DP-3T: Decentralized Privacy-Preserving Proximity Tracing

Implementation Architecture codename ‘Dappper’

This document describes a possible implementation of the DP-3T design for The Netherlands codenamed “Dappper”.

As a technical implementation description this document gives some insight into what is *technically possible* but **does not answer what *should* be done**. That question also needs answering and should be answered before actual technical implementation since it can heavily influence the design.

The bulk of this architecture description follows the C4 model standard (www.c4model.com). After the system context, we identify further details mostly at level C2 (containers) or level C3 (components). Diving into full detail (C4) is out of scope of this document, which we consider the reference architecture for concrete functional implementation as scope becomes clear.

System Context C1

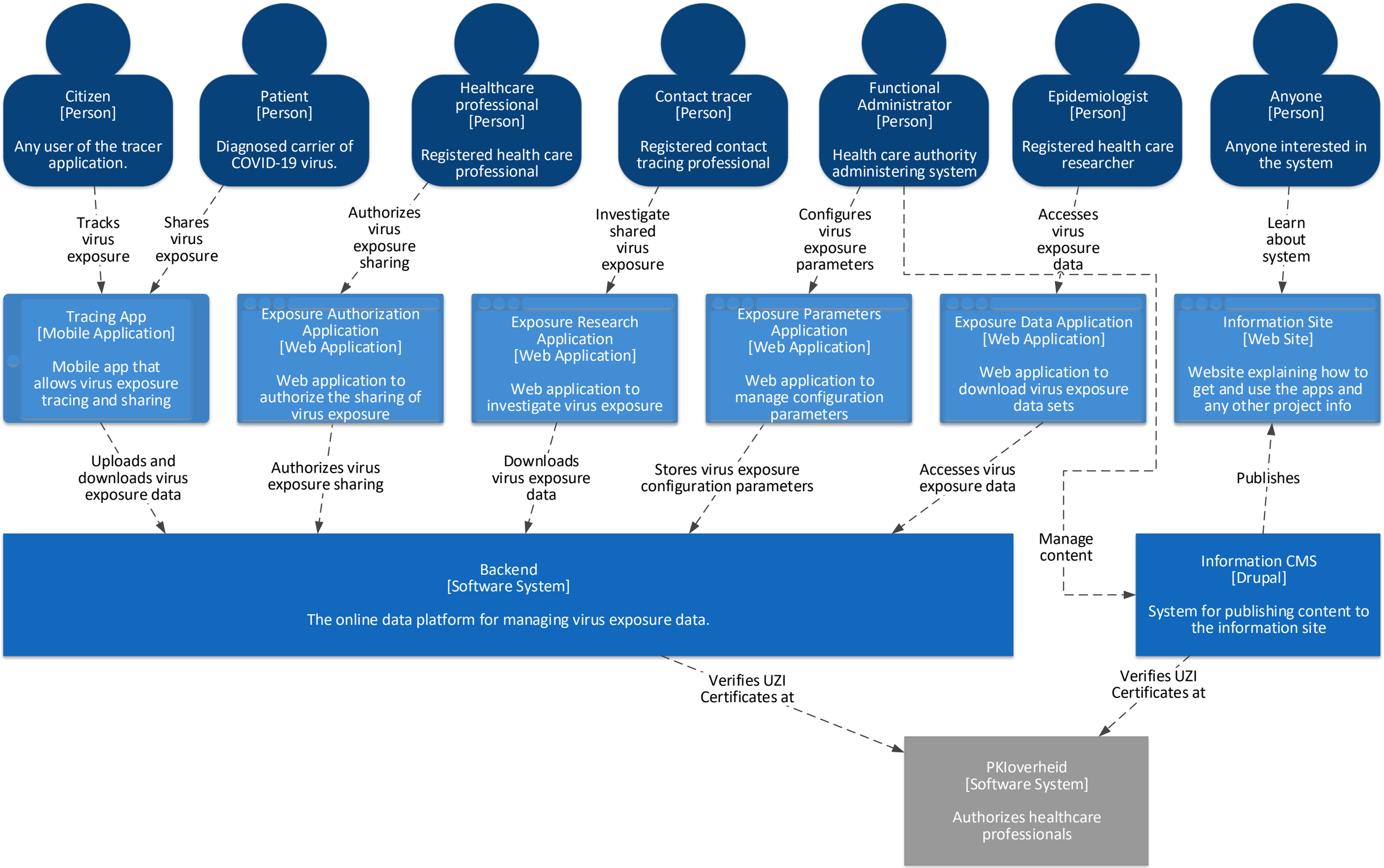
This is a living document that provides a descriptive guideline for implementation and not a set of mandatory constraints. It provides architecture principles and patterns and is designed to evolve with the architecture. You can read more about this at

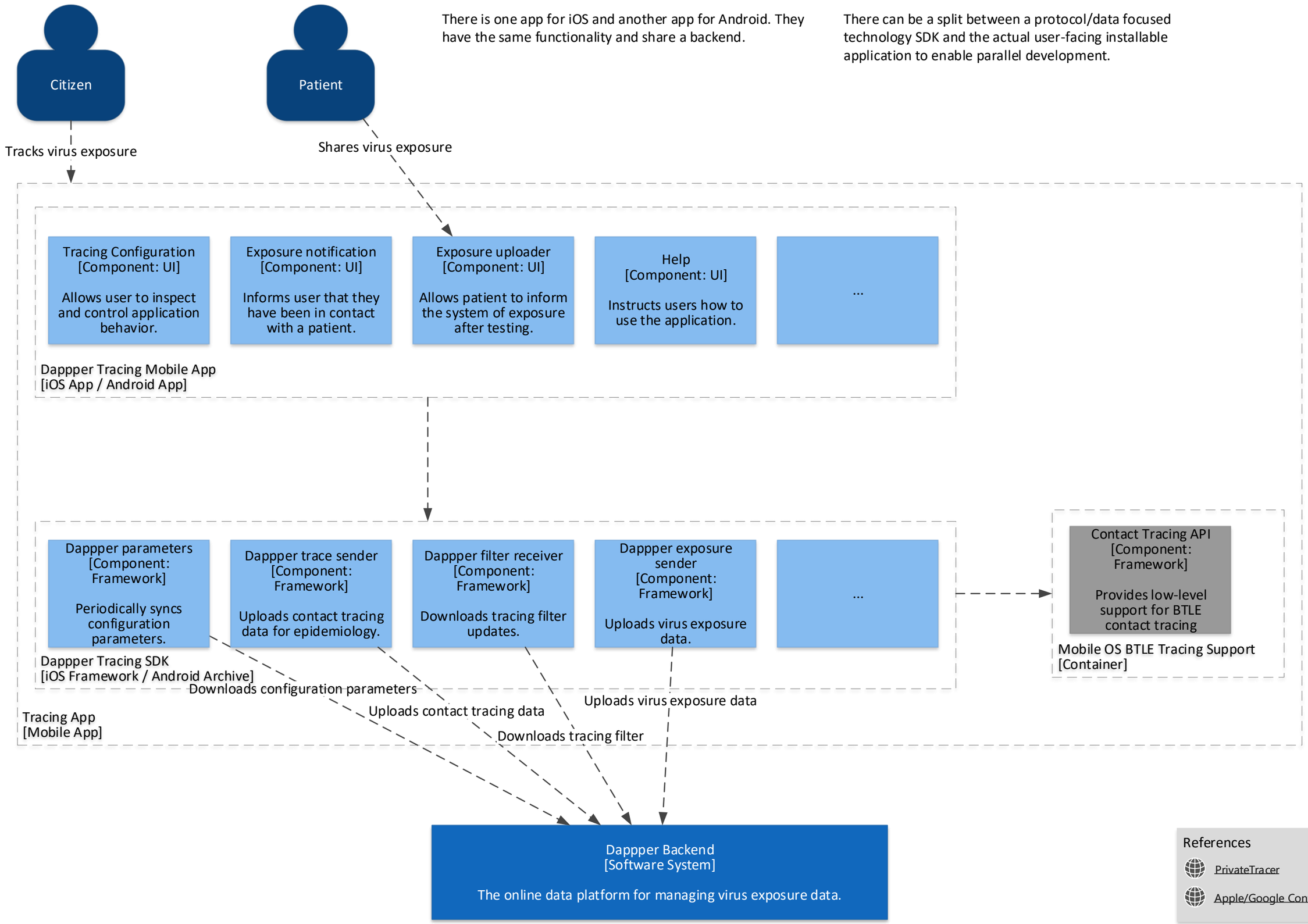
Approach

References

DP-3T

Brief inzake maatschappelijk belang





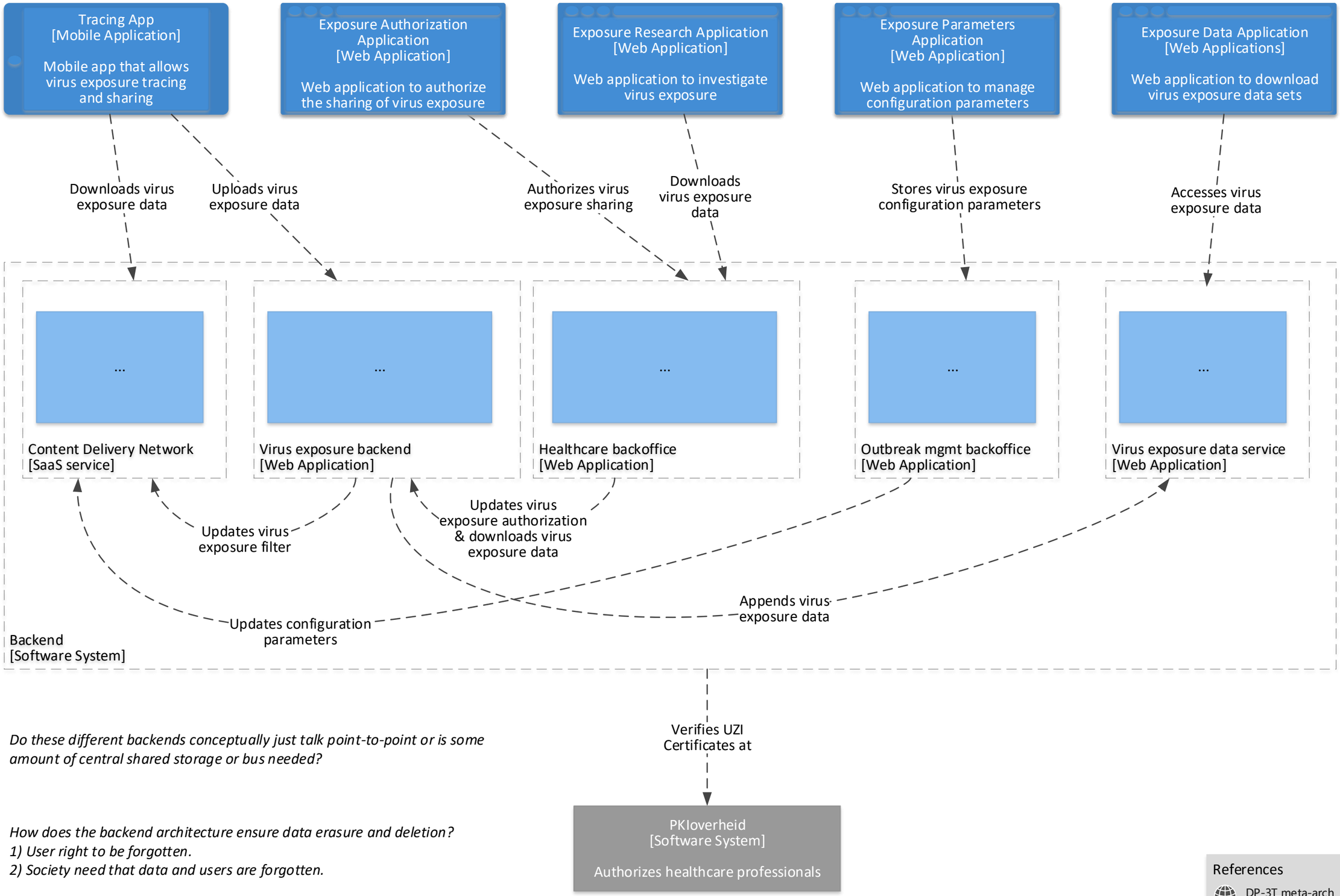
There is one app for iOS and another app for Android. They have the same functionality and share a backend.

There can be a split between a protocol/data focused technology SDK and the actual user-facing installable application to enable parallel development.

References

PrivateTracer

Apple/Google Contact Tracing

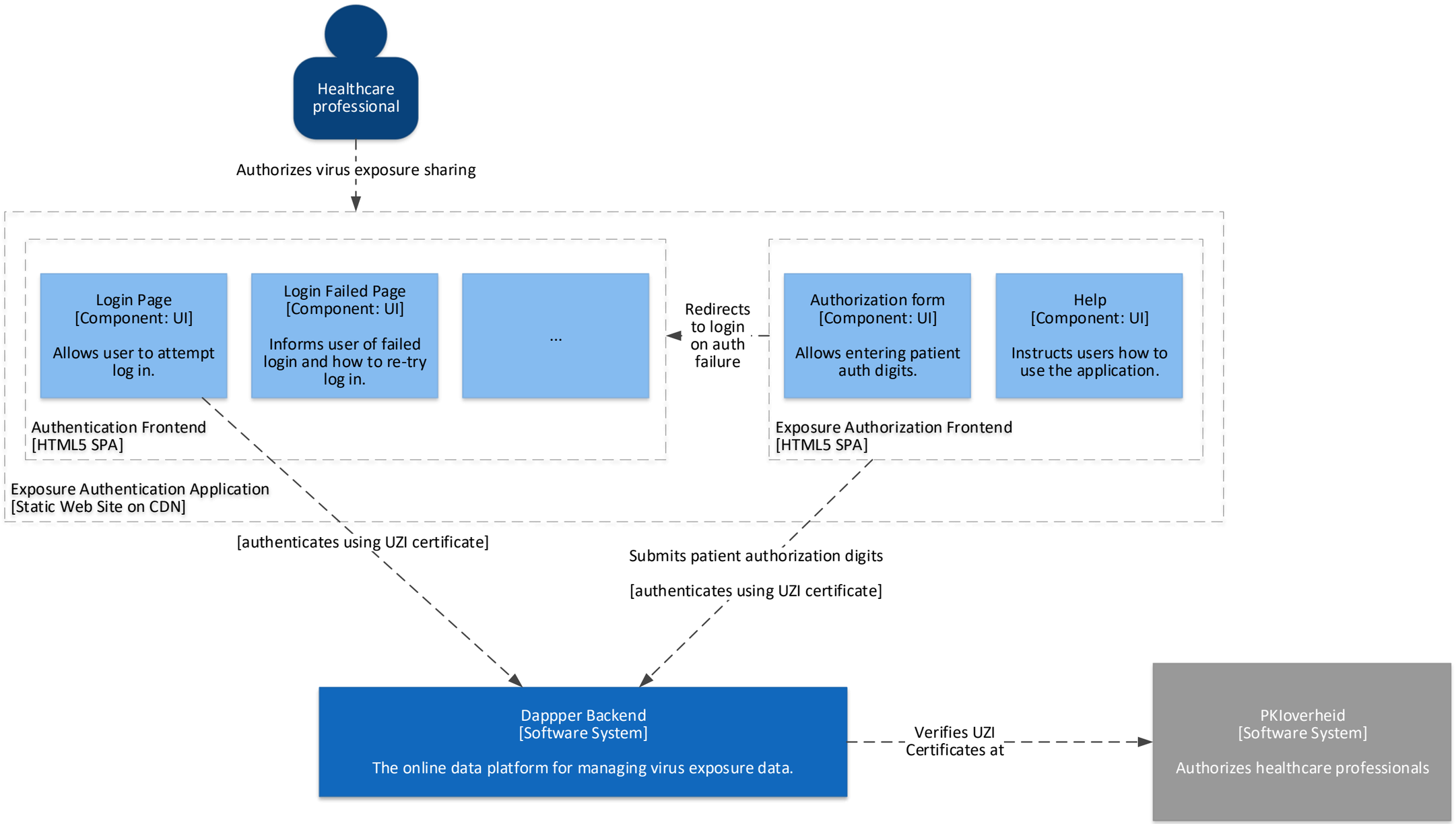


Do these different backends conceptually just talk point-to-point or is some amount of central shared storage or bus needed?

How does the backend architecture ensure data erasure and deletion?
1) User right to be forgotten.
2) Society need that data and users are forgotten.

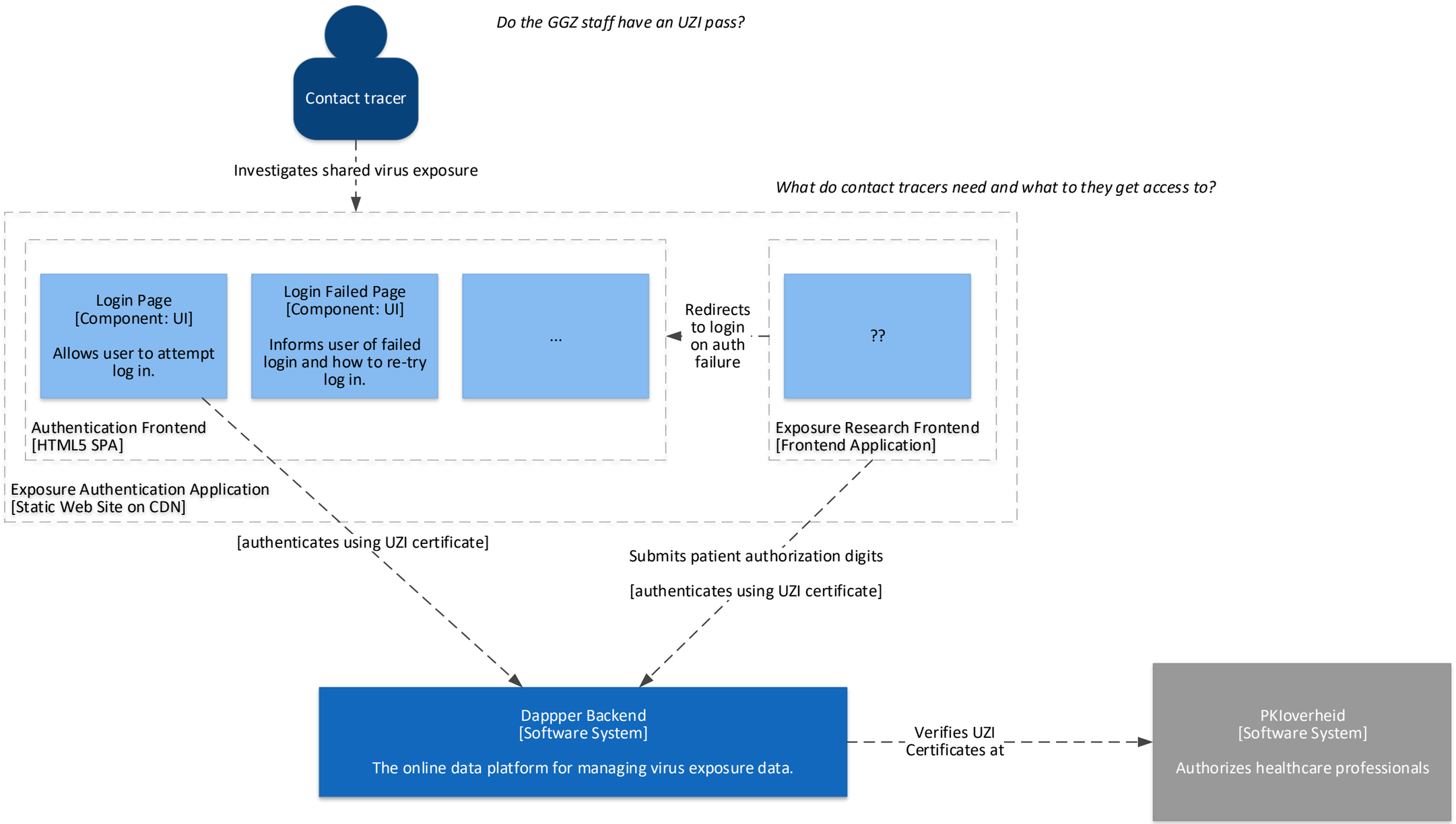
References

- DP-3T meta-arch
- Bert's backend



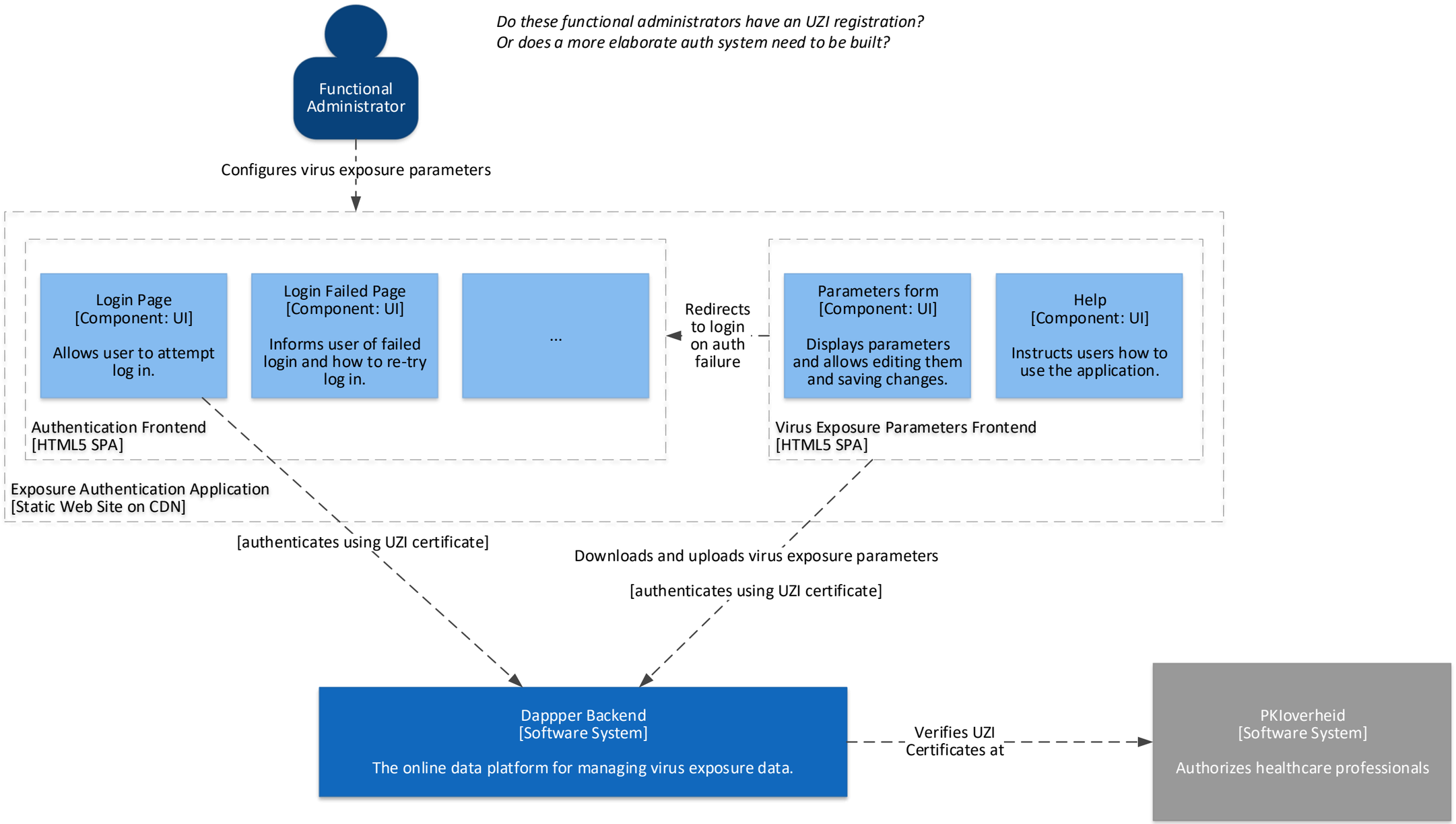
References

- DP-3T backoffice process
- UZI product controleren
- UZI kaartlezer



References

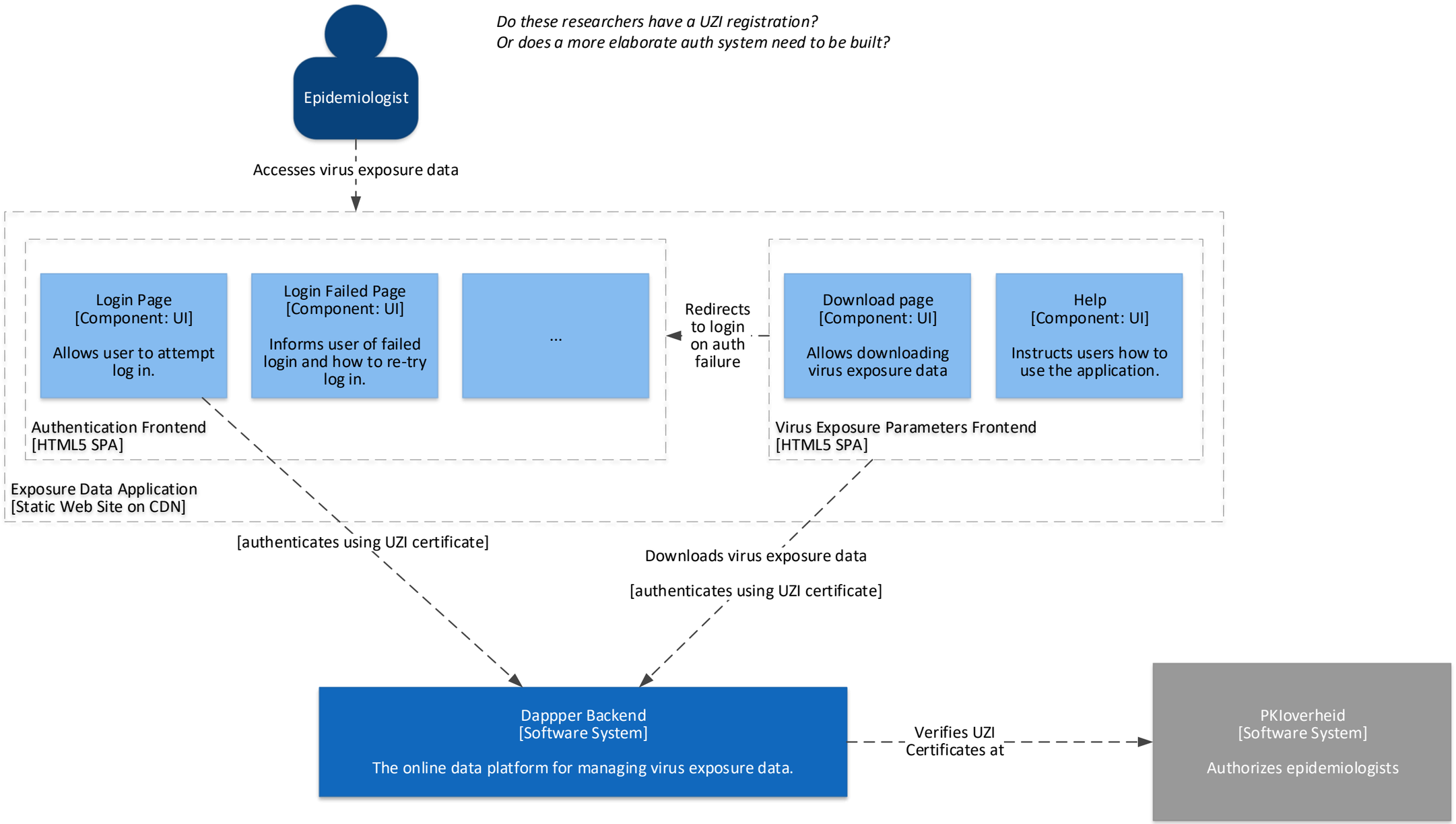
BlueTrace backoffice process



References

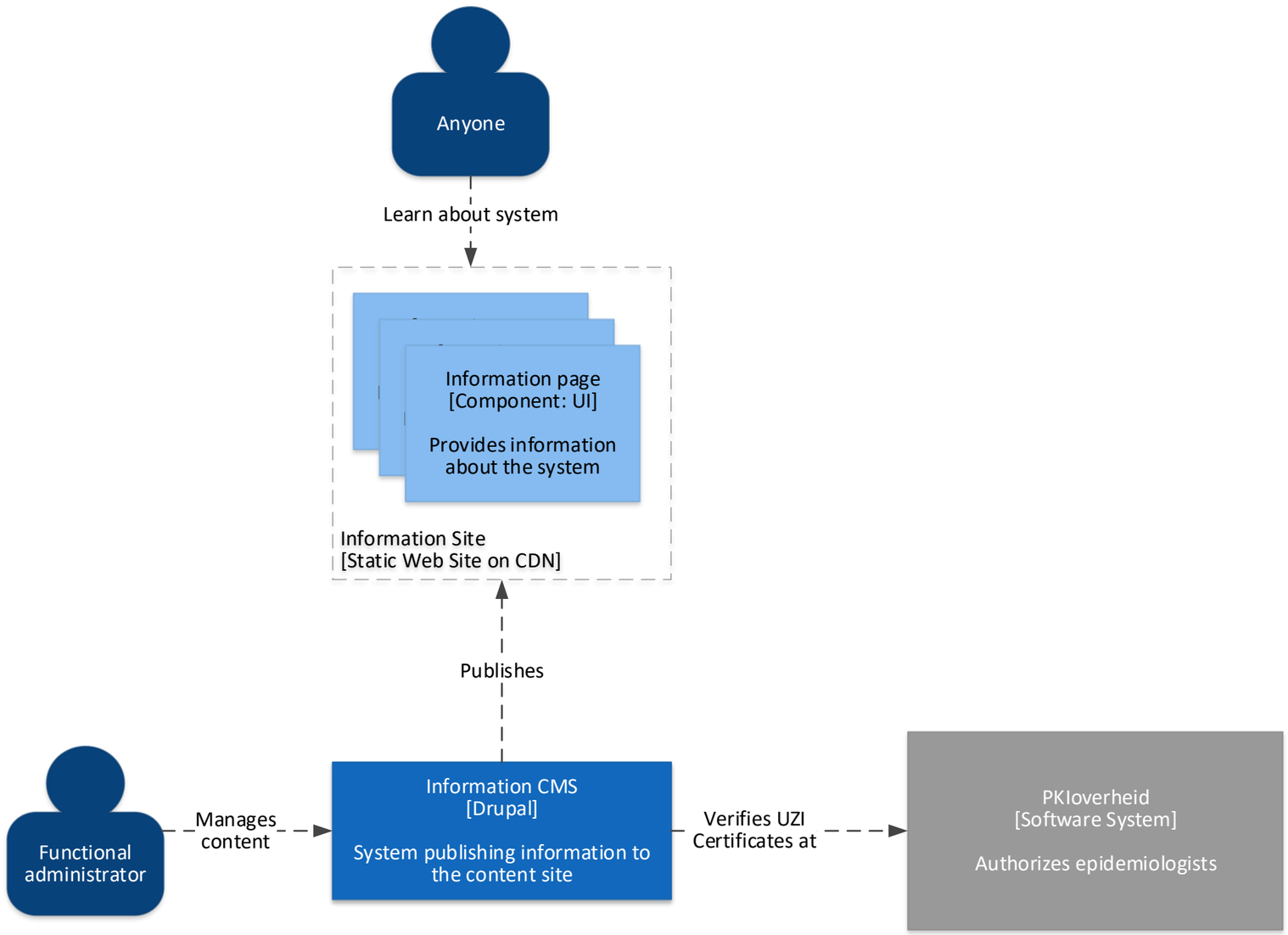
UZI product controleren

UZI kaartlezer



References

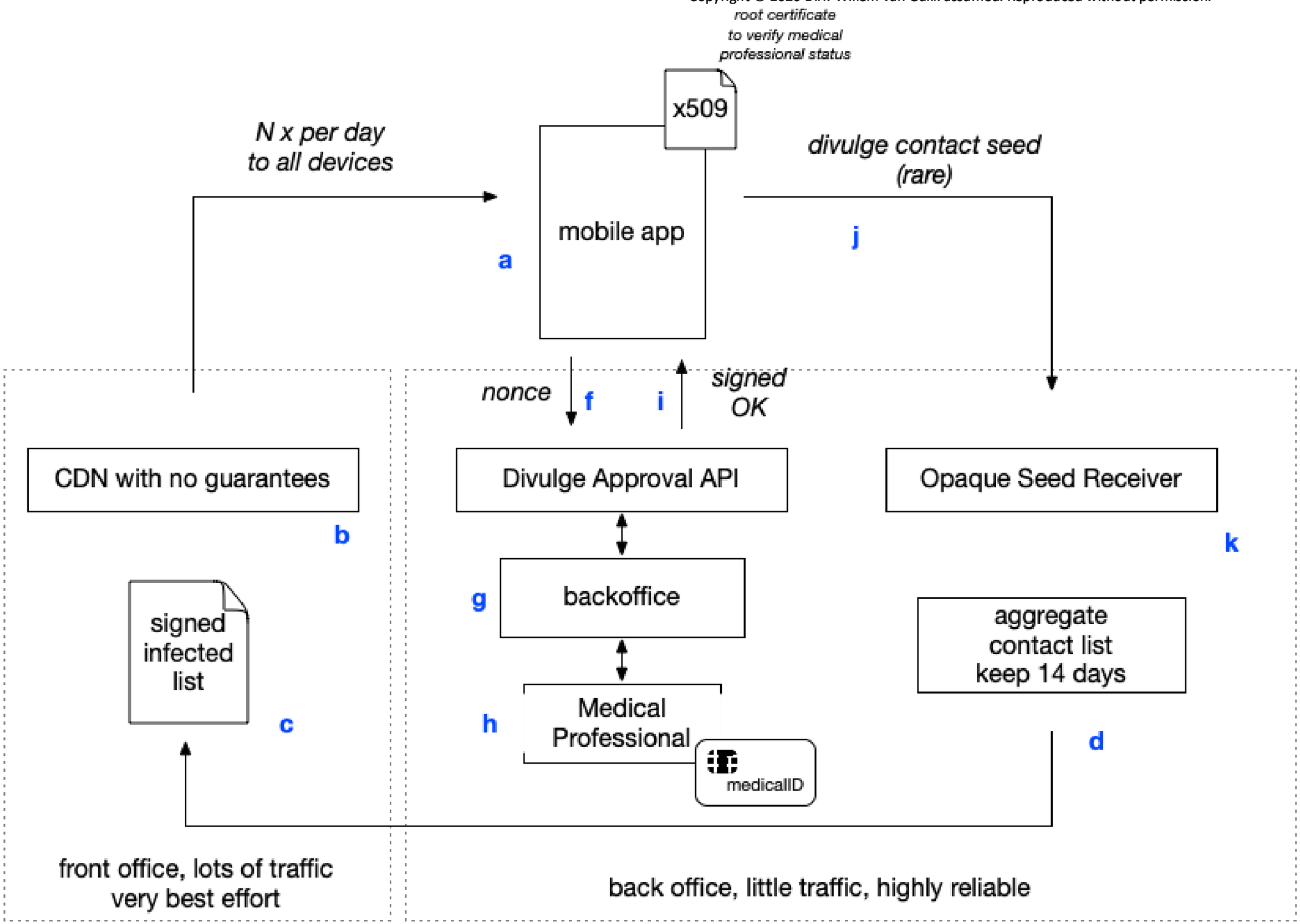
- UZI product controleren
- UZI kaartlezer



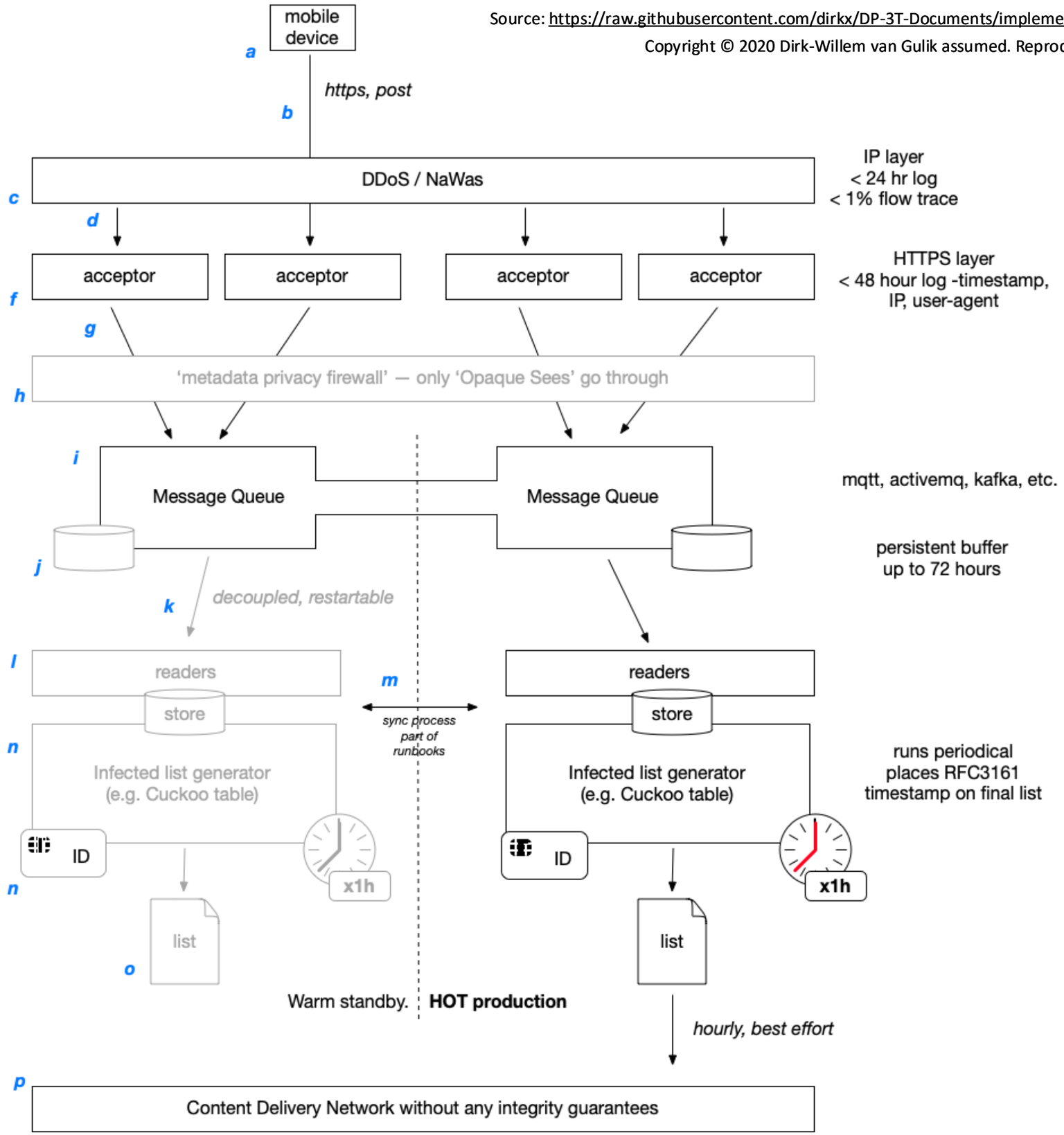
References

DICTU websites

Source: https://raw.githubusercontent.com/dirkx/DP-3T-Documents/implementation-profile-start/meta-arch/overview.png



Source: https://raw.githubusercontent.com/dirkx/DP-3T-Documents/implementation-profile-start/meta-arch/backoffice.png



In modern cloud native development, containers, components, even entire systems can appear and disappear at a rapid pace and it is a challenge to keep architecture documentation such as this up-to-date. That means this architecture documentation starts out *prescriptive* before the system is built, but is intended to become *descriptive* only. It should be a living document that is kept up-to-date, but not ever at the cost of making bad technical decisions or delaying low risk changes to the platform. An agile approach to DevOps is assumed.

This documentation is intended as a shared resource that is updated by everyone that works on the system, and DevOps teams must accept shared responsibility for doing so, ideally adding "update architecture documentation" in their definition of done for stories or at least epics. Alternatively, teams should institute periodic review of their systems and update the documentation.

To help apply a consistent approach, we incorporate several resources by reference:

- **C4 Model**: the diagram style used on many pages (the blue and grey boxes) that breaks any system into Context, Containers, Components and Code. Note Dappper is a "system of systems" and so we have an extra subsystems layers.
- **OSI Model**: a conceptual model for neworking from which the "Layer 3" and "Layer 7" terms derive used in the deployment diagrams.
- **AWS Well-Architected Framework**: even though Dappper is built in the private cloud, the general principles from the AWS well-architected framework can be applied.

  The pillars of the AWS well-architected framework currently are:

  - **Operational Excellence**: running and monitoring systems: managing and automating changes, event response, defining operational standards.
  - **Security**: protecting information & systems: confidentiality and integrity, privilege management, protecting systems, security controls.
  - **Reliability**: prevent and recover quickly from failure: setup, foundational requirements, recovery planning, change management.
  - **Performance Efficiency**: using IT and compute resources effectively: selecting resource types and sizes, monitoring performance, maintaining efficiency.
  - **Cost Optimization**: avoiding un-needed costs: understand and control spend, selecting resource types and amounts, scaling without overspending.
- **Azure Application Architecture Guide**: compared to the AWS well-architected framework, the azure architecture center has more concrete advice.

  Particularly relevant references are:

  - **Multi-region high availability blueprint**: a similar setup is used in Dappper for serving traffic.

The following lists decision that have been made during design process that have impact on the created architecture: