České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jan Helbich**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Energetický dopad technologie zobrazení webového uživatelského rozhraní na mobilním zařízení**

Pokyny pro vypracování:

Cílem práce je vyhodnocení dopadu použití různorodých technologií pro zobrazení uživatelského rozhraní webových aplikací na spotřebu zdrojů mobilního zařízení, včetně energetické náročnosti na spotřebu baterie. Pro dosažení dostatečně kvalitních výsledků navrhněte metodiku pro jednoznačné měření spotřeby zdrojů zařízení. Měření budou probíhat na zařízeních s operačními systémy založenými na Linuxovém jádře, pro které vytvořte prototyp měřícího softwarového nástroje.
Pro testování spotřeby vytvořte ukázkovou webovou aplikaci, jejíž uživatelské rozhraní bude založeno na technologiích Java Server Faces 2, Google Web Toolkit, AngularJS 2, ReactJS a AspectFaces. Z dosažených výsledků vyhodnoťte jak ekonomické dopady, tak i dopady na možné řízení spotřeby baterie mobilních zařízení pomocí technologií pro tvorbu adaptivního uživatelského rozhraní.

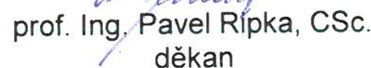
Seznam odborné literatury:

[1] Cerny, T.; Macik, M.; Donahoo, M.J.; Janousek, J., "Efficient description and cache performance in Aspect-Oriented user interface design," in Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on , vol., no., pp.1667-1676, 7-10 Sept. 2014
doi: 10.15439/2014F244
[2] Liu, X.; Ma, Y.; Liu, Y.; Xie, T.; Huang, G., "Demystifying the Imperfect Client-Side Cache Performance of Mobile Web Browsing," in Mobile Computing, IEEE Transactions on, vol.PP, no.99, pp.1-1
doi: 10.1109/TMC.2015.2489202
[3] Singh, V.K.; Dutta, K.; VanderMeer, D., "Estimating the Energy Consumption of Executing Software Processes," in Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing , vol., no., pp.94-101, 20-23 Aug. 2013
doi: 10.1109/GreenCom-iThings-CPSCom.2013.40

Vedoucí: Ing. Tomáš Černý

Platnost zadání: do konce letního semestru 2016/2017

L.S.

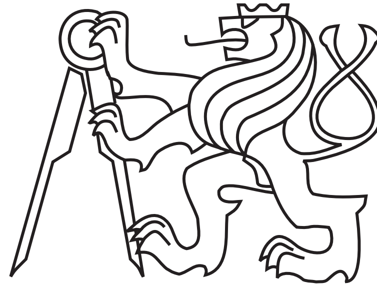


prof. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 26. 11. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Energy impact of web user interface rendering technology on mobile devices

Jan Helbich

Supervisor: Ing. Tomáš Černý, MSc.

Study Programme: Otevřená informatika, Magisterský

Field of Study: Softwarové inženýrství

January 9, 2017

Poděkování

Rád bych poděkoval Ing. Tomáši Černému, MSc. za jeho cenné rady během tvorby této práce. Také děkuji své rodině a přátelům za jejich nekonečnou podporu mého studia.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on January 9, 2017 ....

Abstract

Prezentační technologie webu se průběžně vyvíjí a přizpůsobují požadavkům moderních webových aplikací. Dnes jsou webové aplikace škálovatelnými systémy s vysokým výkoném, které zároveň dobře pracují v distribuovaných prostředích, a grafické rozhraní musí s tímto vývojem přirozeně držet krok. Ale nejsou to jen technologie, který procházejí proměnou. Uživatelé již také často dávají přednost mobilním, baterií napajéným zařízením oproti klasickým stolním počítačům. Je ale možné, že má současný prudký vývoj technologií i jiné dopady, než hezký vzhled webový stránek či rensponzivnost? Nové technologie často přinášení zvýšené nároky na výkon hardware a kapacita baterie je stále omezujícím faktorem při používání smartphonů či laptopů.

Tato práce představuje úvod do tématu měření spotřeby výkonu počítačů a zabývá se aplikací získaných znalostí pro potřeby webových aplikací. Výzkum je primárně zaměřen na vyhodnocení použití různorodých prezentačních technologií na klienta - baterií napájené zařízení. Dále práce popisuje postup návrhu a validaci vlastního softwarového nástroje pro měření výkonu, vhodného pro dané účely, a prozkoumává oblast adaptvního uživatelského rozhraní.

Abstract

Web persentation technologies are continuously evolving, adapting to the requirements of modern applications. Today, web applications are scalable, high-performance systems deployed in distributed environments, therefore even the view layer has to keep up the pace. Also users evolve, moving from desktop computers towards mobile, battery powered devices, such as smartphones. But is there more to the obvious side of contemporary technologies, such as pretty design or responsiveness? Increasing system resources demands negatively influence power consumption of any computer devices, and battery capacity is still a limiting factor for current smartphones or laptops.

This thesis provides an introduction into the field of computer power consumption measurements and applies gained knowledge to web applications. The focus is primarily set on the client - a battery powered device. Futhermore this thesis presents an approach to designing own performance measurement tool and explores the topic of adaptive user interfaces.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Brief background on web applications . . . 2
  - 1.2.1 Application state . . . 3
  - 1.2.2 Stateful server and thin client . . . 3
  - 1.2.3 Thick client and stateless server . . . 3
- 1.3 Requirements catalog . . . 4
- 1.4 Related work . . . 5
  - 1.4.1 Energy efficiency in datacenters . . . 5
  - 1.4.2 Energy efficiency of mobile devices . . . 5
  - 1.4.3 Alternative approaches to power evaluation . . . 6
  - 1.4.4 Comparing web technologies . . . 6
  - 1.4.5 Research summary . . . 7

2 Analysis 8
- 2.1 Approach to measurement . . . 8
  - 2.1.1 Battery consumption measurement method . . . 8
  - 2.1.2 The system resources usage evaluation method . . . 9
- 2.2 Requirements for performance analysis tool . . . 10
  - 2.2.1 Performance analysis tools . . . 11
  - 2.2.2 The /proc file system . . . 12
  - 2.2.3 The /sys file system . . . 14
- 2.3 Evaluating and comparing frontend technologies . . . 15
  - 2.3.1 Testing approach . . . 16

3 Design 17
- 3.1 Measurement method . . . 17
- 3.2 The sample application . . . 18
  - 3.2.1 Tested use cases . . . 20
  - 3.2.2 Backend design . . . 20
  - 3.2.3 Testing adaptive UI . . . 20
- 3.3 System resources utilization tracking tools . . . 20
  - 3.3.1 The performance analysis tool . . . 21
- 3.4 Infrastructure . . . 22
- 3.5 Deployment environment . . . 23

3.6 Tested frontend frameworks . . . 24
3.6.1 JavaServer Faces 2 . . . 24
3.6.1.1 JSF lifecycle . . . 25
3.6.2 Google Web Toolkit . . . 26
3.6.3 AngularJS . . . 28
3.6.4 Angular 2 . . . 29
3.6.5 ReactJS . . . 30
3.6.6 AspecFaces . . . 31

4 Validation of our measurement tool 34
4.1 Validation method . . . 34
4.1.1 Laboratory setup . . . 35
4.2 Results evaluation . . . 35
4.2.1 Evaluation and summary . . . 35

5 Case study: Eventier 37
5.1 Deployment and testing environment . . . 37
5.2 Network traffic comparison . . . 37
5.3 Single client performance . . . 38
5.4 Multiple clients performance . . . 39
5.5 Summary . . . 40

6 Case study: Grafana and adaptive UI 42
6.1 Test setup . . . 42
6.2 Experimental results - graph plotting . . . 42
6.3 Summary . . . 44

7 Conclusion 45
7.1 Future work . . . 45
7.2 Summary . . . 46

8 Eventier - installation guide 51

9 Attached CD 52

List of Figures

3.1 Design of the domain model for our sample application. 19
3.2 Class diagram featuring the core design blocks of our measurement tool. . . . 22
3.3 Deployment diagram featuring used components in our experimental setup. . 23
3.4 Hardware and network model of experimental setup. 24
3.5 The JavaServer Faces application lifecycle 25

6.1 CPU and memory usage for browser rendering. 43
6.2 CPU and memory usage when rendering on server. 43

9.1 Directory tree on attached CD 52

List of Tables

4.1 Energy impact of excessive system resource usage during 3 hours stress tests . 35

5.1 network traffic comparison for UI technologies. 38
5.2 Performance utilization consumption comparison for UI technologies. Single client accesses the application. All values are in MB if not stated otherwise. . 39
5.3 Power consumption comparison and performed tasks statistics for UI technologies. Single client accesses the application. 39
5.4 Performance utilization consumption comparison for UI technologies. Multiple clients access the application. All values are in MB if not stated otherwise. 40
5.5 Power consumption comparison and performed tasks statistics for UI technologies. Multiple clients access the application. 40

6.1 Performance utilization while rendering Grafana plots. Values are in units of MB if not stated otherwise. 43

Chapter 1

Introduction

Throughout the time of its existence, the design of web pages has come a long way from simple textual representation of information to highly interactive presentations in sophisticated graphical form. However the demands on user interface (UI) seem to be ever increasing and today face yet another challenge, that is to adapt to various kinds of client devices and browsers while preserving desired functionality and user experience. The personal computer market has evolved and expanded and desktop computers have become far less common then before and are steadily being replaced by mobile, battery powered devices, such as laptops, tablets or smartphones [1]. Of course this progress has made its mark on web technologies and techniques of development for both front and back-end parts of applications.

There are many obvious manifestations of the evolution that the presentation part of the web has undergone. Be it advanced Human-Computer Interaction (HCI), focusing on usability, adaptive and individual approach to match user's needs. The engineering aspect is focusing on responsiveness and performance just as much as on scalability of applications. Together with well-worked graphical design all these are valuable aspects for overall user experience (UX), aiming to create positive attitudes and perceptions of the person using the system. But there are many other aspects, which seems to be rather of secondary interest to UI technologies since these are not demonstrated as clearly, for example the amount of resources consumed by the device while rendering interacting with web pages.

In this work we focus on further exploring the field of performance and energy consumption of web application's client device. For this purpose we propose an approach to measurement of both and present a comparison of the impact that various different, yet modern and up to date frontend web frameworks have on consuming resource of the client device.

1.1 Motivation

While creating web enterprise systems, the crucial goal is to enable users perform specific business tasks. Here the UI plays a major role, guiding the user through the use case. That is a field of interest to both HCI and UX - the efficiency and comfort while interacting with the application. Battery powered devices may become an obstacle in various cases, though.

Imagine a long reservation process, where suddenly the battery runs out and users are absolutely unaware if the reservation has been placed or not. If the reservation system was aware of the client's devices battery status, it could possibly postpone or take a shortcut in the long process or advice the user to use an alternate way of booking.

Other example may be an application targeting a number of successfully done business cases, where its users have to perform as many tasks as possible. If such application was written in a way that excessively drains client's device battery power, it would greatly disadvantage users accessing it through a battery powered device.

Web applications use web browser as a medium to display content to end users. The content is made out of two factors - the data and the user interface fragments. If we were to just display raw data, there would not be any difference in performance or energy impact on the client device, if we consider using only one type and version of a browser. But it would be confusing and not intuitive for its users, therefore we have to wrap the data into some kind of graphical form that is understandable and successfully guides user through his task. That is the point where application developers can influence the final impact on the client. Choosing various presentation technologies introduces architectural differences that may possibly result in varying impact on performance and consecutively consumed power.

We do not have any ambition to compare UI technologies based on their architecture and believe every approach possesses its qualities and imperfections. It is a matter of fact that whatever technology is chosen, frontend development efforts are not negligible and are on par with the rest of the application. According to [14] 48% of application and code 50% of development time is spend on UI implementation.

There are countless libraries and frameworks making UI development easier and we can choose whatever fits us the best based on a rich set of criteria. Final UX shall never be deprived of its features, though, because of technology limits or hard to achieve functionalities.

We strive to find out whether there is any impact on the application's user in case we were to choose various web application technologies but still preserve both the desired look and feel and application functionality.

1.2 Brief background on web applications

Web applications realize basic kind of client-server communication. A client initiates the communication, requesting some information from the server. Server then replies to the request. This behaviour is not limited only to browsing web pages, but can of course be applied to interserver communication or web mashups[4], but we are mainly interested in web browser to server communication through HTTP(S)[5] based on the TCP/IP handshake, since it involves graphical presentation.

Since client and server are the only actors in the communication, web application's architecture has to cover both sides. Here we can distinguish between system's so called backend and frontend. Backend layer comprises of application logic and data access. The important thing is that backend part of the system should be independent of system's client. Whether the application's client is a CLI tool or a web browser, the backend core system still stays the same. But users are not supposed to interact directly with the backend. The user-computer interaction is handled by frontend, the presentation layer of the system.

1.2.1 Application state

Most web applications have to manage some kind of stateful information. Whether it is only simple authentication data, finding out what form fields did the user fill or a walking through a complex work flow, the actual application state has to be stored somewhere. For webapps, there are two logical places where to keep such data - either the client or the server, since there are no other participants in the communication.

The decision where to put the application's state greatly influences final architecture of web systems. Whatever approach we choose, there is a need for complex set of tools, so called frameworks, that will help us develop the application without reinventing the wheel and minimize the development costs. But such tools also define boundaries which the developer has to stick to. There are two major architectural styles: rich clients with thin stateless server and thin clients with thick stateful server.

1.2.2 Stateful server and thin client

The choice of server-side UI framework inevitably leads to the thick stateful server and thin client model. This way the server produces both the data and the UI fragments, which reduces the client (web browser) into a simple rendering tool (we do not consider additional features such as AJAX support, which may bring certain benefits of the client-side approach).

While developing webapps based on server-side UI framework, one of the great assets is that the whole application context is available in one place. This is beneficial mainly for code reduction and security, because it disregards additional parts of the system that contain any logic.

Further benefits lay in the maintenance phase of system's lifetime. All application fragments that execute any part's of application logic stay on the server. From the developer's point of view this drastically reduces the needed effort to maintain the system. There is simply no need to distribute system updates to all clients, everything needed to deploy a new feature or fix a bug in the code is on the application server and will instantly affect all application's clients.

The final UI the server-side frameworks create is a combination of HTML, JavaScript and CSS tangled together with data in specific application context (i.e. security). The data become hard to separate from the view, which reduces the possibility of distributing data to other applications or UI customizations. Also the network traffic suffers compared to client-side approach, which generates the view fragments and the content of communication with server is solely the data.

Another possibly negative point the server-side UI frameworks bring is higher server load. Server has to reconstruct the UI after user interaction and process the changes. This presents additional computations that have to be done and therefore may become an issue during peak hours. Since application state is on the server, horizontal scalability requires more sophisticated techniques, such as sticky sessions[18], to work properly.

1.2.3 Thick client and stateless server

The objective of thick client architecture is to keep both the logic and generation of view in the browser, leaving server as a simple data endpoint. Since the logic is extracted out of

server's scope, it has no access to the application state in any point of time and therefore becomes fully stateless.

A strict separation of concerns is a great asset of this design, because the application data can be easily reused and eventually there can be multiple types of UI created without any change on the server. Client is also able handle some use cases without communication with the server, reducing network traffic and sparing server resources. Statelessness is also highly appreciated factor for horizontal scaling because there is no need for keeping a connection between a client and one certain server. Clients may freely communicate with any server instance without loss of session.

The separation of the application into two partly independent entities has its negative impacts, of course. First the loss of context results in inevitable need of implementing various parts of the application on both client and server level. As an example may serve data validation. The server cannot accept invalid data, therefore it has to contain the validation logic. The client also has to ensure it has valid data for both enhanced user experience and predicable communication with server. Such restatements lead to higher maintenance complexity and eventually higher costs. This may become a tedious and error-prone task and requires high test coverage and solid integration tests.

1.3 Requirements catalog

As was said earlier, our main goal is to evaluate the impact of web UI rendering technology on the client. But how can we achieve that? And what does it bring with a closer look at the topic? In this section we present a brief list of areas we are interested in, and which comprises the necessary requirements and areas of interested of this work.

First we have to introduce a generic methodology for measuring power and energy consumption of computer devices. The overview of web applications shows that UI technologies are not based on a single architectural type, therefore the final methodology has to be portable. But this alone is a big problem domain, consisting of multiple subareas we have to focus on.

We have to analyze and evaluate contemporary application deployment model and environment to run them. Today virtualization and cloud rule the world of web application, therefore we have to project this fact into our approach. This must result into laboratory conditions equal to standard production deployment model, otherwise our measurements will lose objectivity.

Next there are measurement criteria to consider, be it the way of performance or power consumption evaluation method, or what in the end are we really supposed to measure? Is there any performance factor that can prove one UI technology to be better than the others?

After we design the methodology, we need to create a prototype of an evaluation tool, which is able to track and provide insight on the client device utilization from both system resources usage and power consumption. Our work is primarily focusing on Linux-based system, thus our criteria here are well constraint. Are there any existing programs fulfilling our demands? Or is it possible to assemble such tool from existing and time-proven tools? We will also explore the origins of tracking system and power to understand what information Linux kernel provides and how it works.

Having the whole apparatus ready we will present a case study comparing various UI technologies in terms of client device utilization and energy impact. We also strive to reveal possible benefits of adaptive UI techniques on battery life of mobile computers.

Last but not least we need to consider economical side of the problem, primarily from client's point of view.

1.4 Related work

Our main objective lies in power and resource utilization and UI technologies, while focusing on the connection between the two. We need an insight into both, therefore we will focus on researching each topic independently.

1.4.1 Energy efficiency in datacenters

Many articles regarding power consumption of computers focus on analyzing power utilization in large datacenters and total cost of ownership(TOC). Capra and Merlo proposed a research roadmap to identify a set of software complexity and quality metrics that may be used to assess the energy efficiency of a specific application[17]. The work presents various goals to achieve that lead to the path of energy efficient software and criteria on a tool enabling users to perform energy consumption analysis of the system based on code inspection and hardware utilizations. The measurment method however considers not only the involvement of a physical wattmeter, but even other hardware modifications (using processor clamps). Also only the CPU usage is considered in the computations.

According to Intel[19], the main consumers of server power are processors, draining 45W when idle and up to 200W when fully saturated on average. Following is the memory, ranging from 2W to 12W per module when idle and active respectively. Another considerable amount of energy is lost in computer's power supply, which ranges from 20% to 40% of total power consumed. The paper also suggests a calculation to estimate power consumption based on processors utilization if maximal and idle power demands are known:

$$P_n = (P_{max} - P_{idle}) \times \frac{n}{100} - P_{idle}$$

where P_{max} is the maximal power consumed by the processor, P_{idle} is power drain while idle, n is the utilization factor and P_n the final estimated power for given utilization. They support this equation with additional measurements of actual power drain, where the statistical error ranged around 5%.

1.4.2 Energy efficiency of mobile devices

Of course the boom of mobile market has also brought attention to research of power consumption of devices like smartphones or tablets. In addition to server or most desktop computers, theC devices have additional hardware integrated, such as GSM module. Carrol and Heiser [20] perform present an extensive study on the topic.

Performing benchmarks for common tasks, such as browsing emails or watching videos, the energy drawn from individual hardware components is observed. GSM module clearly has

the most negative influence on battery discharging, with total demands of 824mW (aggregate power exluding backlight) during phone call and 610mW while emailing. Another heavy consumer of power is the backlight, ranging between 8mW and 414mW. There are two interesting facts in regard to the backlight. First the power characteristics is not clearly linear, but rather exponential, consuming 150mW for 67% intensity. Second discovery is that displayed color also impacts required power, being 38.1mW for white and 74.2mW for black color displayed at same backlight intensity.

Experiments were run primarily on Openmoko Neo Freerunner, while using external hardware measurement tool for individual components. The results were validated on HTC dream and Google Nexus One tracking overall power consumption and the relative results show clear correlation to data collected for Openmoko.

The energy demands widespread mobile networking technologies is further addressed in[21], comparing 3G, GSM and WiFi. The work focuses on *tail energy*, an energy overhead inflicted by transition between active and idle state after data transfer. Exprimenting on Nokia N95 and HTC Fuze, the results cleary show that 3G consumes most resources during data transfers with 60% of power wasted on tail energy. 3G is followed by GSM with 30% waste with the period of time lingering in the high-energy post-transfer states reduced to half of 3G. The work shows that WiFi is most efficient at least during high network traffic and states that the tail energy is comparable to 3G, but the transfer is much more effective. The actual amount of wasted power is not mentioned, but for small transfer sizes is compared to GSM.

1.4.3 Alternative approaches to power evaluation

All mentioned works rely on external hardware while measuring power utilization by the device. A software alternative for Linux-based OS is addressed by[22]. Authors present various approaches to energy evaluation and their design of a tool, that requires additional hardware tools only during initial calibration phase. Afterwards the consumed power by a process p is computed as

$$Energy_p = F(cpu_p, disk_p, network_p, memory_p)$$

where F is a function of CPU usage cpu_p , disk usage $disk_p$, network usage $network_p$ and memory usage $memory_p$[22].

The final computations are done using information from *procfs*, the Linux interface to internal kernel structures, which we further explore in the following part of our work. The tool samples chosen resources both on complete system and single process level, computing intermediate utilization from current and previous samples. Through successive empirical measurements the researchers were able to deliver estimates with approximately 95% precision.

1.4.4 Comparing web technologies

Černý and Donahoo[23][2] consider contemporary UI design and delivery approaches from the perspective of resource utilization and energy impact on both the client and the server. The work discusses server and client-side UI technologies and possible improvements

by applying Distributed Concern Delivery (DCD) and browser caching. According to the results, there is a significant positive impact to both server and client by implementing DCD in server-side technologies (specifically JSF).

[24] provides a detailed analysis on how using a different web server influences web application's energy usage. The work targets Ruby on Rails[25] applications and conventionally used servers. Researchers clearly point out that server architecture and the way it handles requests plays major role in application's power consumption. The study notes that the choice of the server due to high throughput and responsiveness may be in conflict with energy efficiency[24]. Another valuable information are the metrics and experimental criteria used throughout empirical measurements, which are primarily in introducing a constant element into the experiments. In case of this particular work it is the application implementation, which is compatible with all presented servers.

In our initial work[16] we focused on performance and power evaluation of the client device while accessing web application with both client and server-side based UI. The paper considers completed business use case as a unit of user performance as well as system resource utilization, and puts additional constraints onto the inter-framework comparison of web UI. The work disregards bottlenecks in network communication by deploying the application and client into LAN with high bandwidth and based on data collected through empirical measurements compares AngularJS to JSF.

Another type of performance comparisons of UI frameworks are often done by members of the frontend development community. An extended benchmark [1] for JavaScript UI frameworks and libraries is being done, evaluating more than 30 trendning presentation technologies. The tests disregard application server involvement and compare technologies based on reimplementing sample application and time tracking of defined tasks.

1.4.5 Research summary

Both UI framework comparison and evaluation of performance and energy consumption of computers are complex fields of study and require generic yet robust approach and deep knowledge of physics, systems and web applications technologies. We have evaluated related works in our area of interest that provide approach and solutions to similar challenges we face in this work and will further analyze these in the next chapter.

[1] https://github.com/krausest/js-framework-benchmark

Chapter 2

Analysis

In this chapter we analyze the requirements needed to fulfill our main goals, which are the evaluation of the energy and performance impact of using various presentation technologies in web applications on the client device. We strive to analyze and present possible methodologies for such measurements, the key criteria to focus on while benchmarking computer performance and energy consumption. Next we present requirements for tools used to evaluate the results a possible optimizations. We also explore the economic side of the topic from both the client and application provider's point of view.

2.1 Approach to measurement

We can divide our areas of interest in measurement into two categories: performance and energy. The performance evaluation of the client device targets the use of system resources, such as CPU or network traffic. Thus this part comprises of the observation of multiple (partly) independent elements of the system. Energy impact on the client however has a single criterion - the absolute consumption of battery of the device while performing certain tasks.

2.1.1 Battery consumption measurement method

There are multiple ways of intercepting the natural process of battery discharging for both hardware and software solutions. Apart from precision, our only requirement is that the used tool is easily applicable to various types of devices, so that there are no additional constraints placed opto the observed device.

The obvious hardware method is connecting a wattmeter to the observed device, collection actual statistics of power usage in units of Watts. Using this method would have the most precise result, since the output data covers the absolute required power consumption while measuring. The employment of the wattmeter can be quite troublesome, though, since it is required to connect it between the power source (in our case the battery) and the load (the device). Practically it would mean to disassemble tested device, i.e. a laptop or a smartphone, remove its battery and replace it with kind of pluggable adapter, which would serve as connector between the battery, wattmeter and the device. For such reason we have

rejected this approach since it requires hardware modifications (even though temporary) and could be quite difficult to set up if we had to measure power consumption of multiple different devices.

Another approach is to collect data from hardware sensors of the battery through the interface with operation system. The Linux kernel does and excellent job here, providing access to actual (values from monitoring hardware) voltage, current, energy and many more information. This is since we can acquire the necessary data without an employment of an external device. Of course a possible problem may arise when a certain power supply does not provide monitoring or there are no kernel drivers available for it. But it is a common practice to display charge status on mobile devices based on Linux, therefore for the purposes of our work we make a simple assumption that all devices have these values available. It would be suitable to express the energy consumption in watt-hours, which actual value can be computed as $P*h$, where P is actual electrical power and h is time in hours. The electrical power is defined as $P = U * I$, where U is voltage and I is current.

As a result, we only have to track the value of current, voltage and time of measurement to derive the value of consumed energy. This is an approach taken by popular Linux tool PowerTOP[26], used for analyzing and optimizing power usage for laptops.

Both earlier mentioned methods depend on continuous data collection and time tracking during measurements to acquire exact energy values. The last presented method differs here and rather takes a statistical way of evaluating the influence of running applications on energy consumption. The idea is to rate every single system resource with based on its power efficiency and create detailed battery discharging statistics for specific hardware configuration. Subsequently it would be possible to statistically accurately predict battery consumption of a process based on its usage of system resources. This is expected to be the case of Apples's OS X Energy Impact[40], a part of the Activity Monitor. It is not clear how Energy Impact really works since it is a closed-source project, but we base our assumptions on educated guess and analysis provided in [40].

2.1.2 The system resources usage evaluation method

We not only aim to understand the characteristics of various UI technologies in energy consumption, but we also want to investigate the origin of these differences. For this purpose we need to track the usage of system resources, such as CPU or I/O operations.

There are multiple works presenting the impact of system resources utilization on power demands for mobile devices, but our main interest is presented in[20]. Following is the list of picked system resources we evaluated as crucial to power management of the web application's client. We will need to observe these throughout our experiments.

- CPU
- RAM
- Disk I/O
- Network traffic
- Backlight

The key element in all of our measures is time, though. If the CPU runs at 100% during the initialization of the application and then its utilization is rather insignificant, it is definitely not the same as if it would run at 20% of utilization for an hour. Therefore we have to evaluate collected data in context of time. A database would present a suitable tool for that matter, both for its data persistence and further evaluation by providing a unified access method, such as SQL.

Next we need to know the scope of measurements, that is if we are supposed to track the behaviour of the whole system or focus on single or multiple processes. A web browser is a medium through which user interacts with web applications. Of course various browsers differ in its architecture, but here we are mostly interested the process model.

Google Chrome is an example of multi-process architecture[28]. There are three types of processes running: browser, renderers and plugins. Browser is always only one, handles interaction with user, network communication, disk I/O, but does not parse any HTML nor interprets JavaScript of web pages. Renderers are reponsible for handling all the logic for processing HTML, JavaScript or CSS, leveraging the open-source WebKit[56] rendering engine. For security reasons are renderers sandboxed, preventing it from direct system interaction. If a suspicious activity is detected, i.e. an exploit, the main browser process may kill the renderer or stop its execution. Chrome spawns a renderer process for each opened tab on a distinct domain. Plugins are all external features run by the browser, for example a PDF reader or Flash player.

Mozilla Firefox introduced multi-process architecture called Electrolysis[55] as a preview testing feature in its version 48, released on 7th March 2016. As of version 52 (19th September 2016) it should be enabled by default. The architecture of Electrolysis is similar to that of Chrome, therefore having a main parent processes and sandboxed children ones for security enhancement.

According to survey[7] Chrome and Firefox are the most commonly used browsers compatible with Linux, having 57.1% and 11.1% of internet users respectively. The outcome is unambiguous - we need to focus on multiple processes while measuring the performance and system resources utilization of the device.

2.2 Requirements for performance analysis tool

We need an instrument to acquire and evaluate performance data. There are numerous tools for performance analysis provided for Linux systems and as was mentioned earlier, we are interested only in a small subset of system resources. We dare say that these are the basics for performance evaluation, thus we should not be limited only to advanced tools.

Other requirement for the tools is a possibility to export data to a reasonable format, so that we store and evaluate results later. A database would be suitable here, so the exported data format should be easily transformable to other formats for the simple purpose of swapping database engine.

Next we need the tool to be as economical as possible in the meaning of system load. For example if we are to evaluate disk I/O, the tool can not excessively write to disk, because as a result our data would be corrupted. The same goes for all other resources, of course.

Even though relative error between measurements would probably not be significant, since all data would be affected the same, it is still preferred to evade such state.

As an optional feature we would welcome a support for registration of input events, realized for example as a HTTP based web service. Using this mechanism we could integrate the client-side code of loaded web page with our tool and pass information between the two, such as start and end of page load, track user's activities on the page or, for testing purposes, remotely control the measurement tool itself.

In the following subsections we provide a short description of common Linux tools for performance analysis and information about kernel interface, the *procfs* and *sysfs* virtual file systems.

2.2.1 Performance analysis tools

Top is a Unix utility that provides a rolling display of top CPU using processes[32]. The program is a customizable tool providing real-time view on the operating system status. Customizations are based on selection of tracked and displayed system statistics. It is a standard tool to be found in most Linux distributions or Mac OS. According to documentation the program is able to sample performance values in tenths seconds. Backlight intensity is not an information provided by *top*. The output of the program is based on columns and units of given values are configurable (e.g. bytes/kilobytes etc.), thus capturing the standard output of *top*, parsing the text and saving these values would be a way to store collected data.

PowerTOP is a power management, diagnosis and consumption tracking tool. One of the main benefits of *PowerTOP* is that it tracks processor's idle wakeups, which are harmful to battery usage. It also provides view onto CPU and GPU usage. The program can export measured values into multiple data formats, such as HTML and CSV. Export to CSV would be beneficial for us, because it contains unambiguous pure data. To obtain measurements throughout test sessions, we would have to continuously collect the data the same way as with *top*. *PowerTOP* does not provide memory or I/O statistics, therefore it must complement other measurement tools if used.

Sar is a Linux utility for collecting and monitoring performance data throughout time. It can be run as a service, thus provides a simple, yet comprehensive and powerful tool for system monitoring. The program support the tracking of all resources we need, except for the backlight intensity. Data are sampled in the interval of seconds based on the configuration. The output of *sar* command is a binary file, which can be easily processed by *sadf* program, providing output to standard text formats like CSV, XML or JSON. The *sar* does not support tracking of single processes, rather provides a view of the system as a whole.

Pidstat is a program similar to *top*, but is much simplified and serves as a process monitor, providing all required resources statistics except of backlight intensity, which does not make sense in context of a process. It is also possible to track groups of processes by name, which is would be a handy way to collect data from multi-process browsers like Chrome. *Pidstat* does not support any form of exporting collected data to other formats and its results sampling interval is in units of seconds.

The *upower* is designed to evaluate and control power management. It offers a simple yet fullfeatured way to access the status of a power supply of the client based device, including

displaying all information provided by the ACPI. *Upower* writes to the standard output and does not offer any mechanism for exporting results to other data formats.

To summarize, none of the presented tools provides all desired functionality, which leaves us with either connecting those programs together or introduce our own tool, which fulfills all of our requirements.

2.2.2 The /proc file system

In this section we examine the topic of system resources usage measurement on operating systems based on Linux kernel for the need of implementing our own tool. If not explicitly mentioned, we take into account only kernels of versions 2.6 and newer.

The commonly used interface to the Linux kernel is the 'proc' filesystem (also known as *procfs*), which is in typically (and in most standard distributions) mounted as '/proc'. For our use case - resources usage measurement - we only need a small subset of all it offers, though. The *procfs* is called a pseudo or virtual file system, since it is not a real filesystem accessing physical harddrive, but is implemented in the Linux kernel itself in is whole entirety. There are no physical files present, rather it offers a real-time view and control for many system properties from user space. For example if we run a simple command *ls/proc*, the kernel will generate the file and directory listings based on the current set of processes and system state. If we were to open any file, i.e. the '/proc/meminfo', the kernel collects current memory statistics at the moment of our file request and then reflects the values as the file we tried to open[33]. This is a certain benefit of the 'Everything is a file'[34] feature of Unix-like operation systems.

All files we need in the *procfs* have their structure defined in the kernel documentation and the *man* pages. For example listings 2.1 shows a piece of content of the '/proc/stat' file regarding CPU usage. This is system-wide information represented as 'time' spent in various states. Time is expressed as units of $1/USER_HZ$, where USER_HZ is a system variable defaulting to 100 for most Linux distributions. We are most interested in the following values (followed by column number in the example file):

- user - time spend in user space (outside the kernel) within all processes (#2)
- nice - time spend in user mode with low priority (#3)
- system - time spend in kernel space (privileged mode) within all processes (#4)
- idle - CPU idle time (#5)

Subsequently we can acquire total CPU usage in percents per system since its startup as follows:

$$process_time = user + nice + system$$
$$CPU_usage = process_time/(process_time + idle) * 100$$

Considering that all time constraints are in $USER_HZ$ means that even if we are able to acquire real-time system statistics about CPU usage, we are bound to count with a maximum

of $1/USER_HZ$ time error (due to time units rounding). One possibility to minimize the error is to make the variable as big as possible. Linux kernel supports values to be 100, 250, 300 and 1000 (see man 7 time), however changing this system variable requires kernel recompilation.

Listing 2.1: CPU related content of /proc/stat.

```
cpu  303098 1956 68564 3018386 11969 0 412 0 0 0
cpu0 76676 484 18545 900668 6067 0 149 0 0 0
cpu1 77521 372 17508 703152 2433 0 160 0 0 0
cpu2 74790 584 16061 706484 2028 0 84 0 0 0
cpu3 74110 514 16448 708081 1441 0 19 0 0 0
```

The *procfs* structure exposes single process information in the '/proc/[pid]/' directory, where 'pid' is the system ID of given process. There are numerous files, each containing pieces of process details, but we are primarily interested in the '/proc/[pid]/stat' file. Listing 2.2 shows an example of such file. All times in the $1/USER_HZ$ units of seconds.

If we follow the previous example of tracking solely CPU usage, there is only a subset of provided values we need. These are:

- utime - time the process spent in user space (#14)
- stime - time the process spent in kernel space (#15)
- cutime - time the children processes spent in user space (#16)
- cstime - time the children processes spent in kernel space (#17)
- starttime - the time the process was started at (#22)

Calculating total CPU usage for a single process is a bit harder since we do not have process idle time. Therefore we have to calculate this value as $totalsystemuptime - processstarttime$.

System uptime is accessible in the '/proc/uptime' file. Therefore the total CPU calculation goes as follows:

$$
\begin{gathered}
total_time(p) = utime(p) + stime(p) + cutime(p) + cstime(p) \\
seconds_since_start(p) = uptime - (starttime(p)/USER_HZ) \\
CPU_usage(p) = (total_time(p)/USER_HZ)/seconds_since_start
\end{gathered}
$$

Listing 2.2: CPU related content of /proc/stat.

```
2876 (atom) S 2041 2250 2250 0 -1 4194304 52031 1623 328 0 9803 4162 7 2 20
    0 22 0 4197 1338150912 28167 18446744073709551615 4194304 60888772
    140721468545600 140721468544496 140690175699165 0 0 4096 134300907
    18446744073709551615 0 0 17 0 0 0 33 0 0 62988688 65506724 99987456
    140721468553327 140721468553348 140721468553348 140721468555235 0
```

The sole method of measurement usage of system resources is of course done by sampling. By storing resources statistics in given moment, it is possible to evaluate the results accurately both during and at the end of the session. The more results we have, the more we minimize the total error.

However this does not affect the measurement error we get while computing actual usage as difference between previous and current sample. This means we have to choose the right sampling interval. For example if we were to set the sampling to 1 millisecond and $USER_HZ$ would be set to 100, there is a great probability that we get only two different values in 10 samples, since the values we read from i.e. '/proc/stat' file are in one hundredths of second. Other 8 samples would be useless.

Memory management for single processes is more complicated, though. Linux memory model works with virtual(VIRT), resident(RES) and shared(SHR) memory[35]. We are mostly interested in RES, which is the physically consumed memory by a process. Note that there shared libraries and memory are not reflected in RES, but in SHR. But SHR may be distributed among multiple other processes, thus computing the total memory of used by a process as $pid_mem = RES + SHR$ is not precise and summing such values for multiple processes may result in values exceeding the total memory of the device.

Disk I/O operations and network statistics for each peripheral device are also accessible through *procfs*, in '/proc/diskstat' and '/proc/net/dev' files respectively. The structure of these files is similar to those presented earlier, therefore it is necessary to parse them according to documentation.

To summarize, the *procfs* provides all information about resources we need, except for backlight intensity battery charge status, which are not published by *procfs* previously mentioned tools for performance evaluation do not support it either. This makes for a great foundation for building custom tools that analyze system performance.

2.2.3 The /sys file system

Throughout the time of Linux evolution the *procfs* became hard to maintain and developers missed that information and data it provides lack more common structure. For that reasons another virtual file system, the *sysfs* was introduced. Usually mounted as '/sys', the *sysfs* exports yet another interface layer between kernel and user space. All information are exposed as files, following the Unix philosophy. The *sysfs* is a complement to *procfs* and is a preferred way to expose various subsystems or devices interfaces now. All drivers should have their directories automatically created while being registered, which is our main point of interest - to access device's power and backlight management. Registering a driver means that the kernel module implements a few predefined functions[35].

The *sysfs* presents a concept of classes. A class is a higher-level view of a device that abstracts out low-level implementation details. Drivers may see a SCSI disk or an ATA disk, but, at the class level, they are all simply disks. Classes allow user space to work with devices based on what they do, rather than how they are connected or how they work[35]. For example power supplies are accessible under the '/sys/class/power_supply' directory, but naming the battery device present is out of *sysfs* scope. Thus there may be a battery called 'BAT0' or one called 'smb347-battery'. The inside of the battery directory contains multiple files regarding actual voltage, current, charge or other values. While the attributes provided are believed to be universally applicable to any power supply, specific monitoring hardware may not be able to provide them all, so any of them may be skipped.[35]. Presented values are usually in micro units, for example voltage is in [µV]. Another interesting value is 'charge' - the battery's capacity in [µWh]. There is not only a current value provided, but

also a full charge capacity, which is a valuable information for our measurements. Tracking the difference between full and current charge will enable us to evaluate absolute power consumption during during our measurements.

Backlight control is also exposed by *sysfs* class under '/sys/class/backlight' directory. We are mainly interested in maximal and current brightness. These are available in the 'brightness' and 'max_brightness' files, containing a positive integer value representing the light intensity.

2.3 Evaluating and comparing frontend technologies

There are multiple ways of testing and comparing various UI technologies for web applications, mostly based on performance evaluation while working with DOM. These may be based on unit or end-to-end tests, performing various use cases and tracking completion time. For JavaScript there are even tools such as *protractor-perf*[36], an end-to-end test runner checking performance regressions originally for AngularJS.

For testing use cases and their performance we may use even industry proven tool like Selenium. The merit of using Selenium for our case is that we do not only strive to benchmark JavaScript technologies, but also server-side frameworks, where using a test tool based on frameworks like Node.js could become useless.

There have been many evaluations done on the performance of UI technologies, but one of the most extensive public tests are done by[1], featuring over 30 contemporary JavaScript frameworks. While the work does not present a clear winner, the results may be a handy guide while picking a client-side UI technology for new projects.

Such benchmarks are not quite what we aim to achieve, though. We want to evaluate the behaviour of the browser and the whole system while using given technology. Some UI frameworks may cache a lot, excessively allocating systems memory. Other may demand too much CPU or contain so many resources that browser cache produces heavy disk I/O. UI frameworks or libraries also differ in their size, thus network bandwith may become a limiting factor while accessing applications built with these. And the most important to us is the final impact the technology has on power consumption of client's device.

Another aspect that is mostly being left out is benchmarking in production-like environments, meaning the UI is tested against mocked remote services. But obtaining data may be a crucial performance aspect when interacting with web applications. Generally we can distinguish between server and client-side applications here, because server side needs to load all data at once (without AJAX) to correctly render the UI. Client side frameworks can benefit from the asynchronicity and separation of loading data from remote endpoints from UI rendering. Furthermore if a page is combining data from various resources it is natural that these can be loaded independently or lazily, which may be quite hard to achieve by server-side tools.

[1] https://github.com/krausest/js-framework-benchmark

2.3.1 Testing approach

As we learned while studying UI technologies performance benchmarking, there is only way to setup the comparison - to build a sample application providing few use cases to accomplish. Then run end-to-end tests on all implementations, track and evaluate obtained results. This approach is commonly measuring the speed of given operations, such as DOM manipulation. We will broaden this to evaluate performance in the scope of successful completion of a use case.

Next we will present an energy impact statistics based on system load testing. We want to confirm previous works[20] done in the field and aim to find possible bottlenecks in web applications UI that have negative influence on power consumption. For this purpose we will run a set of independent stress tests, focusing on putting high load on single resources while tracking battery charge.

We would like to investigate the topic of possible connection between adaptive UI and energy consumption - can we somehow save battery charge due to progressive changes in UI rendering in real-time without rebuilding the application? Modern browsers already support providing battery status information itself and exposes it to web pages through JS[29]. Our main idea lies in moving client-side operations to the server in case of low battery charge, perhaps automatically when battery life decreases below certain level. To explore this topic, we will need a sample web application that employs both client and server side rendering. We will conduct an empirical measurement and provide a summary of our findings and recommendation for future works.

Chapter 3

Design

This chapter proposes the solutions for requirements and challenges described in analysis. We present the methodology to for our measurements, the necessary tools and environment setup needed to achieve objective results and overview the requirements for sample application and design its features. We also provide a description of UI rendering technologies we will compare and in which our test application will be written.

3.1 Measurement method

From three presented approaches of measuring energy consumption in 2.1.1 we have discarded using the wattmeter simply because it would make our experiments both hardware dependent and hardly portable to different devices. Statistical approach by collecting hardware information, acquiring performance data for short time period and then computing long the presumed impact on battery bears a possibility of involving great statistical error. Even though this approach is definitely a great choice for evaluating power consumption for real-time analysis of the system, we are interested in undistorted experiments results. Therefore we will use the method of continuous data collecting for a longer period of time, which we will later evaluate as a whole. This approach is also beneficial for avoiding possible nonlinearites in battery discharge characteristics due to long measurement sessions and diminishes the possibility of bringing statistical errors in results.

As the first step in our experiments we will perform load tests on individual system resources. From the results we will create a power impact comparison, showing the differences of consumed energe in relation with resource usage.

Next we will implement the sample application in all presented UI frameworks and perform the measurements and all of them separately. We also have to take server load into account, so we will divide the tests into two groups. In the first group we will perform the measurement with only one client accessing the application. The other measurements will be run with multiple clients performing the same tasks. We will still measure the performance on one device, the same one we used for standalone testing. Finally we will evaluate the data collected from client device in both low and high server load aspects and in context of performance of all technologies.

The economical optimization of using certain UI rendering technologies is another task we aim to accomplish. We will evaluate the collected data, focusing primarily on number of completed tasks by the time of measurement, overall network traffic and lines of code needed for successful implementation of the sample application.

3.2 The sample application

As a sample application for our experiments we will create a simple web management system for creating and browsing public events. The system, we will call it Eventier, will consist of two pages. The first is a list of events with and a search toolbar, containing input fields for event name, date of the event and number of results displayed per page. Each row of the events table will contain an action button, allowing users to view event details or edit the event.

The second page will serve the purpose of viewing, editing or creating an event. All data will be displayed in a form containing event name, description, entry fee, currency and capacity. Next there will be the start and end date of the event (may be multiple dates) and each of these date pairs will have an optionally rendered subform for the location details. The location contains a total of 10 attribute fields and two control buttons for saving or discarding modifications. Complete domain model of the application is depicted on figure 3.1.

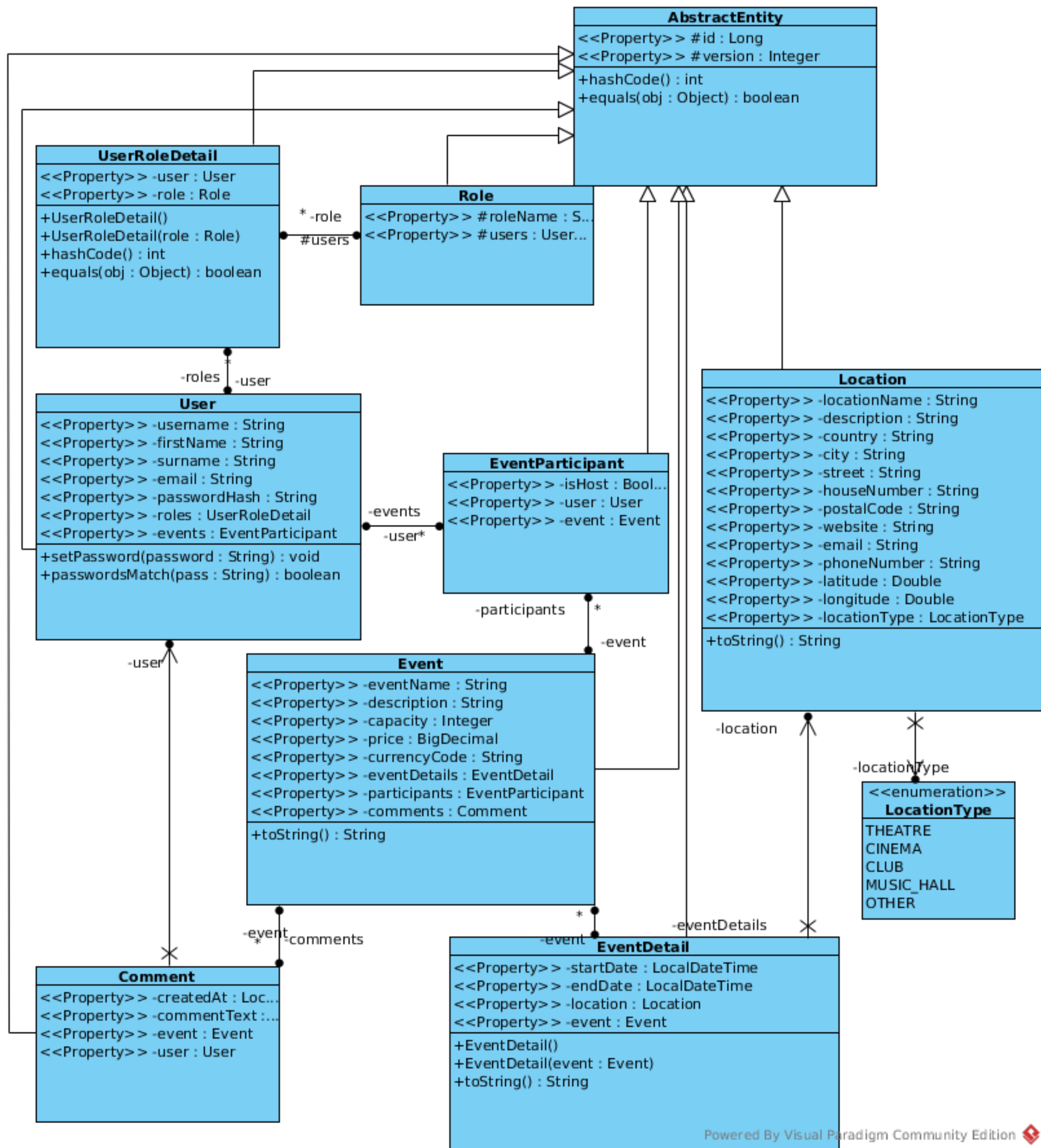


Figure 3.1: Design of the domain model for our sample application.

As a result we will have a CRUD[30] application with a rather simple workflow, at least on the client's site. In our opinion most of today's web applications are CRUD based from the client's point of view, thus even though the features of Eventier are limited, we present it as an example of a real world web application.

3.2.1 Tested use cases

There are two use cases the users of the Eventier application can carry out. These are searching for an event by name or date and editing or creating an event. We will employ both situations in a single test scenario, that is to search for an event, open it for editing, open and view the location details for a certain event date, close the location subform, edit event name, save the event and then finally return to the event listing. We will implement this test scenario using the Selenium framework. The Selenium tests have to deal with differences of the result HTML page a certain UI technology generates, but the main criterion is that the only part of the script that is implementation specific is an XPath or ID of elements we search for.

3.2.2 Backend design

Front end technologies influence the overall architecture of the application, therefore we can not eliminate some differences in the backend while comparing UI technologies. However we need to ensure that the application core, the part of the system handling business logic and data access, has to be the same across all implementations.

To solve this matter we will separate the backend of our sample application into multiple modules with one common to all the others - the core module. The core must handle all data access and logic without exception, otherwise our experiments would become faulty. Other than core, there will be GWT, REST API, JSF and AspectFaces modules, providing an interface for communication with the front end.

3.2.3 Testing adaptive UI

While searching for a data interpretation tool we found Grafana[31], a data visualization and monitoring tool. This tool has another great feature for our experiments, though, and that is a configurable server or client-side rendering of output charts. Rendering interactive charts on the client may heavily use system resources and therefore result in greater energy impact. If rendered on the server, the client receives final chart as an image, thus there is no additional computation needed in the browser. We will perform an energy stress test with both server and client side rendering on data collected during our testings and evaluate the difference in energy consumption of the client device. The results of the tests present a valuable insight on possible employment of adaptive UI technologies in energy critical operations.

3.3 System resources utilization tracking tools

In the analysis chapter we came to the conclusion that we need a central synchronization element in our measurements, and that is the time. Meaning we have to sample system's resource utilization in particular instant, we need a bus-like mechanism that will control the measurement on a time basis. This does not concern the event based part of system, of course.

Since we have not found a universal solution providing all neccessary data we need, we have to build the tool ourselves. Earlier mentioned tools do not necessarily have the same interface on all Linux distribution and combining them could result in portability issues. Therefore we will build a simple program designed to collect system resources utilization samples. Following sections provide an overview of the program's design and its main features.

3.3.1 The performance analysis tool

As we presented in the analysis, all information we need to acquire is accessible on the *procfs* and *sysfs* virtual file systems. Therefore we need to build a program responsible for correct reading and collecting values from these sources. Further in the work we will also refer to the program as MWP. Fundamentally we have to focus on the time factor while reading the system statistics - to objectively evaluate the results, all information within one time iteration of the program has to be collected at a certain instant. A UML class diagram providing an overall insight on the core design is shown on figure 3.2.

There are many system resources that may possible influence energy impact and we do not focus on tracking all of them in our work, but MWP should be easily extensible in case of future usage.

All the collected data have to be exportable to a specified data format. For this purpose we will use a time series database (TSDB)[38], a database engine optimized for handling time data. We have chosen InfluxDB[37], an open source, high performance TSDB. A certain advantage of InfluxDB is that it is controled through RESTful API, therefore no additional dependencies or drivers are needed to integrate our measurement tool with the database except for an HTTP library, which is a core part of most commonly used languages. Furthermore this creates a possibility to keep all the acquired data outside of the client devices, being beneficial for distributing and collecting data from multiple clients at once without need for physical access to the device.

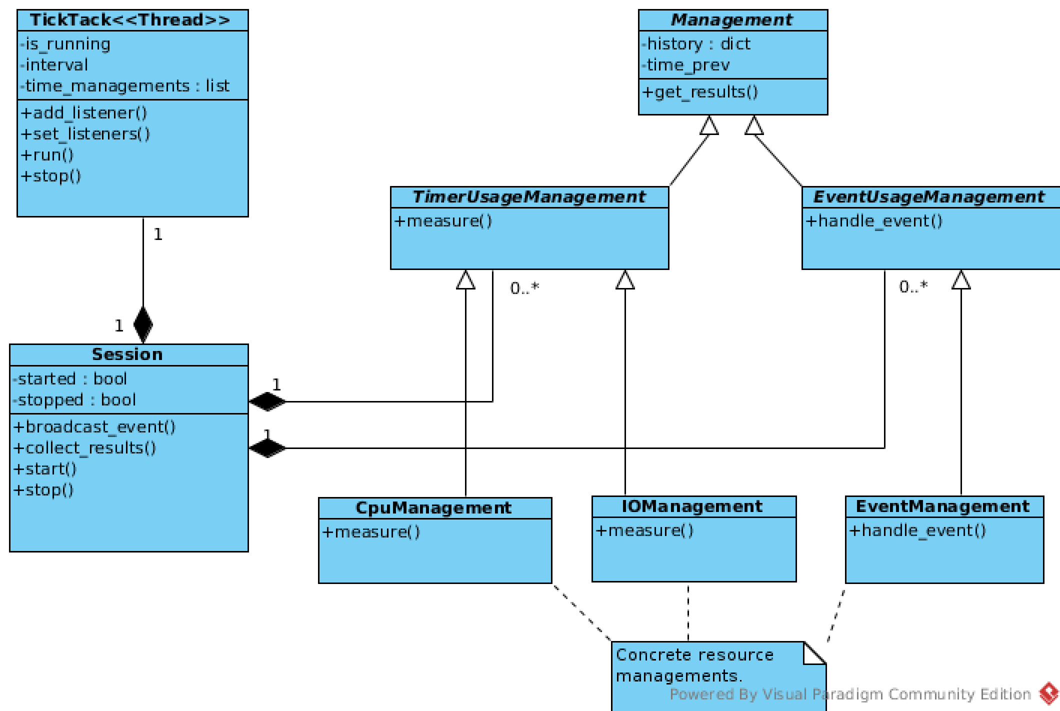


Figure 3.2: Class diagram featuring the core design blocks of our measurement tool.

Since we intend to test web applications, we also want react to certain events, that occur on the tested page. For this purpose we will create two different modes for running MWP: a server and CLI mode. In CLI mode the measurement will start immediately when the program is run, and listening to other events will not be possible. The server mode will run as a minimalistic HTTP server, awaiting either control or application events on predefined port. Control events are to start and stop measurement session, application events may vary, but generally are POST requests containing a predefined data structure as JSON payload. An example of such event may be page load time, sent to the MWP server asynchronously by the loaded page itself after it is rendered.

3.4 Infrastructure

For successful experiments we need at least two physical computer devices. The deployment diagram of used components is depicted on figure 3.3. First a mobile battery powered device (the client) and a server, which on which our sample web application runs. The client device has to be running a Linux based OS with kernel version newer than 2.6, Python3 interpreter and Google Chrome web browser (or its open source spin off Chromium).

We can further divide the server into additional components. These are:

- web server - serving static content, handling SSL etc.
- application server - running the application

- database - application data storage

As a web server we will use the Nginx high performance HTTP server. Nginx will handle static files serving, i.e. HTML, CSS and JavaScript files of client-side UI technologies, and will act as a reverse proxy for the application server. For our experiments we will not use encrypted connection to the server, but if future work requires such features, Nginx would handle the HTTPS protocol. Nginx is an open source project distributed under the terms of BSD-like license.

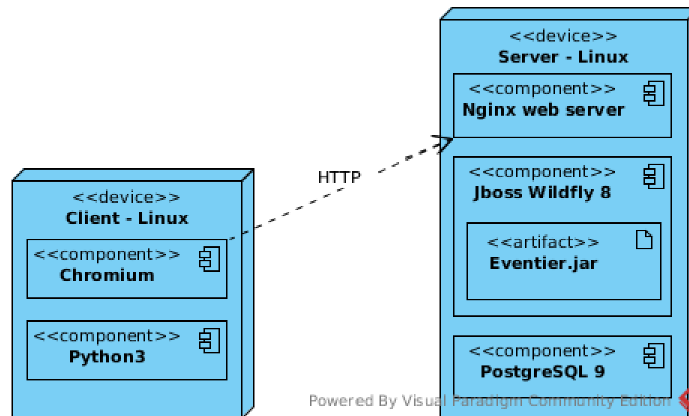


Figure 3.3: Deployment diagram featuring used components in our experimental setup.

The application server we chose for running tested application is Jboss Wildfly 8[53], a Java EE 7 compatible EJB container. Wildfly is an open source project distributed under the terms of LGPL 2.1 license.

We will use the PostgreSQL 9.5[39] as a database engine for our application, an open source technology distributed under the terms of PostgreSQL license, similar to MIT[11]. Since our backend stack is build primarily on Java platform, the application and database are connected through JDBC and Eventier does not leverage any PostgreSQL specific features, we are no limited to single database engine.

We would like to note that the deployment diagram featured on figure 3.3 is rather simplified, at least for the server. Each component on the server can run on a separate machine, there is no need that all are present on one server. This is important if we run our application in highly distributed environments or in a cloud using PaaS[54]. The description of a specific server setup will be provided for all our measurements later in the work.

3.5 Deployment environment

In our preliminary work[16] we have preformed the measurements on a rather resourceful server deployed on-premises with client connecting through 100Mbit/s LAN. Having on-premise server may become costly, at least from the initial costs point of view. Nowadays there are multiple public cloud providers allowing us to use their infrastructure and pay only the rental costs or even offer a free trial period before paying full subscription. For our

purposes we will use the deployment to the public cloud and connecting clients through the internet. Figure 3.4 features a diagram of the hardware and network setup we will use in our experiments.

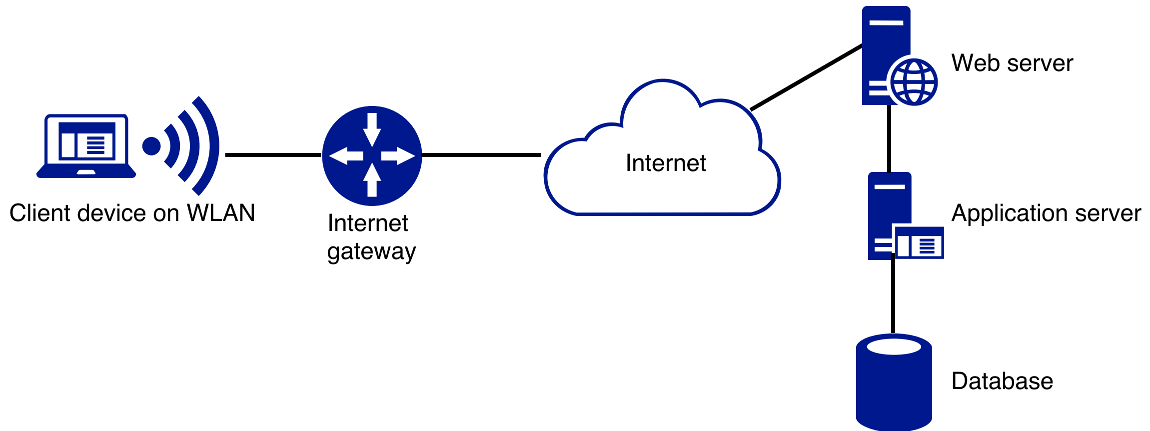


Figure 3.4: Hardware and network model of experimental setup.

3.6 Tested frontend frameworks

We chose from a variety of front end frameworks and libraries both cutting edge and industry proven technologies. Therefore for testing purposes we use:

- JavaServer Faces 2
- Google Web Toolkit
- AngularJS
- Angular 2
- ReactJS
- JavaServer Faces 2 with AspectFaces
- Angular 2 with AspectFaces

Following content of this chapters provides a short description and introduction into each technology used. For every technology we also provide a short description of development experience we gained through the implementation part of our work.

3.6.1 JavaServer Faces 2

The JavaServer Faces 2[41] (also known by abbreviation JSF) is a standard view technology of the Java EE stack, currently in version 2.2, specified in JSR 344. Since JSF is a standard, there are multiple implementations available, but for our purposes we use

the Mojarra, a open-source JSF implementation by Oracle. The JSF is an XML component server-side framework, allowing advanced templating, component composition and extension mechanism. It is a stateful technology by design, build on top of stateless HTTP protocol. The JSF supports multiple output formats, but our main interest is HTML.

3.6.1.1 JSF lifecycle

To grasp how JSF works internally, one has to understand the so-called JSF application lifecycle first, shown on figure 3.5. The lifecycle is a set of actions initiated by client's HTTP request for a resource (a web page), leading to rendering the final view as a response.

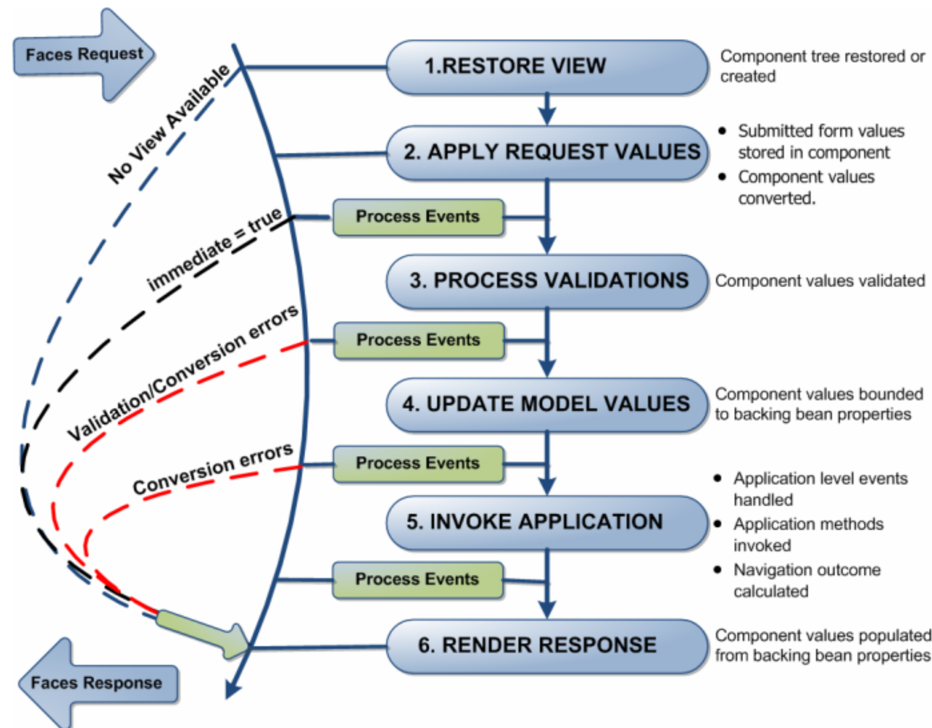


Figure 3.5: The JavaServer Faces application lifecycle

The first phase is *Restore View*, responsible for creating or restoring original component tree. The view can be restored only if it has been saved before our initial request, for example if the request was a submitted form, JSF has saved the previous component tree when user loaded the form page. If no view has been saved before, there is a new one created and all other phases but the last one are immediatelly skipped.

The following *Apply Request* values phase handles the extraction of request parameters and applying these to the component tree. If there are some value converters specified for given component, the conversions are also applied to decoded values in this stage.

The third stage, *Process Validations*, is responsible for validating processed view. All components with a JSF validator attached has their values checked. If there are some validation erros, JSF adds error messages and proceeds with rendering final response to client.

Next phase, *Update Model Values*, handles the updates made through view to the model in the backing bean. Therefore the component tree is traversed and all input's local values are set in model values according to their respective binding defined in the Facelet. If any error occures, JSF throws a conversion error and continues with rendering response.

The fifth phase, *Invoke Application*, handle internal application calls. For example if there is a method for execution defined after form submit, this method is synchronously executed in this phase. Also navigation outcome and is calculated, therefore it is possible to navigace to other pages only once the lifecycle successfully finished without any unexpected exceptions or errors.

The final phase is *Render Response*. This stage is responsible for filling the component tree values from its model in the backing bean. As the view is completely created it is sent back to the client as a HTTP response. The important part here is that the state of the response is saved so it is later accessible by following client requests and the view can be restored in the first phase.

AJAX[10] is also a part of the JSF specification since version 2.0. This allows partial view processing, thus the application lifecycle is also able to handle requests with just a subset of the component tree. Addition of AJAX to JSF is a great asset providing better user experience and a possible network traffic reduction if used properly, since there is no need for whole page re-rendering.

A great asset to the developer's experience is the statefulness of JSF, therefore application state and whole context is accessible on the server in one place. It is easy to develop and maintain JSF controllers (backing beans) and Facelets. Since the framework is server-side, the codebase does not suffer code duplication in stages of data validation or restatement of business logic on the client.

It is common practice to use some component library on top of plain JSF, for example PrimeFaces or RichFaces. Such libraries provide a large set of components and rich features out of the box. It is also possible to use scaffolding, such as Jboss Forge, a rapid application development tool that can generate JSF UI from application model.

The JSF HTTP communication relies heavily on POST and GET methods, thus it may become confusing for a web developer that DELETE and PUT is not used. Another issue to be concerned about is view fragment tangled together with page data. This means it is impossible to get application's data without its view counterpart, making system integration a challenging task.

3.6.2 Google Web Toolkit

Google Web Toolkit[42] (also known as GWT) is a client-side framework for Java with a distinctive approach to web development. Introduced in 2006 by Google, GWT provides a way to create applications fully backed by AJAX (single page applications) without actual need to use the standard web technology stack, that is HTML, CSS and JavaScript. As of today, the framework has reached version 2.8.

The main difference to other mentioned technologies is that the programming model of GWT is to write the application just as if it was a desktop rich client written in Java with a view library like Swing. The GWT library provides wrapper-like classes similar to standard

Swing components, such as buttons or input fields, which are then compiled to JavaScript by the GWT compiler. Cross-browser compatibility is solved by a mechanism call deferred binding. This feature generates multiple versions of result JavaScript code, each suitable for different client browser, such as Google Chrome, Safari, etc. The correct version of the script is then determined by client's browser on request and loaded in runtime. This feature is also enabled for language localization if the GWT Internalization module is used, for example there may be result scripts like "Safari in English" or "Safari in German" as a result of such build. As the documentation states[42], this should be beneficial maily to the client, since the resource size to download is much smaller then if it was full cross-browser and internalized batch of resources.

As was said earlier, GWT library's GUI components are similar to Java desktop frameworks like Swing or JavaFX. Once programmed and ready to build, the framework performs inspection of used components, which have a binding to HTML elements or JavaScript internally defined. This may seem like a limitation for look and feel customization, which GWT solves by the UiBinder feature[42]. The UIBinder is a XML-like templating mechanism to combine plain HTML with GWT components. Each UiBinder XML file is tightly bounded to a Java class, which wraps used GWT components as class fields decorated by mandatory metadata. The result of such approach is a simple, yet comprehensive and powerful creation of custom GWT UI widgets.

Listing 3.1: GWT's UiBinder template example.

```
<g:HTMLPanel styleName="row col-xs-offset-1">
  <div>
    <label class="col-xs-1 control-label">Start date</label>
      <div class="col-xs-2">
        <g:TextBox ui:field="startDateField"
           styleName="form-control"></g:TextBox>
    </div>
  </div>
</g:HTMLPanel>
```

Compared to JSF, GWT does not implicitly define a view-to-data binding. This matter is conveniently solved by the GWT Editor framework[42]. To use the editor, it is necessary to define auxiliary objects (Driver, Adapter and the Editor itself) handling data propagation. This object composition seems slightly difficult to grasp, but once used properly, this approach becomes concise, dynamic and much cleaner than manual extraction of values through components.

There are multiple protocols used for client-server communication. If used with a Java backend technology, the most reasonable way is remote procedure call, known as GWT RPC. GWT creates a standard Java servlet for all properly configured backend services, which handle the communication with client based on standard HTTP calls. The frameworks is responsible for managing both Java objects serialization and internal HTTP calls, leaving out the need to write such boilerplate code by the developer.

Another way to communicate with the server is manual data retrieval through HTTP, a convenient approach while using non-Java backend services. GWT provides handful tools to access the protocol and process its data, mainly supporting JSON and XML data formats.

All calls from client to server are asynchronous. GWT generates an asynchronous service instance for every GWT RPC Java service. Also manual HTTP calls require mandatory

callbacks to process server response. Therefore client-server communication relies on AJAX calls heavily, making GWT pages responsible and improving user experience.

Since the GWT compiler inspects code for translation to HTML or Javascript, the client part of the GWT module must not contain any Java objects without source code. This also applies to all imports of such classes, therefore on the client-side it is practically possible to use only imports from GWT project and basic POJOs. An example of such challenging integration with backend Java EE module may be the use of JPA entities throughout the view module. Even though an entity is a POJO with JPA specific annotations, GWT compiler requires source code even for the JPA implementation. This is unfortunate since it enforces either some kind of DTO[27] layer between backend and view module, or the necessity of publishing some kind of language agnostic API communicating through universal protocols, for example a RESTful web services through HTTP. Also dividing project into multiple modules requires each module to contain a GWT descriptor file, thus making code reuse through shared libraries difficult, since these would have to be polluted by GWT project dependencies.

The support for client-side unit testing combined with the robust strong typing of Java language gives GWT an upper hand when considered as a front end technology to use for environments, where changes occur frequently and regression testing is therefore a must.

3.6.3 AngularJS

AngularJS[6], simply Angular or Angular 1.x, is a complete JavaScript framework for building rich browser-based applications. Founded in 2009 by Google, it has been proven in production over the years and can be considered an industry standard for web applications front end development. Current version is 1.6. Angular emphasizes the decoration of HTML view fragments with JavaScript without manual DOM manipulation and provides a robust way to do so, comprehensible to JavaScript developers.

The preferred design pattern for Angular applications is Model-View-Controller. This means that separation of concerns plays a great role, simplifying both development and maintenance. Every part of the application is separately and easily testable. Controllers use vanilla JavaScript, thus one does not need to get familiar with neither some custom language obscurity nor with possibly complicated project setup. For HTML, Angular provides custom directives, allowing constructs such as loops or conditional rendering in view definitions. Such feature brings forth a powerful templating mechanism, significantly simplifying dynamic DOM manipulation based on specific controller and its data. The view and controller are bind together by the so-called "scope"[43]. Scope is available to the controller or template at any time of its use, providing access to the view layer and vice versa and eliminating any need for manual DOM manipulation.

Listing 3.2: For-each construct by a directive in AngularJS.

```
<tr ng-repeat="event in events">
  <td>{{event.name}}</td>
</tr>
```

Angular provides a two-way data binding, a mechanism providing a link between view data and controller data. In case either one of these changes, the other gets updated accordingly and automatically. This is mainly beneficial for applications with relatively simple

application state management, since all data manipulations are handled by the framework. Though it may also become a problematic matter for complex systems, where cascading model updates occur, resulting in possibly undesired view cascading updates.

Client side validation is included by default in the AngularJS distribution. Adding validation constraints to forms is performed in the HTML template by addition of element attributes. Creating custom validators is also supported out of the box. Of course validation is not required for Angular forms. Turning it off is done simply by appending the *novalidate* directive to the *<form>* HTML element in the template.

While interacting with forms in Angular, the framework checks whether there were some changes made, if fields are valid and so on. As a result, each form's input element has a CSS class assigned, corresponding to its state. For example if the input field has not been touched and its model is valid, it will implicitly have the *ng-pristine* and *ng-valid* CSS classes. If a modification on the field is made, the "ng-pristine" class is removed and "ng-dirty" class takes its place.

For client-server communication is prepared the "ngResource" module built on top of XHR. A simplified API and asynchronous execution guarantee comfortable access to HTTP based services while preserving user experience, since there is no need to wait for server response.

One of the key concepts in AngularJS is dependency injection. Built into the core and being a preferred way of decoupling the application code, the main benefit of using dependency injection lays in an ease of testability on both unit and integration levels. Combining Angular's own ngMock for services mocking with universal JavaScript testing libraries, such as Jasmine[44], Karma[45] or Mocha[46], we can build a full-featured testing framework assuring the application quality.

3.6.4 Angular 2

Angular 2 is a successor of the AngularJS framework, but completely rewritten and not backwards compatible with 1.x version. The main objective of Angular 2 is to solve the imperfections of previous version and fulfill the expectations of modern JavaScript toolset, be it bringing static typing to the client-side with TypeScript (on which the whole framework is based), featuring server-side rendering or performance optimizations. Announced in 2014 and reaching first release version in 2016, Angular 2 has emerged just recently, but seems to be gradually building its community and whole ecosystem.

Following its predecessor, Angular 2 follows the "batteries included" philosophy, meaning all necessary tools are included in standard distribution of the framework. As was said earlier, the preferred language to use for writing Angular 2 applications is TypeScript, a superset of ECMAScript6[47] specification, offering static typing, modularization and decorators, a feature similar to annotations in other languages, which Angular 2 uses heavily.

Angular 2 applications are clearly divided into modules based on language specification rather than custom mechanism, which was one of the downsides of previous version. Modules can be also optimized for loading, there is no need to load all resources at the time of application startup. This improves both testability and logical separation of the application into independent and possibly reusable pieces of code.

Listing 3.3: Example Angular 2 component.

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-component',
  template: '<div>Hello {{name}}. <button
      (click)="sayHello()">Hello</button></div>'
})
export class MyComponent {
  constructor() {
    this.name = 'Max'
  }
  sayHello() {
    console.log('Hello', this.name)
  }
}
```

Templating mechanism was also revised, bringing forth the composition of smaller, individual components, which contain a portion of application logic and respective part of a view. A component is a simple class with the *@Component* decorator added to its signature, specifying at least a component's unique name (called a selector) and full content of the template or a link to a template HTML file. Custom, Angular 2 specific, directives have been preserved, though there are syntactic diffrences to the previous versions. For example the *ng-iterate* directive has been changed for **ngFor*. The behaviour stays intuitive though, thus there should be no issues for AngularJS developers to get familiar with new directives.

3.6.5 ReactJS

ReactJS[48] (or React) is a declarative, component-based view library for JavaScript. Compared to other introduced client-side technologies, ReactJS is not a complete framework, but really handles just the presentation layer. Developed by Facebook, the library was introduced in 2013. ReactJS focuses on using plain JavaScript and JSX[48], an extension enabling the usage of HTML syntax for custom components and composition, which is translatable to pure JavaScript. The philosophy of ReactJS is to manage the HTML part of view by JavaScript, rather than decorating HTML with JS, mainly benefiting in keeping the application state out of the DOM.

One of the key concepts behind React is Virtual DOM, a DOM manipulation engine. The library keeps a pseudo copy of the DOM in memory, represented as pure JS objects. When a state changes, of course we have to propagate it to the DOM. React solves this by creating another copy of Virtual DOM and then computes differences between the old and the new one. Finally only the differences are propagated into the real DOM by browser optimized algorithm. This mechanism gets even powerful when used in combination with immutable data structures, such as the ImmutableJS library provides. The trick here is that immutable data cannot change, only the reference can, thus React does not need to perform full diff of data object properties, it is enough to know, that the data changed. Such feature also gives developers a simplified point of view for handling events, treating the view as if it was completely re-rendered.

React components are JavaScript sources containing both the logic and the view part. An example is shown in listings 3.4. Every component must inherit from *React.Component*

base class or be created by calling *React.createClass()* function. Final requirement final requirement for component is to have a *render()* method, which returns a JSX with exactly one root element - either plain HTML or another component. If a component is exported as a module, it can be reused in other components by its name without restrictions.

State management is also part of the library, but it seems to be a preferred way to handle application state with external tool, such as Redux[49] or MobX[50]. React's own state management is based on preserving only local state in components. Synchronizing state can become difficult while handling multiple nested components, which have the need to mutate this state. As a result, handling the local state may become a code heavily polluted by callbacks or hacks.

Listing 3.4: Example ReactJS stateless component.

```
// renders an unordered list of items passed as component properties
class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}
```

ReactJS is an easily comprehensible technology to embrace by any web developer, because apart from slightly customized syntax of JSX it is just vanilla JavaScript. It is fast, robust and independent of any other technology in front end stack. But there also lies the possibly negative side of using React - one has to build a custom front end technology stack, keep it up to date and ensure that all technologies in the stack work together. Furthermore it is a necessity to know all the technologies in the stack, not only React. Compared to a full-fledged framework like AngularJS, this definitely provides great flexibility, but might be time consuming to setup and maintain.

React's production distribution itself has only about 130kB of size, which is approximately four times smaller then Angular 2 - this may be an advantage for mobile devices accessing the network through slow connection.

3.6.6 AspecFaces

AspectFaces[13] is a Java-based context-aware tool for inspection, evaluation and transformation of Java beans model into a single user specified output. Leveraging both static and dynamic inspection, AspectFaces is a universal, comprehensive and extensible framework. Our main objective is evaluating the performance of various front end technologies, thus we will focus on the user interface generation. Currently there is support for dynamic (runtime) UI generation for JSF 2 and Angular 2, which we will try to evaluate.

While developing UI, one has to consider many aspects, for example:

- data type of the presented information - i.e. string or number - and conversion handling

- data binding - addresses the information binding to data model
- security - can some user access or see the information, i.e. RBAC[51]
- input validation - prevent corrupt data from application]
- presentation - final look and feel of the information
- composition - presented information is related or composed of other UI elements

While developing UI with today's standard technologies, we often run into a state where it is necessary to combine all these into a single output. Without using powerful templating tools, such requirements result into repetitive, tangled and verbose UI code. AspectFaces try to solve this issue in three main stages: inspection, transformation and finally code integration.

Inspection phase has an AspectFaces component as its input, performing Java Bean inspection and construction of meta-model. Such model, containing all separated and extracted aspects for inspected instance, is context-aware, because the original model can define additional security, presentation or other rules.

Transformation phase is the crucial part, since it processes the meta-model using the aspect approach, reusing all of the concerns and producing final UI code fragments. This stage has three subphases: Presentation rules, Template composition and Layout integration. The Presentation rules stage says how to bind data field from model to UI field. For example we may want to present string field in a form as a single input field or as a textarea based on chosen criteria. For such purposes it is possible to map UI fragment files to both Java types and even add conditional subselection of a fragment file based on variables derived from model using Expression Language (EL) as evaluation tool.

Template composition takes the output of the Presentation rules, which is a UI template selection, and evaluates its content in the context of instance meta-model. And example of such template can be seen in listings 3.5. The notable part of the listings is the code encapsulated by the "$" character. These expressions are evaluated in by AspectFaces in the context of instance meta-model. It is also possible to create template nesting - the *template* attribute of the *ui:decorate* element references another JSF template, which can also access the AspectFaces context and EL internal expressions.

Listing 3.5: Example AspectFaces form input template for JSF 2.

```
<ui:decorate xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://xmlns.jcp.org/jsf/html"
  xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
  xmlns:f="http://xmlns.jcp.org/jsf/core"
  xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
  template="#{empty template ? '/WEB-INF/af/profile/bootstrap/simple.xhtml'
      : template}">

  <ui:param name="id" value="#{prefix}$field$" />
  <ui:param name="value" value="#{$entityBean.shortClassName()$.$field$}" />
  <ui:param name="rendered" value="#{empty render$field.firstToUpper()$ ?
      'true' : render$field.firstToUpper()$}" />
  <ui:param name="required" value="#{empty required$field.firstToUpper()$ ?
      $required$ : required$field.firstToUpper()$}" />
  <ui:param name="maxlength" value="$maxLength$" />
```

```
<ui:param name="size" value="$size$" />
<ui:param name="readonly" value="#{empty edit ? false : !edit}" />
<ui:param name="label" value="$label$" />

<ui:define name="input">
  <h:inputText id="#{id}" type="text" required="#{required}"
    value="#{value}"
    class="form-control"
    readonly="#{readonly}">
  </h:inputText>
  <h:message for="#{id}" />
</ui:define>
```

Finally there is the *Layout integration* phase, an XSLT[52] inspired feature allowing to further integrate transformed final UI fragments into layouts by providing support for looping and attribute extraction.

The main objective of AspectFaces is to reduce such restatements, improve system's maintainability and provide rapid development of UI without compromises in terms of features or limiting used front end technology. According to the authors of the framework, there is negligible performance impact while using AspectFaces with JSF 2[13].

Just like JSF tag libraries or other UI extensions in different languages, AspectFaces components can be easily reused between multiple projects. This way one can rapidly build application UI once the component library is built.

It is necessary to get familiar with the framework and its documentation before usage, though. Building custom UI component templates for AspectFaces might be time consuming at first, but it should still save up to 25% of UI development time[13].

Chapter 4

Validation of our measurement tool

This chapter presents a validation and accuracy of our measurement tool, MWP. Since we created our own prototype, we need to prove it provides results of sufficient quality to be used not only for our experiments, but also that it is generic enough to be used as a foundation for future works on performance analysis for Linux-based systems.

Compared to other presented works on the topic [17] [19] [20] we have not used a wattmeter to obtain actual power usage, but rather rely on kernels ACPI interface information about power status of the device. Therefore our validation technique focuses on statistical comparison to previously done research [19] [20] [22], which proved to have at least 95% accuracy.

4.1 Validation method

Our validation method is similar to mentioned related works, targeting extended empirical measurements and tracking system resources utilization and energy consumption. We focus on measurements of individual resources and total power impact on the device.

We focus on tracking all resources MWP features:

- CPU
- RAM
- Disk I/O
- Network traffic (on WiFi)
- Backlight

As [20] suggests, backlight is a static element in the analysis, independent of all other resources, therefore even if its intensity has influence on power consumption, it should always be static. The difference in consumption while displaying different color scale [20] is discarded here by running all tests in black terminal without additional interactions with the device.

We have designed a set of tests for each system resource, creating excessive load while trying to spare usage of other system components. The only exception is for disk I/O test, where the OS tries caches a lot and uses as much memory as possible.

System Resource	Resource load	Charge taken[%]	Power[W]
CPU	99.99%	78.18%	13.59
memory	7631.5MB	37.52%	6.66
disk read	738756.37MB	50.98%	9.04
disk write	67836.9MB	58.15%	10.01
network traffic (WiFi)	5648.05MB	38.41%	6.83
backlight	100%	47.71%	7.86

Table 4.1: Energy impact of excessive system resource usage during 3 hours stress tests

4.1.1 Laboratory setup

In our measurements we ran all tests on Dell Vostro 14 laptop as a client, having processor Intel(R) Core(TM) i3-4005U running at 1.7GHz with 2 cores / 4 threads, 8GB DDR3 RAM at 1600MHz in a single module, and Samsung SSD 850 hard drive. The laptop is running Ubuntu Linux distribution v16.04 with stock kernel 4.4.0-45-generic.

We have performed 3 hours long session for each resource to diminish possible performance peaks and statistical error during measurements. This also enables us to evaluate possible nonlinearities in battery discharge characteristics, which we have to take into account.

Before every test we performed a hard reset of the device, stopped all unimportant system services on the client and flushed system's in-memory caches. The backlight level stays the same for all tests except for its own, where we focus on energy drawn by full intensity.

4.2 Results evaluation

Table 4.1 presents the results the system resource utilization has on power consumption of our tested device. The results clearly show that the CPU is the highest power consumer with 78.18% of the charge taken while operating fully active on all cores and threads. Another major factor we have to consider are disk reads and writes, which respectively consumed 50.98% and 58.15% of the battery.

4.2.1 Evaluation and summary

Our findings are similar to [19] [20], except for the memory usage, which related works suggest to be on equal power terms with disk operations. This state is given because of the fact that there is only one memory installed module in our tested device. According to installed memory datasheet[1], the maximum operating power is 2.7W. Therefore if we were to install additional module, the power consumed during the tests would have risen approximately to the level of disk I/O.

Next we have to consider potential benefits and threats to resource usage from web application's point of view. High CPU and memory can be easily generated by the browser,

[1] <https://www.kingston.com/dataSheets/KVR16LS11_8.pdf>

mostly at applications initialization stage or while using graphics client-side technologies (i.e. graph plotting, drawing tools). Network traffic in units of gigabytes is also not a standard feature or web systems and is rather an issue of big file transfers. Disk I/O power consumption seems to be a problem, but if we consider relative per-hour rates of usage, that is over 200GB read and 20GB written, we dare say that there is not a single existing web application or browser that can generate such huge amount of data on the client's device. We will provide a further look on total disk I/O in our case studies from a real application view.

To summarize, our tool, MWP, which is based on reading values provided from kernel interface, provides sufficient precision and is comparable to tools other researchers [22] have used and proved their quality.

Chapter 5

Case study: Eventier

In this chapter we present a performance and power consumption case study on our sample application, Eventier, featuring functionality described in 3.2. We have reimplemented the application's UI in all presented technologies3.6 with focus on preserving the UX (UI design, process flow) and achieving desired use cases. We compare the implementations in two scenarios: a single and multiple clients accessing the server while repeating desired use case by a Selenium drone.

5.1 Deployment and testing environment

The client device is Dell Vostro 14 laptop, the same as featured in 4. The client is running Google Chrome 53. It is connected to the network through WiFi, connectivity bandwidth is 10Mbit/s for download and 720kbit/s for upload. The server is deployed to Amazon EC2 t2.micro instance[1], having Intel Xeon CPU E5-2676 v3 processor (@2.40GHz), 1G RAM (frequency not provided) and General Purpose SSD drive. The server is placed in West Europe region (Frankfurt). The OS is Ubuntu Server v16.04. On the server we used the Nginx as web and Jboss Wildfly 8 as application server as described in 3.4. Wildfly runs on Java 8 with following parameters: *-Xms256m -Xmx512m -XX:MaxMetaspaceSize=256M -Djava.net.preferIPv4Stack=true*. Nginx is running with default configuration.

PostgreSQL database runs on a separate machine, also in West Europe region (also Frankfurt), running on Amazon RDS db.t2.micro[2] instance. The database runs with default configuration, which should be optimized for machine's resources.

5.2 Network traffic comparison

In this section we explore the differences in network traffic for every UI technology. Taking browser caching into consideration we focus on four basic scenarios: first page load, searching for events with 15 items displayed per page, loading event edit page and submitting event form. The results are shown in table 5.1.

[1] <https://aws.amazon.com/ec2/instance-types/>

[2] <http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.DBInstanceClass.html>

Technology	Initial[kB]	Search[kB]	Edit[kB]	Sumbit[kB]
AngularJS	533	22.2	5.5	1.9
ng2 + AF	4200	24	7.4	1.9
ReactJS	918	22.2	5.5	1.9
GWT	580	1.7	1.2	1.1
JSF	381	9.5	4.2	9.7
JSF + AF	381	9.5	4.2	9.7

Table 5.1: network traffic comparison for UI technologies.

As we can see, there are many similarities in the results. Client side JS based technologies, separating view from data, transfer the same amount of data for test cases except for initial load. Also Angular2 has 1.8kB per some requests for form data model - an additional feature for UI generation provided by AspectFaces. GWT heavily relies on its own data transfer protocol and *gzip* encoding, making it the winner of our tests. JSF in both cases performs well on GET requests (reading data from server), but clearly has the highest payload when submitting the form for server processing. In interesting observation is that JSF needs more than twice smaller data to search events. That is because our sample events have a *description* field containing a long *Lorem ipsum* text, which is shortened to 50 characters on all implementations. And because JSF renders on the server, it is saves unnecessary traffic.

The initial network communication is another issue we need to address. The more data is needed at startup, the slower does the page load. Clearly, Angular 2 needs the larges amount of data (a compiled bundle consisting of all JS)[3]

5.3 Single client performance

As was mentioned earlier, we are interested in the speed of battery discharging and the number of tasks the user is able to successfully perform. To test both JSF and AJS clients we took 1 hour long measurement sessions, running the use-cases described in3.2.1. Tests were executed by a Selenium drone. The system resource utilization results are shown in table 5.2. Backlight is omitted as it was set to a constant value for all tests. Columns Read and Write refer to disk I/O. Next, table 5.3 presents number of completed use cases in the time of one hour and a relative battery charge loss.

The results clearly show the browser does aggressive caching[3], saving file onto the file system. Here GWT is greatly efficient, since its JavaScript compiler focuses on browser optimization of by the deffered binding[42]. Also network communication greatly benefits from the *gunzip* encoding, being nearly 4 times smaller compared to second more efficient JSF.

[3]We have built Angular2 implementation according to original tutorials on <angular.io>, used preconfigured project through *angular-cli* and minimized final build with *uglifyjs*. There is an advanced technique we have not tried for size reduction, called *tree shaking* <https://angular.io/docs/ts/latest/cookbook/aot-compiler.html>, which may have a positive impact on size optimizations.

Technology	CPU[%]	Mem	Read	Write	Network
AngularJS	17.81	1834.60	73.86	183.79	59.31
Angular2 + AF	17.32	1733.91	109.77	212.42	70.48
ReactJS	17.64	1942.05	37.21	179.89	51.32
GWT	16.64	1956.77	2.04	41.32	11.49
JSF	19.78	1662.31	6.58	191.69	41.39
JSF + AF	17.98	1586.10	25.14	188.22	40.20

Table 5.2: Performance utilization consumption comparison for UI technologies. Single client accesses the application. All values are in MB if not stated otherwise.

Technology	Tasks done	Charge taken[%]
AngularJS	823	18.03
Angular2 + AF	844	18.08
ReactJS	799	19.10
GWT	810	17.21
JSF	697	18.72
JSF + AF	695	17.82

Table 5.3: Power consumption comparison and performed tasks statistics for UI technologies. Single client accesses the application.

Another interesting fact is that client side technologies showed higher network traffic then server-side JSF. As was said earlier, this is partly due to data processing of the search results on the server.

The battery drain in the 1 hour long testing session ranges around 2% difference between the most efficient GWT and most consuming ReactJS.

From the task completion point of view, Angular2 was the most successful technology, with 844 use cases done. We can generally say that client-side technologies prevail in the number of use cases done. Server-side technologies have a clear disadvantage here because of successive page rerendering. This is not an issue on high bandwidth networks[16], though.

5.4 Multiple clients performance

Next we are interested in UI performance during higher server load. In this test session we have run the Selenium drone by another 6 clients on a separate machine in through GNU parallel[15] and observed the power and performance on our client. The results are shown in table 5.2.

For higher server load we a slight decrease in completed tasks compared to previous single client tests, which was expected. During the this tests, however, the tested devices shows a slight increase in energy demands of all used technologies.

Technology	CPU[%]	Mem	Read	Write	Network
AngularJS	18.29	1902.62	47.74	186.51	55.96
Angular2 + AF	17.46	1713.47	83.35	215.42	69.61
ReactJS	17.41	2019.73	52.03	191.88	52.30
GWT	16.66	1903.57	36.88	42.86	11.81
JSF	19.50	1639.58	44.29	187.76	40.61
JSF + AF	18.16	1705.84	55.47	232.01	38.20

Table 5.4: Performance utilization consumption comparison for UI technologies. Multiple clients access the application. All values are in MB if not stated otherwise.

Technology	Tasks done	Charge taken[%]
AngularJS	806	18.85
Angular2 + AF	823	19.43
ReactJS	800	19.50
GWT	808	17.60
JSF	680	19.09
JSF + AF	666	19.08

Table 5.5: Power consumption comparison and performed tasks statistics for UI technologies. Multiple clients access the application.

Collected data for multiple clients test correlates with single client, there are no major differences in either system resources utilization or power consumption.

5.5 Summary

In comparison to our initial work[16] we have deployed our test application to a highly distributed and open environment and expanded tested set of UI rendering technologies. Our results show that there are clear benefits of UI code optimizations for specific browser technology (GWT) compared to the universal and portable one. Using *gunzip* encoding of client-server communication also results into much lower network traffic, at least for certain types of web applications. It may conflict with universal RESTful approach of providing data, because not all possible clients may be able to decode server responses, even though major web browsers support this feature.

Compared to our initial work[16], which focused on testing applications on high-throughput intranet we have shown the importance of network connection quality (bandwidth) and server-deployment to the cloud to the UI rendering technology. Successive server dependend page rerendering becomes an obstacle here for application performance, as the number of completed user tasks is significantly lower for server-side technologies.

On the other hand, the energy impact does not vary much between client-side and server-side technologies for applications not relying on intensive CPU usage. However there is a

clear difference in memory consumed while client-side technology is used, which might be an issue for devices with limited system resources.

Chapter 6

Case study: Grafana and adaptive UI

In this chapter we present a short case study on adaptive web applications, supporting optional UI rendering on both client and server side. For our purposes we use Grafana3.2.3, an open source web application for visualizing a collected data, mainly into graphs. Grafana offers plot rendering in the browser and also rendering to PNG on the server using *PhantomJS*, a headless scriptable browser. Both modes are configured by default and are easily accessible through unique URLs.

6.1 Test setup

Grafana has integration with InfluxDB, the time-series database we used for collecting performance and power management data in Eventier case study. Through its UI, it is simple to create dashboards of tables, graphs and many more widgets, which query connected database. We have designed a plot showing captured usage of CPU, memory and discharge characteristics, consisting of 15021 nodes for each entity, therefore 45063 nodes to render in the plot in total.

Grafana server is deployed to Amazon EC2 t2.micro instance with the same configuration as presented in5. The client is also the same Dell Vostro 14 laptop.

The test objective is to render given plot 30 times in a row. Considering time factor, our test case is much shorter than in the Eventier case study, but since we have already verified energy consumption model, therefore we will focus solely on tracking system resources utilization.

6.2 Experimental results - graph plotting

The results of our experiments of client and server rendering performance are provided in table 6.1. Evaluating the data clearly proves that resources were drastically saved when plotting was rendered on the server. Graphical demonstration of CPU and memory usage by Grafana, which we collected during client and server rendering, in shown on figures 6.1 and 6.2.

Renderer	CPU[%]	Mem	Read	Write	Network
client	7.65	1759.14	37.93	12.98	3.50
server	30.93	1851.32	42.91	31.23	16.75

Table 6.1: Performance utilization while rendering Grafana plots. Values are in units of MB if not stated otherwise.

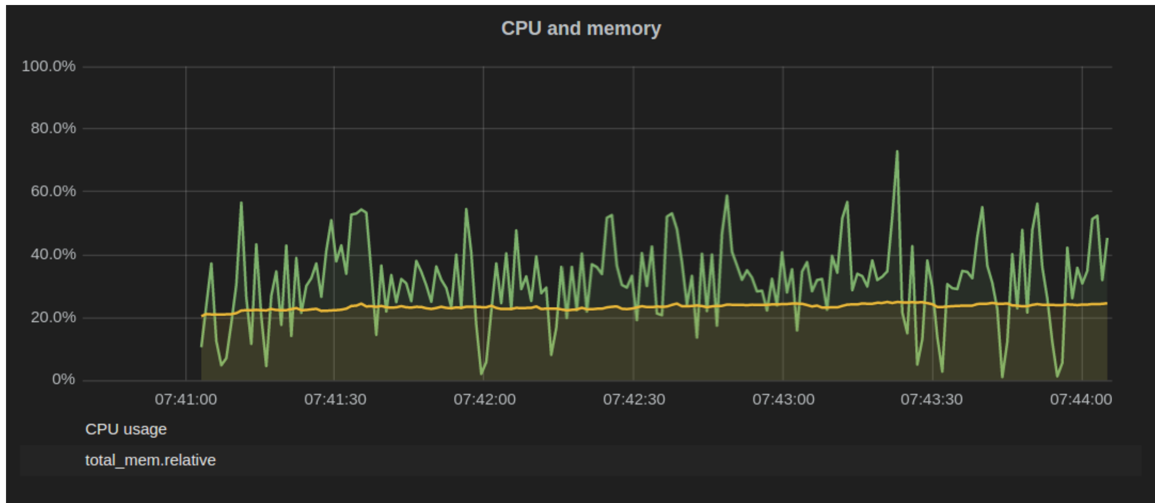


Figure 6.1: CPU and memory usage for browser rendering.

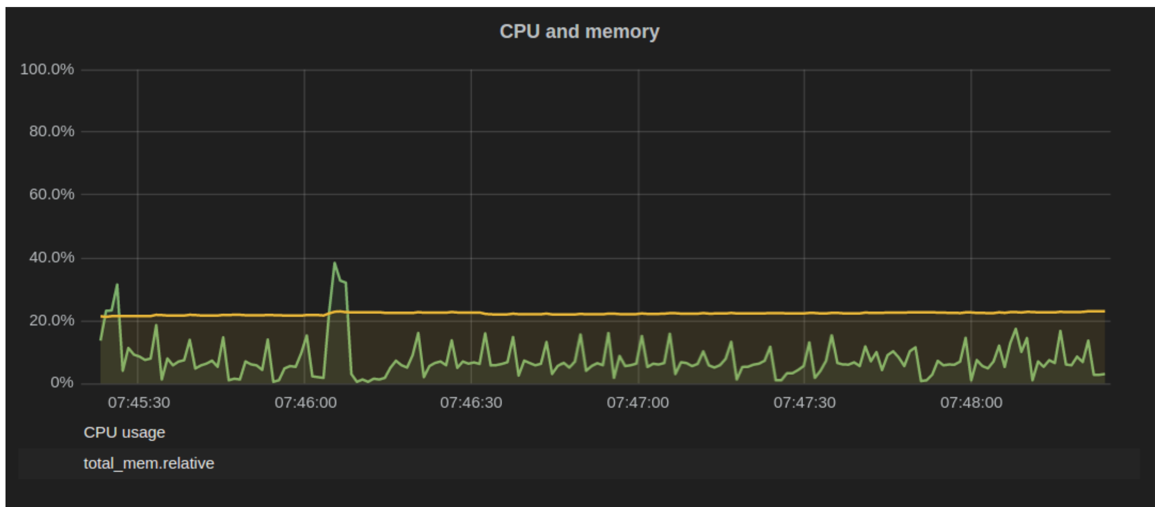


Figure 6.2: CPU and memory usage when rendering on server.

6.3 Summary

From the point of energy impact on a mobile device, adaptive UI may be quite beneficial. As we have shown, there are major savings in the usage of system resources, which positively influence battery life. Comparing collected data to our Eventier case study however suggests, that moving browser functionality to server is probably efficient only for intensive computations, otherwise there not such difference in energy consumption.

Chapter 7

Conclusion

The development of UI technologies for contemporary web applications can be basically divided into two worlds - a client and a server-side. Both approaches have their benefits and shortages in certain environments and use cases. In this work we focused on the impact of UI rendering technologies for the web on power consumption of the device. We have presented an approach for measurements of system resources utilization and energy consumption for computer devices running Linux OS. We have implemented and validated a prototype of such tool.

Featured case study on our sample application addresses the influence a used technology has on client device. Analyzing the state of integral computer components, the experiments have proven that for conventional web applications there is only a little difference in energy impact when deployed to the open network. However it is clear that deployment environment plays a major role in final application performance, considering number of completed use cases in a certain time interval.

Another interesting fact we have found is the impact of browser-specific optimizations of the application's source on the client. In our study only GWT has such feature, which spared a considerable amount of client's system resources. Also encoding communication between client and server is beneficial for network traffic, reducing transferred data approximately 4 times. Though we believe that this optimization is not generically useful for all web applications and should be cautiously considered.

The second presented case study explores possible advantages of adaptive UI on power consumption. We have conducted tests on widely used monitoring tool Grafana. The results prove that delegating certain part of UI rendering to the server may lead to major difference in consumed energy.

7.1 Future work

In our future work we would like to generalize and refine our measurement tool prototype into a standard program compatible with all major Linux distributions and successfully porting it to Android devices.

We believe that the subject of adaptive UI technologies and its impact on energy consumption deserves more attention from both the community and academical sphere, since it

possesses great promises in expanding the users group of computation critical applications of new mobile users.

7.2 Summary

The aim of the work was to explore the field of energy consumption of web applications, focusing on the impact the view technologies presents. We have based our methodology on acknowledged related works and applied the topic onto the area of web applications. We have fulfilled desired requirements, pointing to the pros and cons of presented UI technologies and their architectural designs, created a measurement tool prototype and introduced the topic of adaptive UI in relation to energy consumption.

Bibliography

[1] Tomas Cerny, Michael J. Donahoo. *Impact of Remote User Interface Design and Delivery on Energy Demand.* 2nd International Conference on Information Science and Security (ICISS) 2015, pp. 1-4, doi:10.1109/ICISSEC.2015.7371005.

[2] Cerny, T.; Macik, M.; Donahoo, M.J.; Janousek, J., "Efficient description and cache performance in Aspect-Oriented user interface design," Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on , vol., no., pp.1667-1676, 7-10 Sept. 2014 doi: 10.15439/2014F244

[3] Liu, X.; Ma, Y.; Liu, Y.; Xie, T.; Huang, G. Demystifying the Imperfect Client-Side Cache Performance of Mobile Web Browsing Mobile Computing, IEEE Transactions on, vol.PP, no.99, pp.1-1, doi: 10.1109/TMC.2015.2489202

[4] Mashup (web application hybrid), Wipedia, the free encyclopedia [online], from : <https://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)>

[5] Hypertext Transfer Protocol, Wipedia, the free encyclopedia [online], from : <https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol>

[6] Brad Green and Shyam Seshadri. *AngularJS.* O'Reilly Media, Inc. 1 sedition, 2013.

[7] W3C, browser statistics [online], November 2016, from : <https://www.w3counter.com/globalstats.php?year=2016&month=11>

[8] Representational state transfer, Wipedia, the free encyclopedia [online], from : <http://en.wikipedia.org/wiki/Representational_state_transfer>

[9] Document Object Model, Wikipedia, the free encyclopedia [online], from : <http://en.wikipedia.org/wiki/Document_Object_Model>

[10] Asynchronous JavaScript and XML, Wikipedia, the free encyclopedia [online], from : <http://en.wikipedia.org/wiki/AJAX>

[11] Permisive free software licence, Wikipedia, the free encyclopedia [online], from : <http://en.wikipedia.org/wiki/MIT_License>

[12] JavaScript Object Notation, [online], from : <http://json.org/>

[13] CodingCrayons, AspectFaces wiki page [online], from : <http://wiki.codingcrayons.com/display/af/AspectFaces>

[14] Kennard, R., Edmonds, E., Leaney, J.: Separation anxiety: stresses of developing a modern day separable user interface. In: Proceedings of the 2nd conference on Human System Interactions, HSI'09, pp. 225–232. IEEE Press, Piscataway, NJ, USA (2009). <http://portal.acm.org/citation.cfm?id=1689359.1689399>

[15] Tange, O. GNU Parallel - The Command-Line Power Tool The USENIX Magazine, pp. 42-47, Feb. 2011, vol. 36, <http://www.gnu.org/s/parallel>

[16] Jan Helbich; Tomas Cerny 2016 6th International Conference on IT Convergence and Security (ICITCS), 2016, pp. 1 - 3, DOI: 10.1109/ICITCS.2016.7740329

[17] E. Capra; F. Merlo Green it: everything starts from the software, 2009

[18] Sticky sessions, [online] from : <https://www.nginx.com/products/session-persistence>

[19] Minas, L., Ellison, B.: The Problem of Power Consumption in Servers. Hillsboro;Intel Press (2009)

[20] Carroll, A., Heiser, G.: An analysis of power consumption in a smartphone. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, pp. 21–21. USENIX Association, Berkeley, CA, USA (2010). <http://dl.acm.org/citation.cfm?id=1855840.1855861>

[21] N. Balasubramanian; A. Balasubramanian; A. Balasubramanian: Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. Published by ACM 2009 Article. doi: 10.1145/1644893.1644927

[22] Vivek Kumar Singh; Kaushik Dutta; Debra VanderMeer: Estimating the Energy Consumption of Executing Software Processes. In: Green Computing and Communications (GreenCom), 2013 IEEE DOI: 10.1109/GreenCom-iThings-CPSCom.2013.40

[23] ČERNÝ, T. and M.J. DONAHOO. On Energy Impact of Web User Interface Approaches. Cluster Computing. 2016, 19(73), 1-11. ISSN 1386-7857. <https://link.springer.com/article/10.1007/s10586-016-0665-7>

[24] Irene Manotas; Cagri Sahin; James Clause; Lori Pollock; Kristina Winbladh Investigating the impacts of web servers on web application energy usage. In: Green and Sustainable Software (GREENS), 2013, IEEE DOI: 10.1109/GREENS.2013.6606417

[25] Ruby on Rails, [online] from : <http://rubyonrails.org/>

[26] (2016, April) The PowerTOP website. [Online]. Available: https://01.org/powertop/

[27] Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley Professional, 2003.

[28] Chromium multithread architecture, [online], from : <https://blog.chromium.org/2008/09/multi-process-architecture.html>

[29] Mozilla Developers Network, Battery Status API, [online] from : <https://developer.mozilla.org/en-US/docs/Web/API/Battery_Status_API>

[30] CRUD, Wikipedia, The free encyclopedia, [online] from : <https://en.wikipedia.org/wiki/Create,_read,_update_and_delete>

[31] Grafana, Beatiful Metrics Dashboard, [online] from : <https://grafana.org/>

[32] Unix top, [online] from : <www.unixtop.org/>

[33] Linux meminfo, [online] from : <http://lxr.free-electrons.com/source/fs/proc/meminfo.c>

[34] Everything is a file, Wikipedia, The free encyclopedia, [online] from : <https://en.wikipedia.org/wiki/Everything_is_a_file>

[35] Linux kernel, [online] from : <https://www.kernel.org/>

[36] Protractor E2E test framework, [online] from : <https://github.com/angular/protractor>

[37] InfluxDB, [online] from : <https://influxdata.com>

[38] Time-series database, Wikipedia, The free encyclopedia [online] from : <https://en.wikipedia.org/wiki/Time_series_database>

[39] PostgreSQL database, [online] from : <https://www.postgresql.org/>

[40] Nicholas Nethercote; Insight on Mac Activity Monitor Energy Impact, [online] from : <https://blog.mozilla.org/nnethercote/2015/08/26/what-does-the-os-x-activity-monitors-energy-impact-actually-measure/>

[41] Ed Burns and Neil Griffin. *JavaServer Faces 2.0, The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[42] Robert Hanson and Adam Tacy. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications Co., Greenwich, CT, USA, 2007.

[43] Scope, AngularJS documentation [online] from : <https://docs.angularjs.org/guide/scope>

[44] Jasmine JS, [online] from : <https://jasmine.github.io/>

[45] Karma JS test runner, [online] from : <https://karma-runner.github.io/>

[46] Mocha JS Test framework, [online] from : <https://mochajs.org/>

[47] ECMAScript 6 specification, [online] from : <es6-features.org/>

[48] React JS view library, [online] from : <https://facebook.github.io/react/>

[49] Redux JS, state management library, [online] from : <redux.js.org>

[50] MobX JS state management library, [online] from : <https://mobxjs.github.io>

[51] Role-based access control, Wikipedia, The free encyclopedia, [online] from : <https://en.wikipedia.org/wiki/Role-based_access_control>

[52] Extensible Stylesheet Language Transformations, Wikipedia, The free encyclopedia, [online] from : <https://en.wikipedia.org/wiki/XSLT>

[53] Jboss Wildfly Java EE 7 AS, [online] from : <https://wildfly.org>

[54] Platform as a Service, Wikipedia, The free encyclopedia, [online] from : <https://en.wikipedia.org/wiki/Platform_as_a_service>

[55] Electrolysis for Firefox, [online] from : <https://wiki.mozilla.org/Electrolysis>

[56] WebKit, Open Source Browser Engine, [online] from : <https://webkit.org/>

Chapter 8

Eventier - installation guide

The Eventier sample application is based on the Java 8 platform, therefore as a first step please install the latests Java, and is built by Apache Maven. JS based frontend technologies rely on NodeJs.

For up to date Linux distributions, you can install OpenJDK and other dependencies through your favourite package manager, for example for Debian based systems run:

#> sudo apt-get install openjdk-8-jre mvn node npm

Backend modules running Wildfly AS are *eventier-jsf, eventier-af-jsf, eventier-rest, eventier-af-rest, eventier-gwt*. To install these modules do to their respective directories and run:

#> mvn clean install -DskipTests=true

To quickly run the application, it is convenient to skip the test configuration, for which is the parameter -DskipTests.

Client-side modeles are *eventier-react, eventier-web, eventier-af-ng2*. The projects are NodeJs based. To build and the project, we suggest you first inspect the "scripts" property of "package.json" file in the root of each project.

Installing dependencies is done by running:

#> npm install

Running the application through independent Node server is usually done as:

#> npm run start

Eventier backend requires a running instance of PostgreSQL 9.3+ database on localhost (by default), with preconfigured database 'eventier_dev' and user credentials 'postgres:postgres'.

Chapter 9

Attached CD

```
.
├── eventier.............................Eventier directory.
│   ├── dist.............................Distribution packages of servers.
│   ├── eventier-af-jsf
│   ├── eventier-af-rest
│   ├── eventier-core
│   ├── eventier-gwt
│   ├── eventier-jsf
│   ├── eventier-ng2
│   ├── eventier-parent
│   ├── eventier-react
│   ├── eventier-rest
│   ├── eventier-web
│   └── makescript.sh...................Make script for server-side and GWT.
├── helbich_dp.sha1.....................SHA1 Hash of the thesis PDF.
├── helbich_dp.pdf......................Thesis in PDF.
├── mwp.................................MWP measurement tool.
│   ├── pymwp...........................MWP source code.
│   ├── README.md.......................Installation guide.
│   ├── scripts.........................Helper scripts.
│   └── setup.py........................Setup and install script.
└── thesis..............................Thesis LaTeX source.
    ├── buildtex.sh.....................Build script.
    ├── figures.........................Used figures.
    ├── helbich_dp.tex..................Thesis main body.
    ├── hyphen.tex
    ├── k336_thesis_macros.sty..........Template.
    └── texts...........................Thesis content.

18 directories, 9 files
```

Figure 9.1: Directory tree on attached CD