



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

OPTIMIZATION AND TESTING OF THE FRONTEND ON THE SMART CITY PLATFORM

OPTIMALIZACE A TESTOVÁNÍ FRONTENDOVÉ ČÁSTI NA PLATFORMĚ SMART CITY

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

JÁN ŠPAČEK

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. PETR JOHN

BRNO 2024

Bachelor's Thesis Assignment



154660

Institut: Department of Information Systems (DIFS)
Student: **Špaček Ján**
Programme: Information Technology
Title: **Optimization and testing of the frontend on the Smart City platform**
Category: Web applications
Academic year: 2023/24

Assignment:

1. Study the Internet of Things (IoT) and the Smart City concept.
2. Study the area of testing with a focus on automated testing of user interfaces. Explore available technologies and ways to evaluate web page performance.
3. Analyze the current state of automated testing of user interfaces on Logimic's Smart City platform. Evaluate the gaps and define the requirements.
4. Based on the results of the analysis in the previous section, select methods and libraries and propose methods for automated user interface testing on Logimic's Smart City platform. Suggest ways to increase the efficiency of the user interface parts.
5. Implement the proposed solution in consultation with the supervisor.
6. Test the functionality and usability of the implementation on Logimic's Smart City platform and evaluate whether the proposed modifications have resulted in increased efficiency.

Literature:

- Greengard, S.: The Internet of Things. MIT Press, 2015, ISBN 978-026-2527-736.
- Angular: Introduction to the Angular Docs [online]. 2023 [cit. 2023-08-29]. Available on: <https://angular.io/docs>
- Logimic internal documentation

Requirements for the semestral defence:

Points 1 to 4.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **John Petr, Ing.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: 1.11.2023
Submission deadline: 9.5.2024
Approval date: 30.10.2023

Abstract

The primary objective of this thesis is to develop and implement an automated testing environment for user interface and performance testing on the ACADA platform in collaboration with Logimic company. Additionally, this thesis aims to identify and propose performance enhancements for the ACADA platform. To achieve these goals, the thesis thoroughly analyzes the current state of testing at Logimic, selects the most suitable testing framework, implements a comprehensive testing suite using Cypress, and establishes standards for test structure and workflow. The result of this thesis is a robust testing suite that allows early identification of issues in the development process. By leveraging the capabilities of Cypress, the testing suite ensures a faster and more efficient testing workflow and also contributes to improved user satisfaction by ensuring the platform is reliable and responsive.

Abstrakt

Cílem této práce je ve spolupráci s firmou Logimic vytvořit prostředí pro automatizované testování uživatelského rozhraní a výkonu platformy ACADA. Dalším cílem je odhalit problematické části platformy způsobující omezení výkonu a navrhnout optimalizace. K dosažení stanovených cílů byl analyzován aktuální stav testování ve firmě a navržen nejvhodnější softvér pro testování. Výsledkem práce je soubor testů implementovaný v knihovně Cypress, umožňující časnou identifikaci problémů v procesu vývoje. Využitím schopností knihovny Cypress zajišťuje tento soubor testů rychlejší a efektivnější postup testování a přispívá také k spokojenosti uživatelů zajištěním spolehlivosti a responzivity.

Keywords

Internet of Things, Smart City, automated testing, performance testing, testing framework, Cypress, Logimic, application interface

Klíčová slova

Internet věcí, chytrá města, automatizované testování, výkonnostní testování, testovací knihovna, Cypress, Logimic, uživatelské rozhraní

Reference

ŠPAČEK, Ján. *Optimization and testing of the frontend on the Smart City platform*. Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Petr John

Rozšířený abstrakt

S postupujícím technologickým pokrokem se chytrá zařízení stávají neodmyslitelnou součástí našeho každodenního života. Chytrá zařízení přinášejí inovace do různých oblastí od domácností po průmysl. Senzory, které jsou součástí Internetu věcí (IoT), popisují okolní prostředí tím, že snímají jeho fyzikální vlastnosti. Tím vytvářejí abstrakci reálného světa, která umožňuje monitorovat a řídit zařízení z jakéhokoli místa na světě prostřednictvím internetu. Od chytrých termostátů a senzorů pohybu po průmyslové senzory, tyto technologické inovace otevírají dveře k efektivnějšímu a pohodlnějšímu sledování a řízení zařízení. Propojení těchto zařízení vytváří dynamický ekosystém, který se využívá v široké škále odvětví, od modernizace zemědělství a průmyslu až po vývoj chytrých měst. V rámci chytrých měst slouží IoT ke zlepšení různých služeb, jako jsou doprava, energetická infrastruktura a zdravotnictví, prostřednictvím efektivního sběru a analýzy dat z různých senzorů.

Data z těchto senzorů jsou zásadní pro efektivní fungování chytrých systémů, a proto je nutné je rychle, efektivně a interaktivně zobrazovat. K tomu účelu slouží aplikace a platformy, které propojují různé systémy, usnadňují v reálném čase výměnu dat a poskytují uživatelům přehled o stavu zařízení a celkového systému. Při vývoji a provozu těchto platform je klíčové, aby umožňovaly uživatelům interpretovat data z chytrých zařízení, byly uživatelsky přívětivé a rychlé, aby efektivně reagovaly na potřeby uživatelů. Chybná vizualizace dat může vést uživatele k špatné interpretaci dat a udělení špatného strategického rozhodnutí. Každá sekunda, kterou trvá načtení aplikace, negativně ovlivňuje uživatelskou zkušenost, množství interakcí a míru udržení zákazníků.

Zajištění stability a optimální funkčnosti aplikací vyžaduje systematické a průběžné testování. K ověření funkcionality slouží funkcionální testování, které důkladně prověřuje jednotlivé komponenty aplikace a celkový integrovaný systém. Pro posouzení odolnosti aplikace vůči zátěži a dosažení optimální rychlosti se využívá nefunkcionální testování. Tento přístup je zaměřen na identifikaci problematických částí aplikace, které mohou omezovat její výkon v situacích s krátkodobou i dlouhodobou náporovou zátěží.

Práce se zabývá návrhem testovací strategie pro aplikaci vyvinutou společností Logimic, se zaměřením na platformu ACADA používající framework Angular. Dalším z cílů této práce je implementace automatického testování uživatelského rozhraní pomocí knihovny Cypress, která umožňuje testování jednotlivých komponent a aplikačního rozhraní, s důrazem na klíčové oblasti, které uživatelé nejčastěji využívají a kde jsou potenciální problémy nejvíce patrné. V rámci této práce byly identifikovány konkrétní problémy omezující výkon v těchto oblastech aplikace. Tyto problémy byly detailně analyzovány a následně navrženy optimalizace, jejichž cílem bylo eliminovat identifikované nedostatky. Tím bylo přispěno k zajištění lepší stability, rychlosti a efektivity celkového systému. Úsilí bylo zaměřeno na zlepšení uživatelské zkušenosti prostřednictvím odhalení a následného odstranění konkrétních překážek.

V rámci této práce byly implementovány testy zaměřené na nejpoužívanější části aplikace, jakož i okrajové případy a různá rozlišení obrazovky pro zajištění rozsáhlého pokrytí testů a zabezpečení správného fungování platformy na různých typech zařízení. Práce stanovuje standardy pro testovací strukturu a postup. Tím byla zajištěna přehlednost a konzistence všech provedených testů. Tyto standardy pomáhají udržovat jednotnou a přehlednou testovací sadu, což usnadňuje další práci na vývoji aplikace a údržbě testů v budoucnosti.

Pro implementaci CI/CD pipelines byla zvolena služba GitHub Actions, která nabízí automatizované spouštění testů a možnost přizpůsobit se specifickým potřebám společnosti Logimic. Tímto způsobem může být dosaženo zvýšené efektivity vývoje a nasazování nových

verzí aplikace. Dále bylo jako součást testovací infrastruktury využito Raspberry Pi jako lokálního zařízení pro spouštění testů. Tento přístup vedl k úspoře nákladů, jelikož testy bylo možné provádět na cenově dostupném zařízení.

K analýze výkonu platformy, generování zpráv o výkonu a identifikaci problematických částí aplikace byla použita služba Google Lighthouse. Na základě těchto zpráv a navržených optimalizací z analýzy byly provedeny úpravy s cílem zlepšit celkový výkon.

Účinek provedených optimalizací byl ověřen porovnáním zpráv o výkonu znástroje Google Lighthouse před a po implementaci úprav. Tento přístup umožnil identifikaci a vyhodnocení dopadu provedených úprav na celkový výkon aplikace, poskytující cenné informace pro další optimalizace. Implementované optimalizace vedly k vylepšení skóre v Lighthouse, což potvrzuje jejich pozitivní dopad na výkon aplikace.

V neposlední řadě během testování bylo odhaleno několik chyb, které byly následně hlášeny společnosti Logimic a promptně opraveny.

Optimization and testing of the frontend on the Smart City platform

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Petr John. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ján Špaček
May 3, 2024

Acknowledgements

I would like to sincerely thank my supervisor for his guidance, willingness, and insightful advice throughout my thesis, Ing. Ondřej Šulc and Ing. Michal Valný, Ph.D. from Logimic for their valuable insights and help. I would also like to extend my appreciation to Logimic for allowing me to work on a practical application that has real-life implications.

Contents

1	Introduction	4
2	Internet of Things (IoT)	5
2.1	Architecture	6
2.2	IoT Devices	8
2.3	Technologies	9
3	Smart Cities	11
3.1	Smart City Domains	12
3.2	Dashboard	15
4	Software Testing	17
4.1	Testing Methods	17
4.2	Testing Methods	17
4.3	Testing Tools and Frameworks	19
5	Analysis	23
5.1	Logimic	23
5.2	Current State of Testing at Logimic	26
6	Implementation Design	28
6.1	End-to-end Tests	29
6.2	Performance	30
7	Implementation	33
7.1	Performance Improvements	39
8	Testing	42
8.1	Performance Improvement Evaluation	43
8.2	Impact of Performance Improvements	45
9	Conclusion	46
	Bibliography	47
A	Contents of the included storage media	52

List of Figures

2.1	Global IoT market forecast [47]	5
2.2	IoT architecture layers [24]	6
2.3	Service oriented Architecture [26]	7
2.4	Five layer architecture [57]	7
2.5	Cisco Internet of Things Reference Model [9]	8
2.6	Smart sensor types	9
3.1	Six Smart City characteristics [16]	13
3.2	Eight Smart City domains [6]	14
4.1	Functional testing pyramid [36]	18
4.2	Core Web Vitals	22
5.1	Visualization of system groups	24
5.2	Visualization of group KPIs	24
5.3	KPI alert detail	24
5.4	List of devices	25
5.5	Device detail	25
5.6	Device Statistics	26
6.1	Proposed workflow	29
6.2	Current network flow	31
6.3	Google Lighthouse report of L1 dashboard on Windows before any performance improvements were implemented	32
7.1	Test structure with pseudo-code	34
7.2	A screenshot of a GitHub Action run	37
7.3	Screenshot of a failed test	37
7.4	Lighthouse report with plugins (left) and without (right)	39
7.5	Distribution of Angular bundles stored on AWS S3 bucket, delivered to clients via AWS CloudFront	39
7.6	Coverage report of L1 Dashboard using Coverage DevTool on Google Chrome	40
7.7	loading of user information followed by parallel loading of system information, forms, and profile image	41
7.8	A single query containing user and system information, forms, and profile image	41
8.1	Calendar Modal on L2 that can not be interacted with	42
8.2	Action modal on L3 (left) and User profile image with add and delete button (right)	43

8.3	Google Lighthouse report with extensions (left) and without (right)	44
8.4	Google Lighthouse report on Windows with AMD Ryzen 5 5600X CPU (left), Metric scores from Web Vital extension (right)	44
8.5	Compression comparison on image with variegated background (left) and white background (right)	45
8.6	Google Lighthouse report before performance improvements (left) and after (right)	45

Chapter 1

Introduction

As our world becomes increasingly digitized, the surge in the number of smart devices has become a technological trend. From smart thermostats to industrial sensors, smart devices have become a part of our daily lives. The Internet of Things thrives on the interconnectedness of these smart devices. It transforms them from isolated entities into nodes in a vast network. This transformation facilitates communication, data exchange, and collaborative functionality. In doing so, it creates opportunities for innovation and efficiency across various sectors, including healthcare, agriculture, military, industrial applications, Smart Grids, Smart Homes, and Smart Cities.

Smart Cities utilize interconnected technologies to enhance urban efficiency and responsiveness. Through a variety of initiatives, such as intelligent transportation systems, energy-efficient infrastructure, data-driven governance, and more, these cities promote sustainability and optimize public services, creating a safer, more accessible, and innovative urban environment.

Smart Cities rely on robust platforms and applications to integrate diverse systems, facilitate real-time data exchange, and provide users with insights. Testing plays a vital role in the process of making robust applications. Various testing types, including functional and performance, are instrumental in uncovering potential issues, optimizing system efficiency, ensuring it meets all requirements, and providing satisfying user interaction.

The goal of this thesis is to create a new automated testing environment for a Smart City platform by Logimic. The objective is to analyze and suggest improvements to its performance, implement the proposed enhancements, and subsequently test their effects. This thesis also aims to establish a testing workflow and demonstrate a structured approach to writing end-to-end tests. Chapter 2 provides a more detailed description of IoT and its various architectures, sensors, and technologies. Chapter 3 defines Smart Cities, their characteristics, domains, and data visualization. Chapter 4 explores different approaches to testing and analyzes various popular testing tools and frameworks. Chapter 5 outlines the typical user workflows and the architecture employed by Logimic. It provides an overview of the current state of testing at Logimic, highlighting gaps in testing and areas for performance improvement. In Chapter 6, a framework for end-to-end testing is proposed, along with a testing strategy. This chapter also examines the causes of bottlenecks and suggests optimization strategies. Chapter 7 describes test structure and strategy, integration to CI/CD pipelines, implemented performance improvements, and the use of Google Lighthouse to measure page performance. Chapter 8 shows various bugs discovered by tests and verifies the impact of implemented performance improvements using Google Lighthouse.

Chapter 2

Internet of Things (IoT)

The Internet of Things is a term that is often used in information and communication technologies. It is a broad term and may be difficult to comprehend. So far, there has yet to be a universal consensus on its definition.

One of the first mentions of this term was in 1999 when Kevin Ashton used it to describe a system of real-world devices that can communicate and share information over the internet with minimal human interference [3]. He is often credited with coining the term in the context of radio-frequency identification (RFID) technology. The idea was to enable the devices to be uniquely identified. This concept laid the foundation for the broader vision of the IoT that we see today. Most definitions focus on physical objects as data sources and being part of IoT. Some consider living things like humans and animals part of it, too.

The Internet of Things may be broadly summarized as a network of interconnected objects embedded with sensors, software, and other processing technologies that are uniquely addressable. The network is based on standardized communication protocols that enable the exchange and sharing of data. This thesis will use this broad definition to avoid conflicts between more specific definitions.

Interest in IoT devices is growing as the number of active IoT endpoints is growing each year, reaching approximately 16.7 billion in 2023, and the number is forecasted to grow [47].

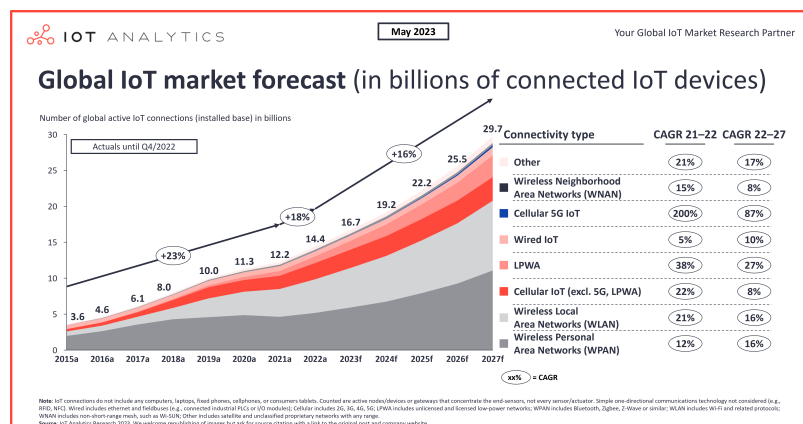


Figure 2.1: Global IoT market forecast [47]

2.1 Architecture

IoT technologies can potentially bring significant advantages to businesses in various sectors. Utilizing IoT applications enables the remote management of devices, leading to improved efficiency, enhanced security, and greater accessibility. Knowledge of IoT architecture may result in the more effective use of resources. Various researchers proposed different architectural models, as seen in Figure 2.2.

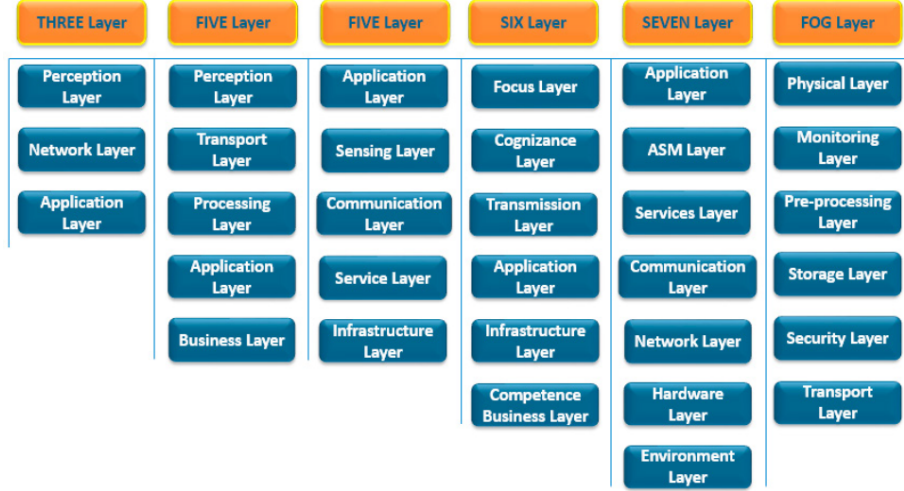


Figure 2.2: IoT architecture layers [24]

One of the most basic IoT architectures introduced is the Three Layer architecture, and other architectures expand it. The layers are described in various papers [37, 27, 24] as:

- The perception layer, sometimes referred to as the sensor layer, consists of smart devices, actuators, and sensors. It aims to connect objects to an IoT network to collect and extract information. Subsequently, the information is transmitted to the upper layer [27].
- Network layer – deals with data transmissions and processing using various communication technologies and protocols. It is sometimes referred to as the Transmission layer.
- Application layer – provides services to applications that also exist on this layer, like Smart Grid or Smart City [27].

Various models of Four Layer Architecture exist, with Service-Oriented Architecture (SoA) being one of the most widely used. It comprises the Perception, Network, Service, and Application layers [27]. The Perception and Application layers are sometimes referred to as the Sensing and Interfaces layers, respectively. The roles of the Perception and Network layers remain consistent with those in the Three Layer Architecture. The Service layer is responsible for creating and managing services required by users or applications [17]. The Application layer is designed to support a variety of applications [27].

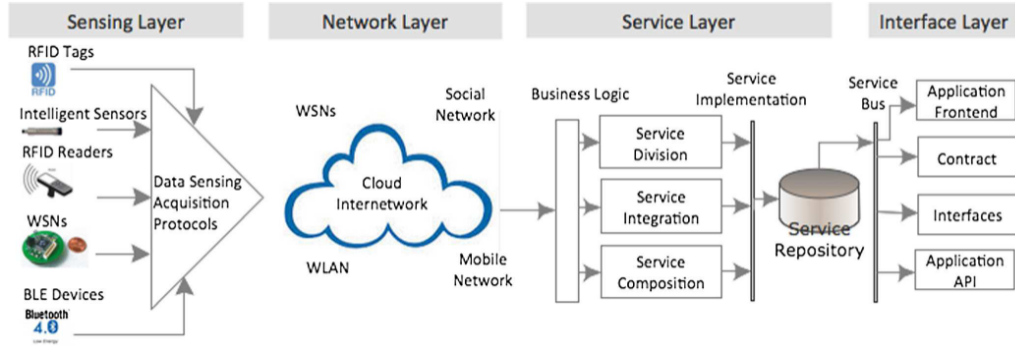


Figure 2.3: Service oriented Architecture [26]

The Five Layer Architecture expands on the implications of using data by introducing the Process and Business layers. Multiple sources [21, 24] define these layers as Perception, Transport, Process, Application, and Business. The Perception and Application layers retain the same purposes as in the Three-Layer Architecture. The Network layer from the Three Layer Architecture is subdivided into the Transport and Process layers. The Transport layer is responsible for transferring the gathered information between the Perception and Processing layers. The Process layer, often referred to as the Middleware layer, handles data processing and analysis, offering valuable insights. The new Business layer manages the entire IoT system and all its applications. It provides business models and live dashboards to facilitate business intelligence.

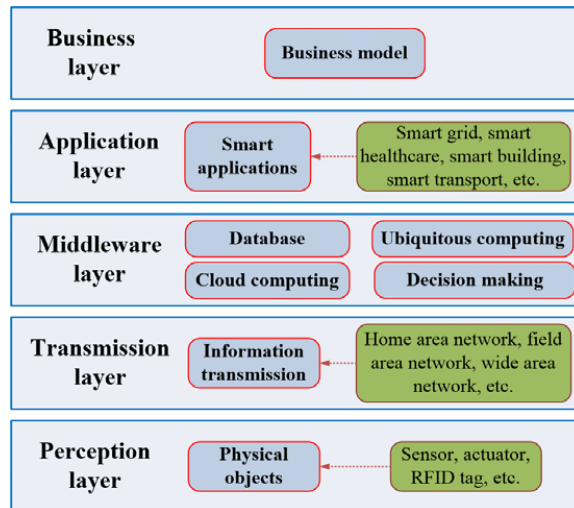


Figure 2.4: Five layer architecture [57]

Several research papers [24, 45, 31] propose distinct Six layer architectures. However, there is a lack of consensus on a standardized six-layer model, as each proposed architecture introduces unique features and emphasizes different aspects of IoT systems.

Among the various seven-layer architectures, one of the most popular is the CISCO Internet of Things Reference Model [9]. The model consists of the following layers:

- Physical Devices & Controllers – contains the sensors, devices, and machines.
- Connectivity – facilitates reliable communication between devices and networks.
- Edge (Fog) Computing – performs high-volume data analysis, evaluation, filtering, and transformation.
- Data Accumulation – converts event-based data to query-based processing. The converted data are stored in databases.
- Data Abstraction – reconciles multiple data formats from different sources, consolidates data into one place, and normalizes and indexes data for applications.
- Application – interprets data from the previous layer, controls applications, and performs business intelligence.
- Collaboration and Processes – involves people and business processes.

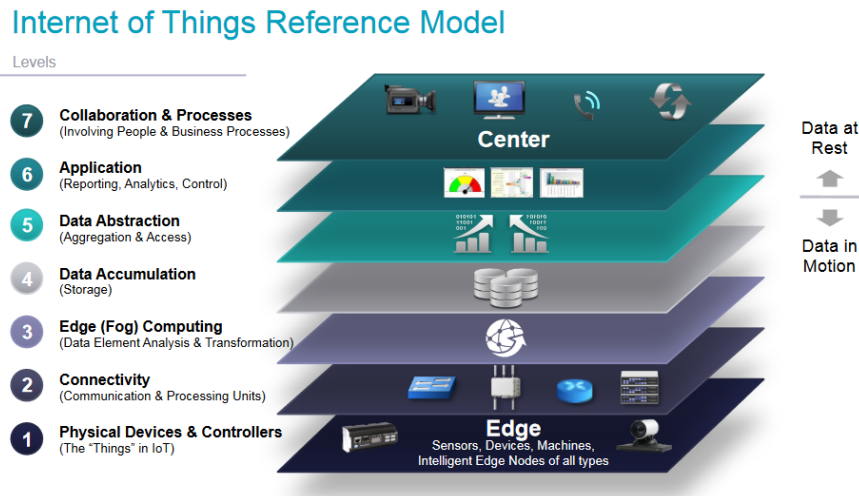


Figure 2.5: Cisco Internet of Things Reference Model [9]

2.2 IoT Devices

IoT devices are equipped with embedded sensors, actuators, processors, and transceivers. An actuator is a device that serves to change the environment, such as the temperature controller of an air conditioner. [45]. Sensors make a significant part of perception or physical layers in the aforementioned architectures. Sensors play a crucial role in the automation of various applications by measuring and processing the collected data to detect changes in physical things. Various sensor types exist. The classification of sensors can be

based on their specifications, conversion methods, the properties they measure, and the specific application areas they serve. Sensors have a wide array of applications in Smart Cities, including parking, noise handling, traffic, and waste management [43]. Chemical sensors can be used to detect forest fires and monitor air pollution. Humidity sensors are essential in greenhouse automation, food storage systems, and weather monitoring. Temperature sensors are used in climate control systems, weather monitoring, and industrial processes that require temperature regulation. Moisture sensors are used in agriculture to monitor soil conditions, helping farmers optimize irrigation and crop management [23, 40]. Motion sensors are commonly used in security systems, smart lighting, and building occupancy detection. Proximity sensors are used in touchless faucets, automatic doors, and robotics. Pressure sensors are used in tire pressure monitoring systems, weather stations, and industrial processes. These sensors can be placed on outside tanks and pipes to monitor water pressure to make leak detection faster and narrow down the location of the leak [43]. Gas sensors are crucial in environmental monitoring, industrial safety, and indoor air quality control. These sensors can be attached to the walls or containers at factories, mines, and industries where there is a danger of leakage [43]. Sound sensors are used in noise monitoring, acoustic quality assessment, and applications like smart cities and public safety. Ultrasonic sensors are employed in parking assistance, robotics, and object detection. Force sensors are used for medical monitoring, industrial equipment, and wearable technology [43, 45].

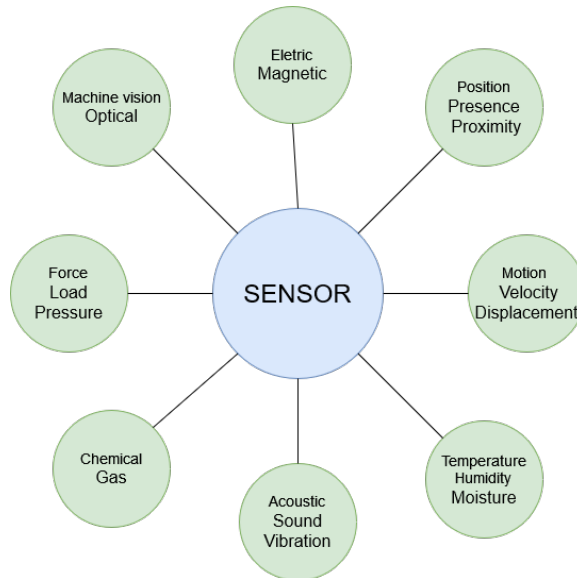


Figure 2.6: Smart sensor types

2.3 Technologies

Sensors collect a vast amount of data. This data forms the backbone of IoT systems and requires technologies that can connect and collect data from a wide array of sources and propagate them to the upper levels of the various layers of the aforementioned architectures. Many technologies can play this pivotal role in IoT. Technologies used for communication can be divided into wired and wireless. Wired communication can use Ethernet, RS485,

Modbus, and Mbus communication formats. Wired sensors are not portable and require additional cabling, which can be very costly when not using an existing power grid [49, 51]. This lack of mobility and potentially high installation cost often makes them inferior to wireless technologies. Therefore, this overview covers some of the most commonly used wireless communication technologies:

- Radio Frequency Identification (RFID) – technology for making objects uniquely identifiable via tags – transponders. Tags are divided into two categories: passive and active. Active tags provide a constant stream of information, enabling real-time tracking and monitoring, while passive RFID tags get activated in the presence of a specific RFID reader [14, 56].
- Wireless Sensor Network (WSN) – a bi-directional, wirelessly connected network of autonomous sensors equipped with sensing, processing, and communication capabilities [14]. The sensors are small, with limited processing and computing resources, and inexpensive compared to traditional sensors [58].
- LoWPAN – Low-power Wireless Personal Area Network (LoWPAN) is a wireless network comprising low-cost devices. It has a number of advantages, like small packet sizes, low power, and low bandwidth [50].
- 6LoWPAN – a protocol that was designed by combining IPv6 and LoWPAN. It offers various benefits, such as excellent connectivity, low energy consumption, and ad-hoc self-organization, and is, therefore, well-suited for the Internet of Things, in which many low-cost devices are included [27].
- LoRaWAN – Long Range Wide Area Network, is a low-power, wide-area networking (LPWAN) protocol designed for the Internet of Things. It enables long-range communication between low-power, battery-operated devices and a centralized network infrastructure [20].
- ZigBee – a wireless communication standard designed for low-power, short-range communication among devices. It operates on the IEEE 802.15.4 standard. It can support different topologies like mesh, star, and tree network topologies [1].

Many other wireless communication technologies like WiFi and Bluetooth are also used, but both have relatively high power consumption, making them suitable for local areas. Bluetooth Low Energy (BLE) addresses the high power consumption, which was introduced as a separate specification within the Bluetooth standard. Connected devices generate vast amounts of data that is transmitted over the internet. Numerous IoT devices necessitate substantial storage capacity and rapid processing capabilities to take immediate decision-making, data visualization, device management, and data analytics [38]. Cloud is the only suitable technology thanks to its centralized, scalable, and efficient infrastructure. It is well suited for managing and extracting value from the vast data connected devices generate. It also facilitates remote device management, analytics, and integration with various services.

Chapter 3

Smart Cities

In the contemporary world, nearly 55% of the world population lives in urban areas, and the percentage is expected to grow [54]. As cities grow, new problems arise, including traffic congestion, waste management, pollution, and parking allocation. Smart Cities can play a pivotal role in making cities more suitable for quality living. A Smart city is a term that lacks any generally accepted definitions, and many city concepts, including “Digital City,” “Wired City,” “Intelligent City,” and “Knowledgeable City,” overlap with the term [30, 10]. Each definition of a Smart city focuses on the term from a different point of view and considers different parts of the concept as a key factor.

Giffinger et al. [16] describe it as “A Smart City is a city well performing built on the ‘smart’ combination of endowments and activities of self-decisive, independent and aware citizens,” emphasizing that a smart city is built upon the active involvement of its citizens. It focuses on citizens’ empowerment and their role in making the city smart. The emphasis is on the people and their capacity to make informed decisions.

Caragliu et al. [2] define it as: “A city to be smart when investments in human and social capital and traditional (transport) and modern (ICT) communication infrastructure fuel sustainable economic growth and a high quality of life, with a wise management of natural resources, through participatory governance.” This definition highlights the importance of investing in human and social aspects and traditional and modern infrastructure. It emphasizes sustainable growth, quality of life, and participatory governance as key goals of a smart city.

Hall [18] describes it as: “A city that monitors and integrates conditions of all of its critical infrastructures, including roads, bridges, tunnels, rails, subways, airports, seaports, communications, water, power, even major buildings, can better optimize its resources, plan its preventive maintenance activities, and monitor security aspects while maximizing services to its citizens.” This definition places its primary focus on the technical and operational aspects of a smart city. It emphasizes the efficient monitoring and integration of critical infrastructure, resource optimization, maintenance, and security while providing better services to citizens.

Dameri [10] defines the term as “A smart city is a well-defined geographical area, in which high technologies such as ICT, logistics, energy production, and so on, cooperate to create benefits for citizens in terms of well-being, inclusion and participation, environmental quality, intelligent development; it is governed by a well-defined pool of subjects, able to state the rules and policy for the city government and development.” highlighting the technological cooperation among various domains and the importance of governance. It emphasizes the use of advanced technologies, well-being, environmental quality, and the

role of a defined group of governing bodies in shaping a smart city. However, they all focus on improving the quality of cities and life. This thesis will broadly define smart city as a concept that refers to the use of IoT and other information and communication technologies and data-driven solutions to enhance operational efficiency, sustainability, and overall quality of urban life.

3.1 Smart City Domains

To better understand what the Smart City term represents, it can be split into various components, dimensions, domains, or characteristics. Giffinger [16] identifies six characteristics: Smart Living, Smart Governance, Smart Economy, Smart Mobility, Smart Environment, and Smart People (Figure 3.1). However, it is important to note that not all these characteristics are universally present in every smart city. Furthermore, there can be overlap among these characteristics [11]. Atitallah et al. [4] also identify six characteristics, but their set differs from those previously mentioned. Companies involved in the implementation of smart cities, like Frost & Sullivan, and authors like Mohanty [30] and Bellini [6] consider eight dimensions. Some extend the model by Giffinger, while others introduce entirely new domains (Figure 3.2). Each city implements its smart project differently, focusing on a subset of domains. Consequently, this diversity leads to variations in the composition and characteristics of smart cities.

The increasing development of Internet of Things technologies plays a crucial role in shaping the landscape of contemporary smart cities and represents one of the key drivers for innovation and sustainable development. The primary objective of deploying and integrating IoT solutions is to allow smart cities to advance further, providing new capabilities and features while reducing human intervention.

Challenges come with the rapid spread of IoT smart city technologies and applications. For example, one of the challenges in this context is interoperability issues arising from diverse IoT protocols, formats, and frameworks. Achieving a higher level of interoperability among devices, applications, and services involves the reduction in costs for creating entirely new and distinct deployments of solutions, thereby allowing backward compatibility through the exploitation of older systems [6].

- Smart Infrastructure and Living – includes all components related to developing smarter city infrastructures (smart homes, smart buildings, and so on) as well as the management and improvement of public services, such as cultural activities, tourism and education. The Internet of Things enables the swift adoption of various features in smart buildings, including controlling air conditioning, managing rainwater drainage, implementing secure access systems, employing video surveillance, and issuing alerts for incidents like fires and gas leaks [6]. IoT sensors can be used to monitor the structural integrity of buildings, allowing repairs to be made before critical failures [43].
- Smart Governance – deals with the integration of information and communication technology into urban governance processes, with the goal of improving operational efficiency, effectiveness, and transparency in organizational management and service delivery. It promotes collaboration among city departments, external organizations, and citizens. For instance, citizens have the opportunity to actively engage in city governance activities and decision-making processes [7].

SMART ECONOMY (Competitiveness) ▪ Innovative spirit ▪ Entrepreneurship ▪ Economic image & trademarks ▪ Productivity ▪ Flexibility of labour market ▪ International embeddedness ▪ <i>Ability to transform</i>	SMART PEOPLE (Social and Human Capital) ▪ Level of qualification ▪ Affinity to life long learning ▪ Social and ethnic plurality ▪ Flexibility ▪ Creativity ▪ Cosmopolitanism/Open-mindedness ▪ Participation in public life
SMART GOVERNANCE (Participation) ▪ Participation in decision-making ▪ Public and social services ▪ Transparent governance ▪ <i>Political strategies & perspectives</i>	SMART MOBILITY (Transport and ICT) ▪ Local accessibility ▪ (Inter-)national accessibility ▪ Availability of ICT-infrastructure ▪ Sustainable, innovative and safe transport systems
SMART ENVIRONMENT (Natural resources) ▪ Attractivity of natural conditions ▪ Pollution ▪ Environmental protection ▪ Sustainable resource management	SMART LIVING (Quality of life) ▪ Cultural facilities ▪ Health conditions ▪ Individual safety ▪ Housing quality ▪ Education facilities ▪ Touristic attractivity ▪ Social cohesion

Figure 3.1: Six Smart City characteristics [16]

- Smart Mobility and Transportation – provides services that manage private and public traffic flows, dynamic traffic routing, and smart parking. IoT sensors and wireless connections enable the creation of self-driving cars and monitoring vehicle performance. By harnessing the data gathered from these vehicles, it becomes feasible to predict and manage traffic congestion on the most frequently used and crowded routes[29].
- Smart Economy – provides e-business and e-commerce (like online shipping) services to increase productivity and delivery. It also includes the concept of a sharing economy, where individuals or private companies provide services by utilizing their assets, often through peer-to-peer marketplaces. Additionally, there are peer-to-peer labor services, where citizens and stakeholders offer their skills and expertise for specific tasks [55]. Using mobiles and smartphones in daily transactions in Shenzhen makes cash and bank cards obsolete [6].

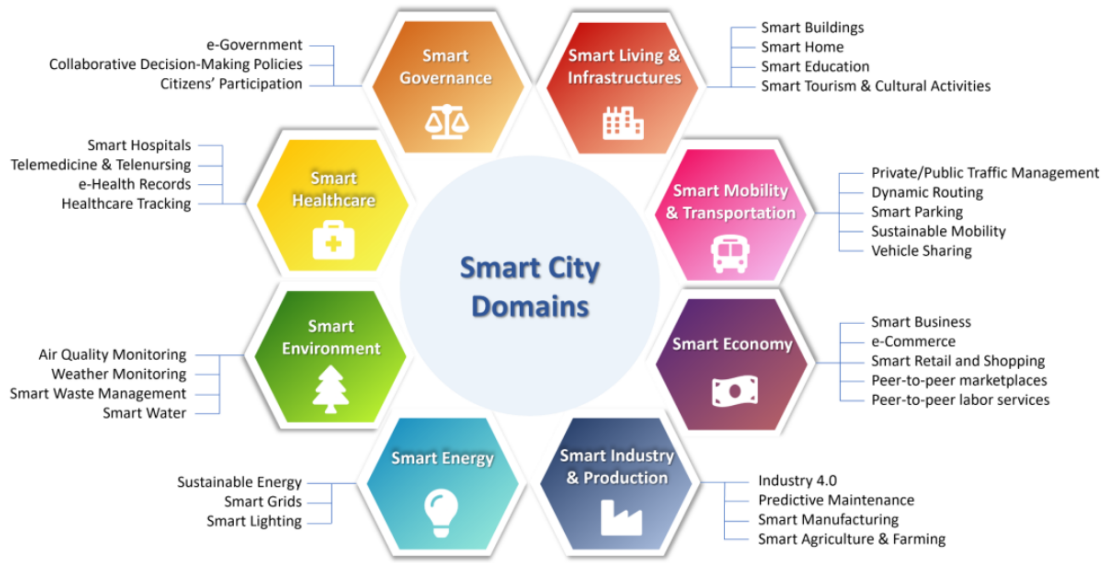


Figure 3.2: Eight Smart City domains [6]

- **Smart Industry and Production** – allows an innovative, more automated, less human-dependent productive environment. It eases supply chain tracking through sensor technologies like RFID or NFC. This enables the real-time collection and analysis of information for tracking shipments and evaluating product quality. This domain also includes agriculture, where IT devices can be used to improve irrigation efficiency, monitor crops, and detect diseases [6].
- **Smart Energy** – encompasses the integration of decentralized, renewable, and sustainable energy sources, along with their efficient distribution, with the goal of optimizing power consumption. Smart grids assist in balancing energy loads, enabling automatic transitions to alternative energy sources and predicting future energy demand while estimating power availability and price. Another important application of smart energy is leak monitoring systems [29].
- **Smart Environment** – includes environmental data collection, monitoring, and analysis to mitigate pollution, monitor water quality, manage water supply, and handle weather and climate events. Smart waste management is also included in this area since it has numerous environmental impacts. Waste production control is managed through smart waste bins equipped with sensors, allowing real-time analysis of their current capacity [32].
- **Smart Healthcare** – provides services for medical staff and patients. IoT technologies are used for remote monitoring, telemedicine, telenursing, patient identification, medical supplies, and instrument management. It can reduce healthcare costs by efficiently using medical equipment and sensors [6, 43].

3.2 Dashboard

One of the primary goals of smart cities is to provide its core services to the public efficiently. Municipalities often publish data about these services, employing visualization tools or APIs. Dashboards serve as a method to visually represent data, enabling the aggregation, visualization, and analysis of performance to support the sustainable development of smart cities¹. They offer valuable tools for assessing and facilitating urban infrastructure elements and services [13]. Dashboards support a broad spectrum of information needs but can also be tailored to specific purposes. It is useful to differentiate dashboards due to the various data and purposes they can be used for. These variations may require changes in visual design and functionality [15]. City dashboards utilize visual analytics, integrating interactive elements such as bar charts, graphs, maps, and 3D models to display information about city performance and structure. They showcase essential data about city resources, encompassing urban services, infrastructure, transportation, energy, security, healthcare, water supply, environmental factors, population, and numerous other elements, all accessible on a single screen, continuously updated as new data is input into the database. Furthermore, these dashboards offer interactive features like data selection, filtering, data queries, and zoom functionalities to enhance user engagement [13]. IoT environments add complexity to the features of current dashboards because of data complexity [42]. Stephen Few [15] believes one of the most useful ways to categorize a dashboard is by the type of business activity that it supports. He breaks them down into three roles: strategic, analytical, and operational.

- strategic – offer a quick overview to enable decision makers to monitor the health and opportunities of the business. These dashboards do not require real-time data; rather, they benefit from snapshots taken monthly, weekly, or daily.
- analytical – often demands greater context, such as rich comparisons and more extensive history. Like strategic dashboards, analytical dashboards also benefit from snapshots of data. Analytical dashboards should support data interactions that include delving deeper into the underlying information, enabling users not only to observe current situations but also to investigate their root causes.
- operational – displays dynamic data in real-time or near-real time, allowing users to monitor operations and make immediate decisions based on the most up-to-date information. The visual elements used in operational dashboards should maintain a high level of simplicity. In situations of urgency, where an immediate response is essential, the significance and the necessary actions must be easily and unmistakably understood.

To effectively visualize and present data in a user-friendly manner, key performance indicators (KPIs) play a crucial role by quickly offering precise insights, aiding decision-making, and comparing current performance against predefined targets necessary for achieving business goals. However, KPIs are not always well-known, and finding suitable KPIs for every business objective can be challenging. In such situations, it is common to refer to established KPI lists. In some cases of smart cities, choosing a KPI can be challenging due to new data sources and the need to meet the service requirements of city users [39].

There is no standardized set of key performance indicators (KPIs) or measurement framework for evaluating the real effectiveness of smart initiatives across all smart cities.

¹<https://data.brno.cz>

This lack of standardization makes it challenging to compare and assess their performance. Numerous frameworks for measuring the results and effects of smart city initiatives have been proposed. However, there are only a few cases where cities have effectively applied these measurement frameworks, or any others, to understand their impact on the quality of life in the city. Even when rankings use different methods, cities often seek to participate. Achieving a top rank in a highly reputed city ranking can significantly improve the international image of a city and, as a result, plays an important role in its marketing strategy [16].

Chapter 4

Software Testing

Ensuring flawless functionality and optimal performance of dashboards is crucial for providing users with reliable and timely insights, allowing them to make informed decisions. Software testing is a technique most often used for verifying and validating the quality of software [33]. Proper testing of applications can ensure these goals are met. Defining factors for successful and effective software testing projects involve careful selection and application of suitable testing methods, as well as choosing and utilizing the correct test automation tool and/or framework [53].

4.1 Testing Methods

Ensuring flawless functionality and optimal performance of dashboards is crucial for providing users with reliable and timely insights, allowing them to make informed decisions. Software testing is a technique most often used for verifying and validating the quality of software [33]. Proper testing of applications can ensure these goals are met. Defining factors for successful and effective software testing projects involve careful selection and application of suitable testing methods, as well as choosing and utilizing the correct test automation tool and/or framework [53].

4.2 Testing Methods

Many approaches to testing are available, depending on its purpose. Methods can be divided into two categories: functional and non-functional testing. Functional testing focuses on the processing results and evaluates whether the application meets the basic, minimum expectations of the user. Paspelava [36] divides testing into four levels (depicted in Figure 4.1):

- Unit tests – the primary purpose is to isolate and test individual units or components to ensure they function correctly before being integrated into the larger system.
- Integration tests – are performed after integrating two or more modules that interact with each other and then testing them as a unified group.
- System testing – examines fully integrated applications to evaluate the interactions between components and their collective behavior within the entire system. This comprehensive testing phase includes end-to-end (e2e) and regression testing. The

primary goal of end-to-end testing is to validate the overall behavior, including its functionality, reliability, performance, and security [8]. Regression testing is performed after a code update to verify that the update has not introduced new bugs and that the application continues to function correctly while ensuring that no previously resolved bugs have resurfaced.

- Acceptance testing – the primary focus is to verify that the application functions as anticipated, is free of any critical defects, and aligns with business requirements. This process includes testing the application with end-users to ensure it meets their expectations and requirements. Additionally, user feedback is collected, allowing critical issues to be quickly addressed before the product is released.

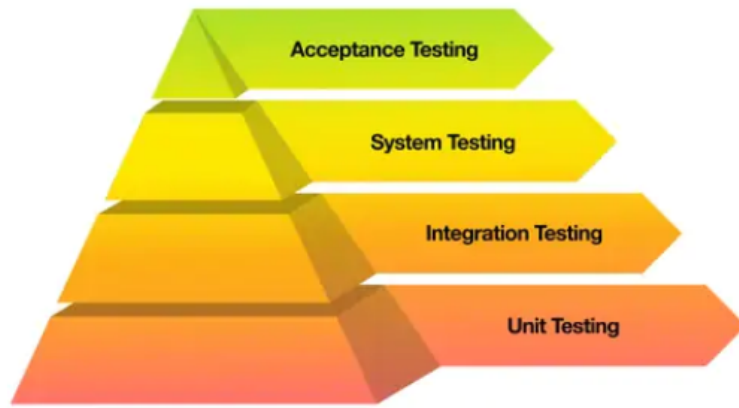


Figure 4.1: Functional testing pyramid [36]

Non-functional testing tests the performance of an application and ensures it is usable and reliable. Sethio [44] divides non-functional into the following types:

- Usability testing – evaluates a software application’s user-friendliness and overall user experience.
- Performance testing – tests the stability, scalability, and response time of an application with the goal of identifying and removing performance bottlenecks, thereby improving the overall user experience. Stress, endurance, and load testing are also considered part of performance testing.
- Compatibility testing – focuses on browser and operating system compatibility, including cross-platform compatibility across various devices such as desktops and mobile platforms.
- Security testing – evaluates the effectiveness of a system in safeguarding against both intentional and unintentional intrusions from internal and external sources. Additionally, it verifies that data is accessible only to authorized users.

Another way to categorize testing methods is by using black-box and white-box testing. White-box and black-box testing are often viewed as complementary approaches. Numerous researchers [33, 46, 28] emphasize covering both specification and code actions.

Black box testing is a technique conducted without familiarity with the internal operations of the application. This method solely assesses the essential components of the system and is minimally connected with the internal logical structure of the system [12]. Black box testing is applicable at system and acceptance testing levels. This technique offers only limited coverage, and without clear specifications, test cases are difficult to design.

On the other hand, White box testing is a technique that focuses on internal logic, structure, and coding. In white-box testing, a tester must have full knowledge of the source code [12]. White box testing is applicable at the integration, unit, and system levels of testing mentioned above. Some of the disadvantages of this technique include high cost, as a skilled tester is required to write and perform them. It is very difficult to find hidden errors and maintain this type of testing manually.

Automating the testing process in large-scale projects like Smart City can lead to long-term time and resource savings. Fragile tests are susceptible to breaking with slight modifications and necessitate maintenance, adding to the overall expenses. While automated tests excel in handling repetitive tasks, they are not as effective in dealing with non-standardized tasks. In contrast, manual testing is preferable for functions that lack repetition [41]. Therefore, it is crucial to thoughtfully consider test cases, determine what needs automated testing, and ensure the creation of reproducible, robust tests for optimal efficiency and reliability in software development.

4.3 Testing Tools and Frameworks

The Smart City platform analyzed in this thesis is built using Angular, a popular web application framework. This section explores Angular testing frameworks. Numerous frameworks for unit testing in Javascript are available. A unit testing framework requires assertion libraries, mocking libraries, and a test runner. While certain frameworks come with assertion, mocking libraries, and a test runner, others require configuration. The three most commonly used and popular frameworks for unit testing are:

- Mocha¹ – commonly used for testing node.js projects. Mocha lacks built-in assertion and mocking libraries. This allows testers to choose which libraries they want to use. This, however, requires additional time to set them up. Mocha has its test runner. Some popular assertion libraries used with Mocha are chai, should.js, expect.js, and better-assert. Mocking libraries include Sinon.js, testdouble.js, Jack, simple-mock, andnock.
- Jest² – primarily designed for React and developed by Facebook (nowadays Meta), which also maintains React. Beyond React, Jest is compatible with Babel, TypeScript, Node, Angular, and Vue projects. One distinctive feature of Jest is its inclusion of built-in assertion and mocking libraries, along with its dedicated test runner, which allows it to work out of the box. This simplifies the setup process but also limits customization. Jest runs tests in parallel, thereby reducing their execution time. Additionally, Jest supports snapshot testing and generating code coverage reports.
- Jasmine³ – a flexible framework with no dependencies, operating independently of the DOM or browsers. It comes with an assertion and mocking library but lacks

¹<https://mochajs.org/>

²<https://jestjs.io>

³<https://jasmine.github.io/>

any test runner. In the case of Angular projects, Karma, partly developed by the Angular team, is commonly used as a test runner. Notably, Jasmine paired with Karma is natively integrated into Angular projects, making them the most common tools for unit testing in Angular projects. Karma has been deprecated as of April 2023. In response, the Angular team introduced experimental support for Jest and Web Test Runner in Angular v16. Web Test Runner⁴ is a modern test runner and can replace Karma when its support becomes stable [35]. At the time of writing this thesis, the newest angular version is v17, and both Jest and Web Test Runner are still experimental and, therefore, discouraged from use in production.

Angular CLI automatically downloads Karma and Jasmine, allowing developers to write unit tests without additional setup. The configuration can be adjusted to use other testing frameworks. When choosing a testing framework, factors such as speed, customizability, learning curve, and setup time should be carefully considered.

Another important part of testing web applications is end-to-end testing, which ensures the application works correctly in real-world scenarios. Automating tests can save resources, as mentioned previously; however, in end-to-end testing, it is sometimes appropriate to do manual testing. For example, when a user interface is expected to change in the near future, any automation will need to be rewritten. Test automation takes longer than manual testing in the short term, so in case of a tight deadline on a project that requires testing to be done, manual testing can be the best solution according to Selenium documentation⁵. Commonly used frameworks for end-to-end testing of JavaScript-based applications include Selenium, Cypress, Playwright, and Protractor.

- Selenium WebDriver – a commonly used e2e testing framework. Employing a client-server architecture, it communicates commands to the browser through specialized browser drivers. Selenium is compatible with a range of popular browsers, including Google Chrome, Internet Explorer, Safari, Opera, Firefox, and Microsoft Edge. Operating on Windows, macOS, and Linux/Unix operating systems, Selenium supports various programming languages such as Ruby, Python, Java, C#, PHP, Perl, and JavaScript. Additionally, Selenium is able to concurrently operate two or more browsers, including support for multiple tabs. It can also run in headless browsers, improving its performance in CI/CD pipelines. However, Selenium does not inherently generate test reports. Instead, it relies on plugins like JUnit or TestNG for this functionality.
- Playwright – developed by Microsoft, is a versatile testing framework that supports modern rendering engines, including Chromium, WebKit, and Gecko. Offering compatibility with JavaScript, TypeScript, Python, C#, and Java programming languages, Playwright is adaptable across various operating systems like Windows, macOS, and Linux. It can be executed locally or on Continuous Integration (CI) platforms, either in headless mode or with a graphical interface. Similar to Selenium, Playwright supports multiple tabs and domains and has the capability to emulate mobile devices. Playwright has its built-in code generation tool, Playwright CodeGen, which records user interactions with the application and automatically generates corresponding test cases. Playwright also enables the execution of tests in parallel and includes built-in test reporters. Moreover, it offers the ability to mock and modify API requests and responses.

⁴<https://modern-web.dev/docs/test-runner/overview/>

⁵https://www.selenium.dev/documentation/test_practices/overview/

- WebDriverIO – a relatively new popular JavaScript automation framework built on top of Selenium WebDriver, utilizing its functionality and browser support. Notably, WebDriverIO goes beyond Selenium by also supporting the Chromium DevTools protocol. Furthermore, it can integrate with the mobile testing framework Appium. WebDriverIo can also do performance testing with Google Lighthouse. WebDriverIO can run in headless browsers, thereby improving performance in CI/CD pipelines.
- Protractor – a testing framework designed for Angular and AngularJS applications, built on top of WebDriverJS, which is the JavaScript binding for the Selenium WebDriver API, thereby inheriting its capabilities. By default, Protractor employs Jasmine as its test interface and supports asynchronous execution of tests. In September 2023, Protractor has reached end-of-life. Consequently, new users are discouraged from adopting it, and there is a recommended migration to alternative end-to-end testing solutions for ongoing and future projects.
- Cypress – a front-end testing framework distinguished by its exclusive support for JavaScript. It supports Chrome, Microsoft Edge, Firefox, and Electron, has experimental support for WebKit, and can run headlessly if the used browser supports it. Unlike some other testing frameworks, Cypress does not offer multi-tab support and is limited to controlling a single open browser at a time because it executes its commands inside a browser [41]. A notable feature of Cypress is its inclusion of Cypress Studio, a tool that facilitates test creation by recording user actions. Developers can control network requests and stub external dependencies. Additionally, Cypress allows API requests to be made, and responses can be validated and mocked. Cypress has built-in support for a variety of Continuous Integration/Continuous Deployment (CI/CD) tools. Cypress offers numerous official and community plugins, further enhancing its capabilities. For example, cypress-audit enables performance measuring using Google Lighthouse or cypress-web-vital to use Web Vitals.

When selecting an end-to-end framework, factors like browser and language support, support for CI/CD, multiple browser windows and tabs, speed and parallel test execution, setup time, reproducibility, and ease of maintenance of tests should be considered, depending on project needs. A survey conducted by the Angular team⁶ shows that 64% of enterprise customers use Cypress.

In addition to evaluating application functionality, application performance is a crucial metric. It involves assessing various aspects of performance, such as speed, scalability, stability, and responsiveness under different workloads. By conducting performance testing, organizations can identify bottlenecks, determine system capacity, and optimize performance to deliver a better user experience [25].

Various types of performance testing exist, including stress and load testing. These assess how an application manages sudden and substantial increases in workload or endures significant workload for extended periods. Popular tools for this type of testing include Apache JMeter, Gatling, BrowserStack, and LoadRunner.

Another performance testing category focuses on page loading and interactivity, incorporating critical metrics such as Largest Contentful Paint (LCP) for measuring how quickly the main content of a page becomes visible to users. First Input Delay (FID) is used to measure the responsiveness of a web page by quantifying the delay between a first user interaction with the page (like clicking a button or a link) and the browser response to

⁶<https://github.com/angular/protractor/issues/5502>

that interaction. This metric was, however, retired in 2024 and replaced by Interaction to Next Paint (INP)⁷. This new metric assesses page responsiveness using data from the Event Timing API. Cumulative Layout Shift (CLS) measures the visual stability of a web page by quantifying how much the content shifts around as the page loads. Google introduced these metrics as part of their Core Web Vitals set, aiding in diagnosing performance issues like slow server response times or render-blocking resources, directly influencing user experience. The Core Web Vitals can be measured using PageSpeed Insights⁸ or in Chrome browser using Web Vitals Chrome Extension.



Figure 4.2: Core Web Vitals⁹

Google Lighthouse is an open-source, automated tool for monitoring front-end software performance. It includes audits for SEO, progressive web apps, performance, and accessibility. It uses the Largest Contentful Paint and Cumulative Layout Shift metrics from Core Web Vitals, but also introduces new metrics. First Contentful Paint (FCP) for measuring how quickly the user can see anything on the screen. Speed Index that measures how quickly content is visually displayed during page load. Total Blocking Time is used to measure the total amount of time after First Contentful Paint (FCP) where the main thread was blocked for long enough (above 50 milliseconds) to prevent input responsiveness¹⁰. It also identifies opportunities to improve the performance. Lighthouse can be run from the command line in Chrome DevTools from a web UI PageSpeed Insights as a node module. It can be integrated with other automation tools like WebDriverIO and Cypress, as mentioned above. Additionally, it offers a Lighthouse CI¹¹ suite that enables Continuous Integration (CI).

Performance testing can pose greater challenges for Angular applications, given that Angular is commonly used for building single-page applications. The advantage of single-page applications is that they work inside the browser and do not require pages to be reloaded every time the user navigates to a new page. While this offers benefits in terms of user experience, it introduces an added layer of complexity for performance testing, as there is no loading of a new URL with each navigation.

⁷<https://web.dev/articles/vitals>

⁸<https://pagespeed.web.dev/>

⁹<https://web.dev/articles/vitals>

¹⁰<https://developer.chrome.com/docs/lighthouse/performance/>

¹¹<https://github.com/GoogleChrome/lighthouse-ci>

Chapter 5

Analysis

Selecting a testing framework is an important decision that can have long-lasting consequences. It is, therefore, crucial to project testing requirements into consideration. One of the most important features is cross-browser testing ability and support for major operating systems. Another important feature is the support for CI integration. It is also important to take into consideration requirements from Logimic, the company developing the application.

5.1 Logimic

Logimic¹ is a Czech company specializing in the development and provision of Asset Control and Data Acquisition (ACADA) systems for industry and municipalities. ACADA is a full-stack cloud platform with cloud and web-based systems. This platform is capable of processing data and presenting it through dashboards, enabling customers to remotely control, manage, and maintain their devices via a web browser or mobile application. The platform integrates a variety of end devices, sensors, and assets and is interoperable with third-party applications through APIs. It offers features such as defining Key Performance Indicators (KPIs), sending notifications, and generating reports. ACADA is divided into four layers. The testing and optimization will be performed on the ACADA web application developed by Logimic.

The application displays a dashboard presenting system groups (refer to Figure 5.1). This view is part of the first layer, characterized by extensive aggregation. Each group additionally displays the statuses of crucial Key Performance Indicators (KPIs) within the group. When a value from any of the devices in a group falls outside the specified range, an alert is triggered, and the color on the dashboard changes accordingly. For instance, Figure 5.1 on the left depicts a group of devices used for air quality monitoring. The Figure shows levels of Formaldehyde and CO₂ are within the set range, but the Volatile Organic Compounds (VOC) and temperature have a red indicator, indicating that they are not within the specified range.

¹<https://www.logimic.com/>

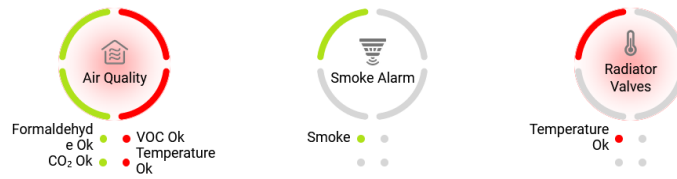


Figure 5.1: Visualization of system groups

When a user clicks on a group, they are directed to a second layer view that displays all Key Performance Indicators (KPIs) within that group (refer to Figure 5.2). In the center of the figure, a specific KPI, Volatile Organic Compounds (VOC), is shown, along with details indicating how many devices in the group have generated an alert regarding its state.

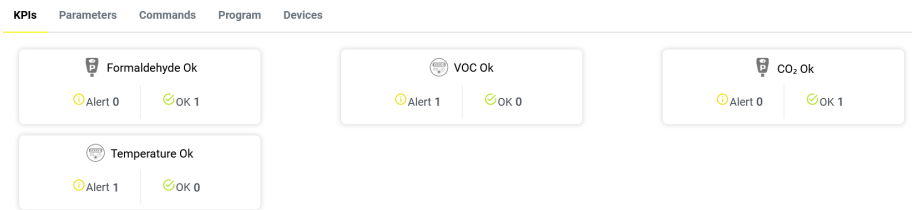


Figure 5.2: Visualization of group KPIs

After clicking on *Alert* or *OK* indicators, the user is taken to a third-layer view displaying a list of devices that triggered the alert and the date and time when the alerts were generated. Additionally, a map illustrating their respective locations is presented, as depicted in Figure 5.3.

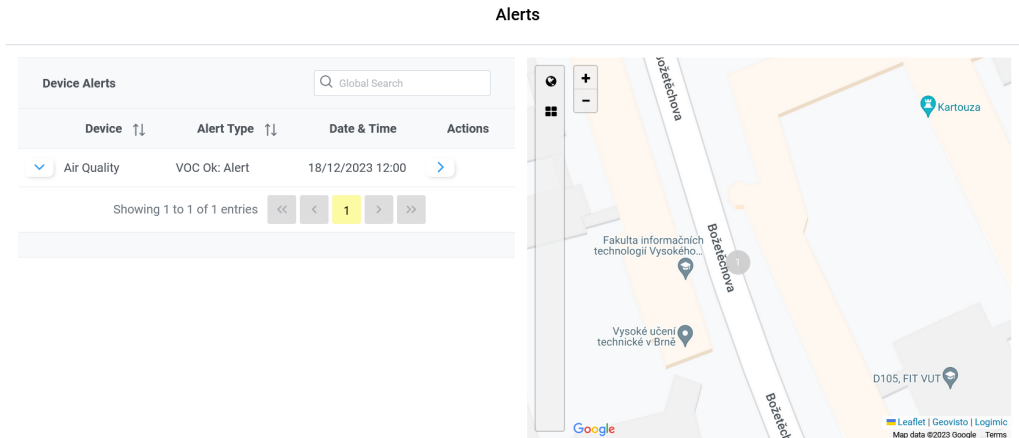


Figure 5.3: KPI alert detail

Users can explore the devices associated with a group by clicking on *Devices* while viewing the group details (refer to Figure 5.2). This action shows a view containing a card with multi-select dropdowns for selecting device models, individual devices belonging to the selected models, and device parameters. The selected devices appear in the Devices overview card (see Figure 5.4). Each device displayed in this card includes its details and an action button that opens a modal window, allowing users to navigate to the device details.

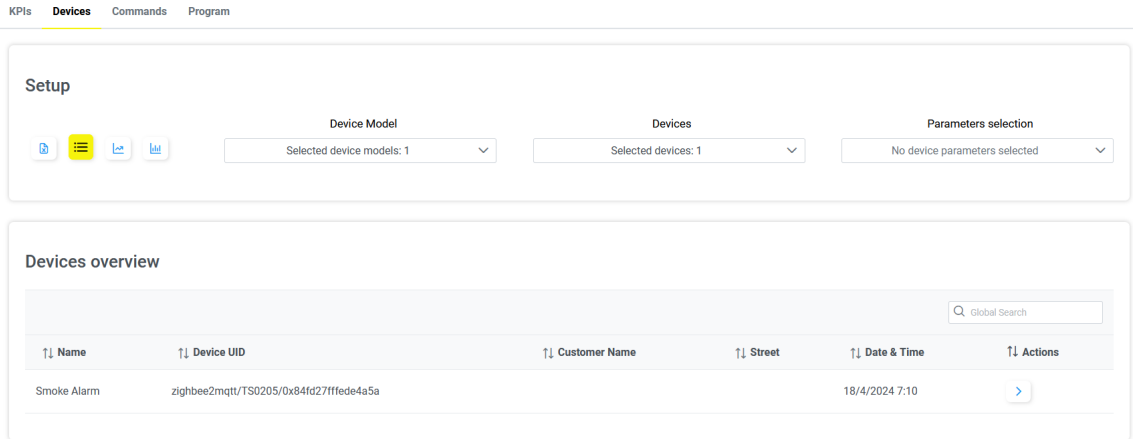


Figure 5.4: List of devices

After clicking on the mentioned button, the user is taken to a view in the lowest, fourth layer. This view showcases the device profile. This view includes information about active alerts, parameters, their current values, and any associated comments (refer to Figure 5.5). Notably, parameter values are continuously updated in real time.

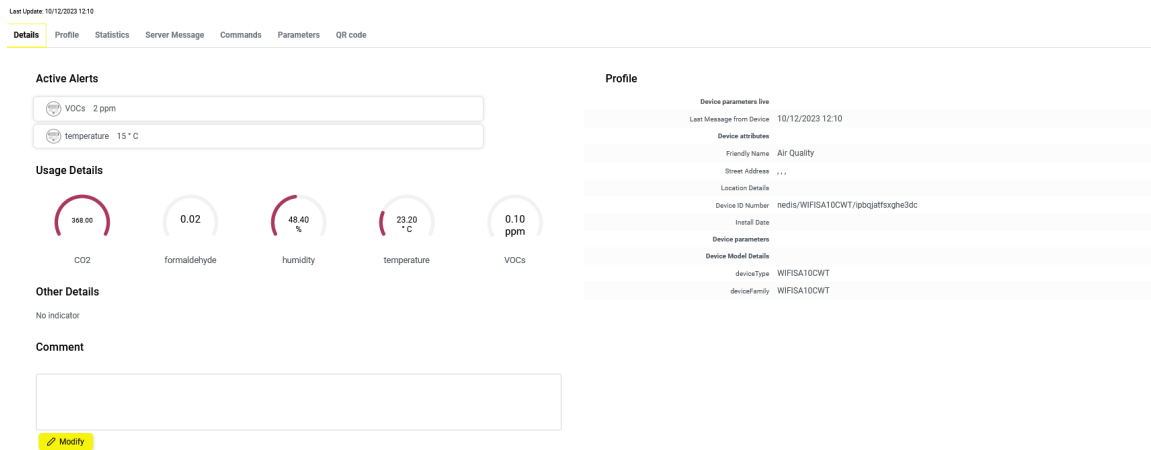


Figure 5.5: Device detail

The user is also able to view statistics of device parameters by clicking on Statistics in device detail (refer to Figure 5.5). Additionally, the user has the flexibility to choose the time frame for the displayed statistics. The available time options include a daily, weekly, or monthly overview, providing a quick snapshot of its performance over different intervals. Furthermore, users can also select a custom interval.

The described views represent the core interactions users commonly engage in within the application. End-to-end testing will be prioritized on these views and workflows, ensuring they function properly and no regression happens. Additionally, monitoring the performance of these application parts is crucial to ensure they maintain optimal speed and responsiveness.



Figure 5.6: Device Statistics

5.2 Current State of Testing at Logimic

The Angular CLI comes with the Jasmine testing framework and Karma test runner. These tools are currently employed for unit testing at Logimic. As mentioned earlier, although the Karma test runner is deprecated, no stable alternatives are available, and the Smart City project is currently running on version 13. In the future, consideration can be given to changing the test runner to, for example, Web Test Runner. However, this should only be pursued when the support for the test runner is stable and the project has been migrated to the appropriate Angular version. If changing the test runner is considered, one of the advantages is the time needed to adapt to the new configuration. The tests need not be modified; only the test runner configuration needs to be modified. Therefore, there is no need to migrate to other tools in the near future. The unit test coverage is minimal, allowing a swift potential transition to alternative testing frameworks such as Jest.

Protractor is currently used for end-to-end testing. However, Protractor reached its end of life in September 2023, so a new testing framework must be chosen. The existing Protractor tests are not extensive, making the transition to other frameworks quick. A lot of UI testing is done manually. Despite a significant amount of UI testing being carried out manually, it's essential to recognize that while manual testing can offer short-term cost reductions and prove useful in projects with tight deadlines, automated testing is essential for long-term efficiency, especially as the project scales [48].

Another important aspect of testing is performance testing. Currently, no performance testing, manual or automated, is being done, and no predefined metrics are set for monitoring. A common approach is using Google Lighthouse or a similar tool to run audits. This can be done manually using testing frameworks like Cypress, which were chosen to offer plugins that allow automation. Chrome DevTools can be used to analyze runtime performance. Google Lighthouse was chosen because it is a free, open-source tool that can be automated. Additionally, network and performance panels included in Chrome DevTools were used to analyze network activity, aiding in identifying potential bottlenecks.

Logimic currently uses API polling to periodically request updates from the server in both the first and fourth layer views. This practice leads to responses in the first-layer dashboard (refer to Figure 5.1) to contain data about all the groups, the number of service requests, and KPI alerts, even when some of the groups and their metrics in the response data are identical to those currently displayed. This leads to the unnecessary downloading of large data sets at regular intervals, negatively impacting application performance and responsiveness.

To address this issue, adopting WebSockets or webhooks is recommended. Both methods enable the server to notify clients of changes in data and deliver only the modified data to the client. This approach would significantly decrease the volume of data that needs to be downloaded, eliminating the need for periodic requests to the server to receive updates. Implementing WebSockets or webhooks is expected to result in an overall improvement in application performance. The main difference between webhooks and WebSockets lies in their communication capabilities. WebSockets allow bidirectional communication, while webhooks only provide one-way communication from server to client. For the specific task of updating data visualized by the dashboard, webhook functionality should prove sufficient.

Chapter 6

Implementation Design

Given the current manual approach to end-to-end testing, utilizing a framework for test automation is proposed. Cypress stands out as one of the most suitable and widely used frameworks for writing and automating end-to-end tests for Angular.

To meet the requirements made by Logimic, the chosen framework must not only offer the same functionality as Protractor but also provide support for video recording of test execution for efficient diagnosis of test failures. Additionally, it should have the ability to test APIs and mock API responses, as well as integrate with CI/CD pipelines.

Cypress supports the capture of snapshots and video recordings during test execution, with the flexibility to store them in a specified folder that can be customized through configuration. It offers options to only record failed tests and adjust video quality, thereby reducing the total size of generated videos during the run and saving resources.

In addition, Cypress provides built-in commands for making HTTP requests on behalf of the web application and asserting the API response. The framework supports spies, stubbing, and fixtures, allowing testers to mock functions, API requests, and responses. Fixtures, in particular, prove valuable when stubbing API responses, especially for large JSON objects, which are commonly returned by Logimic's API. Cypress also comes with an interactive user interface, enabling real-time, step-by-step viewing of tests [41].

Although Cypress exclusively supports JavaScript, its compatibility is not an issue, as Angular is a TypeScript-based framework. Since both TypeScript and JavaScript are languages used by developers at Logimic, the learning curve for Cypress is minimized. Cypress is designed to work “out of the box”, making the installation and setup process fast and easy. In my view, Cypress stands out as the most fitting framework for Logimic, and I highly recommend its use.

Tests written in Cypress will be able to be run locally by testers on their machines but will also be integrated into CI/CD pipelines. Logimic uses GitHub Actions¹ CI/CD pipeline for testing. GitHub provides a limited number of free minutes for GitHub-hosted runners, and beyond that allocation, every additional minute used incurs a cost. Given the limited free minutes offered by GitHub for runners and the resource-intensive nature of end-to-end tests, running these tests frequently on GitHub-hosted runners could lead to substantial costs. GitHub Actions also provides the option to run workflows locally, eliminating the expenses associated with using GitHub-hosted runners. I strongly recommend testing the GitHub workflow locally to reduce costs to only the maintenance of the device acting as the test runner. Another advantage of GitHub Actions is displaying the workflow run status on

¹<https://github.com/features/actions>

the branch or pull request, making the monitoring easier and more organized. Logimic also uses AWS services, providing the option to integrate Cypress into AWS CodeBuild². Similar to GitHub Actions, CodeBuild offers a limited amount of free minutes, with additional minutes incurring costs. While AWS CodeBuild supports local testing of builds, it lacks the functionality to run pipeline tests on a local device. Consequently, the only available option is to utilize Amazon-hosted runners, making this approach more expensive compared to GitHub Actions. This cost consideration makes GitHub Actions a preferable choice over AWS CodeBuild. Both GitHub Actions and AWS CodeBuild can be used together, whereas GitHub Actions, for example, can trigger tests on AWS CodeBuild when changes are pushed to a branch. Regardless of which one is chosen, Cypress integrates with both, offering flexibility in the choice of CI/CD solutions.

Configuring when the tests will be run is a crucial aspect of the testing strategy. Running tests after every push on any branch could consume considerable time, significantly slowing down the workflow. Conversely, running tests exclusively during production releases might lead to the late discovery of issues, making debugging and fixing more challenging and time-consuming, prolonging the release cycle. A recommended approach is to run tests when merging new features or updates to a stable branch. The proposed workflow can be seen in Figure 6.1. This ensures that the introduction of new features or updates is tested properly to prevent regression. Implementing tests at this stage allows early identification of issues in the development process, leading to more efficient workflow. Consequently, Cypress was chosen as the framework for implementing all end-to-end testing.

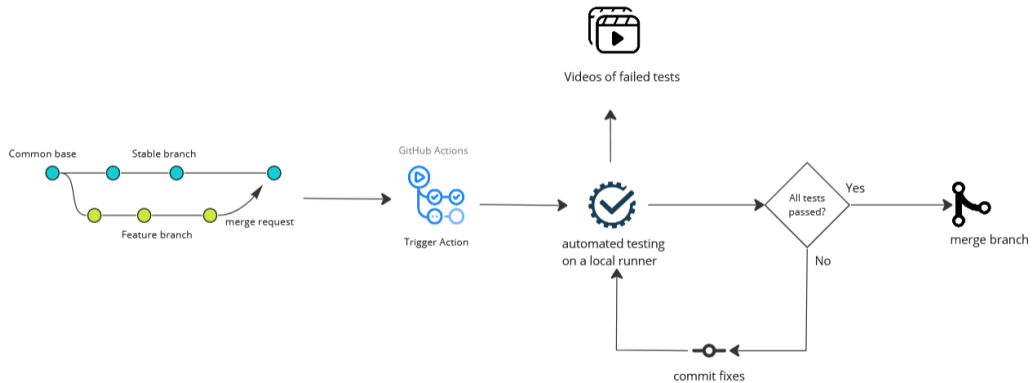


Figure 6.1: Proposed workflow

6.1 End-to-end Tests

Cypress tests are organized within the `e2e` subfolder of the `cypress` directory, located in the project root. The tests will be categorized into folders dedicated to component, API, and UI testing. Cypress allows testing of standalone components and provides the capability to pass inputs and outputs to components while utilizing spies to test event handlers. This allows Cypress to test the functionality, styles, and appearance of components. In the context of Logimic, GUI is built using components, identified by names starting with `lgmc-g`. For example, `lgmc-g-navbar` represents a component displaying a navigation bar, while

²<https://aws.amazon.com/codebuild/>

`lgmc-g-kpi-indicator` shows the state and value of a KPI. I recommend testing `lgmc-g` components to ensure that each UI component is working as expected before proceeding to launch API and end-to-end tests. This step ensures a comprehensive examination of component behavior and appearance in isolation. As was mentioned in Section 4.3, Cypress enables the execution of API requests on behalf of the application and the validation of the corresponding responses. These tests are used to verify the functionality and reliability of the API with which the application interacts. Users most often interact with views from the first to the fourth layer, as described in the Analysis (Section 5.1); the UI tests will primarily focus on these layers. This focus ensures that the layouts in the most frequently used views are thoroughly examined to guarantee their correct functionality.

In the first layer (dashboard with system groups), the emphasis will be on testing the navigation to lower layers. This involves verifying the functionality of clicking on groups to ensure that the correct group details are displayed. Additionally, the tests will validate that the Key Performance Indicators (KPIs) and their states align with those initially presented in the first layer.

In the second layer (group detail), tests will verify if each KPI corresponds to the correct number of devices and whether this count aligns with the number of devices displayed in the *Devices* tab. To test navigation to lower layers, the tests will include clicking on *Alert* or *OK* KPI indicators, leading to the KPI alerts view in the third layer. Here, the objective is to confirm that the number of records in the Device alerts table matches the count indicated by the KPI indicator in the second layer. Additional navigation tests will be conducted in the *Devices* tab to ensure transitions to the lowest layer. In this lowest layer view, verification will be conducted to confirm that the device name and ID match those presented in the second layer.

In the third layer (KPI alerts), the tests will validate that alert details are displayed following the click on an alert. The tests will verify filtering through the search bar works correctly. Additionally, they will verify that clicking on an action button generates a modal and that selecting *More Details* navigates to the device detail view.

Moving to the fourth, lowest layer (Device detail), the tests will focus on the statistics tab. Specifically, they will verify that selecting fields appropriately adds them to both the graph and legend. The tests will also focus on ensuring that all options for selecting the level of detail work correctly.

6.2 Performance

As established in Analysis (see Section 5.2), both WebSockets and webhooks have the potential to improve app performance. Both can be used to send data containing changes (for instance, sending data for only the two groups that require updates instead of all) and eliminate the need for clients to periodically request new data. Currently, the retrieval of the `AccessToken` from AWS Cognito³ triggers parallel loading of user-related data, including user, system, user forms, current system, and avatar, each through separate queries (depicted in Figure 6.2). This process can be optimized by reducing those to a single query, where all the needed user information is returned in exchange for the AWS token in a nested structure. Some images are currently stored in a database in base64 format, and these images lack caching, requiring the user to load the image after reloading. A more

³<https://aws.amazon.com/cognito/>

efficient solution involves using Amazon S3 (Simple Storage Service)⁴ to store images. By doing so, HTTP headers received from S3 can be configured to instruct the client to cache the received image. Additionally, the use of Content Delivery Networks (CDNs), such as Amazon CloudFront⁵, can further reduce server response times by distributing content to cache or edge servers located close to user, improving overall performance [34].

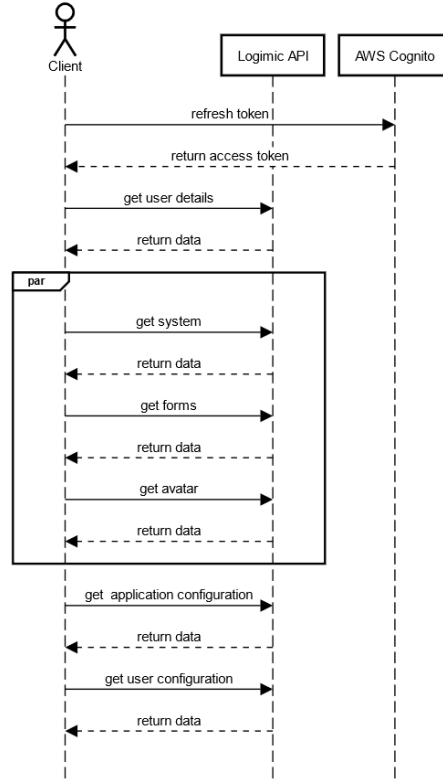


Figure 6.2: Current network flow

Another recommendation is to consider using the *.webp* and *.avif* image formats, as they provide superior compression compared to the currently utilized *.jpg* and *.png* formats. Adopting these formats can reduce the amount of data that needs to be downloaded, contributing to improved efficiency in data transfer and shorter loading times. It was also discovered that Angular bundles are not served compressed. Logimic is already using AWS CloudFront, so enabling compression and caching would improve the performance by further reducing the amount of data that needs to be downloaded.

Cypress can be used to generate Google Lighthouse audits. These audits serve to evaluate metrics such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), and Speed Index, which collectively measure the time it takes for the page to load. To ensure performance consistency, thresholds can be established for each metric, causing tests to fail if values fall below the set thresholds. As shown in Figure 4.2, it is recommended for sites and applications to achieve a Largest Contentful Paint of 2.5 seconds or less. The suggested targets for First Contentful Paint and Speed Index are 1.8 seconds and 3.4 seconds, respectively. The current evaluation of the home page (refer to Figure 6.3)

⁴<https://aws.amazon.com/s3/>

⁵<https://aws.amazon.com/cloudfront/>

shows that First Contentful Paint (22.8 seconds), Largest Contentful Paint (24.9 seconds), Speed Index (22.8 seconds), and Total Blocking Time (1720 milliseconds) are way higher than recommended values. Logimic should aim to approximate these recommended values. However, due to the fact that single-page applications have longer initial load times, which impact FCP and LCP values [19], reaching the recommended values can be very challenging. The main purpose of automating the audit generation and setting thresholds is to prevent regression. Initial thresholds should be set approximately to current values, and updates should be made as performance enhancements are introduced during development.

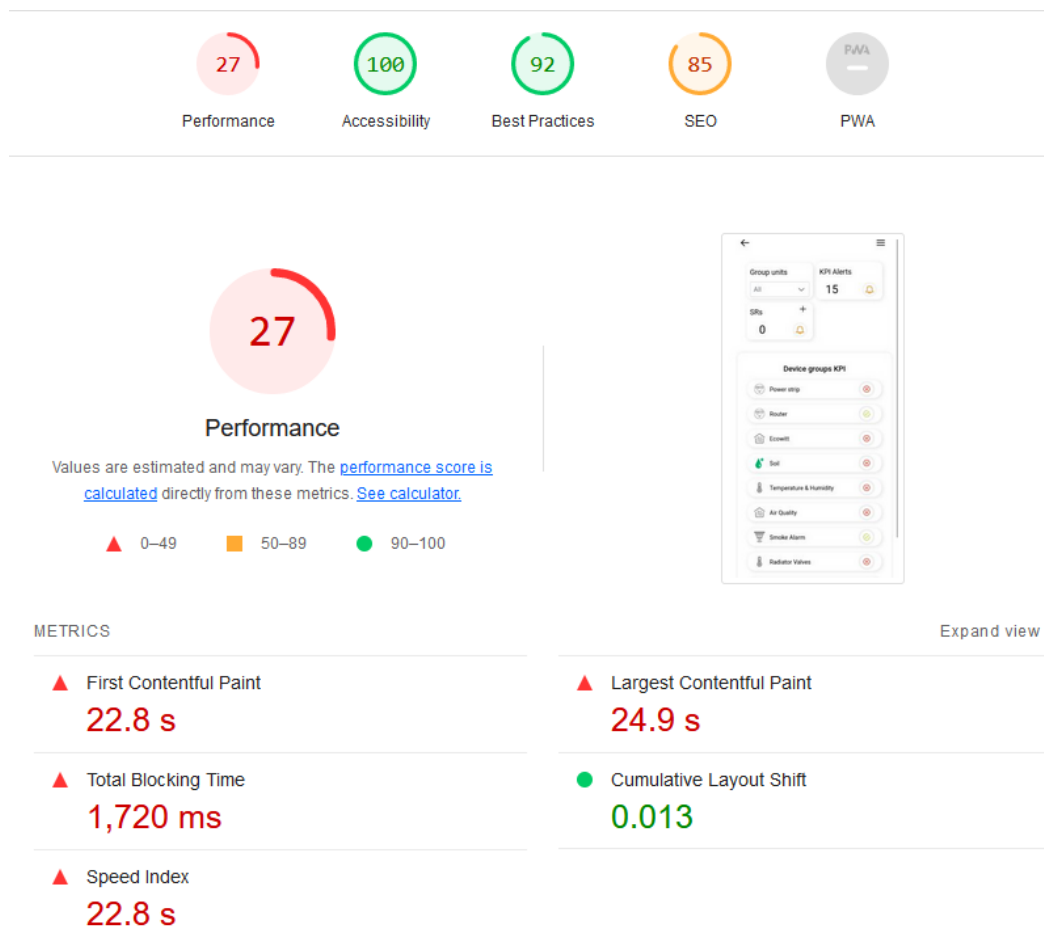


Figure 6.3: Google Lighthouse report of L1 dashboard on Windows before any performance improvements were implemented

Chapter 7

Implementation

Cypress was chosen as the framework for writing end-to-end tests, following the end of life of Protractor in September 2023, which was previously used for end-to-end testing. Given that only a single test was previously implemented, there was no necessity to migrate existing tests from Protractor to Cypress. The new tests written as a part of this thesis target the pages outlined in Chapter 5.

In addition to end-to-end testing, Cypress can also test individual components. However, Logimic currently uses Jasmine for this purpose. Prior to the commencement of this thesis, Logimic had conducted little to no end-to-end testing. Therefore, the focus of this thesis is not only on implementing end-to-end tests but also on establishing standards for test structure and workflow and improving performance. This, combined with the complexity of building isolated components for testing, resulted in component tests not being included in this thesis.

As described in Chapter 5, ACADA is structured into four layers, each with a separate test file in the `e2e` directory. Test files utilize the `describe()` function to group tests into suits (refer to Figure 7.1). Test suits help with test organization, for example, using one test suite for one page. Within each `describe` block, the first code block is a `beforeEach()` hook¹, used to execute a custom command named `login()` before each test, ensuring the platform is logged in and in the desired state before test execution. Test cases are then defined using the `it()` function, each focusing on a specific aspect or functionality of the application.

The purpose of each test case and the commands they utilize to achieve it should be carefully considered in order to avoid dependencies between test cases. For instance, a test case verifying the functionality of a button within a card should not rely on the successful creation of that card by a preceding test. By carefully planning and structuring test cases based on their intended functionality and command usage, dependencies can be minimized, resulting in more robust and reliable test suites.

ACADA renders pages on desktop and mobile differently. This often means changes in layout and use of different components, necessitating the need to test them separately. Cypress allows testing on any screen size thanks to the `cy.viewport()` command. This command was utilized in the thesis to test desktop resolutions and mobile devices. Desktop and mobile tests can be divided into different test suites, allowing control over which test suites are run using environment variables.

¹<https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests#Hooks>

A custom Chai assertion by Gleb Bahmutov² was used to test overflow and clipping of texts and elements. Leveraging this assertion alongside the capability to adjust the viewport, the tests ensure that the application is accurately rendered across various screen sizes. For instance, on devices with small resolutions, such as phones with limited screen sizes, text from buttons or cards may overflow, causing it to extend beyond the boundaries of the element. When users encounter such overflow issues, it can lead to confusion or discourage further usage of the app.

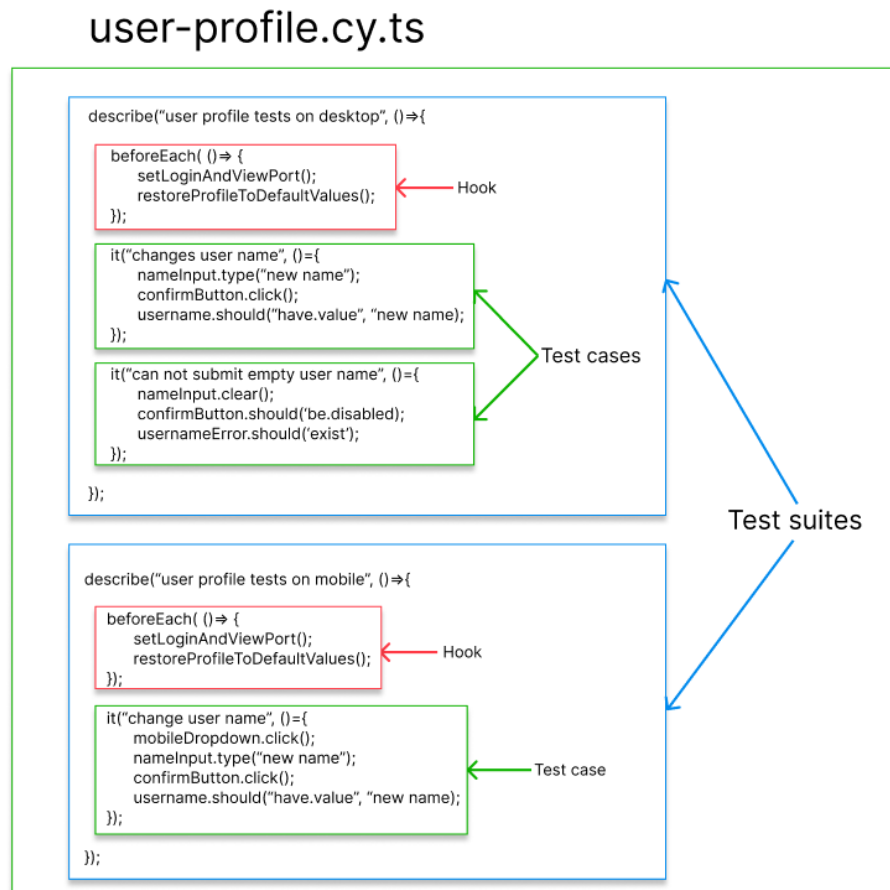


Figure 7.1: Test structure with pseudo-code

Figure 7.1 illustrates an example test file containing two test suites: one for desktop resolution tests and another for mobile resolution tests. Each suite employs a hook to log into the app and set the desired viewport, along with a function that resets the profile to its default state (without this reset, a test could, for example, assume that the user does not have a title and then add one, however, if any previous test already modified the title, the test would fail). Individual test cases interact with page elements to page functionality. Some tests focus on common interactions that users would do (changing user name), and others focus on edge cases (trying to submit an empty user name).

²code source: <https://glebbahmutov.com/cypress-examples/recipes/overflow.html#button-with-overflowing-text>

Cypress provides an API for creating custom commands³ and overriding existing ones, improving code readability and reusability. It is important to create custom commands only when they are used in multiple test files. If a sequence of commands is repeated within individual test cases in a single test file, extracting them into a function within the same file should be considered instead. Custom commands, such as `getDataCy()` for selecting elements using the `data-cy` attribute, align with best practices⁴, isolating elements from CSS and JavaScript changes. This approach proved beneficial, especially as some tested pages underwent UI modifications during the thesis writing process.

Logimic uses PrimeNG⁵ UI component library in ACADA. As a third-party library, its code does not include `data-cy` attributes, making it difficult to select its elements. This results in the chaining of multiple commands instead of using a single `getDataCy()` custom command, negatively impacting code readability. Additionally, this reliance on chaining commands increases susceptibility to changes, particularly when modifications are made to the internal implementation of the library. For instance, upgrading to a newer library version may result in structural changes that disrupt the existing command chains. This is especially problematic for the PrimeNG library, which is used extensively throughout the application. Due to the extensive use of the PrimeNG library, it is advisable to use custom commands only for components used across multiple pages.

Cypress wipes all cookies and session data after every test to ensure test isolation, resulting in a time-consuming login process before each test case. The login process consists of filling out the login form and waiting for a token from AWS Cognito. However, Cypress provides a command `session()`⁶ for caching and restoring cookies and session data, effectively reducing test execution time. This feature is leveraged in `login()` custom command, eliminating the need to fill out the login form and wait for the token before each test. As a result, test execution time is reduced by 36% on average. This optimization holds significant importance within CI/CD pipelines, as it reduces the time developers must wait for tests to complete, particularly when merging branches or during other automated processes.

In addition to leveraging custom commands in Cypress for improved test structure and workflow, integrating Cypress tests into CI/CD pipelines further enhances the testing process. These pipelines automate testing processes, eliminating the need for developers to launch them manually. The pipeline can now be run manually but is ready to be automated and adjusted to Logimic needs.

Sensitive information like login credentials, especially passwords, should not be included in test files [5], posing security risks as it exposes the credentials in plain text, making them vulnerable to unauthorized access. They can instead be passed as environment variables. For example, locally, it can be done using `cypress.env.json` file. Another way is to pass them from the command line using `-env` argument, for example, `npx cypress run -env password=secretpwd`. For CI/CD pipelines, in this case, GitHub Actions, secrets can be passed as environment variables. They must begin with `CYPRESS_*` or `cypress_*` to be parsed by Cypress. This is utilized primarily for setting the username and password by setting the `CYPRESS_username` and `CYPRESS_password` variables.

It is important to note that application performance may fluctuate due to variations in device hardware and network connectivity, potentially leading to test failures caused by timeouts on specific devices. This is an inherent flaw of Google Lighthouse and has to be

³<https://docs.cypress.io/api/cypress-api/custom-commands>

⁴<https://docs.cypress.io/guides/references/best-practices>

⁵<https://primeng.org/>

⁶<https://docs.cypress.io/api/commands/session>

counted. For example, the performance score on a high-end computer was 56 out of 100, on a low-end computer 36, and on Raspberry Pi 27, confirming the fluctuations. Using a test runner from a pipeline can serve as a reference point for evaluating application performance. In this thesis, CI/CD pipelines for testing are implemented using GitHub Actions. Logimic uses the AWS CodeBuild pipeline for building deployments, with GitHub Actions being run before deployment. GitHub Actions were chosen for its versatility and cost efficiency, particularly its ability to support self-hosted test runners. While first-party or third-party test runners, such as those provided by AWS or GitHub Actions, offer convenience and scalability, they also come with certain drawbacks compared to using a local test runner. First or third-party test runners can introduce dependencies on external services, which may result in downtime or service interruptions that are beyond Logimic control. They may also incur higher costs compared to self-hosted alternatives. GitHub offers 2000 free minutes per month on its test runner, followed by per-minute rates. Cloud-based test runners typically operate on a subscription or pay-per-use model, where users are billed based on test duration, resource usage, or number of tests run. These costs can quickly accumulate, especially for projects with frequent test runs or extensive test suites. On the other hand, self-hosted test runners may offer a more cost-effective solution, as they leverage existing infrastructure and resources without incurring additional fees. By utilizing local servers or dedicated hardware, Logimic can minimize recurring expenses associated with cloud-based services. To ensure smooth operation of the Angular application (ACADA) on devices serving as local test runners, these devices must possess adequate hardware capabilities. Logimic already utilizes Raspberry Pis for testing code on devices with ARM processors, making them suitable candidates for serving as test runners. Leveraging existing hardware, such as Raspberry Pis, for testing purposes can further reduce costs and provide greater control over testing environments. In this instance, a Raspberry Pi was used as a self-hosted test runner to run the test pipeline. When configuring Raspberry Pi, dependencies like Node.js and Chromium web browsers must be installed first. Following the official GitHub documentation⁷, the setup process involves adding the Raspberry Pi as a self-hosted runner. The CI/CD pipeline configuration includes steps for installing dependencies, building the application, and running Cypress tests. Additionally, it utilizes GitHub secrets such as Personal Access Token to access the app repository and pull submodules, as well as the username and password for ACADA, which are passed to Cypress during the testing process.

Following test execution, Cypress logs every test run, and GitHub stores these logs and artifacts. Developers can access these logs using the GitHub Action job details. Additionally, GitHub provides a build summary detailing the number of tests that passed, failed, or skipped and the overall build duration (see Figure 7.2).

While the logs indicate which test cases were successful and which failed, providing error messages and screenshots for failed tests, it is important to note that the information provided may not always be sufficient, especially if the error was caused by a previous command (see Figure 7.3). Video recordings help mitigate this issue by allowing testers to view the course of the test. Video recording can be easily enabled by following documentation⁸. Cypress also offers features for video compression and automatic deletion of videos from successful test runs. However, Cypress currently lacks the option to customize when a new recording begins. It records all tests in a test file into a single video, which can be

⁷<https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/adding-self-hosted-runners>

⁸<https://docs.cypress.io/guides/screenshots-and-videos>

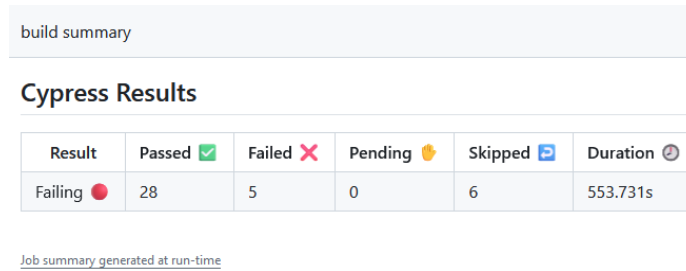


Figure 7.2: A screenshot of a GitHub Action run

cumbersome for files with many test cases, making it harder for testers to quickly locate specific failures. Due to the fact that GitHub Actions run the tests on a local runner, the resulting videos are also stored at by the runner, making it difficult to access them. A solution could be to use GitHub Actions to upload the videos to an AWS S3 bucket. Another

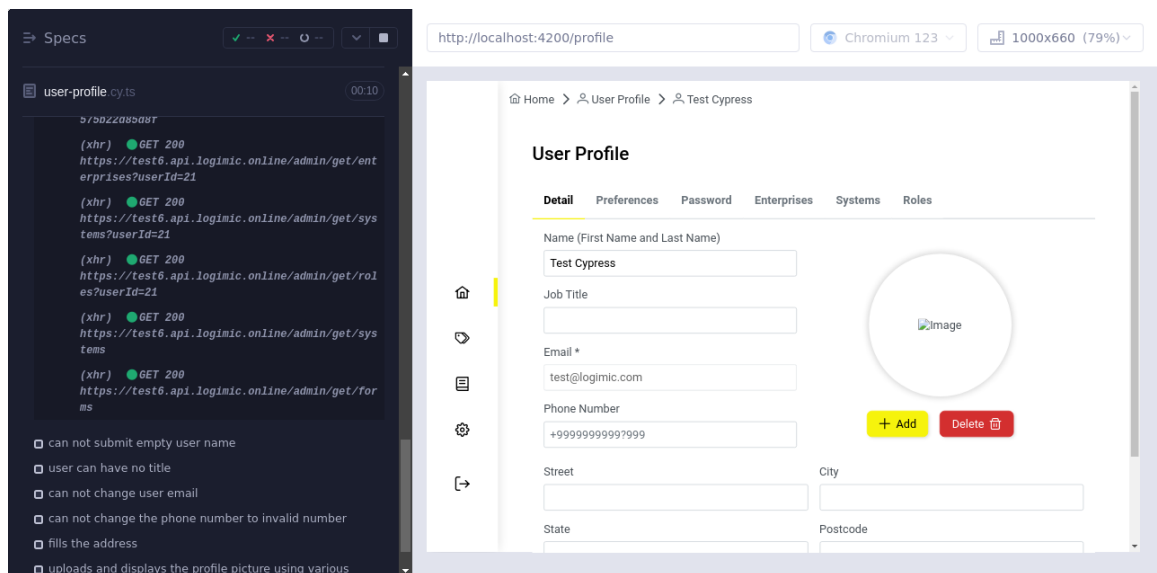


Figure 7.3: Screenshot of a failed test

solution is to use Cypress Cloud, developed by Cypress.io, the company behind the Cypress testing framework. It offers a suite of features to enhance the testing workflow, including parallelization, load balancing, flake detection, dashboards, insights, and uploading video recordings to a cloud, among others. Cypress.io offers a tiered pricing model for Cypress Cloud, starting with a free tier. However, Cypress Cloud was not tried, and Logimic is not fully committed to using it until it is properly investigated and tried. As an alternative, there is Sorry Cypress⁹ which offers most of the features of Cypress Cloud and can be self-hosted, making it a preferable option for Logimic. However, Cypress.io has blocked Sorry Cypress from version 13 onwards due to intellectual property concerns¹⁰. This thesis uses Cypress version 13.6.4, rendering Sorry Cypress unusable. Logimic decided not to utilize either of the aforementioned solutions for now.

⁹<https://sorry-cypress.dev/>

¹⁰<https://www.cypress.io/blog/2023/11/07/update-defense-intellectual-property>

In addition to the insights provided by test logs, screenshots, and videos, comprehensive performance reports using Google Lighthouse are also generated. Utilizing the NPM package `@cypress-audit/lighthouse`¹¹ to generate Google Lighthouse performance reports. These reports proved invaluable for identifying opportunities for performance improvements and verifying the impact of proposed performance improvements. The plugin enables the execution of Lighthouse audits directly within Cypress test suites, facilitating navigation to desired pages and running audits. Additionally, it offers the flexibility to set thresholds for each test or globally, enabling the detection of decreasing performance. However, limitations arise as not all pages have unique URLs, rendering some inaccessible for Lighthouse audits, thereby restricting coverage. Lighthouse conducts audits covering various aspects such as performance, accessibility, best practices, progressive web apps, and SEO. In the context of this thesis, the performance audit holds particular significance. The Performance score, which represents a weighted average of metric scores, is calculated based on key metrics, including First Contentful Paint (FCP), Speed Index (SI), Largest Contentful Paint (LCP), Total Blocking Time (TBT), and Cumulative Layout Shift (CLS). As discussed in Section 6.2, while the Cumulative Layout Shift showed no need for improvement, the remaining metrics exhibited poor scores. Notably, the initial values for FCP, LCP, and SI were approximately 20 seconds, making it challenging to meet the recommended range of 1.8 to 3.4 seconds. Instead, Logimic should prioritize achieving threshold values that significantly impact the performance score. For instance, improving FCP to 3.7 seconds or lower, LCP to 5.5 seconds, SI to 3.9 seconds, and TBT to 1.3 seconds would positively influence the performance score. The Lighthouse scoring calculator¹² offers valuable insights into the score calculation process. Achieving these proposed values would enable the establishment of effective test thresholds. For example, an FCP of 20 seconds has the same detrimental effect on the performance score as 10 seconds, underscoring the importance of meeting these performance targets. This thesis proposes performance improvements with the aim of achieving the recommended threshold values for key metrics. Proposed performance improvements were consulted with Logimic developers, and several were implemented by Logimic. Section 7.1 contains a summary of implemented improvements, and their impact can be seen in Chapter 8.

It is important to note that the performance score can fluctuate based on factors such as device hardware and internet connection. Additionally, browser extensions injecting JavaScript and interfering with network requests, as well as antivirus software, may negatively impact the score, as can be seen in Figure 7.4. Browser extensions, particularly ad blockers, are widely used among users, but developers have no control over their presence and influence. These extensions often contribute to increased Total Blocking Time by blocking the main thread, leading to decreased page responsiveness and slower loading times, as evidenced by the Speed Index metric. Unfortunately, developers cannot mitigate the impact of these extensions directly. Consequently, the performance observed on user devices may frequently be suboptimal. Additionally, discrepancies may arise between reports generated on a test runner without any browser extensions and those produced locally, where developers commonly utilize such extensions.

Antivirus software also interferes with testing. Cypress is able to control and stub network traffic. Therefore, Cypress assigns and manages browser certificates to be able to modify the traffic in real time. However, this functionality can lead to issues with certain antivirus software, such as Kaspersky Antivirus. These antivirus programs often check web

¹¹<https://www.npmjs.com/package/@cypress-audit/lighthouse>

¹²<https://googlechrome.github.io/lighthouse/scorecalc/>

METRICS		Expand view		METRICS		Expand view	
▲ First Contentful Paint	4.8 s	▲ Largest Contentful Paint	11.1 s	▲ First Contentful Paint	4.8 s	▲ Largest Contentful Paint	8.4 s
▲ Total Blocking Time	1,720 ms	● Cumulative Layout Shift	0.011	▲ Total Blocking Time	590 ms	● Cumulative Layout Shift	0.02
▲ Speed Index	12.9 s			▲ Speed Index	6.8 s		

Figure 7.4: Lighthouse report with plugins (left) and without (right)

certificates, and when Cypress issues certificates, it can result in an incomplete chain of certificates. As a consequence, the antivirus may block traffic to sites with these certificates. Given that ACADA relies on communication with Logimic servers and AWS Cognito, this issue renders the platform unusable and consequently untestable. The solution to this problem involves several approaches. One option is to temporarily disable the antivirus software while running Cypress tests. Alternatively, users can add the Cypress certificate as trusted to the antivirus program. Another approach is to configure Cypress to use specific certificates that are recognized by the antivirus software. By implementing one of these solutions, users can ensure that Cypress testing is not hindered by antivirus software interference.

7.1 Performance Improvements

Lighthouse reports revealed a significant issue: the absence of compression for JavaScript and CSS files retrieved from the server, particularly Angular bundles like `main.js` (containing application code, including components, services, and modules) and `vendor.js` (which includes third-party libraries and dependencies used by ACADA). These bundles are quite large, often several megabytes in size.

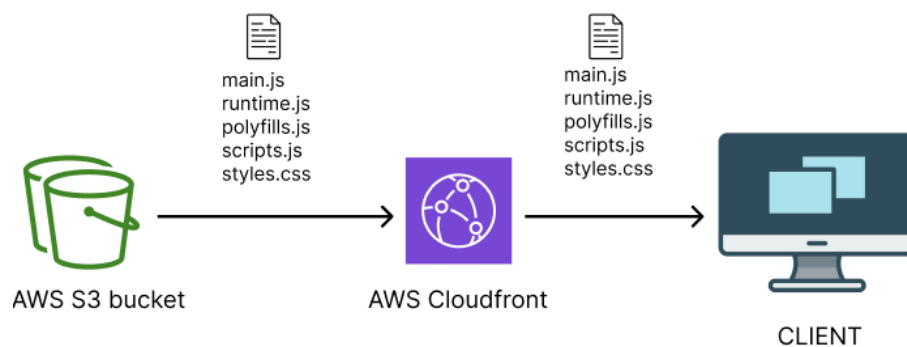


Figure 7.5: Distribution of Angular bundles stored on AWS S3 bucket, delivered to clients via AWS CloudFront

This lack of compression leads to longer loading times, especially on devices with poor internet connections, ultimately blocking the main thread and negatively impacting FCP,

LCP, and SI metrics. Logimic utilizes two deployment methods: the first involves cloud deployment using AWS S3 buckets to store application files and AWS CloudFront as a content delivery network (illustrated in Figure 7.5). While AWS CloudFront can automatically compress certain file types¹³ and serve them to clients if they support it. However, Logimic had disabled this feature due to the significant nearly one-day delay it introduced when deploying new versions, which was deemed unacceptable.

A potential solution would be to compress the files within the S3 bucket or upload compressed files to the bucket. However, attempts to upload compressed files to the S3 bucket and manually configure the required metadata, as suggested by Hicks¹⁴, did not yield the desired outcome, as the files were still being served uncompressed. Further investigation into AWS CloudFront revealed that enabling automatic compression required enabling caching. Unfortunately, Time-To-Live (TTL) values were set to 86400 seconds (equivalent to one day), which is the default AWS policy, resulting in significant delays in deployment. This occurred because clients would still be served old files due to the TTL of files cached on edge locations of CloudFront. Consequently, Logimic opted to previously abandon automatic compression on CloudFront.

After discussions with Logimic, the TTL value was adjusted to 15 minutes, effectively reducing the lengthy delays and enabling compression and caching simultaneously. Compressing these files using Brotli or gzip also reduced their size, consequently cutting down transfer times.

Second, local deployment method uses an Nginx webserver with Docker. In this scenario, the configuration was nearly complete. Enabling the `gzip_static` directive¹⁵ allowed the server to serve compressed files stored in its local storage to clients. A new npm target was created to achieve file compression. This target compresses Angular bundle files and CSS files using gzip while keeping the original files. This approach ensures that if the compressed file is either unavailable or unsupported by the client, Nginx serves the uncompressed version of the file.

URL	Type	Total Bytes	Unused Bytes	Usage Visualization
▶ http://localhost:4200/vendor.js	JS (per fu...	15,215,588	8,521,026 56%	
▶ http://localhost:4200/main.js	JS (per fu...	9,138,609	5,443,542 59.6%	
▶ http://localhost:4200/scripts.js	JS (per fu...	910,854	666,053 73.1%	
http://localhost:4200/styles.css	CSS	281,867	275,186 97.6%	
http://localhost:4200/saga-blue.css	CSS	179,294	173,710 96.9%	
▶ http://localhost:4200/styles.js	JS (per fu...	176,205	163,820 93%	

Figure 7.6: Coverage report of L1 Dashboard using Coverage DevTool on Google Chrome

Compressing Angular bundles addresses only a part of the size problem. Initially, the `main.js` file contained all modules and components, regardless of their use, resulting in unused code as shown in Figure 7.6. While Angular offers lazy loading of modules as a solution, Logimic initially did not implement it. To enable lazy loading, slight modifications were made to the configuration files of modules that did not need to be initially loaded. Specifically, the `component` declaration was replaced with `loadChildren` and moved the `import` statement accordingly. This adjustment allows Angular to load these modules dynamically as needed rather than including them in the initial bundle. This has slightly reduced the size of the `main.js` file, but more is required in order to impact performance.

¹³<https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/ServingCompressedFiles.html>

¹⁴<https://medium.com/@graysonhicks/how-to-serve-gzipped-js-and-css-from-aws-s3-211b1e86d1cd>

¹⁵<https://docs.nginx.com/nginx/admin-guide/web-server/compression/>

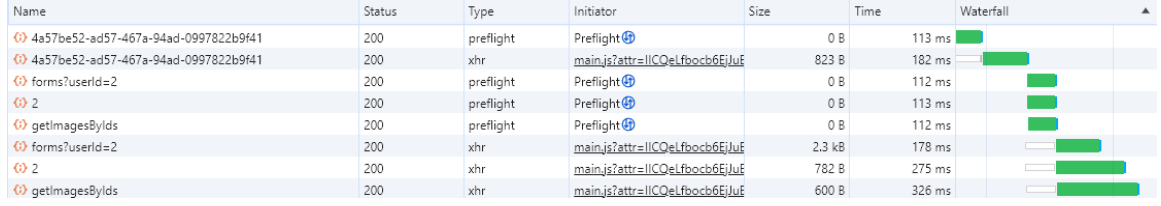


Figure 7.7: loading of user information followed by parallel loading of system information, forms, and profile image

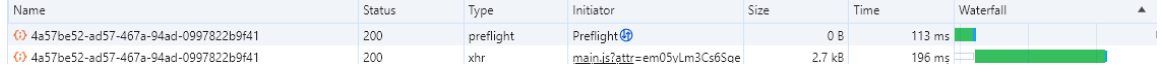


Figure 7.8: A single query containing user and system information, forms, and profile image

However, modules are not the only issue. ACADA has many dependencies, some of which are unused, duplicate, or barely used, resulting in further unused code (see `vendor.js` in Figure 7.6). For example, code using the `aws-sdk` npm package was not utilized and was consequently removed. Additionally, functionalities provided by certain libraries, such as `pdf.js` for generating PDF files and `xlsx.js` for generating Excel sheets, can potentially be moved to the backend. Moreover, some libraries, like `quill.js`, are only partially used, and their functionality could be replaced with custom implementations. However, replacing libraries with custom implementations may be time-consuming and may not be impactful, making it cost-prohibitive. Similar to modules, not all dependencies are utilized throughout the application. Starting from Angular version 17, third-party dependencies can be lazy loaded [52, 22], which can significantly reduce bundle size. However, Logimic currently uses version 13, making lazy loading of third-party dependencies impossible. Therefore, this thesis recommends Logimic to consider migrating to version 17 to take advantage of lazy loading for third-party dependencies.

Section 6.2 proposed optimizing the loading of user-related data (user and system information, forms, and profile image) by consolidating it into a single query. Initially, the system loaded user information sequentially, followed by parallel loading of system information, forms, and profile images, as depicted in Figure 7.7. This sequential loading introduced delays as subsequent data waited for the user information to complete loading. Although consolidating user and system information, forms, and profile images into a single query increased transfer size and slowed server response compared to individual requests, it still resulted in roughly 50% reduction in loading time compared to the original method, as shown in Figure 7.8.

In Section 6.2, the use of WebSockets or webhooks was proposed. While webhooks could have sufficed for sending updated data to clients, WebSockets offer greater versatility with bi-directional communication. Recognizing the benefits of bi-directional communication for future development, Logimic expressed a preference for WebSockets. However, integrating either webhooks or WebSockets would have necessitated significant changes to the existing codebase. Logimic had higher-priority tasks at the time of writing this thesis, resulting in neither option being implemented. Nevertheless, Logimic is actively considering the implementation of WebSockets in the near future.

Chapter 8

Testing

Cypress tests are designed to cover both typical user interactions within the application, such as viewing alerts and graphs and checking KPIs, as well as edge cases, such as submitting empty usernames or incorrect file formats. Through this testing approach, several bugs were discovered, promptly reported to Logimic, and subsequently addressed.

One unexpected issue emerged during the testing of L2 and L4 graphs, which exclusively appeared when using Firefox on devices equipped with touchpads. This anomaly resulted in the **p-calendar** component from the PrimeNG library becoming noninteractive, as illustrated in Figure 8.1. To address this issue, Logimic implemented a solution by disabling support for touch screens, effectively resolving the bug and restoring the expected functionality.

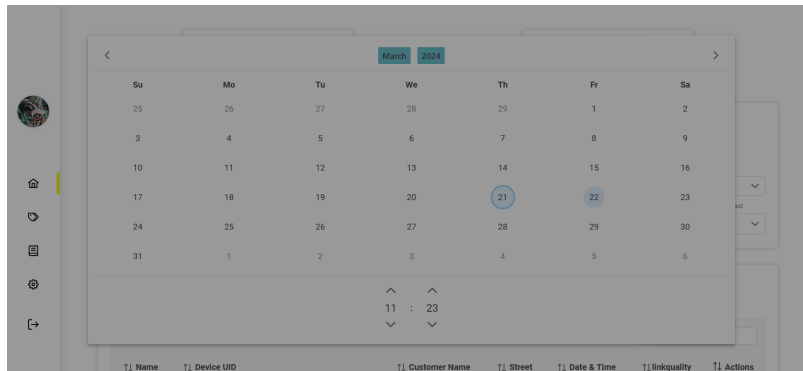


Figure 8.1: Calendar Modal on L2 that can not be interacted with

In another instance, during an L3 test of the Alerts page, aimed at verifying the correct functioning of the Actions modal window (refer to Figure 8.2), it was observed that upon clicking the “Send Mail” button and completing the Email Message form, no email was sent or received, and instead, a blank page was displayed. This dysfunctional feature was promptly identified and subsequently removed by Logimic, along with the “Send Mail” button, to prevent any further issues.

Furthermore, a specific test targeting the user profile page revealed a functional issue: although users could click on the “add” button (as illustrated in Figure 8.2) and upload a file, the image failed to update accordingly.

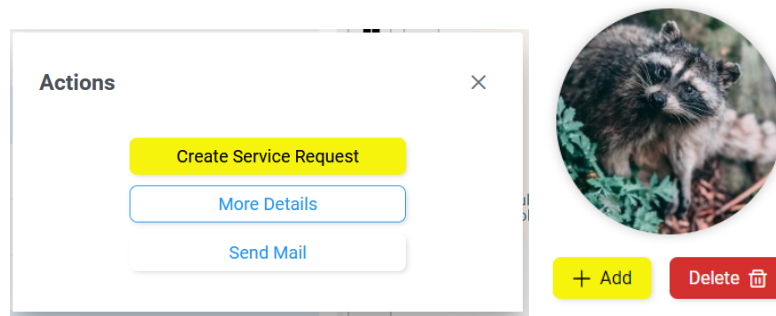


Figure 8.2: Action modal on L3 (left) and User profile image with add and delete button (right)

Following the recommendation outlined in Section 6.2 to use WebP and AVIF files instead of PNG, a subsequent test was conducted to validate the support for these alternative file formats. However, it was discovered that AVIF files were not accepted as expected.

Thanks to the CI/CD integration, the tests were able to prevent regression. In one instance, they uncovered a modification made by another student to the L1 dashboard, which resulted in previously clickable elements becoming unclickable. This would have prevented users from navigating to L2 group details, rendering a large part of the application unreachable.

8.1 Performance Improvement Evaluation

As outlined in Chapter 7, the evaluation of Google Lighthouse audits reveals that performance scores can fluctuate due to various factors. Notably, the presence of browser extensions emerges as a significant contributor to performance discrepancies. When active, these extensions inject additional JavaScript code into web pages, which can consume CPU resources and impede page rendering. For instance, Figure 8.3 illustrates a higher Total Blocking Time metric on a report with extensions compared to a report where the browser operates in incognito mode, mitigating extension interference. This discrepancy suggests that browser extensions indeed contribute to blocking the main thread, leading to increased Total Blocking Time and slower page rendering.

While browser extensions play a pivotal role in performance variations, hardware components, particularly the CPU, have a significant impact on performance scores. The CPU plays a critical role in processing various tasks involved in loading and rendering a web page. These tasks include executing JavaScript (affecting the Total Blocking Time metric), parsing HTML and CSS, rendering visual elements (impacting metrics like First Contentful Paint, Largest Contentful Paint, and Speed Index), and handling network requests. Devices with more powerful CPUs can execute these tasks more quickly and efficiently, resulting in faster page loading times and better performance scores on Google Lighthouse reports. For instance, reports shown in Figure 8.3 were generated on the same laptop with an Intel Core i5-10300H CPU. In contrast, Figure 8.4 displays the report of the same page viewed in incognito mode on Chrome on a desktop featuring an AMD Ryzen 5 5600X CPU, illustrating the impact of CPUs on performance scores.

Despite the availability of Google Lighthouse in Microsoft Edge as a built-in tool, the `@cypress-audit/lighthouse` plugin exclusively supports Google Chrome and Chromium.

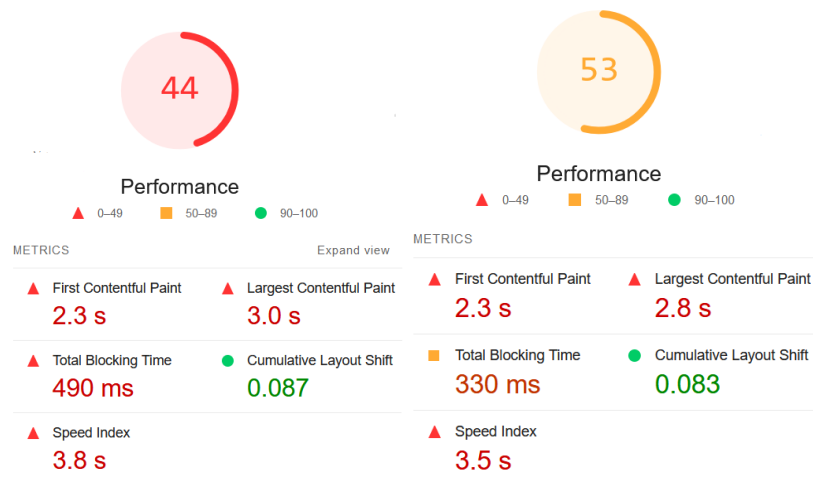


Figure 8.3: Google Lighthouse report with extensions (left) and without (right)

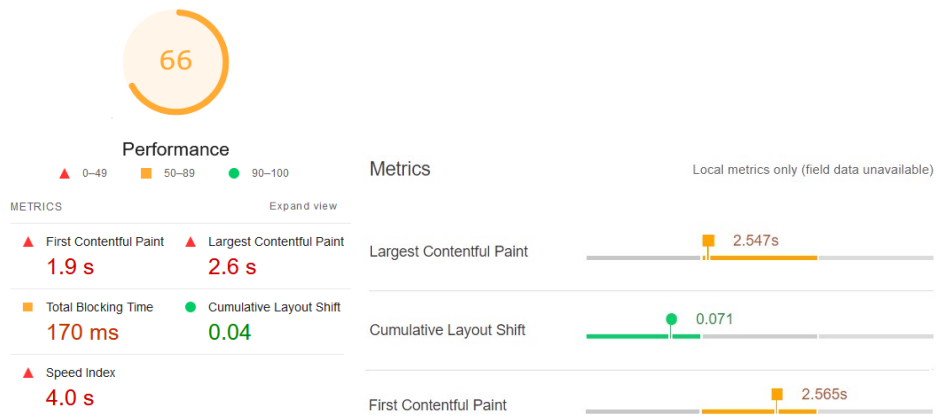


Figure 8.4: Google Lighthouse report on Windows with AMD Ryzen 5 5600X CPU (left), Metric scores from Web Vital extension (right)

Since Chrome is not available on ARM-based architectures like the Raspberry Pi used for the local test runner, Logimic is left with Chromium as the only viable option for running tests and generating Lighthouse reports.

In addition to Google Lighthouse, the Web Vitals Chrome extension serves as an alternative tool for assessing page performance. This extension offers rapid access to metric values, alleviating the need to generate comprehensive Lighthouse reports, which can be time-consuming. A comparison of values obtained from the Web Vitals extension, as depicted in Figure 8.4, with reports like Figure 8.3, shows similar results. This similarity suggests developers can effectively utilize either tool interchangeably based on their specific needs.

Many images and icons are served in PNG format. However, utilizing WebP format can reduce image sizes by an average of 26% compared to PNG, while AVIF can achieve even greater reductions of up to 50% on average. The extent of compression varies depending on the nature of the images; for instance, images with a white background tend to achieve higher compression rates. In Figure 8.5, images with homogeneous backgrounds, such as

white, have smaller file sizes compared to those with variegated backgrounds despite having slightly higher resolutions. While the compression rates between AVIF and PNG formats remain similar, WebP compression is notably less effective for images with variegated backgrounds. Given that the default profile picture and most icons feature a white background, transitioning to WebP or AVIF formats can have a significant impact on file size. Consequently, this reduction leads to smaller load sizes, thereby slightly improving performance on devices, particularly for users with slower internet connections.

racoon.avif	AVIF File	35 KB	rat_white_background.avif	AVIF File	13 KB
racoon.jpg	JPG File	87 KB	rat_white_background.jpg	JPG File	40 KB
racoon.png	PNG File	261 KB	rat_white_background.png	PNG File	147 KB
racoon.webp	Chrome HTML Do...	204 KB	rat_white_background.webp	WEBP File	17 KB

Figure 8.5: Compression comparison on image with variegated background (left) and white background (right)

8.2 Impact of Performance Improvements

It is crucial to verify the effectiveness of implemented performance improvements. This validation process involves comparing Google Lighthouse reports generated before and after the implementation of improvements on the same machine, as illustrated in Figure 8.6. Enabling compression significantly reduced the file size transferred to the client, and when combined with the consolidation of loading user-related data into a single query, resulted in improved load times and subsequently reduced the time for First Contentful Paint (FCP), Largest Contentful Paint (LCP), and Speed Index (SI). Similarly, removing third-party libraries and enabling lazy loading not only reduced the file size transferred to the client but also decreased the CPU execution time for JavaScript tasks. This reduction in Total Blocking Time (TBT) allowed the browser to initiate rendering earlier, further decreasing the time for FCP, LCP, and SI. The improvement in these performance metrics serves as evidence of improved application performance on first loads.

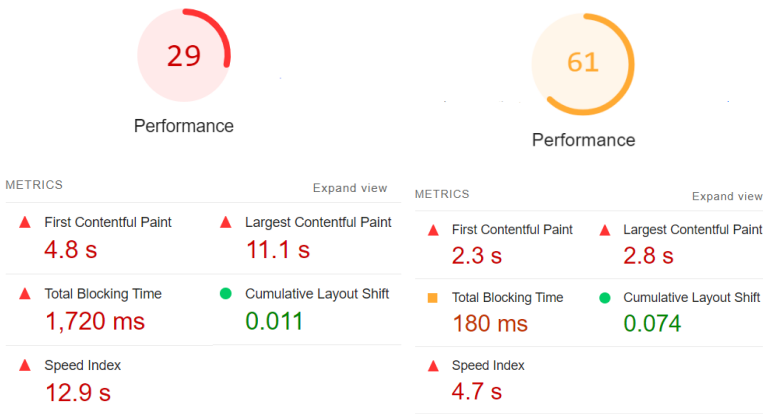


Figure 8.6: Google Lighthouse report before performance improvements (left) and after (right)

Chapter 9

Conclusion

This thesis aims to establish a new automated testing environment for the ACADA platform while optimizing its performance. An in-depth exploration of the Internet of Things (IoT) and Smart Cities, covering diverse architectures, sensors, communication technologies, and domains that contribute to functionality and efficiency, was conducted to achieve these objectives. Commonly used testing methods and frameworks were cataloged, with Cypress identified as the preferred choice for automated end-to-end testing. Integration with CI/CD pipelines using GitHub Actions was proposed, offering automation and flexibility to adapt testing strategies according to Logimics requirements. The analysis of performance metrics has led to the identification of areas for improvement, and corresponding suggestions have been proposed.

During implementation, a testing environment was developed for the ACADA platform, covering both typical user interactions and edge cases to ensure comprehensive coverage. Standards for test structure and workflow were established to maintain clarity and consistency across the testing suite.

A CI/CD pipeline was set up using GitHub Actions to automate the testing process. This pipeline allows automatic execution of Cypress tests, allowing for efficient and frequent testing of the platform and being adjustable to the needs of Logimic.

Google Lighthouse was utilized to measure the performance of ACADA and identify areas for improvement. Based on the performance metrics and suggestions from the analysis, suggested improvements were implemented to optimize performance.

The testing section documents bugs that were discovered by the tests. The impact of the implemented improvements was analyzed using Google Lighthouse reports to measure the effectiveness of the enhancements and ensure that they contribute positively to the overall performance of ACADA. The resulting higher scores in Google Lighthouse reports serve as evidence of improved performance.

The automated testing environment allows Logimic to expand its testing efforts. Following the established end-to-end testing strategy, Logimic can continue writing tests to cover user interactions and edge cases, improving test coverage and ensuring platform functionality. As new features are developed, the testing suite can be expanded to accommodate changes, maintaining platform reliability.

Additionally, Google Lighthouse serves as a valuable tool for monitoring platform performance. Regular Lighthouse audits enable Logimic to track key performance metrics, identify regressions, and optimize performance, ensuring optimal user experiences.

Bibliography

- [1] AL SARAWI, S., ANBAR, M., ALIEYAN, K. and ALZUBAIDI, M. Internet of Things (IoT) communication protocols: Review. In: *2017 8th International Conference on Information Technology (ICIT)*. IEEE, 2017, p. 685–690. DOI: 10.1109/ICITECH.2017.8079928. ISBN 978-1-5090-6332-1.
- [2] ANDREA CARAGLIU, C. D. B. and NIJKAMP, P. Smart Cities in Europe. *Journal of Urban Technology*. 1st ed. Routledge. 2011, vol. 18, no. 2, p. 65–82. DOI: 10.1080/10630732.2011.601117. Available at: <https://doi.org/10.1080/10630732.2011.601117>.
- [3] ASHTON, K. et al. That ‘internet of things’ thing. *RFID journal*. Hauppauge, New York. 2009, vol. 22, no. 7, p. 97–114.
- [4] ATITALLAH, S. B., DRISS, M., BOULILA, W. and GHÉZALA, H. B. Leveraging Deep Learning and IoT big data analytics to support the smart cities development: Review and future directions. *Computer Science Review*. 2020, vol. 38, p. 100303. DOI: <https://doi.org/10.1016/j.cosrev.2020.100303>. ISSN 1574-0137. Available at: <https://www.sciencedirect.com/science/article/pii/S1574013720304032>.
- [5] BAHMUTOV, G. *Keep passwords secret in E2E tests* [online]. Dec 2023. [Accessed 18. April 2024]. Available at: <https://glebbahmutov.com/blog/keep-passwords-secret-in-e2e-tests/#do-not-hardcode-passwords>.
- [6] BELLINI, P., NESI, P. and PANTALEO, G. IoT-Enabled Smart Cities: A Review of Concepts, Frameworks and Key Technologies. *Applied Sciences*. 2022, vol. 12, no. 3. DOI: 10.3390/app12031607. ISSN 2076-3417. Available at: <https://www.mdpi.com/2076-3417/12/3/1607>.
- [7] BERNARDO, M. d. R. M. Smart city governance: from e-government to smart governance. In: *Smart cities and smart spaces: Concepts, methodologies, tools, and applications*. IGI Global, 2019, p. 196–232.
- [8] BOSE, S. *What is end to end testing?* [online]. Feb 2023. [Accessed 11. November 2023]. Available at: <https://www.browserstack.com/guide/end-to-end-testing>.
- [9] CISCO. *The Internet of Things Reference Model*. Cisco Systems, Inc., 2014. Available at: <https://dl.icdst.org/pdfs/files4/0f1d1327c5195d1922175dd77878b9fb.pdf>.
- [10] DAMERI, R. Searching for Smart City definition: a comprehensive proposal. *International Journal of Computers & Technology*. october 2013, vol. 11, p. 2544. DOI: 10.24297/ijct.v11i5.1142.

- [11] DAMERI, R. P. *Smart City Implementation*. 1st ed. Springer Cham, 2017. ISBN 978-3-319-83369-9.
- [12] EHMER, M. and KHAN, F. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*. 2012, vol. 3, no. 6, p. 12–15. DOI: 10.14569/ijacsa.2012.030603.
- [13] FARMANBAR, M. and RONG, C. Triangulum City Dashboard: An Interactive Data Analytic Platform for Visualizing Smart City Performance. *Processes*. 2020, vol. 8, no. 2. DOI: 10.3390/pr8020250. ISSN 2227-9717. Available at: <https://www.mdpi.com/2227-9717/8/2/250>.
- [14] FAROOQ, M. U., WASEEM, M., MAZHAR, S., KHAIRI, A. and KAMAL, T. A review on internet of things (IoT). *International journal of computer applications*. Foundation of Computer Science. 2015, vol. 113, no. 1, p. 1–7.
- [15] FEW, S. *Information Dashboard Design: Displaying Data for At-a-glance Monitoring*. Analytics Press, 2013. ISBN 9781938377006.
- [16] GIFFINGER, R., FERTNER, C., KRAMAR, H., KALASEK, R., MILANOVIĆ, N. et al. *Smart cities - Ranking of European medium-sized cities*. Vienna University of Technology, january 2007.
- [17] GOKHALE, P., BHAT, O. and BHAT, S. Introduction to IOT. *International Advanced Research Journal in Science, Engineering and Technology*. IARJSET. 2018, vol. 5, no. 1, p. 41–44.
- [18] HALL, P. Creative Cities and Economic Development. *Urban Studies*. 2000, vol. 37, no. 4, p. 639–649. DOI: 10.1080/00420980050003946. Available at: <https://doi.org/10.1080/00420980050003946>.
- [19] HARDING, B. *A guide to single-page application performance* [online]. Raygun, Aug 2023. [Accessed 17. April 2024]. Available at: <https://raygun.com/blog/spa-performance/>.
- [20] HAXHIBEQIRI, J., DE POORTER, E., MOERMAN, I. and HOEBEKE, J. A Survey of LoRaWAN for IoT: From Technology to Application. *Sensors*. 2018, vol. 18, no. 11. DOI: 10.3390/s18113995. ISSN 1424-8220. Available at: <https://www.mdpi.com/1424-8220/18/11/3995>.
- [21] HOWELL, J. *An Introduction To IoT Architecture* [online]. 2023. [Accessed 9. October 2023]. Available at: <https://101blockchains.com/iot-architecture/>.
- [22] JANIUK, J. *Use Angular's new deferrable views now* [online]. Angular Blog, Dec 2023. [Accessed 24. April 2024]. Available at: <https://blog.angular.io/use-angulars-new-deferrable-views-now-20ca9f64f1e5>.
- [23] KIM, W.-S., LEE, W.-S. and KIM, Y.-J. A Review of the Applications of the Internet of Things (IoT) for Agricultural Automation. *Journal of Biosystems Engineering*. 2020, vol. 45, p. 385–400. DOI: 10.1007/s42853-020-00078-3. Available at: <https://doi.org/10.1007/s42853-020-00078-3>.

- [24] KUMAR, N. M. and MALLICK, P. K. The Internet of Things: Insights into the building blocks, component interactions, and architecture layers. *Procedia computer science*. Elsevier. 2018, vol. 132, p. 109–117. DOI: <https://doi.org/10.1016/j.procs.2018.05.170>. ISSN 1877-0509. Available at: <https://www.sciencedirect.com/science/article/pii/S1877050918309049>.
- [25] KUMAR, P. *Top 10 performance testing tools in 2023* [online]. BrowserStack, 31. July 2023. [Accessed 3. December 2023]. Available at: <https://www.browserstack.com/guide/performance-testing-tools>.
- [26] LI, S., XU, L. D. and ZHAO, S. The internet of things: a survey. *Information systems frontiers*. Springer. 2015, vol. 17, p. 243–259. DOI: 10.1007/s10796-014-9492-7.
- [27] LIN, J., YU, W., ZHANG, N., YANG, X., ZHANG, H. et al. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE internet of things journal*. IEEE. 2017, vol. 4, no. 5, p. 1125–1142.
- [28] LIU, H. and KUAN TAN, H. B. Covering code behavior on input validation in functional testing. *Information and Software Technology*. Feb 2009, vol. 51, no. 2, p. 546–553. DOI: 10.1016/j.infsof.2008.07.001.
- [29] MAHDAVINEJAD, M. S., REZVAN, M., BAREKATAIN, M., ADIBI, P., BARNAGHI, P. et al. Machine learning for internet of things data analysis: a survey. *Digital Communications and Networks*. 2018, vol. 4, no. 3, p. 161–175. DOI: <https://doi.org/10.1016/j.dcan.2017.10.002>. ISSN 2352-8648. Available at: <https://www.sciencedirect.com/science/article/pii/S235286481730247X>.
- [30] MOHANTY, S. Everything You Wanted to Know About Smart Cities. *IEEE Consumer Electronics Magazine*. july 2016, vol. 5, p. 60–70. DOI: 10.1109/MCE.2016.2556879.
- [31] MZAHM, A., AHMAD, M. and TANG, A. Agents of Things (AoT): An Intelligent Operational Concept of the Internet of Things (IoT). In: *2013 13th International Conference on Intelligent Systems Design and Applications*. IEEE, December 2013. DOI: 10.1109/ISDA.2013.6920728. ISBN 978-1-4799-3516-1.
- [32] NAVEEN, S. Study of IoT: Understanding IoT Architecture, Applications, Issues and Challenges. In: *1st International Conference on Innovations in Computing & Networking (ICICN16)*. IJANA, May 2016. ISSN 0975-0282.
- [33] NIDHRA, S. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications*. june 2012, vol. 2, no. 2, p. 29–50. DOI: 10.5121/ijesa.2012.2204.
- [34] PALLIS, G. and VAKALI, A. Insight and perspectives for content delivery networks. *Communications of the ACM*. ACM New York, NY, USA. 2006, vol. 49, no. 1, p. 101–106.
- [35] PARKER, D. *Moving Angular CLI to Jest and Web Test Runner*. [online]. Angular Blog, 27. April 2023. [Accessed 18. November 2023]. Available at: <https://blog.angular.io/moving-angular-cli-to-jest-and-web-test-runner-ef85ef69ceca>.

- [36] PASPELAVA, D. *What is unit testing in software testing and why is it important?* [online]. March 2021. [Accessed 10. October 2023]. Available at: <https://www.exposit.com/blog/what-unit-testing-software-testing-and-why-it-important/>.
- [37] PENG, S.-L., PAL, S. and HUANG, L. *Principles of internet of things (IoT) ecosystem: Insight paradigm*. Springer, 2020. ISBN 978-3-030-33595-3.
- [38] PENG, S.-L., PAL, S. and HUANG, L. *Principles of Internet of Things (IoT) Ecosystem: Insight Paradigm*. January 2020. ISBN 978-3-030-33595-3.
- [39] PERAL, J., MATÉ, A. and MARCO, M. Application of Data Mining techniques to identify relevant Key Performance Indicators. *Computer Standards & Interfaces*. 2017, vol. 54, p. 76–85. DOI: <https://doi.org/10.1016/j.csi.2016.11.006>. ISSN 0920-5489. SI: New modeling in Big Data. Available at: <https://www.sciencedirect.com/science/article/pii/S0920548916301763>.
- [40] QAZI, S., KHAWAJA, B. A. and FAROOQ, Q. U. IoT-Equipped and AI-Enabled Next Generation Smart Agriculture: A Critical Review, Current Challenges and Future Trends. *IEEE Access*. 2022, vol. 10, p. 21219–21235. DOI: 10.1109/ACCESS.2022.3152544. ISSN 2169-3536.
- [41] RAIKULA, K. *Implementation of Automated End-To-End Testing in Web Applications*. 2023. Bachelor thesis. Metropolia University of Applied Sciences.
- [42] ROJAS, E., BASTIDAS, V. and CABRERA, C. Cities-Board: A Framework to Automate the Development of Smart Cities Dashboards. *IEEE Internet of Things Journal*. 2020, vol. 7, no. 10, p. 10128–10136. DOI: 10.1109/JIOT.2020.3002581.
- [43] SEHRAWAT, D. and GILL, N. S. Smart Sensors: Analysis of Different Types of IoT Sensors. In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. IEEE, 2019, p. 523–528. DOI: 10.1109/ICOEI.2019.8862778. ISBN 978-1-5386-9439-8.
- [44] SETHI, M. A. A review paper on levels, types & techniques in software testing. *International Journal of Advanced Research In Computer Science*. 2017, vol. 8, no. 7, p. 269–271. DOI: 10.26483/ijarcs.v8i7.4236. ISSN 0976-5697. Available at: <https://doi.org/10.26483/ijarcs.v8i7.4236>.
- [45] SETHI, P. and SARANGI, S. R. Internet of Things: Architectures, Protocols, and Applications. *Journal of Electrical and Computer Engineering*. jan 2017, vol. 2017, p. 25. DOI: 10.1155/2017/9324035. ISSN 2090-0147. Available at: <https://doi.org/10.1155/2017/9324035>.
- [46] SHAW, M. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer*. Oct 2002, vol. 4, no. 1, p. 1–7. DOI: 10.1007/s10009-002-0083-4.
- [47] SINHA, S. *State of IoT 2023: Number of connected IoT devices growing 16to 16.7 billion globally* [online]. 2023. [Accessed 6. October 2023]. Available at: <https://iot-analytics.com/number-connected-iot-devices/>.

- [48] SRIVASTAVA, N., KUMAR, U. and SINGH, P. Software and Performance Testing Tools. *Journal of Informatics Electrical and Electronics Engineering (JIEEE)*. Jan. 2021, vol. 2, no. 1, p. 1–12. DOI: 10.54060/JIEEE/002.01.001. Available at: <https://jieee.a2zjournals.com/index.php/ieee/article/view/11>.
- [49] SURESH, P., DANIEL, J. V., PARTHASARATHY, V. and ASWATHY, R. H. A state of the art review on the Internet of Things (IoT) history, technology and fields of deployment. In: *2014 International Conference on Science Engineering and Management Research (ICSEMR)*. IEEE, 2014, p. 1–8. DOI: 10.1109/ICSEMR.2014.7043637. ISBN 978-1-4799-7613-3.
- [50] TAN, J. and KOO, S. G. A Survey of Technologies in Internet of Things. In: *2014 IEEE International Conference on Distributed Computing in Sensor Systems*. IEEE, 2014, p. 269–274. DOI: 10.1109/DCOSS.2014.45. ISSN 2325-2936.
- [51] TANASIEV, V., PĂTRU, G. C., ROSNER, D., SAVA, G., NECULA, H. et al. Enhancing environmental and energy monitoring of residential buildings through IoT. *Automation in Construction*. 2021, vol. 126, p. 103662. DOI: <https://doi.org/10.1016/j.autcon.2021.103662>. ISSN 0926-5805. Available at: <https://www.sciencedirect.com/science/article/pii/S0926580521001138>.
- [52] THALHAMMER, A. *How to lazy load large 3rd-party dependencies with angular 17's @defer* [online]. Jan 2024. [Accessed 24. April 2024]. Available at: <https://www.angulararchitects.io/en/blog/how-to-lazy-load-large-3rd-party-dependencies-with-angular-17s-defer/>.
- [53] UMAR, M. A. and CHEN, Z. A Study of Automated Software Testing: Automation Tools and Frameworks. *International Journal of Computer Science Engineering (IJCSSE)*. december 2019, vol. 8, no. 6, p. 217–225. DOI: 10.5281/zenodo.3924795.
- [54] UNITED NATIONS. *World Urbanization Prospects: The 2018 Revision [key facts]*. United Nations, 2018. Available at: <https://population.un.org/wup/Publications/Files/WUP2018-KeyFacts.pdf>.
- [55] VINOD KUMAR, T. M. and DAHIYA, B. Smart Economy in Smart Cities. In: VINOD KUMAR, T. M., ed. *Smart Economy in Smart Cities: International Collaborative Research: Ottawa, St.Louis, Stuttgart, Bologna, Cape Town, Nairobi, Dakar, Lagos, New Delhi, Varanasi, Vijayawada, Kozhikode, Hong Kong*. Singapore: Springer Singapore, 2017, p. 3–76. DOI: 10.1007/978-981-10-1610-3_1. ISBN 978-981-10-1610-3. Available at: https://doi.org/10.1007/978-981-10-1610-3_1.
- [56] WEINSTEIN, R. RFID: a technical overview and its application to the enterprise. *IT Professional*. 2005, vol. 7, no. 3, p. 27–33. DOI: 10.1109/MITP.2005.69.
- [57] XUYANG, L., LAM, K., ZHU, P. K., ZHENG, C., LI, X. et al. Overview of Spintronic Sensors, Internet of Things, and Smart Living. august 2016.
- [58] YICK, J., MUKHERJEE, B. and GHOSAL, D. Wireless sensor network survey. *Computer Networks*. 2008, vol. 52, no. 12, p. 2292–2330. DOI: <https://doi.org/10.1016/j.comnet.2008.04.002>. ISSN 1389-1286. Available at: <https://www.sciencedirect.com/science/article/pii/S1389128608001254>.

Appendix A

Contents of the included storage media

The included storage media holds these files:

- README.md: readme file describing content of the storage media
- thesis.pdf: thesis paper in PDF format
- thesis/: directory with all LaTeX source files
- source/: directory with all the source files
- source/README.md: readme file explaining the installation steps and usage