Documenting Architecture

DevOps, Maintenance, and Evolution @ ITU

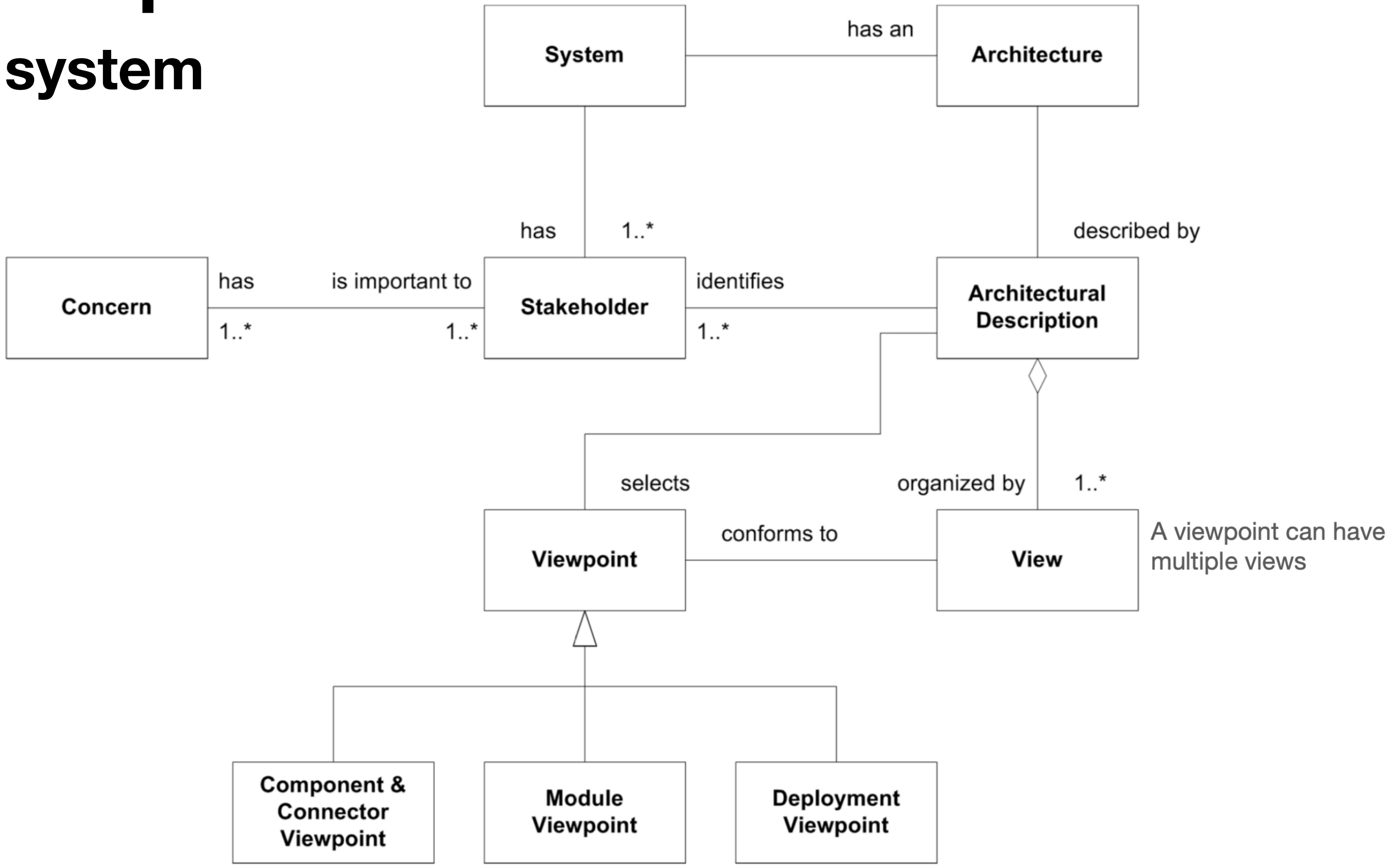
Mircea Lungu

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



Viewpoints

Answer different questions

Popular “catalogues” of viewpoints

- 4+1 by Kruchten
- 3+1 by Christensen
  - Module
  - Components and Connectors
  - Allocation
- …*

* 7 min abs?

Module Viewpoint

How is the functionality organized in code?

Elements

- Packages
- Classes
- Interfaces
- Modules

Relationships

- Dependencies

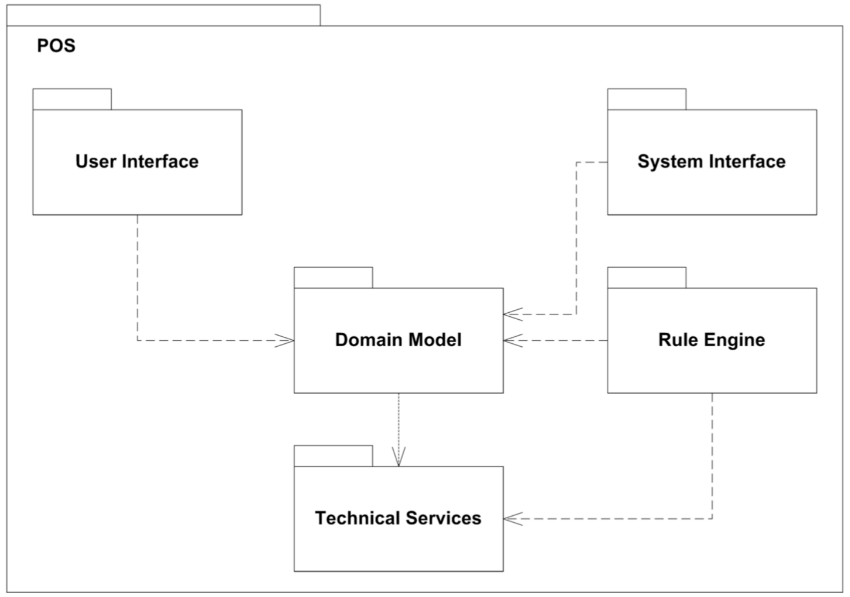


Figure 2: Package overview diagram for the POS system

Components And Connectors

How Does the System Achieve It's Functionality at Runtime?

Components = units of functionality

Connectors = communication channels between components

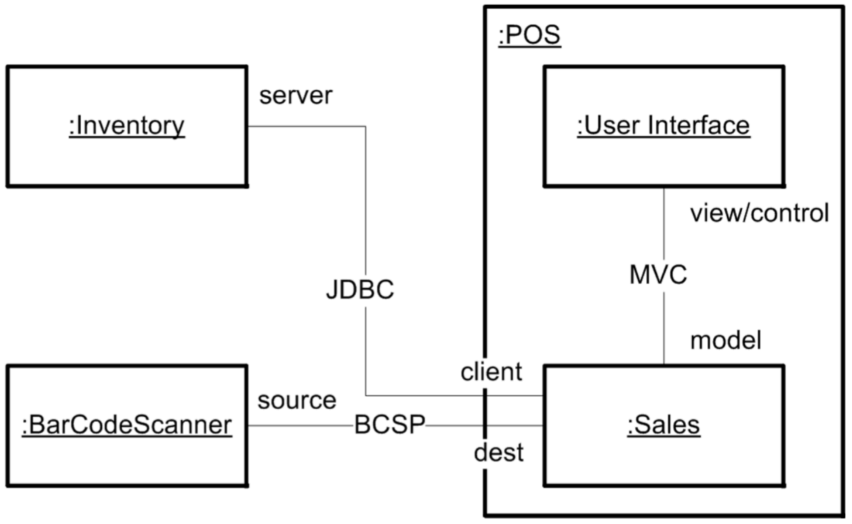


Figure 5: C&C overview of the POS system

Deployment

How are elements mapped on the infrastructure?

Elements

- Processes
- Infrastructure

Relationships

- Allocated-to
- Depends-on
- Protocol links

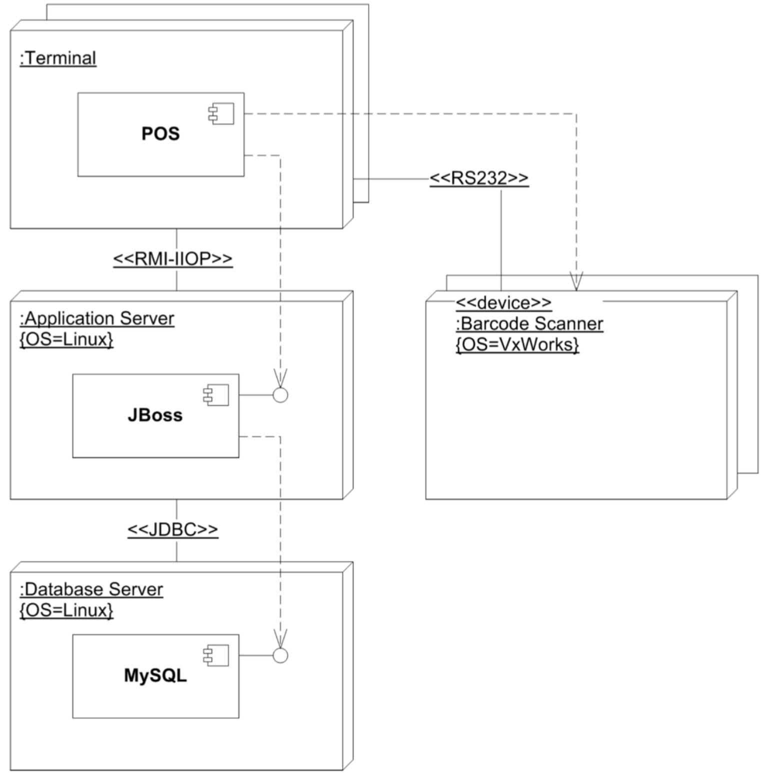


Figure 7: Deployment view of the NextGen POS system

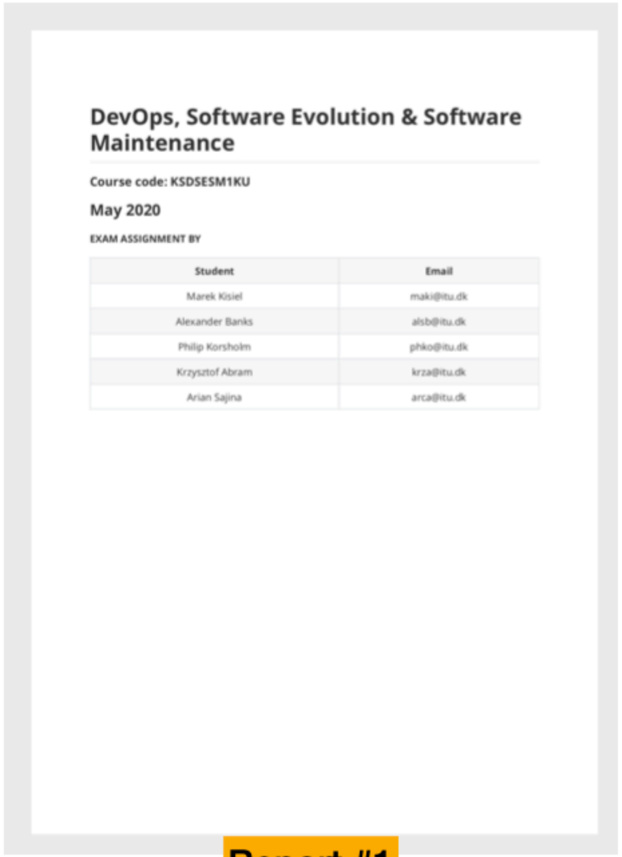
Visual Representation

- Use a Standard Notation (e.g. UML)
  - Deployment Diagrams
  - Component Diagrams
- Or Create / Adapt Your Own, but then
  - Explain it
  - Add a legend

Formatting Your Report

Make it as readable as possible

A Report Has a Title and Authors



DevOps, Software Evolution & Software Maintenance

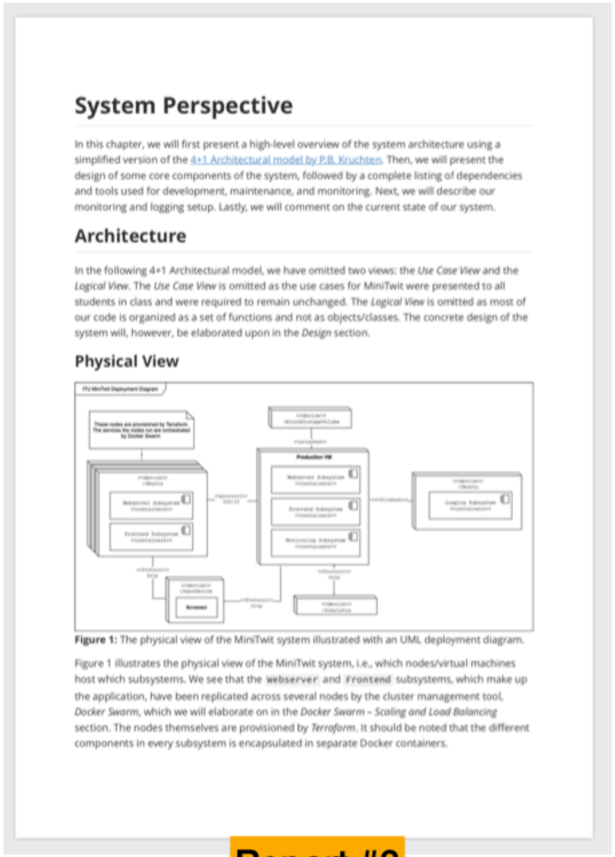
Course code: KSDSESM1KU

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.

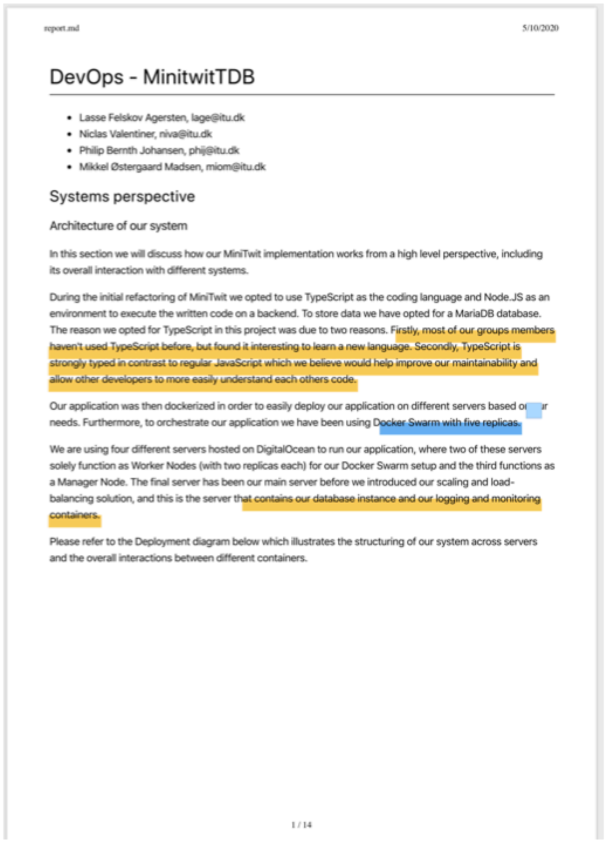
Physical View

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.

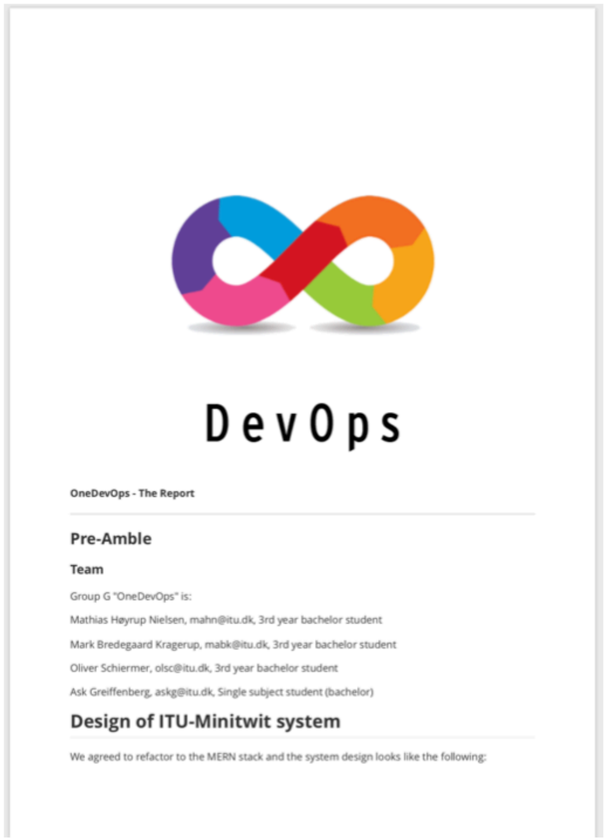
During the initial refactoring of MiniTwit we opted to use TypeScript as the coding language and Node.JS as an environment to execute the written code on a backend. To store data we have opted for a MariaDB database. The reason we opted for TypeScript in this project was due to two reasons. Firstly, 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 / 14



OneDevOps - The Report

Pre-Amble

Team

Group G "OneDevOps" is:

Mathias Høyrup Nielsen, mahn@itu.dk, 3rd year bachelor student

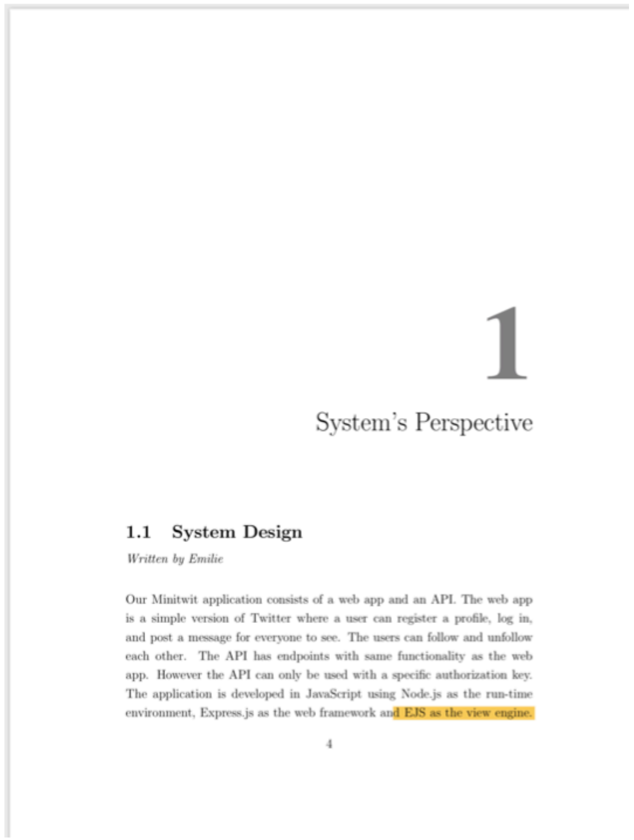
Mark Bredegaard Kragerup, mabk@itu.dk, 3rd year bachelor student

Oliver Schiermer, olsc@itu.dk, 3rd year bachelor student

Ask Greiffenberg, askg@itu.dk, Single subject student (bachelor)

Design of ITU-Minitwit system

We agreed to refactor to the MERN stack and the system design looks like the following:


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

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

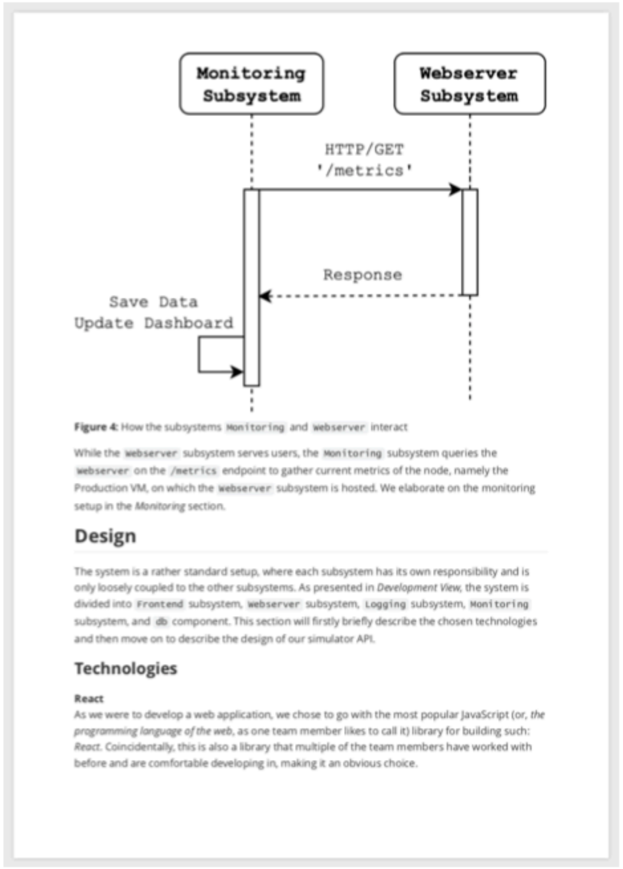


Figure 4: How the subsystems `Monitoring` and `Webserver` interact

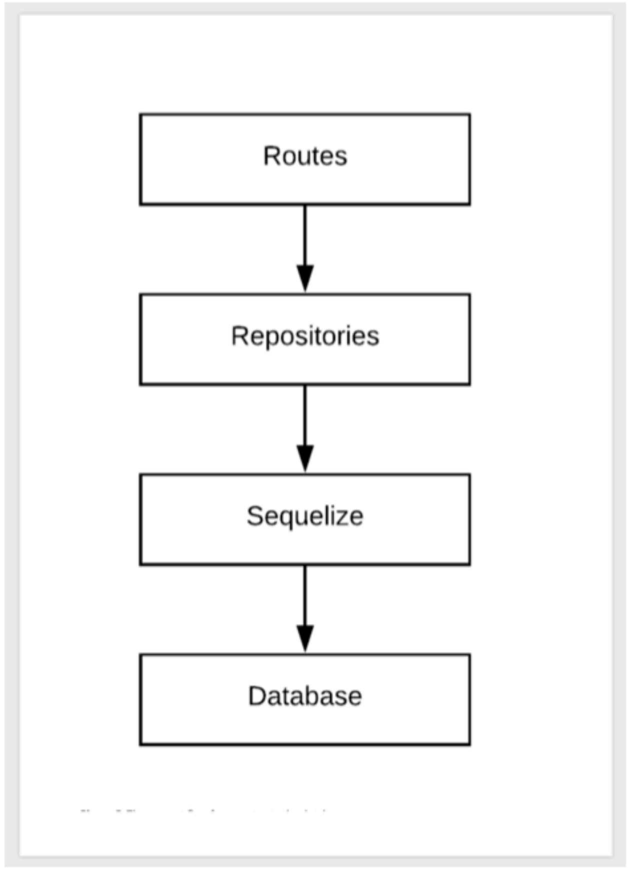
While the `Webserver` subsystem serves users, the `Monitoring` subsystem queries the `Webserver` on the `/metrics` endpoint to gather current metrics of the node, namely the Production VM, on which the `Webserver` subsystem is hosted. We elaborate on the monitoring setup in the *Monitoring* section.

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.



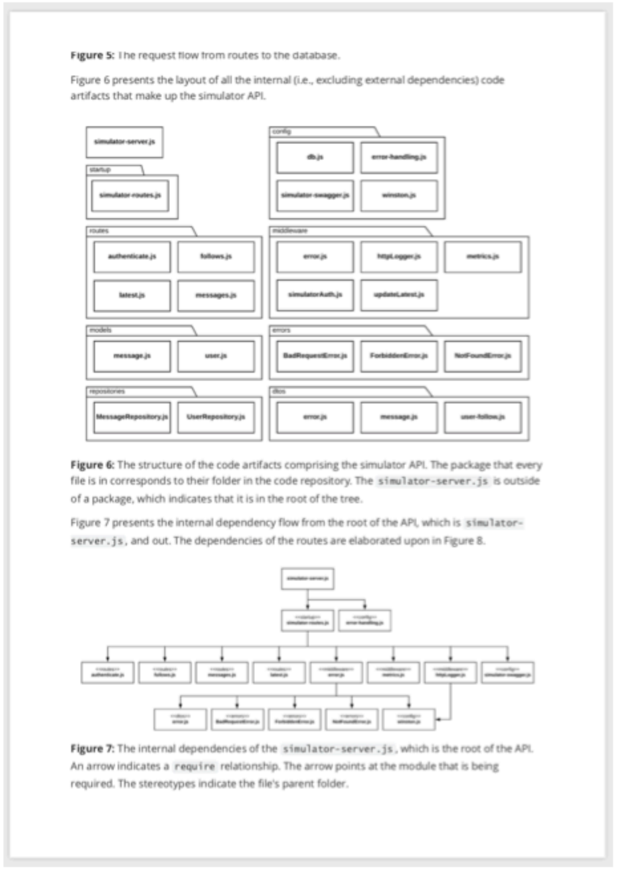


Figure 5: The request flow from routes to the database.

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.

Multiple Views for the Same Viewpoint

Make Complexity More Manageable

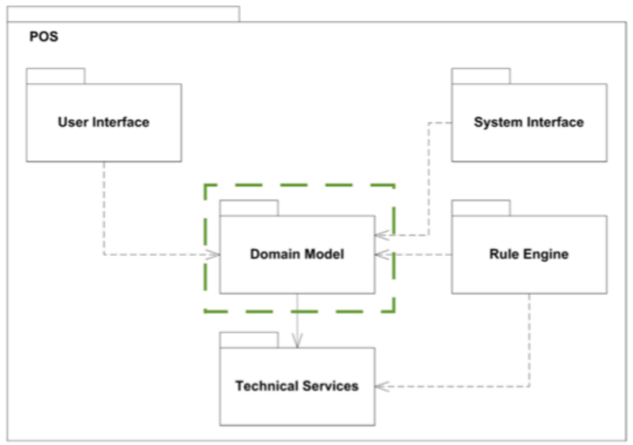


Figure 2: Package overview diagram for the POS system

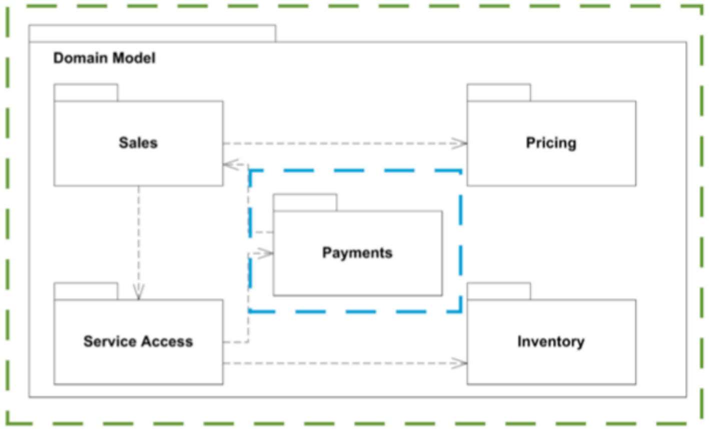


Figure 3: Decomposition of the Domain Model package of the POS system

If you end up showing class diagrams you should only show the essential ones to make your point. It's architecture after all!

References

1. An Approach to Software Architecture Description Using UML Revision 2.0. Henrik Bærbak Christensen, Aino Corry, and Klaus Marius Hansen

2. Architectural Blueprints—The “4+1” View Model of Software Architecture. Philippe Kruchten

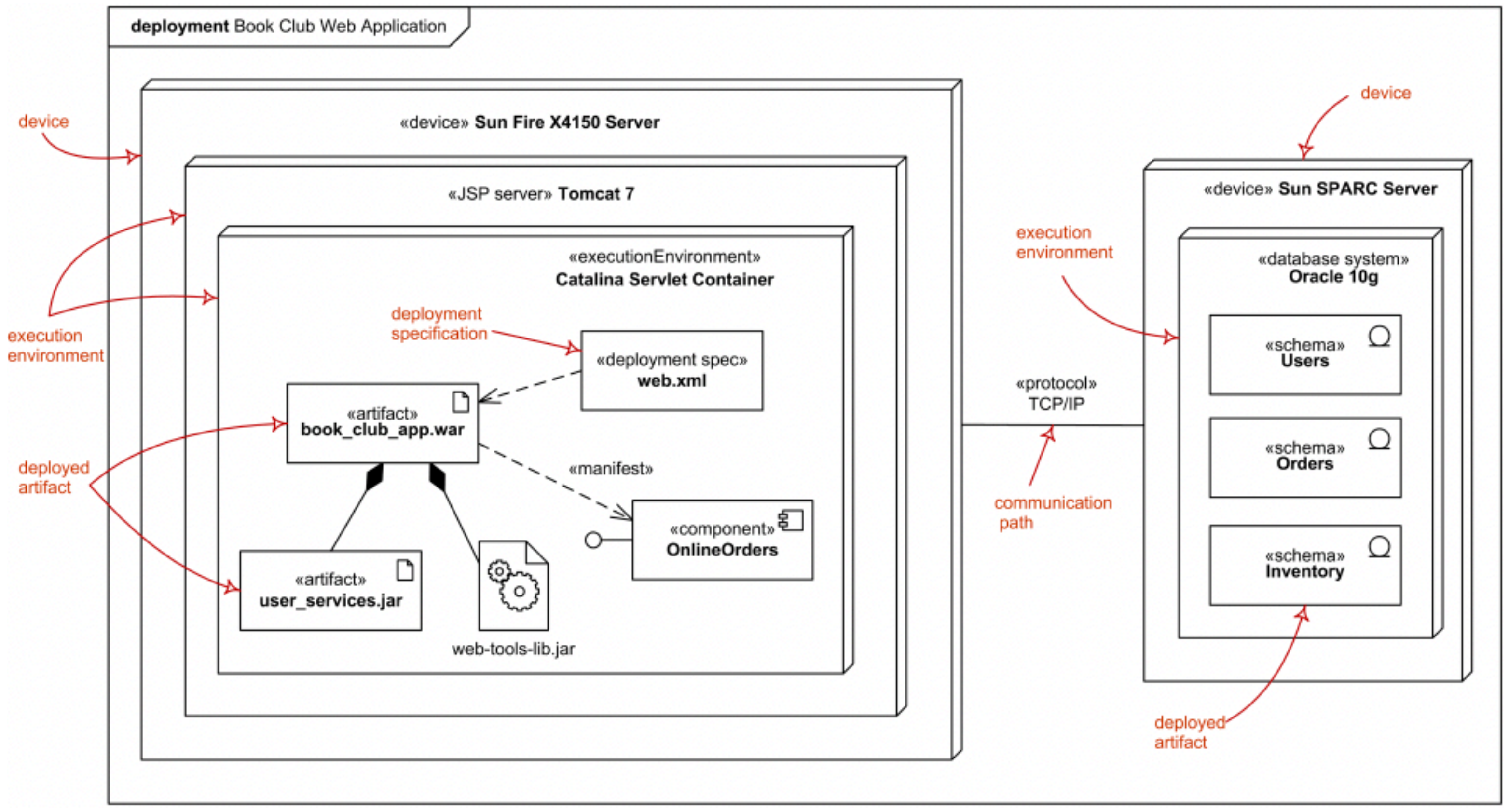
3. https://www.uml-diagrams.org

    1. Deployment Diagrams

    2. Component Diagrams

4. Writing Guidelines. M. Lungu (Github)

UML Deployment Diagram


deployment Book Club Web Application
«device» Sun Fire X4150 Server
«JSP server» Tomcat 7
«executionEnvironment» Catalina Servlet Container
«deployment spec» web.xml
«artifact» book_club_app.war
«manifest»
«component» OnlineOrders
«artifact» user_services.jar
web-tools-lib.jar
«protocol» TCP/IP
«device» Sun SPARC Server
«database system» Oracle 10g
«schema» Users
«schema» Orders
«schema» Inventory
device
execution environment
deployment specification
deployed artifact
communication path
device
execution environment
deployed artifact