Documenting Architecture

DevOps, Maintenance, and Evolution @ ITU

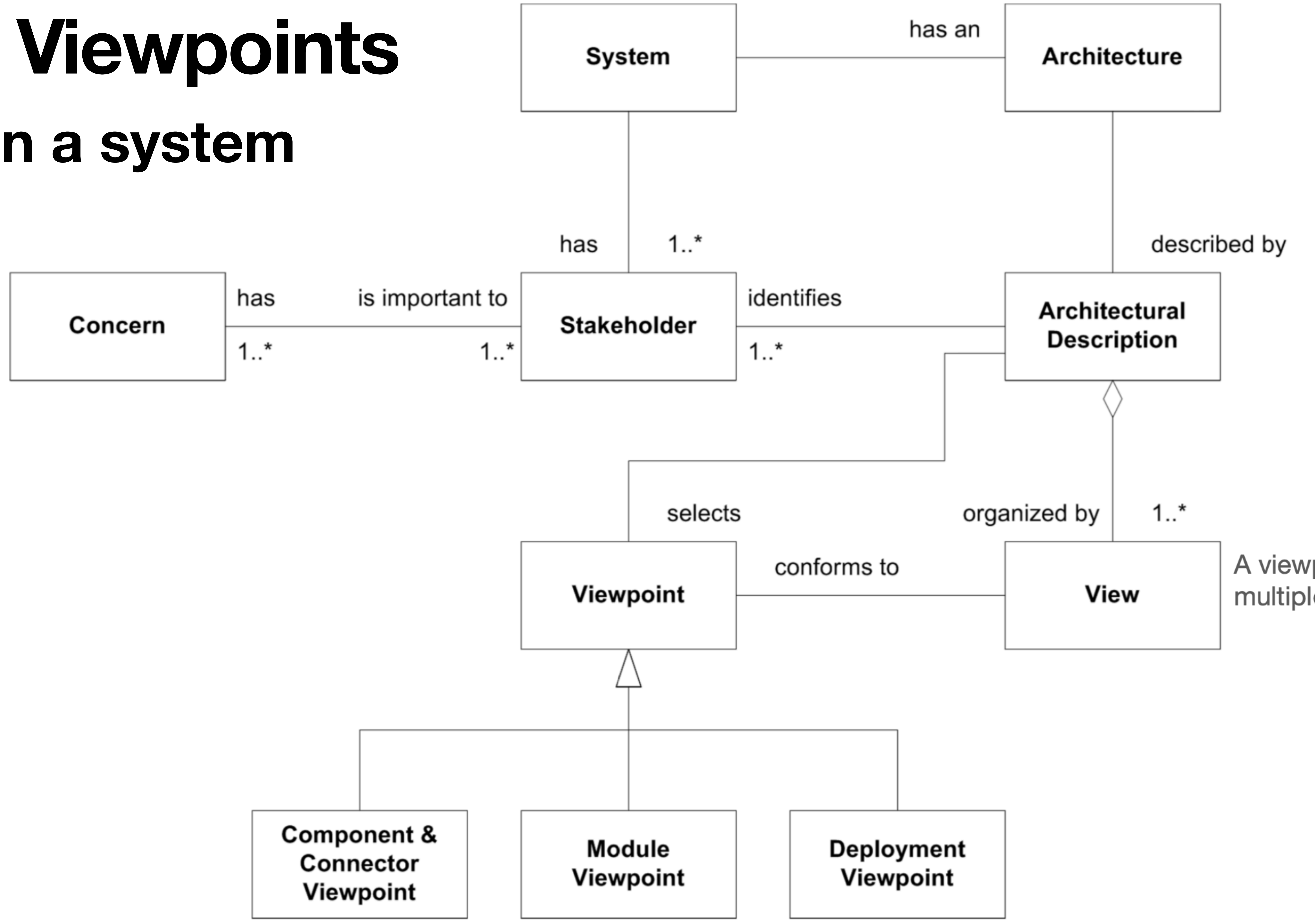
Mircea Lungu

The ideal development environment is one for which the documentation is available for essentially free with the push of a button

Len Bass

Architectural Viewpoints

= *Perspectives* on a system

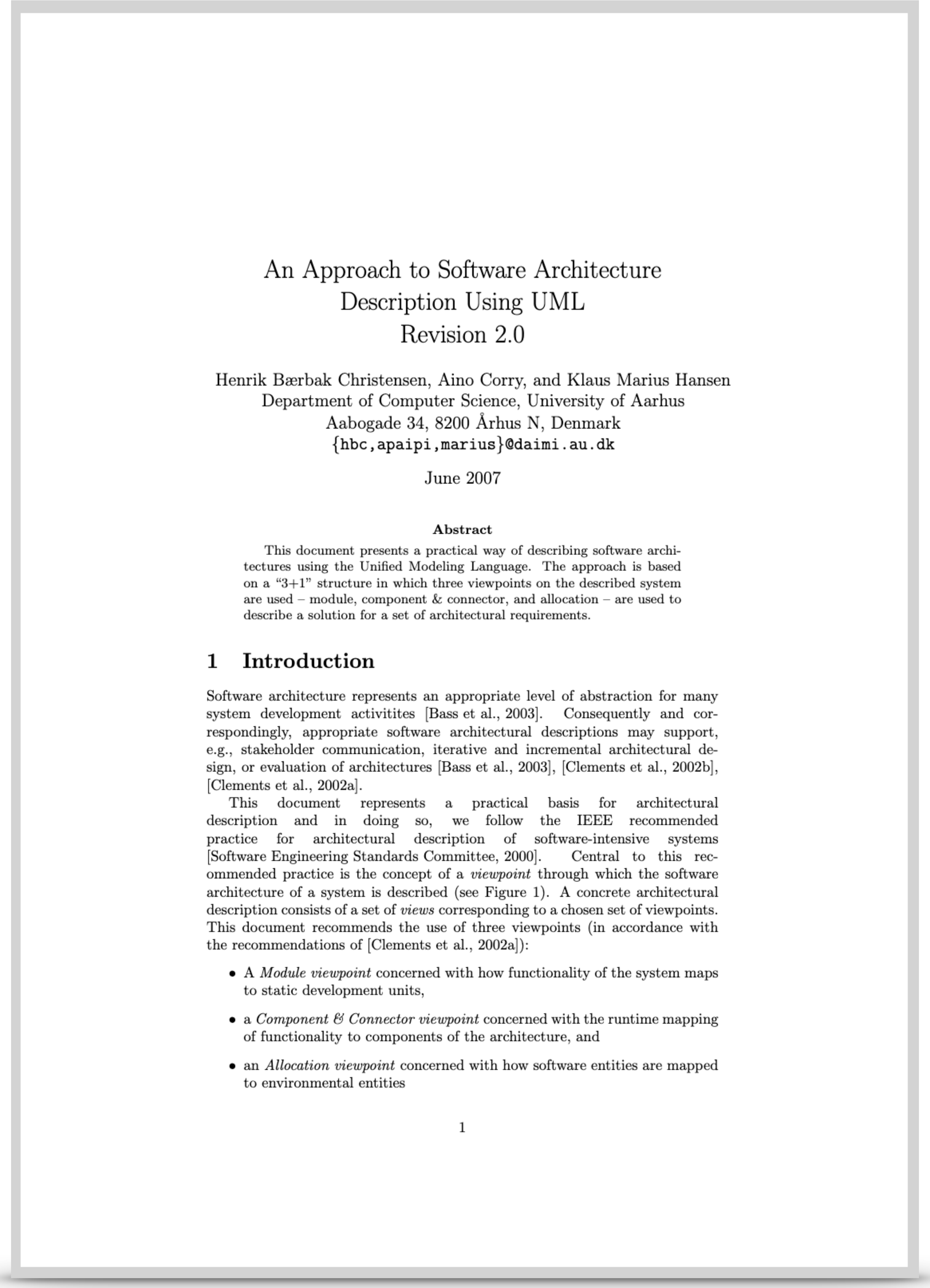


A viewpoint can ha
multiple views

Viewpoints

Popular as “catalogues”

- **4+1 by Kruchten**
- <u>**3+1 by Christensen**</u>
- …*



An Approach to Software Architecture
Description Using UML
Revision 2.0

Henrik Bærbak Christensen, Aino Corry, and Klaus Marius Hansen
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{hbc,apaipi,marius}@daimi.au.dk

June 2007

Abstract

This document presents a practical way of describing software architectures using the Unified Modeling Language. The approach is based on a “3+1” structure in which three viewpoints on the described system are used – module, component & connector, and allocation – are used to describe a solution for a set of architectural requirements.

1 Introduction

Software architecture represents an appropriate level of abstraction for many system development activitites [Bass et al., 2003]. Consequently and correspondingly, appropriate software architectural descriptions may support, e.g., stakeholder communication, iterative and incremental architectural design, or evaluation of architectures [Bass et al., 2003], [Clements et al., 2002b], [Clements et al., 2002a].

This document represents a practical basis for architectural description and in doing so, we follow the IEEE recommended practice for architectural description of software-intensive systems [Software Engineering Standards Committee, 2000]. Central to this recommended practice is the concept of a *viewpoint* through which the software architecture of a system is described (see Figure 1). A concrete architectural description consists of a set of *views* corresponding to a chosen set of viewpoints. This document recommends the use of three viewpoints (in accordance with the recommendations of [Clements et al., 2002a]):

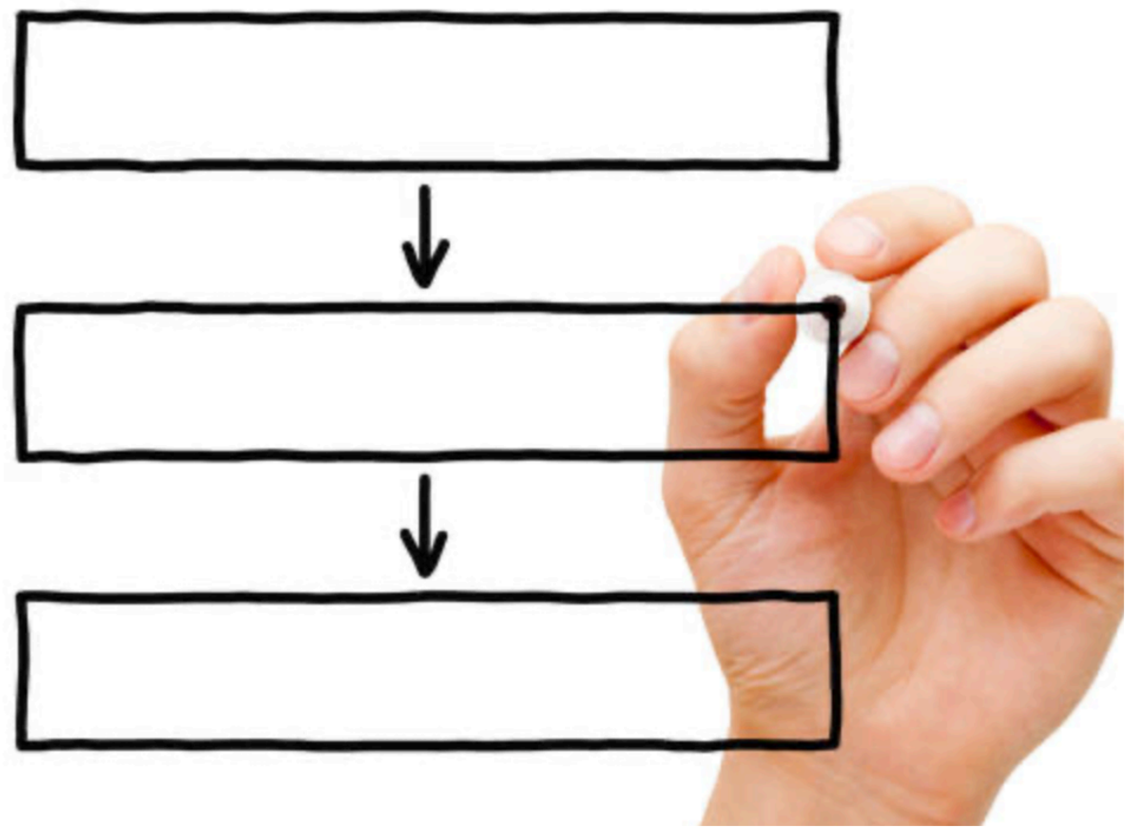
- A *Module viewpoint* concerned with how functionality of the system maps to static development units,
- a *Component & Connector viewpoint* concerned with the runtime mapping of functionality to components of the architecture, and
- an *Allocation viewpoint* concerned with how software entities are mapped to environmental entities

1

* remember the "7 minute abs" scene from There's Something About Mary?

Viewpoints

1. **Concern** - what is it presenting?
2. **Elements** - what does it depict?
3. **Relationships** - relationships between elements?
4. **Representation**



Module Viewpoint

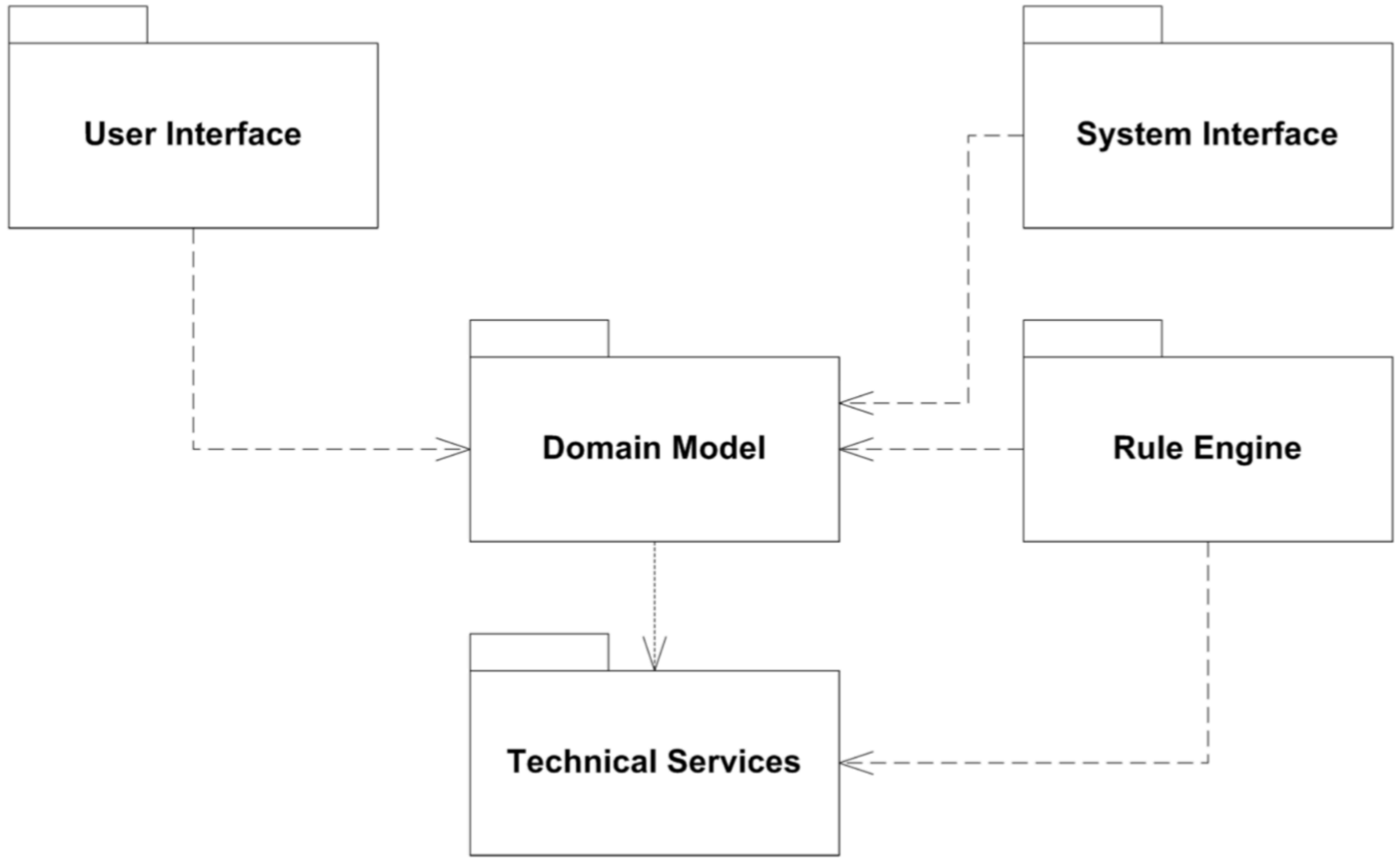
How is the functionality organized in code?

Elements

Packages, Modules

Relationships

Compile-time Dependencies



Components And Connectors

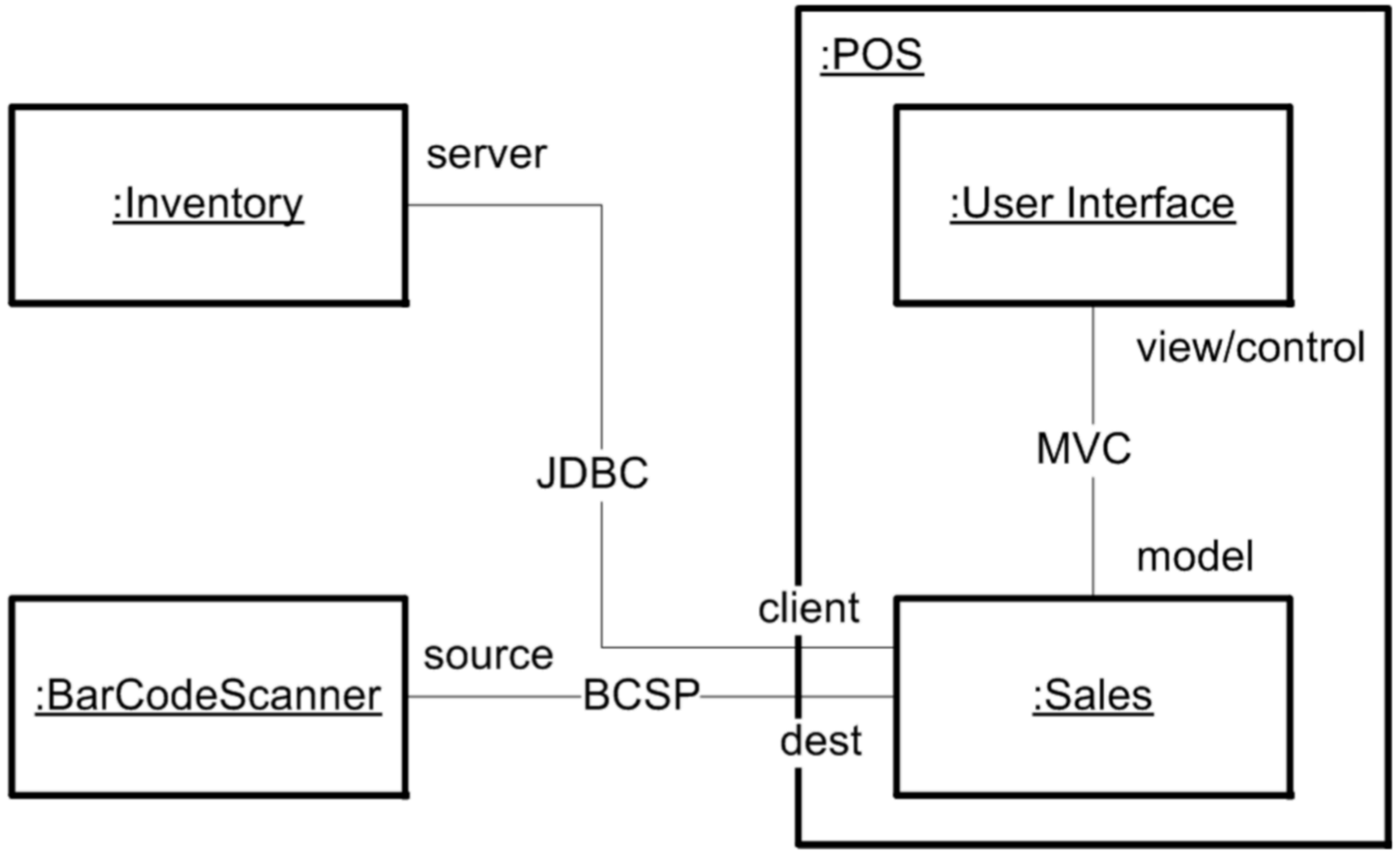
How does the system achieve its functionality at runtime?

Elements

Units of functionality

Relationships

Communication channels



Ensure to annotate with the communication protocol if one is known!

Deployment

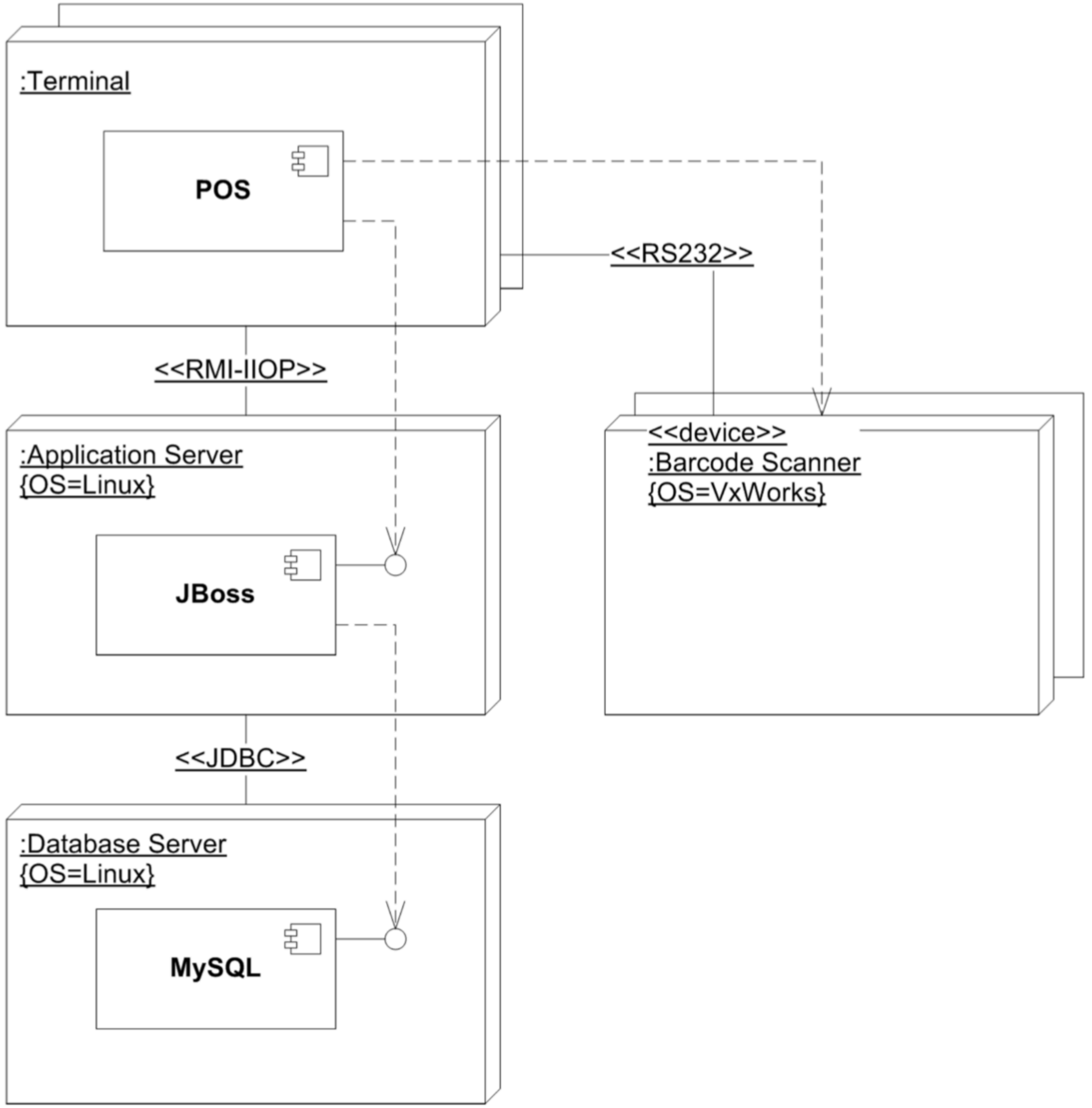
How are elements mapped on the infrastructure?

Elements

- **Processes, Infrastructure**

Relationships

- **Depends-on, Protocol links, Allocated-to**



Visual Representation

Prefer standard notation when available

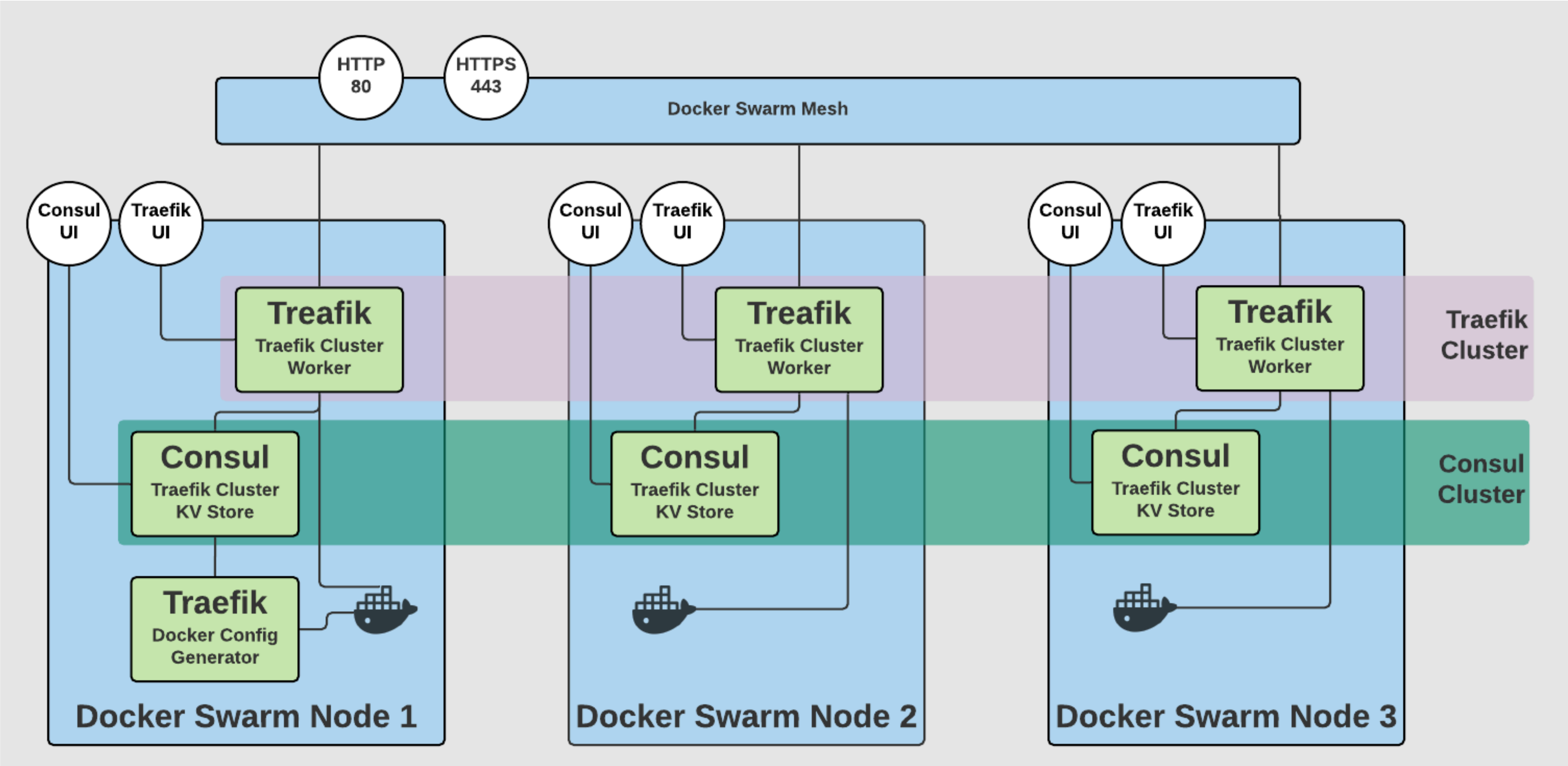
- UML Deployment Diagrams
- UML Component Diagrams

Create your own if you need to, but …

- **Ensure consistency** in visual language
- **Add a legend** for non-standard elements

Custom Visual Notation Example

Highlighting Swarm Nodes and Clusters



You're free to invent your own notation, but then, you should add a legend. And make sure that it makes sense!

Formatting Your Report

Make it as readable as possible

A Report Has a Title and Authors

DevOps, Software Evolution & Software Maintenance

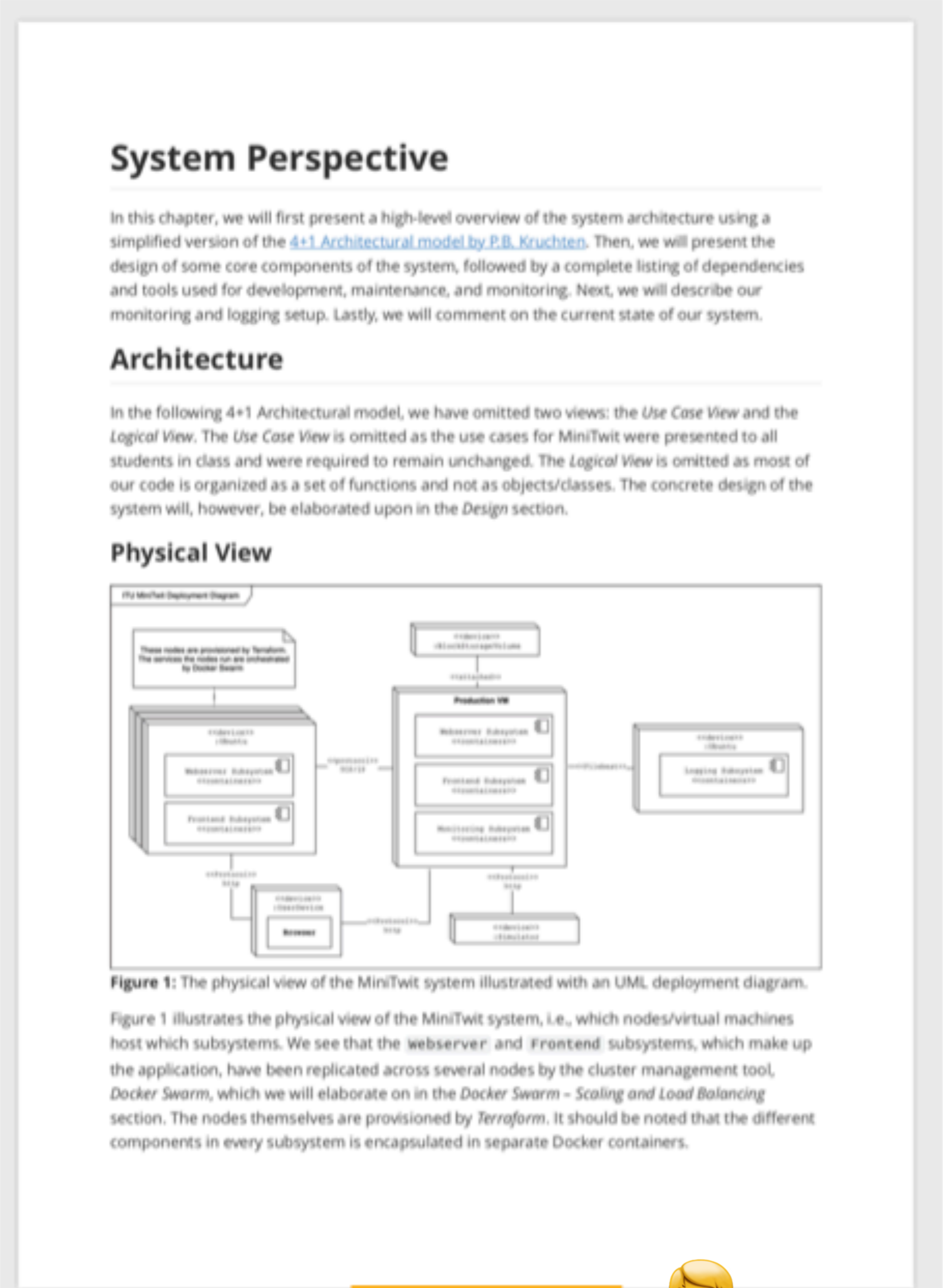
Course code: KSDSESM1KU

May 2020

EXAM ASSIGNMENT BY

Student	Email
Marek Kisiel	maki@itu.dk
Alexander Banks	alsb@itu.dk
Philip Korsholm	phko@itu.dk
Krzysztof Abram	krza@itu.dk
Arian Sajina	arca@itu.dk

Report #1 👍



System Perspective

In this chapter, we will first present a high-level overview of the system architecture using a simplified version of the 4+1 Architectural model by P.B. Kruchten. Then, we will present the design of some core components of the system, followed by a complete listing of dependencies and tools used for development, maintenance, and monitoring. Next, we will describe our monitoring and logging setup. Lastly, we will comment on the current state of our system.

Architecture

In the following 4+1 Architectural model, we have omitted two views: the *Use Case View* and the *Logical View*. The *Use Case View* is omitted as the use cases for MiniTwit were presented to all students in class and were required to remain unchanged. The *Logical View* is omitted as most of our code is organized as a set of functions and not as objects/classes. The concrete design of the system will, however, be elaborated upon in the *Design* section.

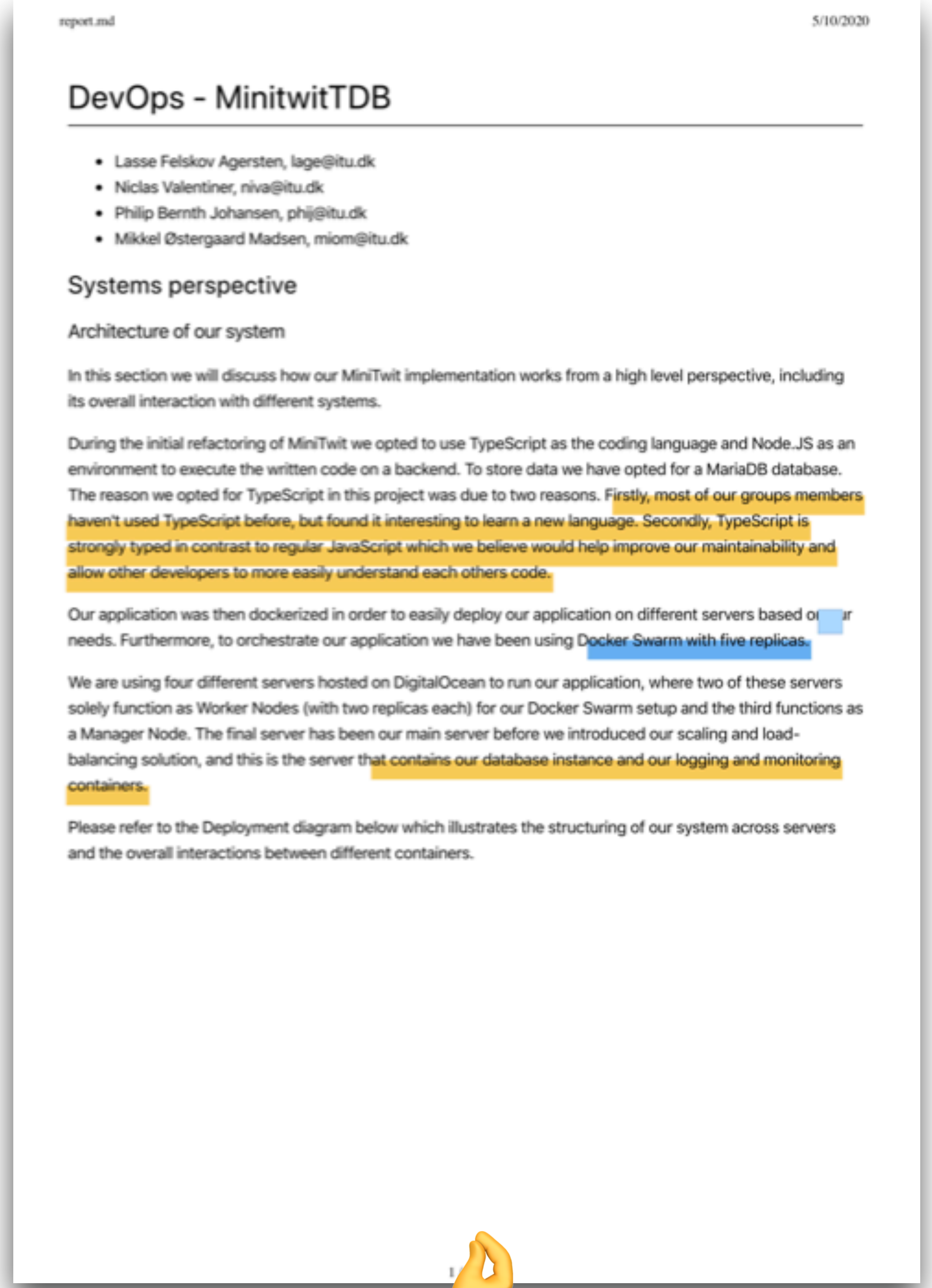
Physical View

Figure 1: The physical view of the MiniTwit system illustrated with an UML deployment diagram.

Figure 1 illustrates the physical view of the MiniTwit system, i.e., which nodes/virtual machines host which subsystems. We see that the `webserver` and `Frontend` subsystems, which make up the application, have been replicated across several nodes by the cluster management tool, *Docker Swarm*, which we will elaborate on in the *Docker Swarm - Scaling and Load Balancing* section. The nodes themselves are provisioned by *Terraform*. It should be noted that the different components in every subsystem is encapsulated in separate Docker containers.

Report #2 🙅

A Report Has a Structure



report.md 5/10/2020

DevOps - MinitwitTDB

- Lasse Felskov Agersten, lage@itu.dk
- Niclas Valentiner, niva@itu.dk
- Philip Bernth Johansen, phij@itu.dk
- Mikkel Østergaard Madsen, miom@itu.dk

Systems perspective

Architecture of our system

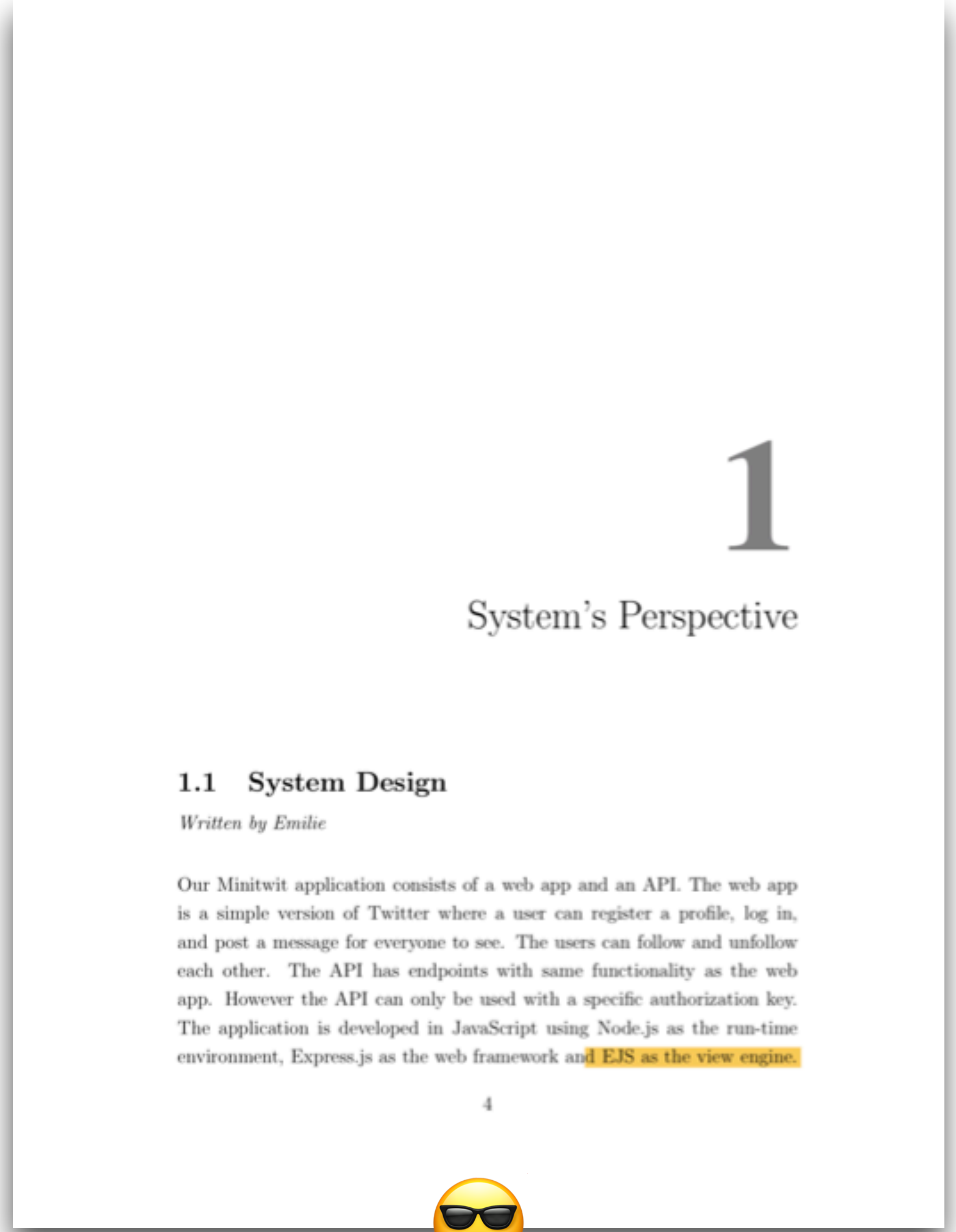
In this section we will discuss how our MiniTwit implementation works from a high level perspective, including its overall interaction with different systems.

During the initial refactoring of MiniTwit we opted to use TypeScript as the coding language and Node.JS as an environment to execute the written code on a backend. To store data we have opted for a MariaDB database. The reason we opted for TypeScript in this project was due to two reasons. Firstly, most of our groups members haven't used TypeScript before, but found it interesting to learn a new language. Secondly, TypeScript is strongly typed in contrast to regular JavaScript which we believe would help improve our maintainability and allow other developers to more easily understand each others code.

Our application was then dockerized in order to easily deploy our application on different servers based on our needs. Furthermore, to orchestrate our application we have been using Docker Swarm with five replicas.

We are using four different servers hosted on DigitalOcean to run our application, where two of these servers solely function as Worker Nodes (with two replicas each) for our Docker Swarm setup and the third functions as a Manager Node. The final server has been our main server before we introduced our scaling and load-balancing solution, and this is the server that contains our database instance and our logging and monitoring containers.

Please refer to the Deployment diagram below which illustrates the structuring of our system across servers and the overall interactions between different containers.



1

System's Perspective

1.1 System Design

Written by Emilie

Our Minitwit application consists of a web app and an API. The web app is a simple version of Twitter where a user can register a profile, log in, and post a message for everyone to see. The users can follow and unfollow each other. The API has endpoints with same functionality as the web app. However the API can only be used with a specific authorization key. The application is developed in JavaScript using Node.js as the run-time environment, Express.js as the web framework and EJS as the view engine.

4

Section Numbers Matter

Reconstruction IV: Dynamic Analysis

We've looked at the source code, we've looked at history, we can not *not look at the running system*.

There are several ways in which we can do this: - add ad-hoc logging statements to the system - "instrument" the code that is being executed by using reflection - monitor network traffic for distributed systems

We will then discuss how can this kind of information be used in architecture recovery.

Limitations of Static Analysis

Case Study: Dead Code Detection

Let us assume that we want to discover whether a given system has code that is not used. This happens quite often actually. - How are we going to do it with static analysis? - What are the limitations of static analysis in this particular problem? - code might look connected to the rest of the call graph but never be called in practice - code might look disconnected but be called using reflectio

Limitations

Some of the limitations of static analysis:

- **Overestimates some relationships** that are only instantiated at runtime
  - runtime polymorphism - from the source code one can not know which of the many alternative implementations is actually used
- **Some information is only really available at runtime**
  - dynamic code evaluation (e.g. `eval`)
  - code that is dependent on user-driven input
  - usage of reflection
- Can not provide **information about runtime properties**
  - E.g., memory consumption and timing might be architecturally relevant

What Is Dynamic Analysis?

Dynamic analysis is a **technique of program analysis** that consists of **instrumenting** and **observing** the **behavior** of a program while it is executing.

Dynamic analysis collects **execution traces** = records of the sequence of actions that happened during an execution.

Think again about the previous *dead code detection* scenario.

> If we had information from the execution of the system we could exclude some candidates if we see that they are used at runtime.

1 Reconstruction IV: Dynamic Analysis

In the previous sessions, we have looked at the source code, we have looked at history, and we saw that interesting information is available there. However, we have now to face the elephant in the room: the *running system itself*. We can not *not look* at it in our attempts to understand the system's architecture, even if, this is going to be the most challenging aspect.

There are several ways in which we can do this: - add ad-hoc logging statements to the system - "instrument" the code that is being executed by using reflection - monitor network traffic for distributed systems

We will then discuss how can this kind of information be used in architecture recovery.

1.1 Limitations of Static Analysis

1.1.1 Case Study: Dead Code Detection

Let us assume that we want to discover whether a given system has code that is not used. This happens quite often actually. - How are we going to do it with static analysis? - What are the limitations of static analysis in this particular problem? - code might look connected to the rest of the call graph but never be called in practice - code might look disconnected but be called using reflectio

1.1.2 Limitations

Some of the limitations of static analysis:

- **Overestimates some relationships** that are only instantiated at runtime
  - runtime polymorphism - from the source code one can not know which of the many alternative implementations is actually used
- **Some information is only really available at runtime**
  - dynamic code evaluation (e.g. `eval`)
  - code that is dependent on user-driven input
  - usage of reflection
- Can not provide **information about runtime properties**
  - E.g., memory consumption and timing might be architecturally relevant

1.2 What Is Dynamic Analysis?

Dynamic analysis is a **technique of program analysis** that consists of **instrumenting** and **observing** the **behavior** of a program while it is executing.

Dynamic analysis collects **execution traces** = records of the sequence of actions that happened during an execution.

Think again about the previous *dead code detection* scenario.

Use —number-sections with pandoc to convert .md files (e.g. https://github.com/mircealungu/reconstruction/blob/master/tools/md_to_pdf.sh)

Text in Figures Should be of Comparable Size to Text in Page

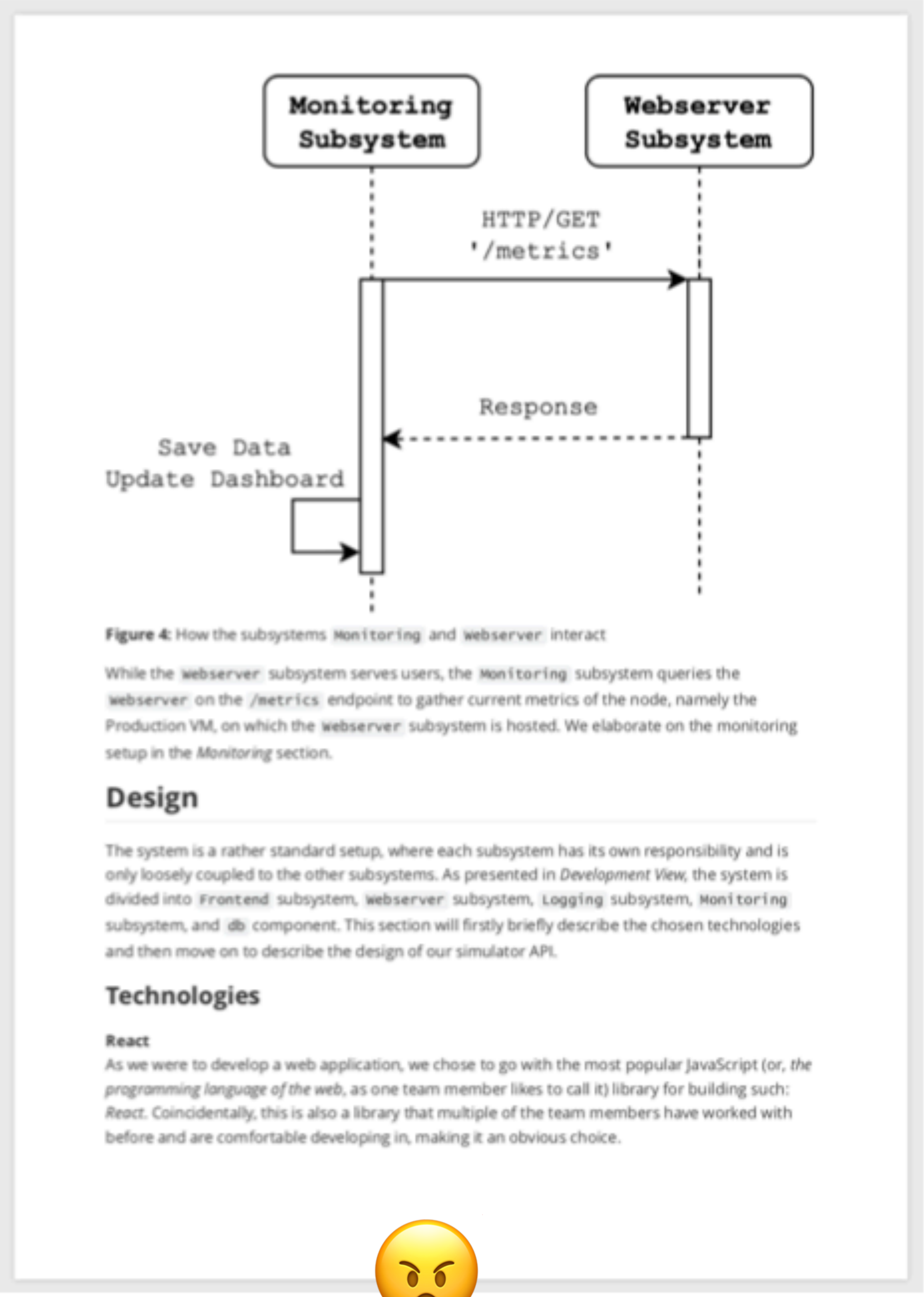


Figure 4: How the subsystems `Monitoring` and `webserver` interact

While the `webserver` subsystem serves users, the `Monitoring` subsystem queries the `webserver` on the `/metrics` endpoint to gather current metrics of the node, namely the Production VM, on which the `webserver` subsystem is hosted. We elaborate on the monitoring setup in the *Monitoring* section.

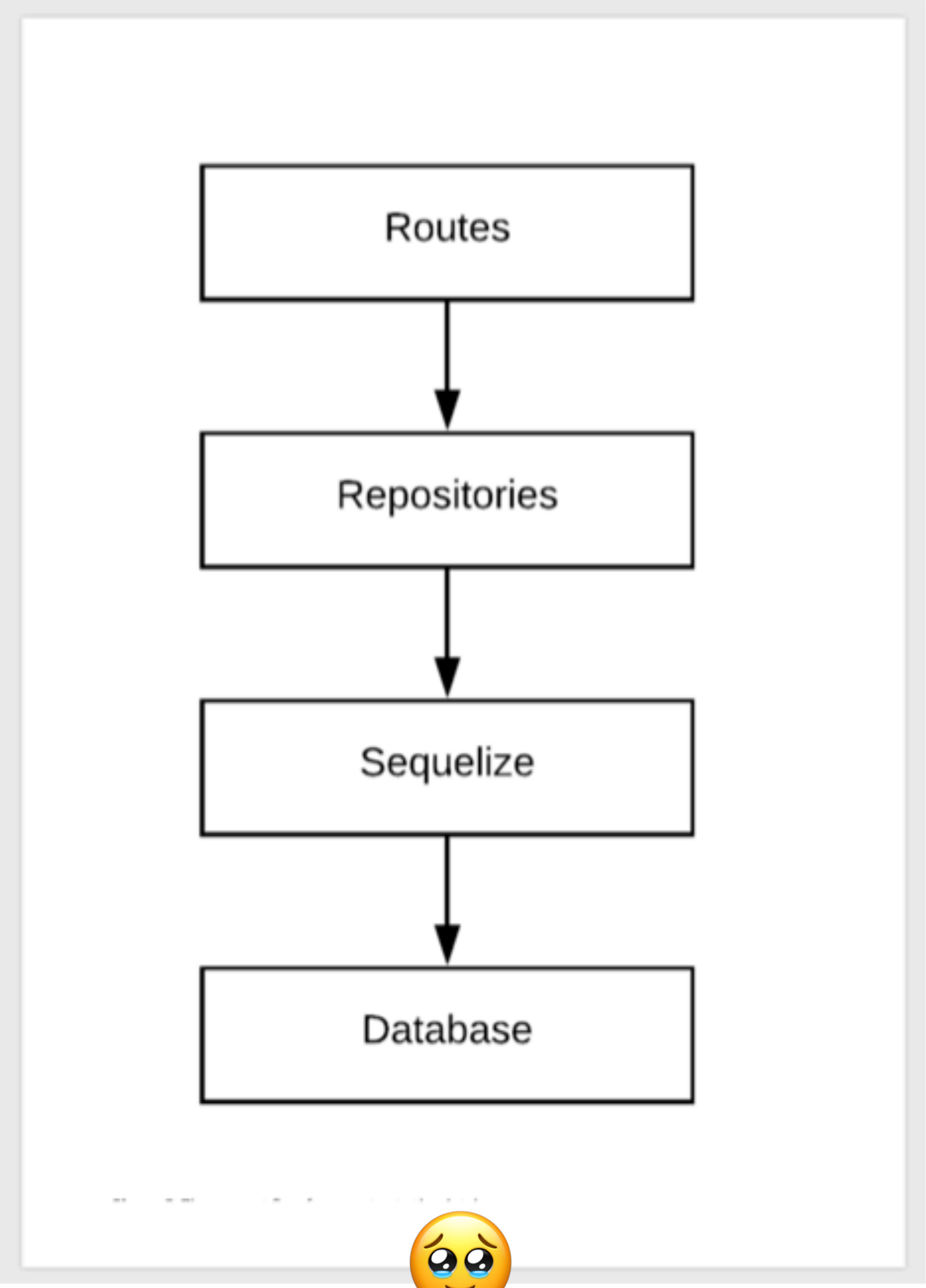
Design

The system is a rather standard setup, where each subsystem has its own responsibility and is only loosely coupled to the other subsystems. As presented in *Development View*, the system is divided into `Frontend` subsystem, `webserver` subsystem, `Logging` subsystem, `Monitoring` subsystem, and `db` component. This section will firstly briefly describe the chosen technologies and then move on to describe the design of our simulator API.

Technologies

React

As we were to develop a web application, we chose to go with the most popular JavaScript (or, *the programming language of the web*, as one team member likes to call it) library for building such: *React*. Coincidentally, this is also a library that multiple of the team members have worked with before and are comfortable developing in, making it an obvious choice.



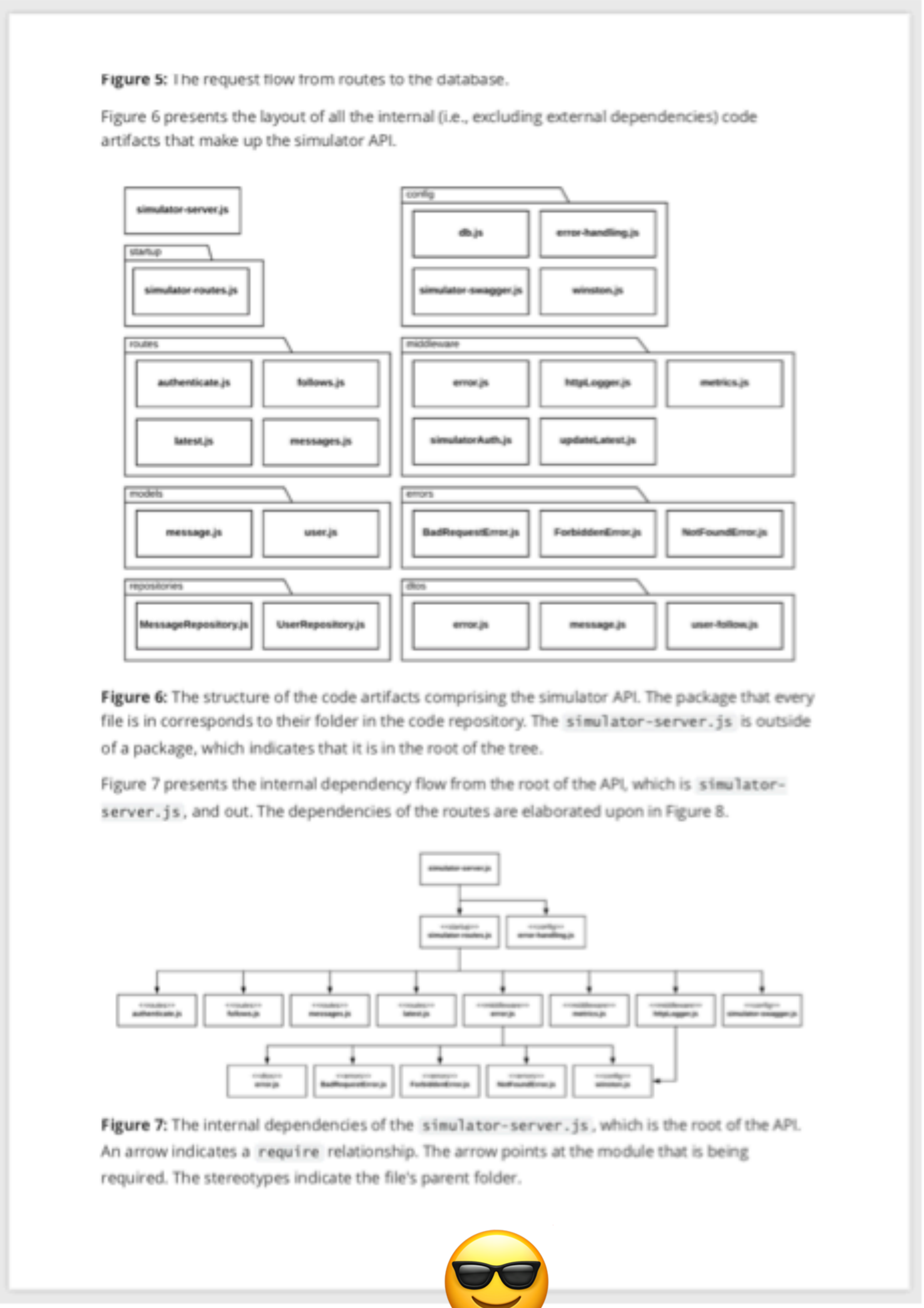


Figure 5: The request flow from routes to the database.

Figure 6 presents the layout of all the internal (i.e., excluding external dependencies) code artifacts that make up the simulator API.

Figure 6: The structure of the code artifacts comprising the simulator API. The package that every file is in corresponds to their folder in the code repository. The `simulator-server.js` is outside of a package, which indicates that it is in the root of the tree.

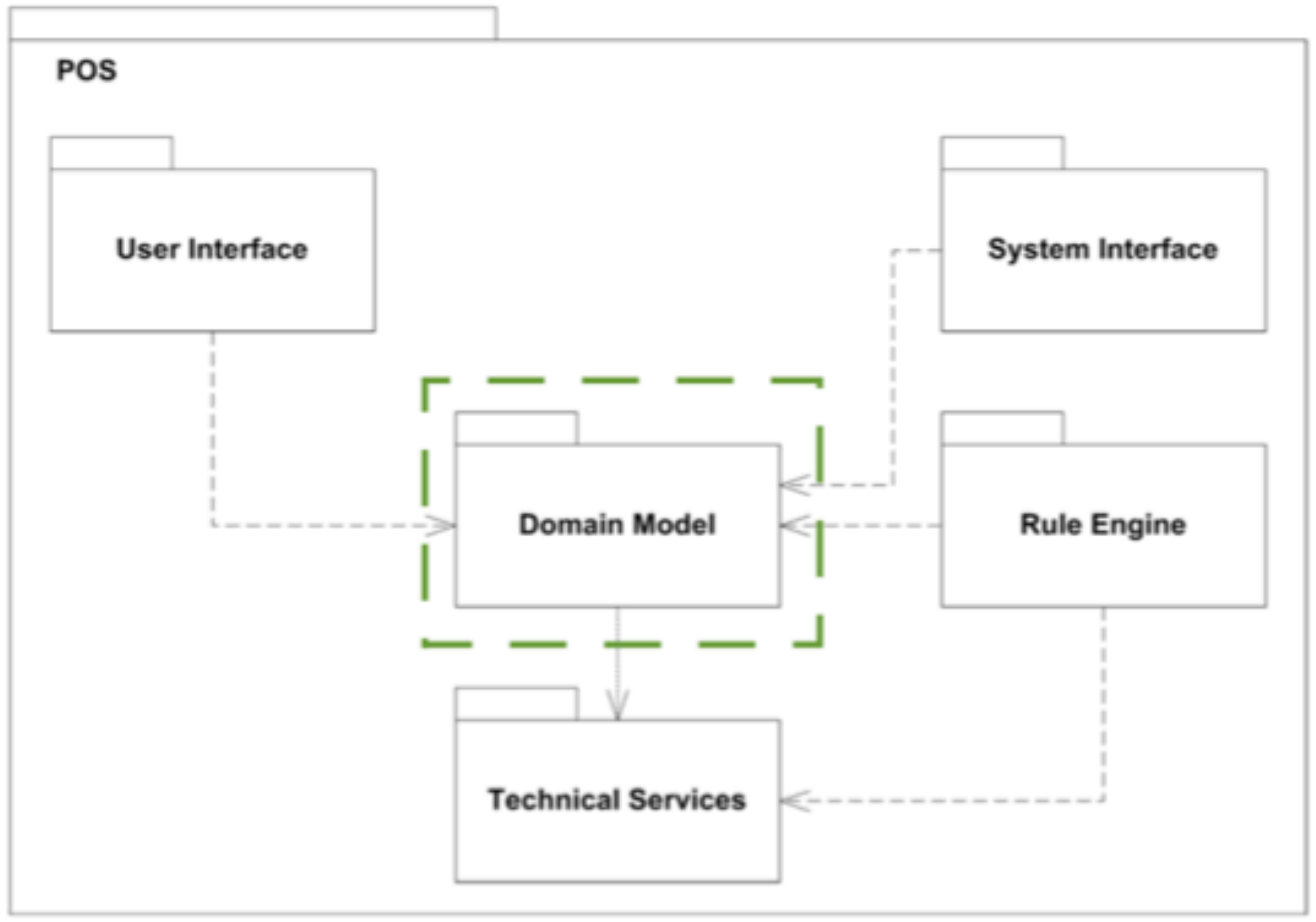
Figure 7 presents the internal dependency flow from the root of the API, which is `simulator-server.js`, and out. The dependencies of the routes are elaborated upon in Figure 8.

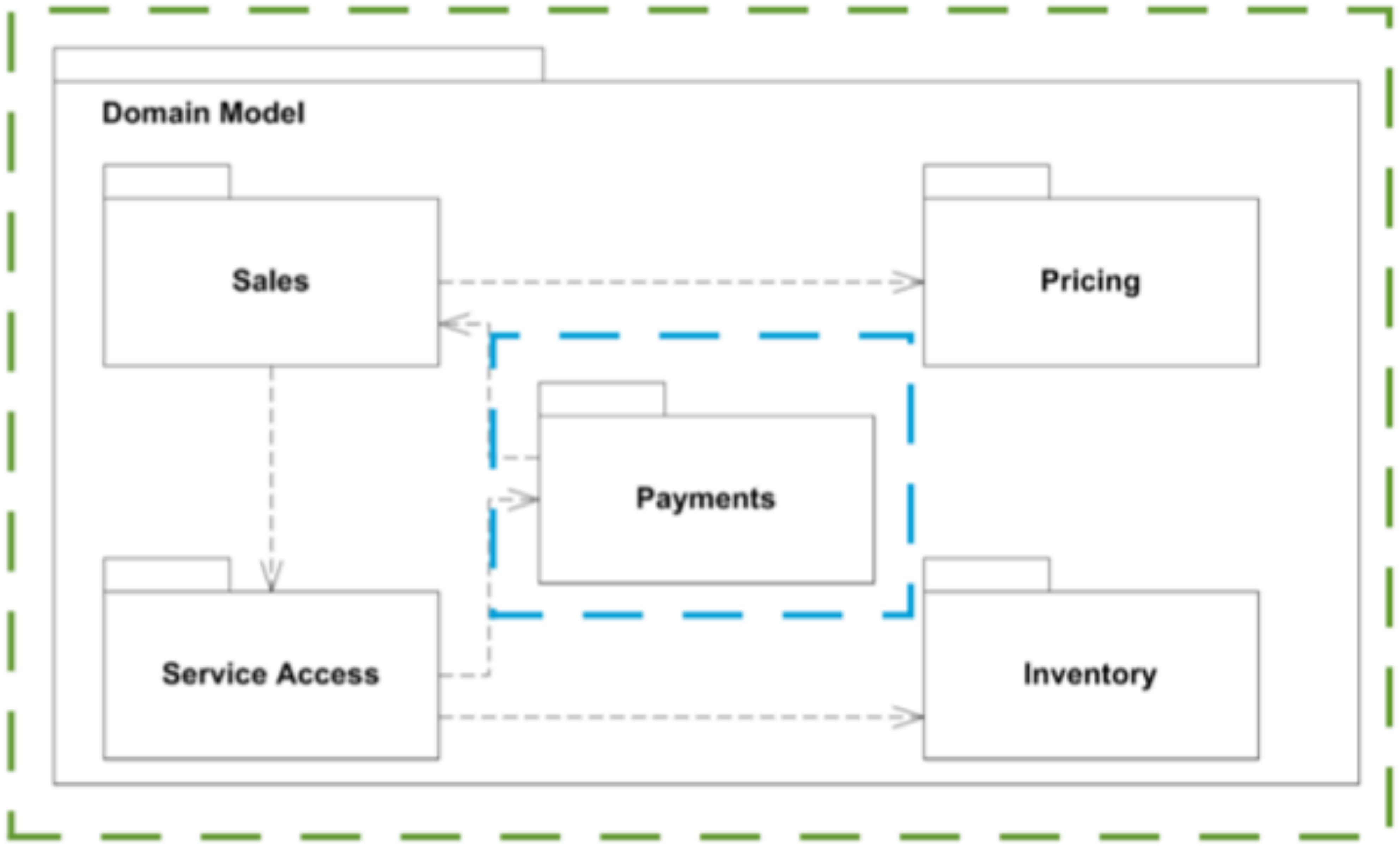
Figure 7: The internal dependencies of the `simulator-server.js`, which is the root of the API. An arrow indicates a `require` relationship. The arrow points at the module that is being required. The stereotypes indicate the file's parent folder.

https://github.com/mircealungu/student-projects/blob/master/writing_guidelines/Use_the_Right_Font_Size_in_Images.md

Use multiple views for the same viewpoint

To make complexity more manageable





The information visualization mantra:
“Overview, Zoom, Details on Demand” (B. Schneiderman)

Is this image worth a hundred words?

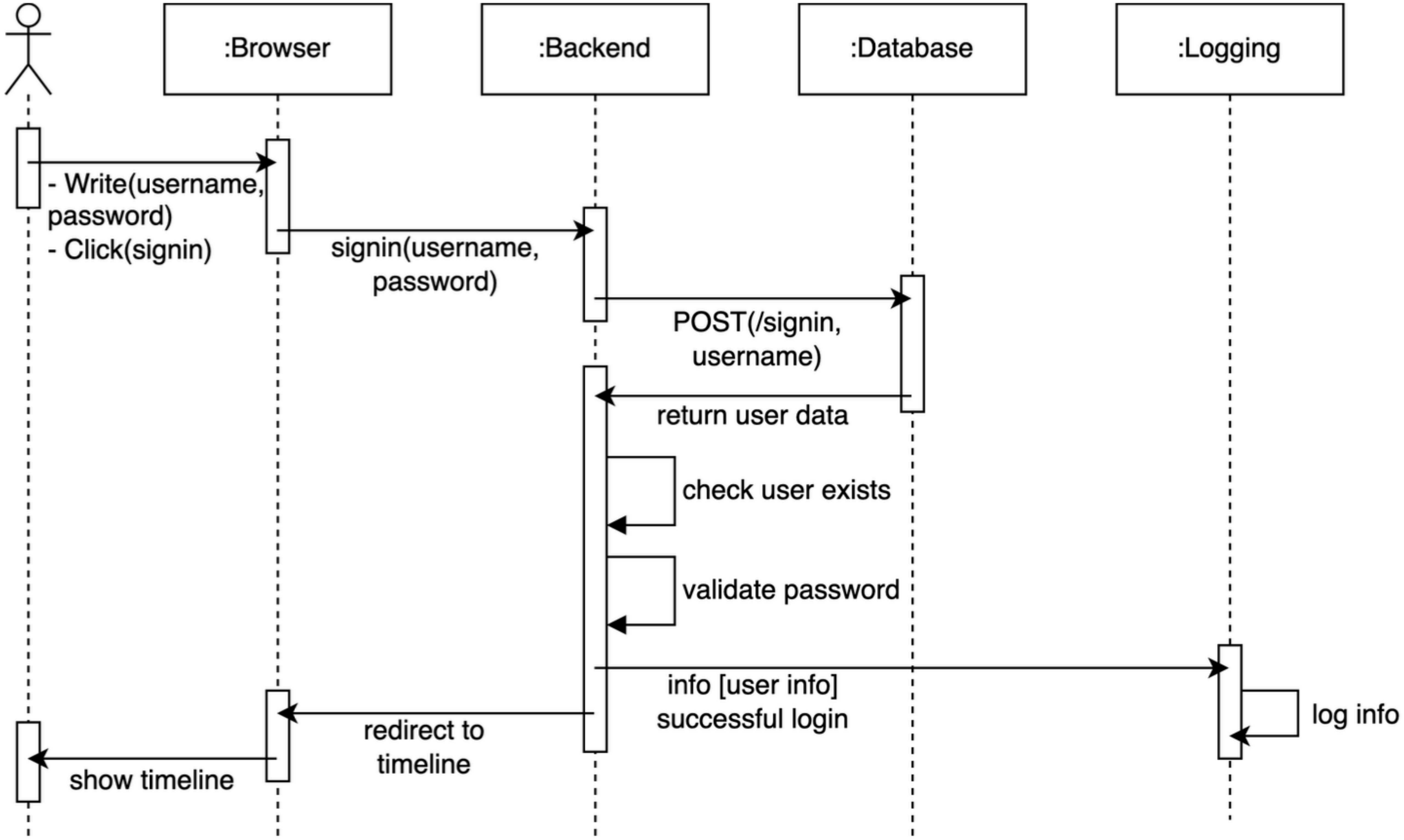


Figure 4: Sequence diagram of a successful login scenario

Are these hundred words worth an Image?

1.3 Important interactions of subsystems

The frontend relies on interaction with the backend to get the information needed to show the user. The backend also relies on the database for data to process and pass to the frontend.

The monitoring subsystem consists of Prometheus and Grafana containers. Prometheus relies on the backend for data scraping. Grafana needs a data source in Prometheus, to display information.

The logging setup is composed of ElasticSearch and Kibana containers, as well as our logging provider in Serilog. Serilog relies on the backend for logging data which it sends to the ElasticSearch sink, which feeds that data to Kibana.

References

1. An Approach to Software Architecture Description Using UML Revision 2.0. Henrik Bærbak Christensen, Aino Corry, and Klaus Marius Hansen **(<—— THIS IS THE 3+1 :)**

2. Architectural Blueprints—The "4+1" View Model of Software Architecture. Philippe Kruchten (this is just for reference)

3. https://www.uml-diagrams.org

    1. Deployment Diagrams

    2. Component Diagrams

4. Writing Guidelines. M. Lungu (Github)