of challenges, especially when it comes to securing event deliveries and making them reliable. This thesis researches webhooks and related industry best practices and designs a framework for integrating webhooks in web applications that implements said best practices. The result of this work is an open-source and easily extensible software package for the Haskell programming language.

Keywords Webhooks, REST, API, events, Haskell, framework.

Contents

Introduction **1**

1 Goals and Methodology **3**

2 Webhooks and State-of-the-Art **5**
2.1 Events and Web APIs . . . 5
2.2 Webhooks Introduction . . . 6
2.3 Webhook Interactions . . . 6
2.3.1 Overview . . . 6
2.3.2 Subscribing To Events . . . 7
2.3.3 Event Delivery . . . 7
2.3.3.1 Retry Mechanism . . . 8
2.3.3.2 Event History . . . 9
2.3.4 Event Data . . . 9
2.3.4.1 Thin vs Thick Event . . . 10
2.4 Securing Webhooks . . . 12
2.4.1 Event Authenticity . . . 12
2.4.2 Webhook Consumer Authenticity and Encryption . . . 12
2.4.3 Webhook Endpoint Idempotency . . . 12
2.4.4 Replay Prevention . . . 13
2.4.5 Server-Side Request Forgery (SSRF) . . . 13
2.4.6 Event Sender IP Whitelisting . . . 13
2.5 Standard Webhooks . . . 14

3 Existing Frameworks and Libraries **15**
3.1 Thorn . . . 16
3.1.1 Events . . . 16
3.1.2 Subscriptions . . . 16
3.1.3 Dispatchers . . . 16
3.1.4 Reliability . . . 17
3.1.5 Security . . . 17
3.2 Svix . . . 18
3.2.1 Reliability . . . 18
3.2.2 Security . . . 19

3.2.3 Architecture . . . 19

4 Requirements **21**
4.1 Functional requirements . . . 21
4.2 Non-functional requirements . . . 21
4.2.1 Ease of use . . . 21
4.2.2 Flexibility . . . 21
4.2.3 Reliability . . . 22
4.2.4 Security . . . 22
4.2.5 Usability . . . 22

5 Implementation **23**
5.1 Whoopr Module . . . 23
5.1.1 Subscription Management . . . 23
5.1.2 Task Queue . . . 24
5.1.3 Event Sender . . . 24
5.1.3.1 Caveats . . . 25
5.2 Whoopr.Basic Module . . . 26
5.2.1 Concretizing Subscription Management . . . 26
5.2.2 Subscription API . . . 26

6 Evaluation **27**
6.1 Example Application . . . 27
6.1.1 Defining Application Environment . . . 27
6.1.2 Launching the Event Sender . . . 28
6.1.3 Sending Events . . . 28
6.2 Assessment . . . 28
6.3 Further Work . . . 29

7 Distribution **31**

Conclusion **33**

Bibliography **35**

A Contents of attachments **39**

List of Figures

2.1 GitHub form for registering a webhook for a repository. [1] 8
2.2 Sequence diagram of creating a webhook subscription. (Based on [2].) 9
2.3 Sequence diagram illustrating the delivery of single event notification. (Based on [2].) 10

3.1 Sequence diagram illustrating single event send through Svix. (Based on [3].) 19

Introduction

As modern web development evolves, so do the requirements for modern web services. Applications that integrate with web services need to be responsive and react to changes immediately. Thus, services must provide concepts such as events and notifications, which poses a problem, as such concepts are not supported by the common REST architecture [2].

A solution that is becoming more and more popular are *webhooks*. However, webhooks are not standardized, and so, for each webhook provider, the answer to the question "What are webhooks, exactly?" differs. General statements, such as "webhooks are event handlers for the web" or "a webhook is an HTTP callback", can be agreed upon, but details such as the problem of verifying webhooks not so much. This poses the question of what the advantages and disadvantages of the different designs that can be found among webhook providers are, and it is this question which the first part of this thesis tries to find the answer to. [2, 4]

Embedding webhooks in any single application presents a specific set of challenges due to this ambiguity. The second part of this thesis is concerned with designing a solution to enable the embedding of webhooks in *any* application. Attempting this only creates additional challenges, because one size does not fit all. The additional challenge becomes the integration with different pieces of technology and the need to accommodate different designs, without overcomplicating things. All this is emphasized by the use of a statically typed, lazy, and pure functional programming language like Haskell, as existing solutions that work well in the popular imperative languages may not be easily transferrable.

Chapter **1**

Goals and Methodology

The goal of this thesis is to develop a framework for the Haskell programming language to support webhook development in web applications. First, in the Webhooks and State-of-the-Art chapter, I research existing literature and other resources and provide a description of webhooks and how they fit in with web applications. In the following chapter, Existing Frameworks and Libraries, I review projects that solve similar problems.

Then, in the following chapters, I proceed according to the traditional methodologies of software engineering. Using the findings of these two chapters, in chapter Requirements I specify the requirements of my solution. The Implementation chapter then provides a description of the design and implementation of the solution. In the Evaluation chapter, I provide an example usage of the solution, assess my solution, and outline avenues for possible future work and improvements. Finally, chapter Distribution describes the process of open-sourcing the solution and making it ready for use by others.

Chapter 2

Webhooks and State-of-the-Art

2.1 Events and Web APIs

Traditional RESTful APIs are built around operations on *resources*. These operations correspond to HTTP verbs like GET, POST, DELETE, etc., with their precise meaning more-or-less consistent across API providers. However, this communication model does not fit well in situations where the API consumer wants to be notified about a change on the side of the API provider in a timely manner, or in other words, receive *events*. This stems from the synchronous nature of the HTTP protocol, where a client sends a request and expects a response immediately, while such notifications occur asynchronously [1] from the point of view of the client.

There are numerous approaches to this problem, each with its own advantages and disadvantages. With *polling*, the client simply repeats a request over and over and checks every response for a change. Other approaches use HTTP streaming, Server-Sent Events or WebSockets, where a single HTTP connection (or TCP connection in the case of WebSockets) between a client and the server is maintained indefinitely. With this connection established, the server can then send notifications to a client in an asynchronous manner. Compared to polling, this offers reduced overhead and latency in communication.

Yet another approach is to reverse the client-server roles for the notification communication. Here, the API consumer exposes an HTTP endpoint and makes it known to the API provider. The API provider can then send notifications as HTTP requests to the API consumer, and thus the API consumer, originally acting as an HTTP client, now acts as a server and vice-versa. It is *inversion of control*, a *push model* of communication. This approach comes with its own set of complications however, for example, due to security concerns it may not be viable to expose such endpoint to a public network, or the endpoint of the consumer may not be available at the time of the notification. Nonetheless, this is the approach employed with *webhooks*. [2]

[1] Not to be confused with asynchronous programming models found in many modern programming languages like JavaScript.

2.2 Webhooks Introduction

The term ”webhook” was originally coined by Jeff Lindsey in a blog post titled ”Web hooks to revolutionize the web” [5]. Other terms used are also ”reverse APIs” or ”push APIs” [6]. There is no formal description of webhooks, instead: ”What we call webhooks is merely a collection of concepts and a collection of best practices based on these concepts” [2, pp. 35]. These concepts are built around the approach to asynchronous communication mentioned above. Red Hat [6] defines a webhook as: ”an HTTP-based callback function that allows lightweight, event-driven communication between 2 APIs”.

Webhooks are similar to callbacks or event handlers – be it registering a callback or subscribing to some event, the user is registering a *hook*. With webhooks, the user (API consumer) creates a hook by registering an HTTP endpoint with an API provider. [2]

The role reversal involved in webhook communication may lead to confusion when it comes to terms like ”client” and ”server”, as the meaning changes depending on the context. I will use the terms *”webhook consumer”* and *”webhook provider”*, e.g.: when an event notification is sent to the webhook consumer, it acts as an HTTP server and the webhook provider acts as an HTTP client.

2.3 Webhook Interactions

2.3.1 Overview

In ”Webhooks: Events for REST APIs” [2], Biehl provides a summary of the roles and their respective responsibilities in webhook interactions. Webhook consumers need to:

- implement the webhook endpoint, and
- register the webhook endpoint with the provider.

For the webhook providers, the responsibilities are to:

- define possible events,
- manage webhook subscriptions, and
- deliver events to subscribers.

Webhook providers define their events in their developer documentation, usually alongside traditional API documentation. This definition includes the types of events; for example, PayPal [7] defines an event for when an invoice is created, another for when one is paid, and many more. It also includes descriptions of the data and metadata included in the event notification payloads, and offers information on how to properly handle these payloads, for example, how to verify payload signatures. The documentation will also often include an explanation of webhooks.

Webhook consumers implement webhook endpoints according to this documentation. To register these endpoints, i.e. subscribe to certain events, consumers will use a dedicated subscription API or a *webhook dashboard*, a web application, which provides webhook subscription management and is commonplace among webhook providers.

Once a subscription is established, the webhook provider will deliver event notifications to the webhook consumer.

2.3.2 Subscribing To Events

The only absolutely necessary part of creating a webhook subscription is to let the webhook provider know where to send notifications, i.e. URL of the webhook endpoint. However, providers also allow consumers to specify the events they are interested in. While consumers may filter events as they receive them, this approach allows consumers to manage the load imposed by sending or receiving large quantities of notifications, as the provider might impose quota or the consumer's server may become overwhelmed. [2]

Another very common and important subscription configuration among providers is a shared secret or a key that is required to ensure the security of the notification (see Section 2.4.1).

Figure 2.1 is a screenshot showing an example of a webhook registration form, where GitHub users can create a webhook for their repository. Here, "payload URL" corresponds to the aforementioned webhook endpoint. Using the "which events would you like to trigger this webhook?" option, the webhook consumer can select the events they are interested in, with one of the provided options, and also a good example being Git pushes made to this repository. These are sometimes called *topics*. The "secret" field concerns security, and its value is used to generate a hash-based message authentication code (HMAC) for each notification; more on that later in Section 2.4. [1]

The subscription process may also involve the provider sending a special notification to validate or verify the endpoint. For example, GitHub will send a "ping" event after creating the webhook by clicking "Add webhook" in the registration form. This "ping" is a test that the webhook endpoint responds correctly to the HTTP POST request of the notification by responding with a 2xx (200–299) status code. [1] Some webhook providers send a one-time verification challenge before sending any real notifications, which verifies that the creator of the subscription owns the webhook endpoint. The challenge notification will, for example, include a random string in its payload, which the webhook consumer must echo back in the response. This helps mitigate abuses where a malicious party uses the webhook provider infrastructure to perform unsolicited HTTP requests, such as distributed denial of service (DDoS) attacks. [8]

Figure 2.2 shows an overview of the communication involved in creating a subscription, including the optional one-time verification challenge.

2.3.3 Event Delivery

Figure 2.3 illustrates the HTTP communication involved in sending a single event to a single webhook endpoint. Note that the diagram does not show when does the consumer actually process the event. The preferred approach is for the consumer to respond to the provider immediately and process the event when convenient. This is because an untimely response, due, for example, to the webhook consumer being under high load, may unnecessarily result in more attempts at delivery, as will be described next. [2] (See also Section 2.4.3 on webhook idempotency.)

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, *etc*). More information can be found in our developer documentation.

Payload URL *

https://example.com/postreceive

Content type

application/x-www-form-urlencoded

Secret

Which events would you like to trigger this webhook?

- Just the push event.
- Send me **everything**.
- Let me select individual events.

Active

We will deliver event details when this hook is triggered.

Add webhook

Figure 2.1: GitHub form for registering a webhook for a repository. [1]

2.3.3.1 Retry Mechanism

Event deliveries may fail due to a myriad of reasons. For example:

- the notification HTTP request or its corresponding response got lost during transit, and the request reaches timeout on the webhook provider side, or
- the processing of the event fails on the side of the webhook consumer, and it responds with a 500 HTTP status code, or
- the webhook consumer was temporarily unavailable at the time of the delivery,
- etc.

Most webhook providers recognize this, and to make the process of event delivery more reliable, they will attempt to redeliver failed deliveries. Details of how long the delay between individual attempts is and what the maximum number of retries is vary between different webhook providers. A common and recommended strategy is to decrease the rate of delivery attempts using an *exponential backoff* algorithm (meaning that the delays between subsequent attempts form a geometric progression) and stop attempts after some days of retrying. [2, 8, 4]

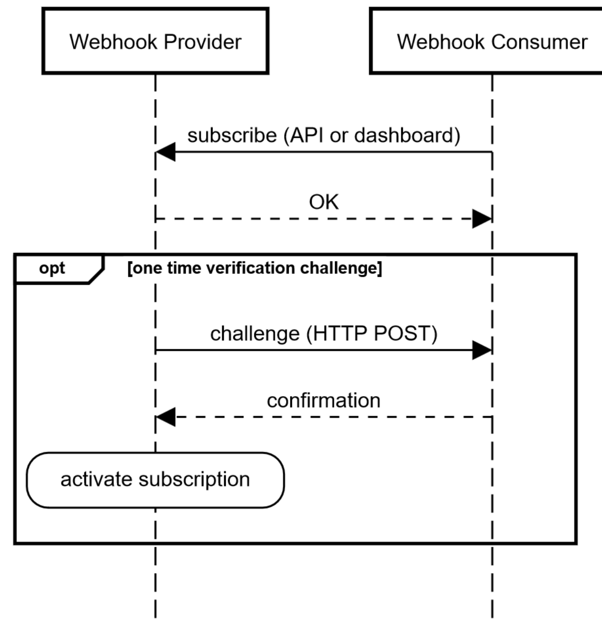

Figure 2.2: Sequence diagram of creating a webhook subscription. (Based on [2].)

What happens after the maximum number of retries is reached and the event remains undelivered varies between different webhook providers. If enough deliveries fail, webhook providers will usually disable the corresponding subscription, requiring attention and manual intervention from the consumer to reestablish it. For example, Adobe Acrobat Sign [9] events will be irrecoverably lost. Other providers implement some concept of *event history*. [2]

2.3.3.2 Event History

Some webhook providers store information about the events sent for each subscription and expose this information to webhook consumers via an API and/or the webhook dashboard. Using the API or the dashboard, consumers can inspect individual deliveries and their responses (useful for development and debugging), or trigger another delivery even if the maximum number of retries has already been reached (*resynchronization*). [2]

For example, Svix will retry with increasing delay for a little over a day. If the maximum number of retry attempts is reached without successfully delivering the event, the subscription is disabled. Svix retains all events (not just undelivered) for a duration of 90 days by default, during which the consumer can manually resynchronize. [3]

2.3.4 Event Data

The data describing the event is specific to the domain and the webhook provider; however, notifications share a lot in common across different providers when it comes to the metadata, i.e. data about the event data. Apart from

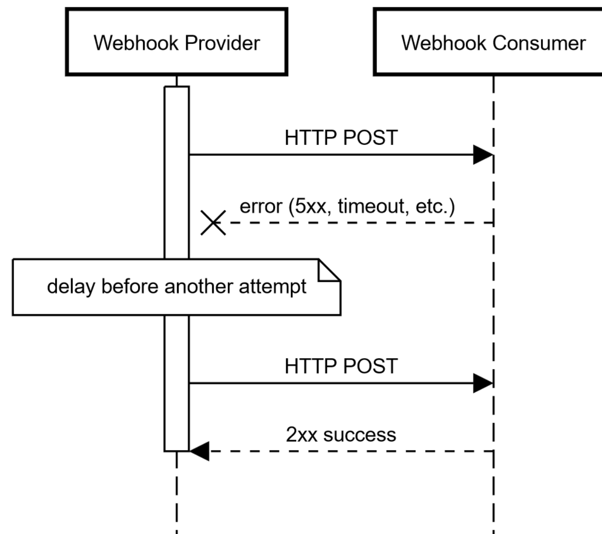

Figure 2.3: Sequence diagram illustrating the delivery of single event notification. (Based on [2].)

the normally metadata included in HTTP headers, such as `Content-Type`, the notification requests will include the information required by webhook consumers to correctly process the event. These metadata may be included in the notification HTTP request as custom headers or in the payload together with the event data. Common metadata include a unique identifier for the event, timestamp, type of the event, and signature.

Although event data may be sent in any format, as with REST APIs, JSON is the de facto standard.

2.3.4.1 Thin vs Thick Event

When designing events, webhook providers are faced with the question: Which data describing the change that has occurred in the system should be included in the event notification payload and which data should be left out? If the answer is to include the entirety of the affected resource and any relevant information about the change, the provider uses *thick events*. On the other hand, if the answer is to only include a URL of the affected resource and the type of event, the provider uses *thin events*. [2] Listing 1 and Listing 2 illustrate this on an event describing a new customer entity being created, showing an example of a thick and a thin event payload, respectively.

Biehl [2] argues for thin events as the more advantageous design that mitigates some of the problems related to delivering webhook events. One reason is security – no sensitive data is included in the payload; hence there is no data to be compromised. Sensitive data, if needed, must be accessed through a regular API. Standard Webhooks [4] and webhooks.fyi [8] also advocate for thin events, citing advantages such as better performance due to smaller pay-

[2] One may also encounter terms like "dataless notifications" for the concept of thin events, or "full events" for thick events.

```json
{
  "eventType" : "customer.created",
  "timestamp" : "2024-02-01T04:19:53Z",
  "eventId" : "1234",
  "data" : {
    "id" : "5678",
    "firstname" : "John",
    "surname" : "Doe",
    "address" : "Fake St. 1",
    "email" : "john.doe@example.com",
    "ssn" : "000-00-0000"
  }
}
```


Listing 1: Example of a thick event payload.


```json
{
   "eventType" : "customer.created",
   "timestamp" : "2024-02-01T04:19:53Z",
   "eventId" : "1234",
   "link" : "https://example.com/customers/5678"
}
```


Listing 2: Example of a thin event payload.

load sizes and flexibility. It is a more future-proof design, as more data can be included as the webhooks provided evolve in a backward compatible way, but not the other way around. Payload size should also be limited to some reasonable amount (Standard Webhooks proposal recommends approximately 20 kb), so as not to overwhelm the API consumer. This, again, becomes a non-issue with thin events, and traditional approaches like pagination may be used for large quantities of data once the consumer queries the linked resource.

The major disadvantage of thin events is the additional implementation complexity imposed on the webhook consumer, which needs to perform additional requests to gather the information necessary to process events.

Using thin or thick event payloads does not pose a binary decision for webhook providers; rather, they represent opposing ends of a spectrum. GitHub's event payloads, for example, fall somewhere in between; it is unrealistic for the payload to include much, let alone all, information about the affected git repositories due to their size, but it does include a lot more information than just the event type and a URL. For instance, a push event, triggered when a `git push` is performed on the repository, in its payload will include, along with the URL of the resource of the commit, information like the hash of the commit or a timestamp.

Stripe [10] uses thick events – the entire entity affected is sent together with the event metadata. In the case where the object was updated, it also sends the previous values of the affected fields. PayPal [7] event payloads, which, although not entirely thin, will also contain links to related resources and actions that the webhook consumer can perform on them, thus implementing

the hypermedia as the engine of application state (HATEOAS) principles.

2.4 Securing Webhooks

2.4.1 Event Authenticity

Webhook consumers need to be able to ensure the authenticity of the event and the integrity of the message. This can be achieved by signing the notification. [2, 11]

The most popular approach is to create a hash-based message authentication code (HMAC) from the sensitive data and metadata of the event and then include it in a header of the notification HTTP request. To create the HMAC, both parties, the webhook provider and consumer, need to know the same secret key. Recall the ”secret” field in the GitHub form for registering a webhook: Figure 2.1. [8]

Some use cases may require non-repudiation of notifications. However, this cannot be achieved with HMAC as there is a shared key; instead, asymmetric keys should be used.

With asymmetric keys, the webhook provider generates a private and public key pair, preferrably for every subscription. The provider then uses the private key to sign the notification and uses some other channel to distribute the public key; some, for example, will provide it in the webhook dashboard, and some will offer a URL to retrieve it from in the notification request. Compared to HMAC verification, this approach is more difficult to set up for webhook consumers and it is also computationally more expensive to sign and validate events.

Another option is to rely on mutual transport layer security (TLS) for communication, where not only the identity of the server (webhook consumer) is verified using a certificate, but also that of the client (webhook provider); however, it is again difficult to set up, especially compared to signing with HMAC, and due to this, it is not considered worthwhile for most use cases. [8]

2.4.2 Webhook Consumer Authenticity and Encryption

So that the webhook provider can ensure that event notifications with potentially sensitive data are sent to their intended recipients, HTTPS should be used. This requires the consumer to properly configure a TLS certificate. The provider then needs to properly verify the certificates, e.g. treat self-signed certificates as not trustworthy.

However, it is not always feasible for consumers to enable HTTPS for their webhook endpoints, leaving the delivery process susceptible to man-in-the-middle attacks. If encryption cannot be used, the events should not contain sensitive data. Instead, event payloads should be designed to be thin, with sensitive data accessible to the webhook consumer via an API on the side of the webhook provider. [2]

2.4.3 Webhook Endpoint Idempotency

The same event may be delivered more than once. This can happen, for example, when the webhook consumer does not respond to the first delivery in

time, and so the provider attempts another delivery. Therefore, it is important that webhook endpoints are *idempotent* [2]. If the provider includes a unique event identifier in the notification that is consistent between delivery attempts, consumers can store the identifier and use it to ensure that the same event is not processed twice. [8]

2.4.4 Replay Prevention

In a replay attack, a malicious party records the HTTP request of the event and sends it again to the webhook consumer.

If the strategy from the previous section is used and the event identifier is included in the signature (Section 2.4.1), it can serve as replay prevention; however, this would require the consumer to store the event identifiers indefinitely.

A better strategy to help mitigate this is to include a timestamp with each attempt to deliver the event. Again, this timestamp must be included in the signature described in Section 2.4.1 to prevent tampering. When the webhook consumer receives a notification from the provider, they verify that the timestamp is within some predefined tolerance from the current time. It is therefore needed that the timestamp of the current delivery attempt is used, and not the timestamp of the original event. [2]

2.4.5 Server-Side Request Forgery (SSRF)

Extra care must be taken when sending requests to the user-defined webhook endpoint URLs. Since event notification HTTP requests are sent from inside the network of the webhook provider, a malicious actor could use a webhook subscription to send HTTP requests to the internal infrastructure of the provider. This can be especially dangerous in combination with tools to inspect events and their responses that webhook providers often include in their webhook dashboards to aid development.

One way of mitigating this issue is to maintain a blacklist of sensitive IP addresses, perform domain name resolution on webhook endpoint URLs using domain name system (DNS), and filter out endpoints that correspond to blacklisted IPs. Since DNS records change, it is important to filter blacklisted IP addresses before each event notification HTTP request, and not only once at the time of subscription. Implementors should also be careful not to perform domain name resolution again after the address was checked, as this could be abused in a *DNS re-binding attack*. Any HTTP redirect could also point to a blacklisted IP address and should be filtered out.[2, 11]

Another approach is to use a proxy for outgoing traffic that is isolated from sensitive internal systems. There are existing solutions created specifically for use with webhooks, such as Smokescreen [12] and Webhook-sentry [13].

2.4.6 Event Sender IP Whitelisting

Webhook providers often publish a list of static IP addresses that will be used to send the notification HTTP requests from. As an additional security measure, webhook consumers should configure their event receiver to block communications coming from non-whitelisted IP addresses. [2, 4]

2.5 Standard Webhooks

At the time of writing this thesis, there is an ongoing effort to standardize webhooks[4]. It is an attempt to codify industry best practices by some of the major players in the industry. The proposal specifies requirements and recommendations, with the signature scheme (recall Section 2.4.1), together with the headers needed to verify it, being the only requirement. This signature scheme allows both symmetric and asymmetric signatures, and also multiple signatures for a single event to support *zero downtime secret rotation*. The Standard Webhooks repository also includes reference implementations of signature generation and verification for a number of programming languages.

The majority of the recommendations in the proposal have already been discussed in this chapter.

Chapter 3

Existing Frameworks and Libraries

There are many tools for implementing webhook consumers, i.e. the receivers of events. Often webhook providers themselves maintain libraries to correctly receive their own flavor of webhooks; however, due to the differences between these webhook flavors, a universal tool is not possible. Ngrok[14] is a good illustration of this problem. It is a reverse proxy software that can be configured to receive webhooks, verify them, and then send them to the actual consumer application through a tunnel, one advantage of which is that the actual consumer application does not need to be exposed to the public Internet. However, ngrok needs to implement special handling for each webhook provider, precisely because there are differences between each.[3]

The options for tools for implementing a webhook provider are limited. There exist frameworks such as Thorn [**?**] and Django REST Hooks [15], which integrate with Python and the Django web framework; however, they are not actively maintained. There is the ActionHook [16] gem for Ruby, which does not build on any framework, also without any recent commits to the GitHub repository. A post on the Microsoft .NET blog [17] describes a relevant ASP.NET related project; however, it is now obsolete and abandoned, according to the Deveel Webhooks documentation [18], which is another .NET framework providing such capabilities.

There is another category of tools that webhook providers can use: Webhook-as-a-Service (WaaS) solutions. Here, the examples are Svix [3] and Hook0 [19]. They have the advantage of being programming language agnostic, although perhaps not as flexible.

I decided to review one of the solutions based on an existing web framework and one of the WaaS solutions. For the solution based on an existing web framework, I chose Thorn. While it is not actively maintained, its documentation is the most extensive and complete, it is one of the more popular solutions (judging by GitHub stars), it integrates with a web framework, which offers interesting possibilities that are worth showcasing, and Django and Python are the technologies that I am more familiar with compared to .NET or Ruby. For a WaaS solution, I chose Svix using similar criteria.

[3]This is one of the problems motivating the Standard Webhooks [4] initiative.

3.1 Thorn

Thorn is a webhook framework for Python, perhaps most similar in concept to the subject of this thesis out of the existing solutions reviewed in this chapter. Although Thorn states that the framework can be extended to integrate with any Python web framework, out of the box, only Django [20] is supported. A convenient interface is provided to define events, send events, and manage subscriptions.

This section is based on the information found in the documentation of Thorn [21] and its source code [22].

3.1.1 Events

Events may be sent manually from anywhere; however, more interestingly, Thorn allows users to define events on top of Django models through *model events*.

A model in Django corresponds to the model in the Model-View-Controller (MVC)[4] architecture; it is responsible for managing the data in the application, and each model usually maps to a single table in a relational database via object-relational mapping (ORM). Changes to these models, and consequently the database, can be easily set up to trigger events.

Users can also declaratively specify *filters* that narrow down the conditions when certain model events are sent. Listing 3 illustrates usage of model events.

3.1.2 Subscriptions

Subscriptions are implemented as yet another Django model that is stored in the relational database alongside other models of the Django application. Users can choose to include a default REST API implementation to manage subscribers in their Django application. By default, a subscription will be owned by a user managed by the Django user authentication system.

3.1.3 Dispatchers

Thorn offers multiple pluggable *dispatchers*. The job of a dispatcher is to perform the notification HTTP requests to registered webhook endpoints.

The "default" dispatcher sends requests directly from the current process. This has the disadvantage that the request that originally triggered some event in the Django application cannot be finished until all notification requests have been sent out to subscribed webhook endpoints and their responses processed. For this reason, the default dispatcher should only be used in development scenarios.

The dispatcher intended for production is the "celery" dispatcher. Celery is a distributed task queue software [23], and Thorn uses it to distribute the work of a dispatcher over a pool of workers.

[4]Technically, Django uses Model-Template-View, however, these can be viewed as different terminology for Model-View-Controller respectively.

```python
from django.db import models
from thorn import ModelEvent, webhook_model

@webhook_model
class Article(models.Model):
    # Django fields to define database columns for this model.
    uuid = models.UUIDField()
    title = models.CharField(max_length=128)
    state = models.CharField(max_length=128, default='PENDING')
    body = models.TextField()

    # This class is used to set up Thorn events.
    class webhooks:
        # Declares an event that will be sent every time
        # an Article is changed and the value of the state
        # field is equal to 'PUBLISHED'.
        on_publish = ModelEvent(
            'article.published',    # Event type/topic for the event.
            state__eq='PUBLISHED', # Filter over the state field.
        ).dispatches_on_change()   # Trigger on every change.

        # Defines the payload of the events.
        # Data from the model object may be used.
        def payload(self, article):
            return {
                'title': article.title,
            }
```

Listing 3: A Django model extended with Thorn model events. (Based on [21].)

3.1.4 Reliability

Celery can be configured to automatically retry a failed task, which Thorn takes advantage of to enable a configurable retry mechanism for delivering events. If event delivery fails, despite possible retries, Thorn will not react in any way (e. g. the subscription associated with the webhook endpoint is not canceled). More broadly, there does not appear to be any communication of the results of the deliveries from the workers back to the main application[5], and consequently no concept of event history is provided by Thorn.

3.1.5 Security

Thorn signs notification requests using an HMAC as described in Section 2.4.1 and includes the signature in a custom HTTP header `Hook-HMAC`. The specific hashing algorithm used is customizable. However, the request and signature do not include a timestamp of the delivery attempt or a unique identifier for the event, leaving recipients potentially vulnerable to replay attacks. Of course,

[5] Celery itself can facilitate this using "backends", e. g. a key-value store like Redis that the task producer polls for task results.

users can include an event ID in the payload of each event themselves. However, including delivery attempt timestamps would require a modification of the source code of the framework.

Another security feature are ”validators”, which validate the recipient webhook endpoint URL before making the notification request. There are predefined validators that check that the protocol is either HTTP or HTTPS, or that the domain name does not resolve to an IP address in a reserved private block. Custom validators may be provided.

3.2 Svix

Svix is a WaaS product. It is written in the Rust programming language and a mostly compatible modified version of the software is open source, and thus Svix can also be self-hosted [24].

Similarly to the previous section, this section is based on the Svix documentation [3] and the source code available on GitHub [24].

Svix differs from a framework like Thorn in that webhook-related functionality is separated from the main application into a standalone service. To send an event[6], webhook providers call the Svix REST API, and the Svix server does the actual work of filtering and sending the event to registered webhook endpoints. To manage subscriptions, inspect results of event deliveries, etc., webhook providers also use the Svix REST API – the provider's webhook consumers cannot communicate with Svix directly, and so such requests must be sent to the provider first and then delegated. [3]

To inform the webhook provider about events, such as failure to deliver an event, Svix provides *operational webhooks*.

Figure 3.1 illustrates the HTTP communication involved in sending a single event through Svix.

Svix also maintains libraries to aid development for both the webhook provider (e. g. wrappers around calls to the Svix REST API) and the webhook consumer (e. g. functions to verify signatures) for many popular programming languages.

3.2.1 Reliability

Unlike Thorn, Svix stores all messages sent, including information about each delivery attempt for each endpoint. Webhook providers can access all this information using the Svix REST API and can also instruct Svix to make another attempt at delivery. To allow webhook consumers to access this functionality, additional work is required on the provider's side, as they cannot call the Svix REST API directly – the provider must do it on their behalf. Svix alleviates this problem to an extent by providing a customizable autogenerated webhook dashboard.

[6] Svix documentation uses different terminology, e. g. ”message” vs. ”event”. For the sake of simplicity and consistency with the rest of this text, I map the concepts used in Svix onto the concepts described here so far. For example, I ignore the Svix concept of ”applications”. Instead of ”message”, ”create message”, etc., I use ”event”, ”send event”. Svix's concept of an ”endpoint” also corresponds well to what I described as subscriptions.

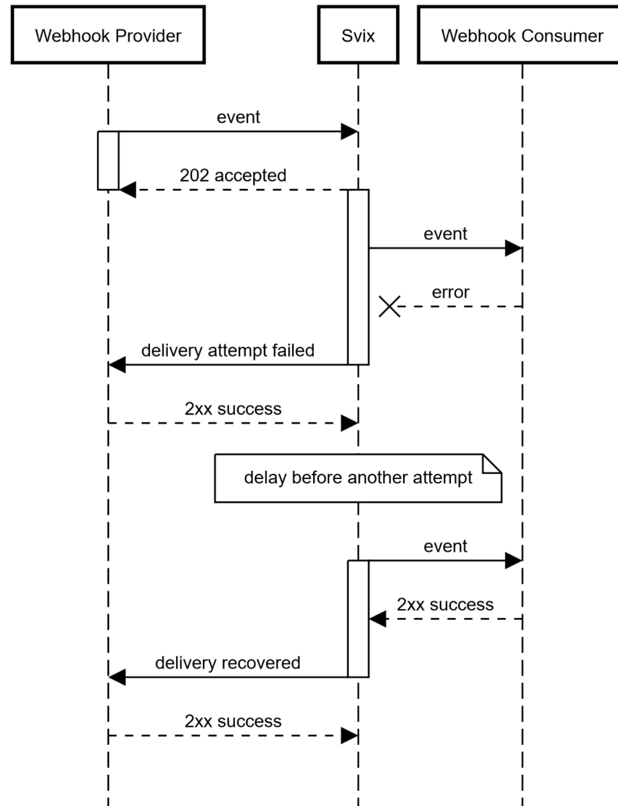

Figure 3.1: Sequence diagram illustrating single event send through Svix. (Based on [3].)

3.2.2 Security

Svix recommends and uses HMAC verification by default; however, signatures using asymmetric keys are also possible. As many providers do, Svix also provides a list of IP addresses from which notification requests will be sent. In general, Svix adheres to the Standard Webhooks (draft) specification [7] and therefore implements current best security practices.

3.2.3 Architecture

Svix stores all information about events alongside the subscription data in a single PostgreSQL database. This database is separate from the database the webhook provider will certainly have, and so to enable relating entities between these databases, Svix allows the provider to set a custom identifier for some of its entities, which can then be used in communications with the Svix REST API.

Unlike Thorn, with its Celery-based dispatcher, workers in the Svix server run in the same process as asynchronous tasks and access the database directly

[7]The CEO of Svix is a member of the technical steering committee for Standard Webhooks.

to store the results of sending notifications. Similarly to Thorn, work (i. e. an instruction to send an event to some endpoints) is distributed to workers using a message queue (in-memory, Redis, or RabbitMQ) [24].

Chapter 4

Requirements

4.1 Functional requirements

- **Must Have** Ability to send events to subscribers accordingly.
- **Must Have** Subscriptions management.
- **Should Have** Subscriptions management REST API.
- **Should Have** Event history and accompanying REST API.
- **Could Have** Webhook dashboard (browser user interface to manage webhook subscriptions and inspect event history).
- **Won't Have** Means to declare possible event types.

Declaring possible event types is mainly useful for documentation. It is only a necessity if the solution is to include features such as automatic documentation generation.

4.2 Non-functional requirements

4.2.1 Ease of use

The interface of the solution should fit well with the approaches commonly used in Haskell web programming and in popular web programming libraries and frameworks in the ecosystem. It should also be possible to fit the subscription and the event history API with users' APIs without much hassle.

4.2.2 Flexibility

As shown in Chapter 2, there is no single way to implement webhooks. Thin vs. thick events, reconciliation strategies, different security requirements, etc. – the framework should be customizable enough to support such different needs.

4.2.3 Reliability

As described in Chapter 2, retry strategies and keeping a record of the dispatched events and the results of their corresponding requests are essential to implement many of the reliability strategies. Using thin events is also mentioned; however, the content of actual payloads of events is up to the user of the framework.

4.2.4 Security

As discussed in Section 2.4, some security features, such as server-side request forgery (SSRF) protection, can be handled by a third-party software. On the other hand, signing event payloads is essential and should preferrably follow the scheme prescribed by Standard Webhooks [4].

4.2.5 Usability

There are two perspectives:

- the developers using the framework to create a webhook enabled web application, and
- the consumers of that web application.

In the first case, usability means e. g. well-commented code and documented functionality with examples. There should be a simple interface to trigger events.

In the second case, this can mean providing features like the ping event so that consumers can test their webhook endpoints, or providing an OpenAPI specification for the subscription and event history APIs.

Chapter 5

Implementation

5.1 Whoopr Module

This section describes the top-level module of Whoopr, which implements the core of the framework.

5.1.1 Subscription Management

An important aspect of any webhook provider is subscription management. Whoopr is unique in that it should be able to integrate with any database management system (DBMS) and it should support use cases such as Svix, where the subscription data is kept in a database separate from the main application, and also use cases such as Thorn, where the subscription data is stored as part of a single application. Clearly, an abstraction is needed.

In Haskell, it is very common to structure programs around *monads* [8] and *monad stacks* [9]. Monads are also inherent to Haskell – all IO operations must be carried out through the `IO` monad. The `IO` monad may, for example, be combined with the `Reader` monad to provide the IO computations with some context, such as the information needed to connect to a database and perform a query. This is essentially the `Session` monad in Hasql [25], a PostreSQL library for Haskell.

For the purposes of this section, monads encapsulate a computation and some context. To define a common interface to the subscription data, a type class is used, where operations such as `getAllSubscribers` are defined within the terms of some monad. Users can therefore supplement any monad to Whoopr, and thus arbitrary context, as long as it satisfies the necessary constraints. Listing 4 shows (a slightly simplified version of) the declaration for this type class.

Notice that the type of the subscription `s` is also polymorphic in this declaration. This leads to more flexibility for implementations since a specific representation is not prescribed. Instead, Whoopr defines type classes such as `HasId` or `HasEndpointURI`, which, in addition to flexibility, allow for more granular constraints in different parts of the system.

[8] I will rather not attempt a proper explanation.
[9] A combination of multiple monads, allowing to use their joined functionalities.

```
class (Monad m, HasId s) => MonadSubscriptions s m | m -> s where
    getAllSubscriptions :: m [s]
    getSubscription :: IdType s -> m (Maybe s)
    createSubscription :: s -> m s
    deleteSubscription :: IdType s -> m (Maybe s)
```

Listing 4: `MonadSubscriptions` type class definition.

Whoopr then declares a subclass of the `MonadSubscriptions` type class, as shown in Listing 5, which is crucial when sending events. Its purpose is to find relevant subscriptions for a specific event. Note the type parameter `fd`, standing for *filter descriptor*, which describes the event for the purposes of this filtering process. A naive implementation of this filtering functionality would simply combine the results of `getAllSubscribers` with a common higher-order function like `filter`; however, usually this filtering can be performed in a database query, yielding better performance. This type class makes both approaches possible.

The simplest example for `fd` is a simple string that contains the type of event. How this string is then matched against the subscribers is up to the instances of this type class. This declaration of `MonadSubscriptionsFilter` also allows different instances for the same subscriber type, but with different filtering. For instance, the simple event type string could be compared simply by equality or hierarchically. (This hierarchical matching is used in Thorn [21], where, for example, `"user.*"` matches `"user.created"`.)

```
class (MonadSubscriptions s m)
        => MonadSubscriptionsFilter s m fd where
    filterSubscriptions :: fd -> m [s]
```

Listing 5: `MonadSubscriptionsFilter` type class definition.

5.1.2 Task Queue

Whoopr's architecture is organized around a central task queue. When a user wants to send an event after a change has occurred in their system, a task is created and inserted into the task queue. On the other end, there is a worker running constantly in a separate lightweight thread, which will receive tasks from the queue and process them. Whoopr defines a pair of type classes that are used to work with the task queue in a manner that does not require a concrete implementation. Listing 6 shows the definitions.

5.1.3 Event Sender

The job of the event sender is to take event data, find relevant subscribers, and, for each one, construct the notification HTTP request and make sure it is delivered. Generating notification metadata, such as signatures, and dealing with retries is also in the scope of the event sender.

```
class TaskQueueProducer q a where
    tqSend :: q -> a -> IO ()

class TaskQueueConsumer q a where
    tqRecv ::  q -> IO a
```

Listing 6: Task queue type class definitions.

Whoopr provides the function `consumeTaskQueue`, which is to be used to launch the event sender. The function (Listing 7) takes as a parameter not only the consuming end of the task queue, but also the producing end. The main reason for this is that failed delivery attempts are re-inserted into the task queue after the retry delay. The `Proxy fd` parameter is where Whoopr pays the price for its flexibility; this parameter is only needed to help the compiler. Finally, the `runner :: m () -> IO ()` parameter is needed to run the filtering monad (or, in other words, to provide the necessary context to the filtering computation).

```
consumeTaskQueue :: (
        TaskQueueConsumer cq (Task fd),
        TaskQueueProducer pq (Task fd),
        HmacSigner s, HasEndpointURI s,
        MonadSubscriptionsFilter s m fd,
        MonadIO m
    ) => Proxy fd -> (m () -> IO ()) -> cq -> pq -> IO ()
consumeTaskQueue proxy runner cq pq = do
    task <- taskRecvHelper proxy cq
    liftIO . async $ case task of
        NotificationDeliveryTask nd -> ...
        NotificationInitializationTask ni -> ...
    return ()
```

Listing 7: Definition of the `consumeTaskQueue` function.

To initiate an event delivery, users must insert a notification initialization task into the task queue, which encapsulates the event data and its filter descriptor. Later, Chapter 6 shows how this can be performed and how it can be wrapped into a more convenient interface.

As the type signature of this function suggests, Whoopr's event sender signs event deliveries using a HMAC. This signature is computed according to the Standard Webhooks requirements [4] and will be included in the notification HTTP POST request in the `webhook-signature` header, along with the `webhook-id` and `webhook-timestamp` headers.

5.1.3.1 Caveats

Whoopr's event sender does not implement security measures such as checking that endpoint URLs do not resolve to private IP addresses. However, this is not a fatal flaw – as mentioned in Section 2.4.5 users can use a proxy such as

webhook-sentry [13] or smokescreen [12], which are made specifically for use with webhooks, to protect their webhooks.

These checks would most conveniently be performed at the time of binding a network socket (an IP address and port pair); however, this is very low-level functionality, and not all HTTP client libraries expose it. Healthchecks [26] solves this by using libcURL.

5.2 Whoopr.Basic Module

The definitions from the core module described in the previous section are not very useful on their own. For that, instances for the abstract typeclasses need to be provided. The definitions from this module can then, together with the previous module, be put together to form a working webhoook provider.

5.2.1 Concretizing Subscription Management

Whoopr.Basic assumes that subscriptions are stored in-memory in a list and that this list is accessible using the `MonadReader` interface. With these assumptions in place, `MonadSubscriptions` can be instantiated (Listing 8).

Next, a concrete subscription type is provided and with the assumption that strings containing the types of events will be used as a filter descriptor, an instance of `MonadSubscriptionsFilter` is provided.

```haskell
class HasSubscriptionsList env s | env -> s where
    hslGetAllSubscriptions :: env -> MVar [s]

instance (HasSubscriptionsList env s,
          MonadReader env m, MonadIO m,
          HasId s, Integral (IdType s))
    => MonadSubscriptions s m where
    ...
```

Listing 8: `MonadSubscriber` instance from the Whoopr.Basic module

5.2.2 Subscription API

Using these concrete definitions and Servant, a Haskell library for writing web applications [27], the Whoopr.Basic module provides a Web Application Interface (WAI) [28] `Application`, which can be combined with other such applications, possibly created using other web frameworks built on top of WAI.

Chapter 6

Evaluation

6.1 Example Application

In this section, I describe an example web application, Whoopr-example, using Whoopr to demonstrate its use. The application exposes a simple web API built using Servant [27] to manage users. Any change to a user will trigger an event and subscribers will be notified using the functionality Whoopr and the Whoopr.Basic module provide.

6.1.1 Defining Application Environment

Listing 9 contains definition of the application environment and instantiates `HasSubscriptionsList` for it using the subscription type from the Whoopr.Basic module. This is enough to make a monad such as `ReaderT AppEnv IO` an instance of the `MonadSubscriber` class, thanks to the instance definition also from the module shown in Section 5.2.1.

The definition of `AppEnv` also shows that the application will use the standard `Chan` as the task queue. (The Whoopr module provides instantiations of the related queue classes for `Chan`.) `ByteString` in the `eventQueue` type is the type of the filter descriptor. The Whoopr.Basic module provides an instance of `MonadSubscriptionFilter` for `ByteString` and `BasicSubscription` and the compiler will automatically use it here.

```
data AppEnv = AppEnv {
    subscriptions :: MVar [BasicSubscription],
    userDb :: MVar [User],
    eventQueue :: Chan (Task ByteString)
}

instance HasSubscriptionsList AppEnv BasicSubscription where
    hslGetAllSubscriptions = subscriptions

runSubscriptionMonad :: AppEnv -> ReaderT AppEnv m a -> m a
runSubscriptionMonad env m = runReaderT m env
```

Listing 9: Application environment for Whoopr-example.

Listing 9 also defines an important function `runSubscriptionMonad`, which converts from the subscription-enabled monad to a different one. This is needed to pass the `AppEnv` context to the event sender (recall the `runner` parameter mentioned in Section 5.1.3) and also to the subscription API, both of which work in the context of a different monad `m`.

6.1.2 Launching the Event Sender

To run the event sender, the example application in its main function launches a new worker thread, which performs the `consumeTaskQueue` function in an infinite loop. Listing 10 show the important parts of the code.

```
main = do
    ...
    let env = AppEnv { ... }
    forkFinally (forever $ consumeTaskQueue' env) (\case
        Right _ -> print "sender finished OK"
        Left e -> print $ "sender finished with error: " ++ show e)
    ...

consumeTaskQueue' env@AppEnv{..} = consumeTaskQueue
    (Proxy :: Proxy ByteString)
    (runSubscriptionMonad env)
    eventQueue
    eventQueue
```

Listing 10: Launching event sender worker.

6.1.3 Sending Events

As described in Section 5.1.3, to initiate event delivery, users must insert a `NotificationDeliveryTask` into the task queue. This is a tedious detail that can be encapsulated in a monadic action using the same pattern as with the `MonadSubscriptions` instance.

Listing 11 first defines constraints to describe a monad, in the context of which events can be sent. Then, using these constraints, the `notify` function is defined. This function takes the filter descriptor, i.e. the event type in the case of this application, and the event data, which can be anything that can be converted to JSON. Before creating the task, it modifies the event data so that it also includes the event type, demonstrating that users are in full control of the contents of their event payloads.

6.2 Assessment

The previous section demonstrates that Whoopr, despite its abstract core, can be used to implement webhook support for a web application effectively, with a small amount of boilerplate code, provided that there are intermediate building blocks that can be used, such as the Whoopr.Basic module. Thanks

```
class HasWebhooks env where
    getTaskQueueProducer :: env -> TaskQueueType

instance HasWebhooks AppEnv where
    getTaskQueueProducer = eventQueue

type (MonadWebhooks env m) = (MonadIO m, MonadReader env m, HasWebhooks env)

notify :: (MonadWebhooks env m, ToJSON ed) => ByteString -> ed -> m ()
notify fd ed = do
    q <- getTaskQueueProducer <$> ask
    let edWithEventType = object [
                "eventType" .= decodeUtf8 fd,
                "eventData" .= ed
            ]
    let task = NotificationInitialization {
        eventData = DynEventData edWithEventType,
        filterDescriptor = fd
    }
    liftIO $ tqSend q $ NotificationInitializationTask task
```

Listing 11: Defining convenience functions to send events.

to the strong static type system of Haskell, it is difficult to make a mistake in the boilerplate code if the types fit together.

Compared to the solutions reviewed in Chapter 3 Whoopr does not provide as comprehensive and out-of-the-box experience; however, it is more flexible.

6.3 Further Work

There are many possible avenues in which to improve Whoopr.

- Whoopr does not implement any concept of event history described in Section 2.3.3.2, and so, as with Thorn, if the maximum number of retries is exceeded, the event is lost.
- The intermediate layer, now represented by the Whoopr.Basic module, is rather limited. Whoopr would greatly benefit from having such a module that builds on PostgreSQL, for example.
- A more complete example application demonstrating usage of Whoopr using popular technologies would not only serve as an instructive resource for users but also help validate the design of the framework.
- Generating a template for a webhook dashboard, binding automatically to the subscriptions and event history APIs is also within the realm of possibility.
- Implement protections against SSRF.

- Implement an asymmetric signature scheme according to Standard Webhooks, allowing users to choose.
- Tests for the framework and a CI/CD pipeline for the GitHub repository.
- And more…

Chapter 7

Distribution

The source code for the Whoopr package is available as a GitHub repository under the MIT license [10]. It has not yet been uploaded to a package repository such as Hackage. To use it, users can instead download the repository and using a `cabal.project` file make it visible to other local packages, which is what was used to develop the Whoopr-example in Chapter 6.

Assuming the following directory structure:

```
cabal.project
whoopr
  src
  whoopr.cabal
  ...
whoopr-example
  app
  whoopr-example.cabal
  ...
```

If `cabal.project` includes the following line:

```
packages: */*.cabal
```

Then the Whoopr package becomes visible in `whoopr-example.cabal` as any other regular package, e.g.:

```
executable whoopr-example
    ...

    build-depends: whoopr, ...

    ...
```

Although this is perhaps not as convenient, it is not unexpected for users to find themselves in a situation where they need to tweak Whoopr to fit their

[10] https://github.com/vojtechbalik/whoopr

needs, as the package is in the more experimental stages of development at this moment.

The Whoopr-example package is also available as a GitHub repository [11]

[11] https://github.com/vojtechbalik/whoopr-example/

Conclusion

This work investigates many different existing webhook providers, looking closely at their similarities and differences. It also researches previous investigations and analyses on the matter. The considerations that webhook providers implementors need to make and the problems they face, particularly when it comes to the security and reliability of webhook deliveries, are discussed together with the related best practices.

An overview of existing frameworks and libraries designed to aid in webhook provider development is provided, and they are divided into two main categories. Out of each category, one solution is reviewed in detail. The attributes of these solutions are also compared to the findings from the preceding research chapter.

The specification of requirements builds on the researched best practices and features that existing webhook providers offer, to which I assign priorities. The subsequent design and implementation, albeit satisfying mostly only the most prioritized of the outlined requirements, is a fully functional solution, and also a solution that is extensible, on top of which missing less prioritized features can be built. It is also likely the only solution in the Haskell space.

The solution, which I called "Whoopr", is then evaluated by building an example webhook-enabled web application. A brief comparison with the existing solutions reviewed in detail is provided as well. Finally, the possible ways in which Whoopr can be developed and improved are also discussed.

The repository containing the final version of the source code for the solution is freely available under an MIT license.

Bibliography

[1] GitHub, Inc. GitHub Docs. https://docs.github.com/en, 2024, [Online; accessed 28-January-2024].

[2] Biehl, M. *Webhooks – Events for RESTful APIs*. API-University Series, CreateSpace Independent Publishing Platform, 2017, ISBN 9781979717069.

[3] Svix. Svix Documentation. https://docs.svix.com/, 2024, [Online; accessed 7-February-2024].

[4] Standard Webhooks. Standard Webhooks specification. https://github.com/standard-webhooks/standard-webhooks/tree/5569aeae6fa0035bb01eb0167ab74bcaaeabedc1, 2023, [Online; accessed 18-January-2024].

[5] Lindsay, J. Web hooks to revolutionize the web. https://progrium.github.io/blog/2007/05/03/web-hooks-to-revolutionize-the-web/, 2008, [Online; accessed 15-December-2023].

[6] Red Hat. What is a webhook? https://www.redhat.com/en/topics/automation/what-is-a-webhook, 2024, [Online; accessed 15-January-2024].

[7] PayPal. PayPal API reference. https://developer.paypal.com/api/rest, 2024, [Online; accessed 24-January-2024].

[8] ngrok. Webhooks.fyi. https://webhooks.fyi, 2023, [Online; accessed 20-January-2024].

[9] Adobe. Webhooks Overview. https://helpx.adobe.com/sign/developer/webhook/overview.html, 2023, [Online; accessed 13-February-2024].

[10] Stripe. Stripe API Reference. https://stripe.com/docs/api, 2024, [Online; accessed 6-February-2024].

[11] Lokare, A. Sending webhooks securely. https://www.ameyalokare.com/technology/webhooks/2021/05/03/sending-webhooks-securely.html, May 3 2021, [Online; accessed 21-January-2024].

[12] Stripe. Smokescreen. https://github.com/stripe/smokescreen, 2023, [Online; accessed 21-January-2024].

[13] Lokare, A.; Lissner, M.; et al. Webhook Sentry. https://github.com/juggernaut/webhook-sentry, 2023, [Online; accessed 31-January-2024].

[14] ngrok. ngrok Docs. https://ngrok.com/docs, 2024, [Online; accessed 9-February-2024].

[15] Zapier Inc. Django REST Hooks. https://github.com/zapier/django-rest-hooks, 2016, [Online; accessed 9-February-2024].

[16] smsohan. ActionHook. https://github.com/smsohan/actionhook, 2020, [Online; accessed 9-February-2024].

[17] Nielsen, H. F. Sending WebHooks with ASP.NET WebHooks Preview. https://devblogs.microsoft.com/dotnet/sending-webhooks-with-asp-net-webhooks-preview/, 2015, [Online; accessed 9-February-2024].

[18] Provenzano, A. Deveel Webhooks. https://github.com/deveel/deveel.webhooks, 2024, [Online; accessed 9-February-2024].

[19] Hook0. The Hook0 Developer Hub. https://documentation.hook0.com/, 2024, [Online; accessed 13-January-2024].

[20] Django Software Foundation. Django documentation. https://docs.djangoproject.com, 2024, [Online; accessed 16-January-2024].

[21] Robinhood Markets, I. Thorn documentation. https://thorn.readthedocs.io, 2016, [Online; accessed 16-January-2024].

[22] Robinhood Markets, Inc and individual contributors. Thorn repository. https://github.com/robinhood/thorn, 2016, [Online; accessed 14-February-2024].

[23] Solem, A.; contributors. Celery User Manual. https://docs.celeryq.dev/, 2023, [Online; accessed 5-February-2024].

[24] Svix. Svix GitHub Repository. https://github.com/svix/svix-webhooks, 2024, [Online; accessed 7-February-2024].

[25] Volkov, N. Hasql. https://github.com/nikita-volkov/hasql, 2014, [Online; accessed 15-February-2024].

[26] Caune, P. Healthchecks. https://github.com/healthchecks/healthchecks/, 2015, [Online; accessed 15-February-2024].

[27] Servant Contributors. Servant. https://docs.servant.dev/, 2022, [Online; accessed 15-February-2024].

[28] Snoyman, M. WAI. https://github.com/yesodweb/wai, 2017, [Online; accessed 15-February-2024].

Acronyms

DBMS database management system. 23

DDoS distributed denial of service. 7

DNS domain name system. 13

HATEOAS hypermedia as the engine of application state. 12

HMAC hash-based message authentication code. 7, 17, 19, 25

MVC Model-View-Controller. 16

ORM object-relational mapping. 16

SSRF server-side request forgery. 22, 29

TLS transport layer security. 12

WaaS Webhook-as-a-Service. 15, 18

WAI Web Application Interface. 26

Appendix A

Contents of attachments

```
├── src.............................................directory of source codes
│   ├── thesis...............................LaTeX source codes of the thesis
│   ├── whoopr.............................. whoopr package implementation
│   └── whoopr-example............whoopr-example package implementation
└── text .................................. directory with text of the thesis
    └── thesis.pdf....................................thesis in PDF format
```