Assignment of bachelor's thesis

Title:	Comparison of Apache JSP and React for Online Books/Comics Reader Implementation
Student:	Kyrylo Ponomarov
Supervisor:	Ing. Marek Suchánek
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Single-Page Application (SPA) is gradually replacing traditional server-side template rendering technologies in web applications. However, SPA also brings with it a number of drawbacks that must be eliminated using other techniques such as Server-Side Rendering or Progressive Web Apps. The goal of this work is to develop a system for online book/comic reading and to thoroughly compare the use of traditional JSP and SPA technology in React:

- Analyze and describe JSP and React (and possibly other related) technologies.
- Research existing comparisons of traditional and SPA approaches to frontend web application development.
- Establish metrics and a process for comparing the two approaches.
- Specify functional and non-functional requirements for an online book/comic reader according to existing similar and widely used systems.
- Design an information system that meets the requirements. Divide the architecture into a backend with a Java API and two frontend applications (JSP and React).
- Implement according to the design and test the resulting application. Both frontend applications should provide identical functionality and appearance.
- Compare the approaches according to the specified metrics and summarize the resulting findings.

Electronically approved by Ing. Michal Valenta, Ph.D. on 17 November 2022 in Prague.

Bachelor’s thesis

COMPARISON OF APACHE JSP AND REACT FOR ONLINE BOOKS/COMICS READER IMPLEMENTATION

Kyrylo Ponomarov

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Marek Suchánek.
May 10, 2023

Contents

Acknowledgments vii

Declaration viii

Abstract ix

Introduction 1

1 Analysis 3

1.1 Server-Side Rendering . . . 3
1.2 Client-Side Rendering . . . 3
1.3 Existing comparisons of SSR and CSR . . . 5
1.3.1 Comparison between client-side and server-side rendering in the web development [2] . . . 5
1.3.2 On the Comparison of Software Quality Attributes for Client-side and Server-side Rendering [4] . . . 6
1.3.3 Summary . . . 7
1.4 Online books/comics reader system definition . . . 7
1.4.1 Requirements . . . 7
1.4.2 Use cases . . . 8
1.4.3 Domain model . . . 9
1.5 Metrics definition . . . 9
1.5.1 Metrics from the client's perspective . . . 9
1.5.2 Metrics from the server's perspective . . . 9

2 Design 11

2.1 Wireframes . . . 11
2.2 Server architecture . . . 11
2.2.1 API design . . . 11
2.2.2 Back-end application design . . . 15
2.3 Architecture of React applications . . . 15
2.3.1 React Components . . . 15
2.3.2 React Props . . . 16
2.3.3 React State Hook . . . 16
2.4 Architecture of Struts2 applications . . . 17
2.4.1 Struts2 Models . . . 17
2.4.2 Struts2 Views . . . 18
2.4.3 Struts2 Controllers . . . 20
2.5 Tools used for metrics collecting . . . 20

3 Implementation **23**
3.1 Server implementation . . . 23
3.1.1 Technology decisions . . . 23
3.1.2 Persistence layer . . . 23
3.1.3 Application layer . . . 24
3.1.4 Presentation layer . . . 24
3.2 React implementation . . . 28
3.2.1 Technology decisions . . . 28
3.2.2 Directories structure . . . 29
3.2.3 API components generation . . . 29
3.2.4 React components pre-rendering . . . 29
3.3 Struts2 implementation . . . 29
3.3.1 Technology decisions . . . 29
3.3.2 Directories structure . . . 31
3.3.3 Spring configuration . . . 31

4 Testing **33**
4.1 Server application testing . . . 33
4.2 Client applications testing . . . 33
4.2.1 The same appearance . . . 35
4.2.2 The same functionality . . . 35

5 Comparison based on the collected metrics **37**
5.1 Collected metrics from the client's perspective . . . 37
5.2 Collected metrics from the server's perspective . . . 38
5.3 Final comparison . . . 39

Conclusion **41**

Contents of enclosed CD **45**

List of Figures

1.1 Market position of PHP in terms of popularity and traffic compared to the most popular server-side programming technologies 4
1.2 Sequence diagram showing server and client communication in case of SSR . . . 4
1.3 Sequence diagram showing server and client communication in case of CSR . . . 5
1.4 Online books/comics reader system, use case diagram 8
1.5 Online books/comics reader system, domain model diagram 9

2.1 Online books/comics reader system, add book page wireframe 12
2.2 Online books/comics reader system, library page wireframe 12
2.3 Online books/comics reader system, book page wireframe 13
2.4 Online books/comics reader system, main page wireframe 13
2.5 Online books/comics reader system, the API endpoints 14
2.6 The MVC pattern in Struts2 18

List of Tables

5.1 Collected metrics from the client's perspective 37
5.2 Collected metrics from the server's perspective 38

List of code listings

2.1 Example of `MyApp` React component 15
2.2 Example of props passing in React 16
2.3 Example of `useState` hook usage in React 17
2.4 Example of a model class in MVC 19
2.5 Example of a simple JSP page 19
2.6 Example of a View-Controller in Struts2: `struts.xml` 20
2.7 Example of a View-Controller in Struts2: `index.jsp` 21
2.8 Example of a View-Controller in Struts2: `LoginAction.java` 21
3.1 Example of a DAO implementing `JpaRepository` 24
3.2 Code of the `ImageService` class 25
3.3 Code of the `BookSPI` interface 25

3.4 Code of the `PageSPI` interface 26
3.5 Code of the `ImageController` class 27
3.6 Example of global exception handling for controllers 28
3.7 Example of the generated `BooksApi` object usage 30
4.1 Example of unit tests coverage for the `ImageController` class 34
4.2 Definition of the auxiliary `NamedByteArrayResource` class 35
4.3 Example of unit tests coverage for the `ImageService` class 36

I would like to thank my supervisor Ing. Marek Suchánek for the guidance and help he provided me in the preparation of this thesis. Also, I would like to express my deepest gratitude to my family and friends who have always supported and inspired me.

Declaration

In Prague on May 10, 2023 . . . .

Abstract

This bachelor's thesis describes and compares two approaches to client-side web applications development: client-side rendering and server-side rendering. React and Struts2 with JSP (Java Server Pages) technologies were chosen as representatives of each approach, and our own metrics were defined for the comparison.

The comparison is based on a sample online books/comics reader system, whose implementations have the same look and functionality in both technologies.

The result of this thesis is the implementation of a server application with API in Java, two client applications in React and Struts2 with JSP, and then a comparison of the chosen approaches for client-side applications development.

Keywords web development, SSR, CSR, React, Struts2, JSP, JMeter

Abstrakt

Tato práce popisuje a porovnává dva přístupy pro vývoj klientských webových aplikací: renderování na straně klienta a na straně serveru. Pro reprezentaci každého z uvedených přístupů byly zvoleny technologie React a Struts2 s JSP (Java Server Pages) a pro srovnání byly definovány naše vlastní metriky.

Porovnání je založeno na ukázce systému pro online čtení knih/komiksů, jehož implementace mají v obou technologiích stejný vzhled a funkcionalitu.

Výsledkem této práce je implementace serverové aplikace s API v Java, dvou klientských aplikací v React a Struts2 s JSP, a následné porovnání zvolených přístupů pro vývoj klientských aplikací.

Klíčová slova vývoj webu, SSR, CSR, React, Struts2, JSP, JMeter

Acronyms

AJAX Asynchronous JavaScript and XML. 3, 39

API Application Programming Interface. v, 1, 3, 7, 11, 14, 15, 17, 27–29, 33, 39

CPU Central Processing Unit. 9, 22

CRUD Create, Read, Update, Delete. 14, 24

CSR Client-Side Rendering. v, 1, 3, 5–7, 37, 39–41

CSS Cascading Style Sheets. 3, 6, 22, 31, 35

DAO Data Access Object. v, 23, 24

DI Dependency Injection. 17, 23, 31

DOM Document Object Model. 9, 29

DTO Data Transfer Object. 27, 29, 31

HTML HyperText Markup Language. 3, 6, 15, 18, 28, 29, 31, 35, 40

HTTP HyperText Transfer Protocol. 15, 24

IoC Inversion of Control. 17, 23, 31

JSP Java Server Pages. v, 1, 3, 7, 18–20, 28, 31, 35, 41

JSX JavaScript XML. 15, 16

MVC Model View Controller. v, 17–20, 31

ORM Object Relational Mapping. 23

PHP HyperText Preprocessor. v, 3, 4, 6

REST Representational State Transfer. 11, 15

SEO Search Engine Optimization. 3, 5, 9, 39

SPI Service Provider Interface. 24, 31

SQL Structured Query Language. 23, 24

SSR Server-Side Rendering. v, 1, 3–7, 37, 39–41

UI User Interface. 15, 18, 28

URL Uniform Resource Locator. 29

UX User Experience. 5, 39

XML Extensible Markup Language. 31

Introduction

Server-Side Rendering, which will be referred to as SSR in this thesis, is a time-proven approach for client-side applications development that is still used in some commercial projects. Like any technique, this approach has advantages and disadvantages, some of which are solved by Client-Side Rendering, which is a relatively new approach for client-side applications development and will be referred to as CSR in this thesis. The natural question is in which case the first or the second approach should be chosen, or whether it makes sense to consider switching to the new approach for existing applications.

Having a constructive comparison of SSR and CSR would be useful both for developers who want to expand their knowledge of these software development approaches and for project managers who need to decide which approach is the best choice for their particular project. Currently, there exist some comparisons of SSR and CSR concepts, however, not many of them cover the actual performance results based on system implementations with identical appearance and functionality developed using each approach. Moreover, no objective comparison based on the technologies we will use for implementation has been found.

The aim is to describe and compare the basic concepts of SSR and CSR on the example of Struts2 with Java Server Pages (JSP) and React technologies. The comparison should be done according to predefined criteria and metrics. As an example task, a system for online reading books/comics has to be implemented. This system must consist of a Java-based back-end application with Application Programming Interface (API), and two client applications. Each part of the system should be tested, which means that the back-end application must be covered by unit tests and that the manual testing must be performed to ensure that both client applications have the same appearance and functionality.

This thesis is split into 5 chapters. The analysis chapter covers a description of the general concepts of SSR and CSR, examples of existing comparisons of these approaches, the definition of metrics that are used for our own comparison, and the definition of requirements for our online books/comics reader system according to already existing similar systems. The design chapter defines the architecture of the system in the chosen technologies, whilst the implementation chapter goes into more detail about the development process of the system. The testing chapter describes how each part of the system is tested. Finally, our own comparison of SSR and CSR is done in the last chapter.

Chapter 1
Analysis

In this chapter we will describe the general concepts of SSR and CSR, look at some existing comparisons of these approaches, and then define the metrics we will use for our own comparison. Finally, we will define functional and non-functional requirements for an online books/comics reader system, which will be used as a sample task for our comparison.

1.1 Server-Side Rendering

SSR was the first approach used for serving dynamic content in web applications. It appeared in the early 1990s, while the first public version of HyperText Preprocessor (PHP) was released in 1995. According to the W3Techs - World Wide Web Technology Surveys report [1], PHP is still the most popular SSR technology, used by 77.4% of websites. Other popular technologies used for SSR are ASP.NET, Ruby on Rails and Java Server Pages (JSP). The popularity of these technologies is demonstrated in Figure 1.1.

As the approach's name implies, the resulting view of the page that will be later delivered to a customer is rendered directly on the server, which means that every time the customer needs to get a new page view he has to send a new request to the server. Figure 1.2 demonstrates communication between a server, an API providing the data needed for rendering, and a client in case of SSR. That being said, at least the following statements can be made:

1. The websites developed using SSR are easily accessible to Search Engine Optimization (SEO) crawler bots because the server returns fully rendered pages.
2. If SSR is used, then the server will be requested quite often, whenever a new page is needed. However, the number of requests can be reduced by using different caching techniques.

1.2 Client-Side Rendering

The CSR approach was the successor to SSR. With the growing popularity of JavaScript for adding interactive client-side behavior, such CSR frameworks as AngularJS, React, and Vue.js have emerged.

In contrast to SSR, the rendering of web pages is primarily done on the client side in CSR. Clients receive a bundle containing HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript code that will later be used to render all pages view by the browser. Any additional content that is required to render a page will be delivered later by doing Asynchronous JavaScript and XML (AJAX) calls to some API. Unlike in SSR, communication between

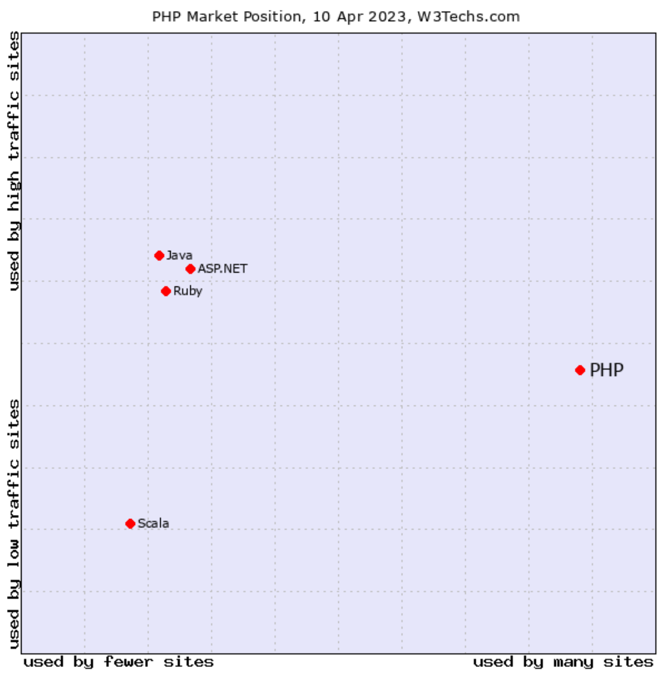


Figure 1.1 Market position of PHP in terms of popularity and traffic compared to the most popular server-side programming technologies

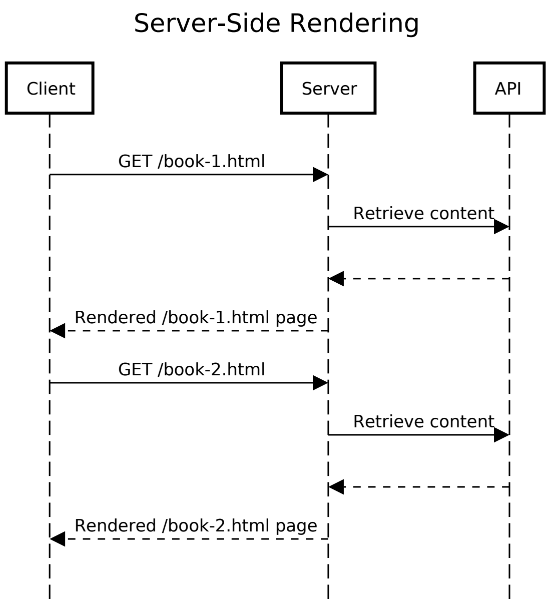


Figure 1.2 Sequence diagram showing server and client communication in case of SSR

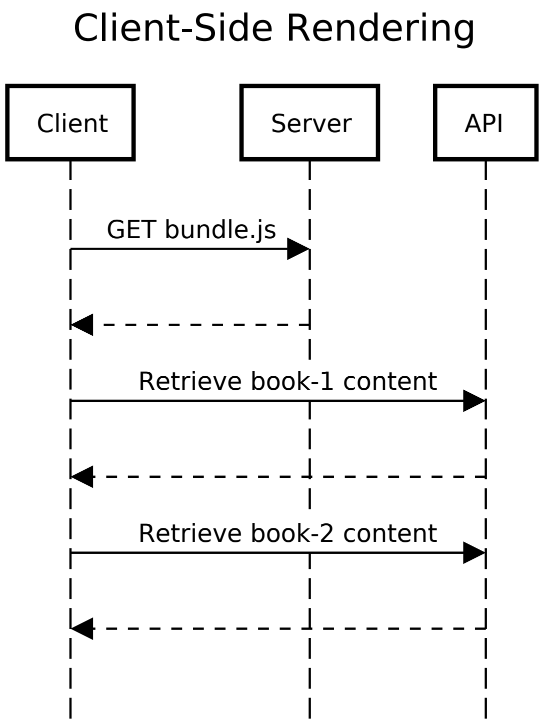


Figure 1.3 Sequence diagram showing server and client communication in case of CSR

client and server happens only once, which is demonstrated in Figure 1.3. In conclusion, at least the following statements can be made:

1. The websites developed using CSR are difficult for SEO crawlers to access because the server returns "raw" and unrendered data to clients, whose responsibility is to take care of rendering later.

2. If CSR is used, then the server will be requested to deliver a bundle with the code necessary to render any page of our web application only once. However, it means that the rendering job is delegated to the client, which can increase the load on the clients' side and affect his User Experience (UX).

1.3 Existing comparisons of SSR and CSR

The comparison of CSR and SSR is an interesting topic from the perspective of development and project management, that is why there already exist some comparisons of these approaches. By using Google Scholar search, we managed to find some of them. Having these examples, we will be able to define what is already done and what new can our thesis bring.

1.3.1 Comparison between client-side and server-side rendering in the web development [2]

Goals As the authors mentioned in the abstract: "*The purpose of this paper is to analyse the comparison between client side and server side method in the respect of technical aspects in* *term of first content paint, speed index, time to interactive, first meaningful paint, first idle CPU and estimated input latency that present better performance*".

Metrics It turned out that the chosen metrics are exactly the same as those provided by Google Lighthouse [3], which is understandable because this tool was used for the metrics generation in that work: "*For the website testing, it used Google Audit, which can look the score of performance, accessibility, best practice, search engine optimization and progressive web app*".

Sample task As the sample task, the authors decided to implement a login page: "*In this test, the study implemented simple login page with the test case of wrong password input*".

Used technologies Talking about the technologies stack, the authors decided to use HTML, CSS, PHP for SSR, and JavaScript, HTML, CSS for CSR: "*On the other hand, in Server side rendering have been used PHP, HTML and CSS while the client side rendering have been used Java script, HTML and CSS because Java script can implement OOP*".

Conclusion This paper is noteworthy, however, it would be interesting to compare the CSR and SSR methods not only from the client's perspective, but also from the server's perspective. It might also be interesting to compare the metrics collected based on some more complex sample task, not just on the login page.

1.3.2 On the Comparison of Software Quality Attributes for Client-side and Server-side Rendering [4]

Goals As the author of this master's thesis mentioned in its abstract: "*The main goal of this study is to investigate the differences between client-side rendering and server-side rendering and to advise developers on making the choice between those two rendering paradigms*".

Metrics The author decided to create a pool of metrics, based on which respondents could later vote for those that were the most important to them. As a result, the following metrics were selected: "*The following software quality attributes will be investigated in this research: performance (page loads, throughput and bandwidth), development effort, scalability, availability*".

Sample task As a sample task, the author decided to implement a simple content management system: "*The web application written for this research is a simple implementation of content management system. It contains simple pages which can be read/browsed like on a real-world website*". It is worth mentioning that the author decided to collect the defined metrics not only from his sample project, but also from some larger open-source projects.

Used technologies Regarding the technologies that were chosen as the representatives of CSR and SSR in this thesis, the author decided to have two implementations of SSR in PHP and Go, while Vue.js was used for CSR.

Conclusion This master's thesis is a very worthwhile work, especially its theoretical part. Unlike the previous paper, it takes into account not only metrics from the perspective of clients' experience, but also from the perspective of servers' performance. However, it was not described in details how the author managed to guarantee that all implementations of his application would be executed in environments with the same amount of resources during metrics collecting.

1.3.3 Summary

Even though there already exist some comparisons of CSR and SSR, the result of this thesis will bring something new as we will use different technologies and several new metrics. However, before collecting any metrics, we need to implement a sample system. The definition of metrics and system requirements will be made in the following sections.

1.4 Online books/comics reader system definition

In this section, we will define how the implementations of a sample system should look like. This definition consists of requirements (functional and non-functional) and use cases. At the end of this section, we will be able to create a domain model for our sample system.

1.4.1 Requirements

This subsection describes the functional and non-functional requirements for our sample system. When defining the requirements, our goal was neither to make them too simple, which would not lead us to an illustrative example from real-world web development, nor to complicate them with features that would not be used for the representative metrics collecting (e.g. user account management).

In order to get a rough idea of what requirements an online books/comics reader system might have, we decided to look at the already existing implementations of such systems[1]. Based on what we have seen, we came up with the following requirements:

Functional requirements clarify what needs to be done and identify the necessary activities.

- **F1: Library management** The system should provide a web page for adding new books. The user should also be able to delete already existing books.
- **F2: Library overview** The system should provide a list of the available books to read. It should be possible to search through the available books by book title. The system should also track what was the last read page when the user stopped reading the book. Otherwise, if the book has not yet been opened, it should be indicated with the corresponding label.
- **F3: Book reading** The system should allow its users to read books. Pagination should be implemented on the book reading web page, which means that the book pages must be split into bundles with a fixed size. Navigation through the bundles should also be implemented in the form of moving to the next/previous bundle or to a bundle with a defined number.

Non-functional requirements are all requirements that are not functional, which usually relates to the performance, usability, and user experience aspects of the system.

- **N1: Clients** Two client applications with identical appearance and functionality should be implemented, one written in Struts2 with JSP, and the other in React.
- **N2: Responsiveness** Both client applications should have a responsive design.
- **N3: Server** A back-end application with a provided API written in Java should be implemented.

[1]For example, `https://www.marvel.com/comics` and `https://www.dc.com/comics`

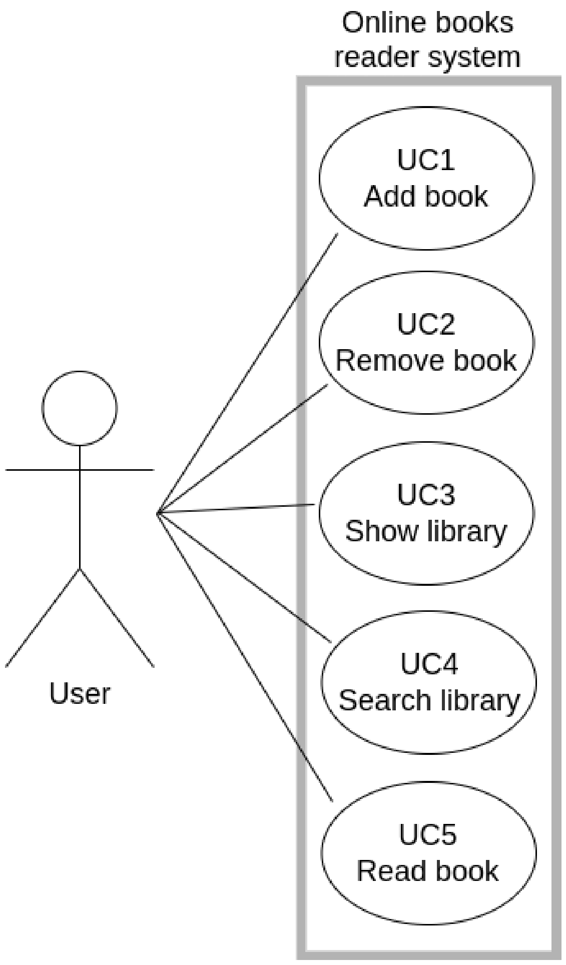


Figure 1.4 Online books/comics reader system, use case diagram

1.4.2 Use cases

This subsection describes the use cases that are derived from our system's requirements. The use cases are depicted in Figure 1.4.

UC1: Adding a new book to the library Users should be able to add new books to their library by entering information about a new book and attaching images of its pages.

UC2: Remove an existing book from the library Users should be able to remove a book from their library.

UC3: Show the library Users should be able to view a list of books that are in their library.

UC4: Search through the library Users should be able to search through their library by book title.

UC5: Read a book Users should be able to read the books they choose.

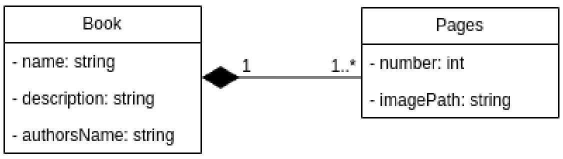


Figure 1.5 Online books/comics reader system, domain model diagram

1.4.3 Domain model

Having the definition of the requirements and use cases, we can create the domain model of our system. The resulting model is illustrated in Figure 1.5

1.5 Metrics definition

In this subsection, we will define the metrics against which we will be able to compare two client implementations of our sample system. We will be interested in metrics from both the client's and server's perspective.

1.5.1 Metrics from the client's perspective

There are several tools for collecting different metrics on client's side. One of them is Chrome DevTools kit [5], which also includes Google Lighthouse [6]. These tools are open-source and can provide us with a lot of information about our application, such as audits for performance, accessibility, progressive web apps and Search Engine Optimization (SEO), that is why it was decided to use Chrome DevTools in this thesis.

After analyzing what information we can get from Chrome DevTools, the following metrics were chosen:

First Contentful Paint measures how long it takes the browser to render the first piece of Document Object Model (DOM) content after a user navigates to a page.

Largest Contentful Paint measures when the largest content element in the viewport is rendered to the screen. This approximates when the main content of the page is visible to users.

Total Blocking Time measures the total amount of time that a page is blocked from responding to user input, such as mouse clicks, screen taps, or keyboard presses.

Speed Index measures how quickly content is visually displayed during page load.

Packets Size measures the size of network packets received by clients. We will not include the size of such external resources as images in this metric.

1.5.2 Metrics from the server's perspective

From the server's perspective, we would like to know which of our implementations is more computationally demanding. One of the main resources that can be monitored in this case is the usage of Central Processing Unit (CPU), which leads to the following metric:

Average CPU Usage measures what the average CPU usage was while processing requests from clients.

Chapter 2

Design

We will start this chapter with creation of wireframes based on the requirements we defined in Section 1.4 to get a rough idea of what our application will look like. After that, we will discuss the design of our server, which consists of an API and a back-end application. Afterward, we will take a closer look at the architecture of applications based on React and Struts2. Finally, we will describe the process of metrics collecting and the additional tools we will need for that.

2.1 Wireframes

In order to have a rough idea of what our application will look like, wireframes of individual web pages were created based on the requirements and use cases we defined in Section 1.4:

- Figure 2.1 demonstrates the page for adding a new book to the library, which covers the use case UC1.
- Figure 2.2 shows the library page where we have an overview of available books. On this page, we can also search through the library by book title and delete books we see. This covers the use cases UC2, UC3 and UC4.
- Figure 2.3 illustrates the book page where we can read a book by navigating through the bundles with its pages. This covers the use case UC5.
- Figure 2.4 displays the main page, which serves as the entry point to our website and which contains links to the other pages we described earlier.

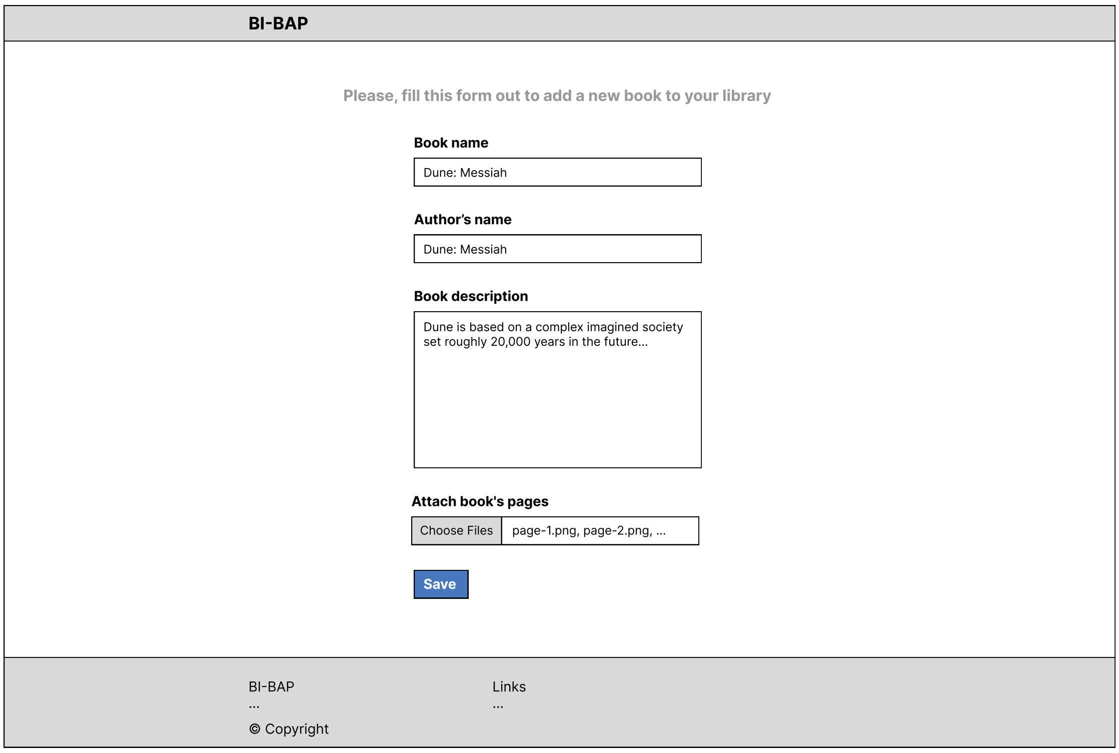
By implementing these wireframes, we make sure that our application covers all the defined use cases.

2.2 Server architecture

In this section, the back-end application with its API design will be described.

2.2.1 API design

Both of our client implementations will reach the back-end application using its API to get data for rendering. We will adhere to Representational State Transfer (REST) standards when designing the API endpoints. We will also use the OpenAPI Specification [7] to help us create a well-documented API with code generation capabilities.


BI-BAP

Please, fill this form out to add a new book to your library

Book name

Dune: Messiah

Author's name

Dune: Messiah

Book description

Dune is based on a complex imagined society set roughly 20,000 years in the future...

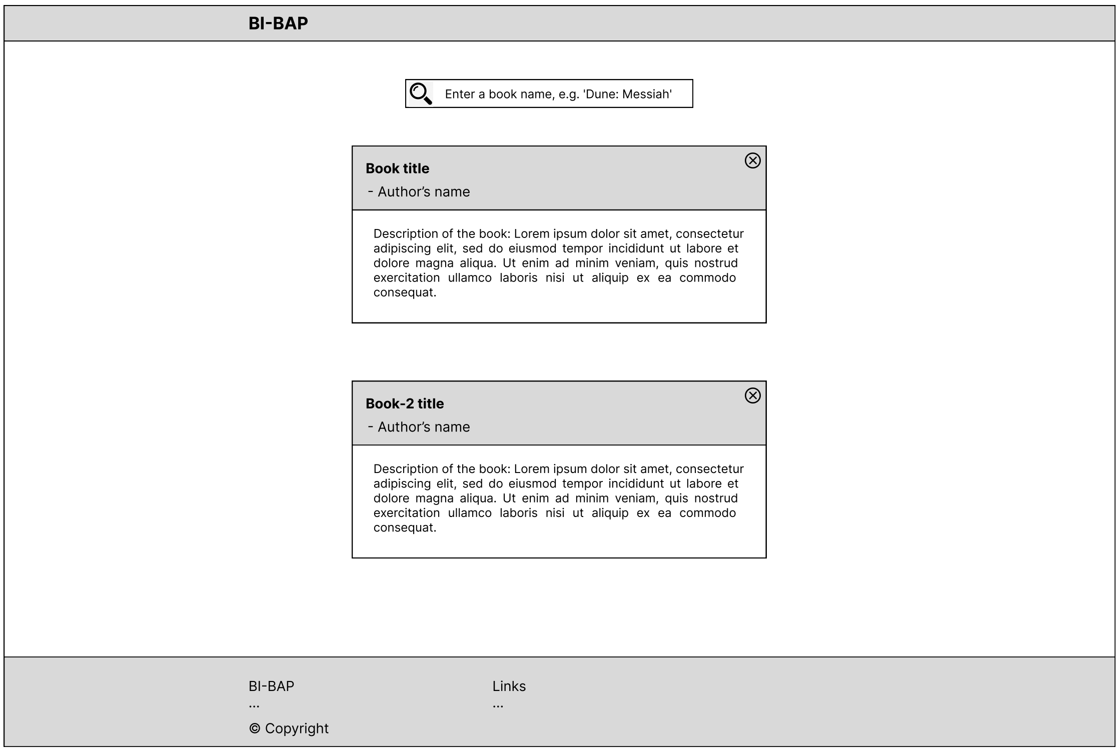
Attach book's pages

Choose Files page-1.png, page-2.png, ...

BI-BAP
...
© Copyright

Links
...

Figure 2.1 Online books/comics reader system, add book page wireframe


BI-BAP

Enter a book name, e.g. 'Dune: Messiah'

Book title
- Author's name

Description of the book: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

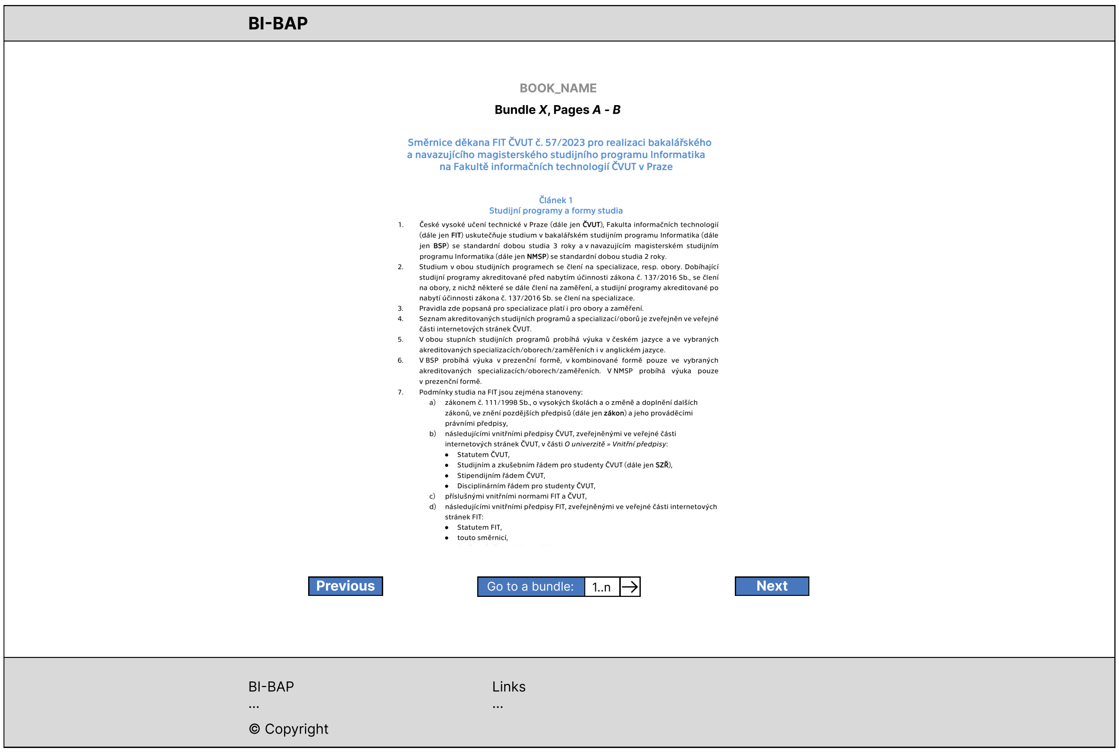
Book-2 title
- Author's name

Description of the book: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

BI-BAP
...
© Copyright

Links
...

Figure 2.2 Online books/comics reader system, library page wireframe



BOOK_NAME

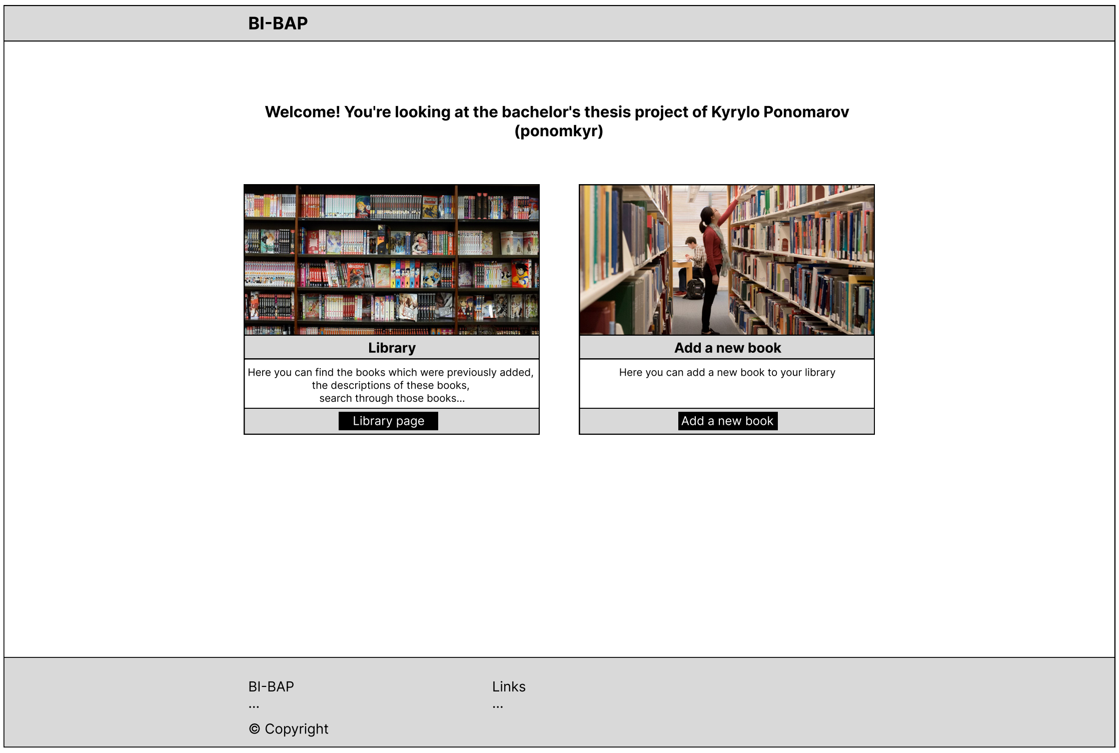
Bundle *X*, Pages *A* - *B*

Směrnice děkana FIT ČVUT č. 57/2023 pro realizaci bakalářského a navazujícího magisterského studijního programu Informatika na Fakultě informačních technologií ČVUT v Praze

Článek 1
Studijní programy a formy studia

1. České vysoké učení technické v Praze (dále jen **ČVUT**), Fakulta informačních technologií (dále jen **FIT**) uskutečňuje studium v bakalářském studijním programu Informatika (dále jen **BSP**) se standardní dobou studia 3 roky a v navazujícím magisterském studijním programu Informatika (dále jen **NMSP**) se standardní dobou studia 2 roky.
2. Studium v obou studijních programech se člení na specializace, resp. obory. Dobíhající studijní programy akreditované před nabytím účinnosti zákona č. 137/2016 Sb., se člení na obory, z nichž některé se dále člení na zaměření, a studijní programy akreditované po nabytí účinnosti zákona č. 137/2016 Sb. se člení na specializace.
3. Pravidla zde popsaná pro specializace platí i pro obory a zaměření.
4. Seznam akreditovaných studijních programů a specializací/oborů je zveřejněn ve veřejné části internetových stránek ČVUT.
5. V obou stupních studijních programů probíhá výuka v českém jazyce a ve vybraných akreditovaných specializacích/oborech/zaměřeních i v anglickém jazyce.
6. V BSP probíhá výuka v prezenční formě, v kombinované formě pouze ve vybraných akreditovaných specializacích/oborech/zaměřeních. V NMSP probíhá výuka pouze v prezenční formě.
7. Podmínky studia na FIT jsou zejména stanoveny:
   a) zákonem č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů, ve znění pozdějších předpisů (dále jen **zákon**) a jeho prováděcími právními předpisy,
   b) následujícími vnitřními předpisy ČVUT, zveřejněnými ve veřejné části internetových stránek ČVUT, v části *O univerzitě » Vnitřní předpisy*:
      - Statutem ČVUT,
      - Studijním a zkušebním řádem pro studenty ČVUT (dále jen **SZŘ**),
      - Stipendijním řádem ČVUT,
      - Disciplinárním řádem pro studenty ČVUT,
   c) příslušnými vnitřními normami FIT a ČVUT,
   d) následujícími vnitřními předpisy FIT, zveřejněnými ve veřejné části internetových stránek FIT:
      - Statutem FIT,
      - touto směrnicí,

Figure 2.3 Online books/comics reader system, book page wireframe

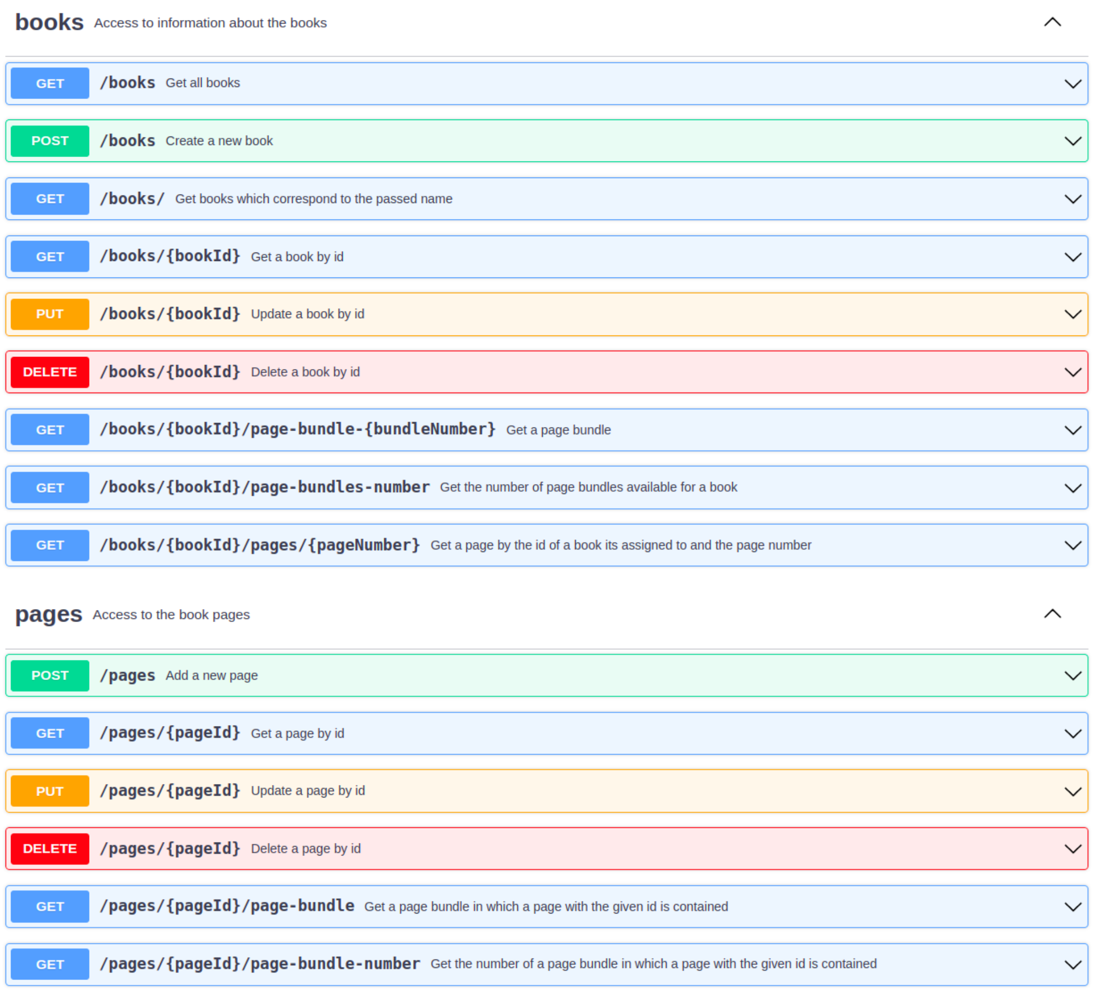


Welcome! You're looking at the bachelor's thesis project of Kyrylo Ponomarov (ponomkyr)

Here you can find the books which were previously added, the descriptions of these books, search through those books...

Here you can add a new book to your library

Figure 2.4 Online books/comics reader system, main page wireframe


books Access to information about the books

GET /books Get all books

POST /books Create a new book

GET /books/ Get books which correspond to the passed name

GET /books/{bookId} Get a book by id

PUT /books/{bookId} Update a book by id

DELETE /books/{bookId} Delete a book by id

GET /books/{bookId}/page-bundle-{bundleNumber} Get a page bundle

GET /books/{bookId}/page-bundles-number Get the number of page bundles available for a book

GET /books/{bookId}/pages/{pageNumber} Get a page by the id of a book its assigned to and the page number

pages Access to the book pages

POST /pages Add a new page

GET /pages/{pageId} Get a page by id

PUT /pages/{pageId} Update a page by id

DELETE /pages/{pageId} Delete a page by id

GET /pages/{pageId}/page-bundle Get a page bundle in which a page with the given id is contained

GET /pages/{pageId}/page-bundle-number Get the number of a page bundle in which a page with the given id is contained

Figure 2.5 Online books/comics reader system, the API endpoints

Resources definition

Based on the use cases we defined in Section 1.4.2, the following resources were identified:

Books The endpoints for this resource start with "/books" prefix and provide basic Create, Read, Update, Delete (CRUD) operations for books, a more complex GET operation that returns books whose titles match the passed value, GET operations for pages and page bundles contained in some particular book.

Pages The endpoints for this resource start with "/pages" prefix and provide basic CRUD operations for pages, along with GET operations for bundles in which some particular page is contained.

It is worth to mention that we do not have a separate entity for page bundles, so we had to implement retrieval of bundle number and bundle content as two different endpoints. The resulting endpoints of our API are depicted in Figure 2.5.

```
const MyApp = () => {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton />
    </div>
  );
}

const MyButton = () => {
  return (
    <button>I am a button</button>
  );
}
```

■ **Code listing 2.1** Example of `MyApp` React component

2.2.2 Back-end application design

Using of multi-layered architectures in application development brings such benefits as separation of concerns, scalability, maintainability, flexibility, reusability and testability, that is why our back-end application adheres to a multi-layered approach with three main layers:

Presentation layer is responsible for exposing the server's functionality through the REST API. It consists of REST controllers that manage communication over the HyperText Transfer Protocol (HTTP).

Application layer contains the business logic and rules that define the application's functionality. This layer acts as an intermediary between the presentation and persistence layers, receiving requests from the presentation layer, processing them, and fetching or updating data from the persistence layer as needed.

Persistence layer is responsible for managing data storage and retrieval. In our application, the PostgreSQL database will be used as a persistent data storage.

2.3 Architecture of React applications

In this section, we will describe the basic architecture and concepts used in React applications. For more detailed information, please refer to the official documentation [8].

2.3.1 React Components

Client applications written in React follow the so-called *component*-based approach. React component is defined as a piece of User Interface (UI) that has its own logic and appearance. Using this definition, we may say that any page can be recursively described as a component.

Imagine a web page that consists of a header and a button. Listing 2.1 demonstrates how this page can be implemented as a component called `MyApp` that includes another component called `MyButton`. As we can also mention from the code example, React uses its own markup syntax called JavaScript XML (JSX) which allows us to inject HTML elements directly in React code.

```
const ParentComponent = () => {
  const name = 'Karel';
  const age = 21;
  const city = 'Prague';

  return (
    <div>
      <h1>Parent Component</h1>
      <ChildComponent name={name} age={age} city={city} />
    </div>
  );
};

const ChildComponent = (props) => {
  return (
    <div>
      <h2>Child Component</h2>
      <p>Name: {props.name}</p>
      <p>Age: {props.age}</p>
      <p>City: {props.city}</p>
    </div>
  );
};
```

Code listing 2.2 Example of props passing in React

2.3.2 React Props

The nested structure of React components would not make much sense if we could not share data between components somehow, typically from a parent component to a child. In React applications, information passed from a parent component to a child component is called *props*, which stands for properties.

In Listing 2.2 we can see an example of passing props, where the `ParentComponent` renders a `ChildComponent` and passes it three props: `name`, `age`, and `city`. The `ChildComponent` receives these props as an object named `props`, which can then be accessed using dot notation to display the values in the child component's JSX.

2.3.3 React State Hook

React *hooks* allow us to use different React features, but in this section we will only cover the `useState` hook, which is probably the simplest and most used one.

Sometimes we may want our components to remember and then display some information. This can be done using the `useState` hook, which allows us to define *state variables* (also known as *states*) for our components. We can manipulate with these state variables, and, what is the most important feature of states, our components will re-render every time any of their state variables changes.

One of the tasks we can use React states for is handling of user input. Listing 2.3 demonstrates how state variables can be used to handle user input in React applications. In that example, we define a component called `InputForm` that uses the `useState` hook to manage the state of `userInput`. The `useState` hook returns an array with two elements: the current state

```
const InputForm = () => {
  const [userInput, setUserInput] = useState('');

  const handleInputChange = (event) => {
    setUserInput(event.target.value);
  };

  return (
    <form>
      <label>
        User Input:
        <input
          type="text"
          value={userInput}
          onChange={handleInputChange}
        />
      </label>
      <p>Input: {userInput}</p>
    </form>
  );
};
```

Code listing 2.3 Example of `useState` hook usage in React

value (`userInput`) and a function to update the state (`setUserInput`). We use array destructuring to assign these values to the `userInput` and `setUserInput` variables, respectively. Then we render a form with an input field, and bind the `userInput` value to the input field. We also pass a callback function `handleInputChange` to the `onChange` event handler of the input field. This function updates the state variable using `setUserInput` whenever the user types something into the input field.

2.4 Architecture of Struts2 applications

In this section, we will describe the basic architecture and techniques used in applications based on the Apache Struts 2 (also known as Struts2) framework. For more detailed information, please see the official documentation [9].

The Struts2 framework is based on Java Servlet API [10] and follows the Model View Controller (MVC) architectural pattern, which helps developers separate the application logic into distinct components for better organization and maintainability. Figure 2.6 shows the relationships of MVC components in terms of the Struts2 framework. The following subsections describe how each of these components is implemented in Struts2.

2.4.1 Struts2 Models

In the MVC paradigm, the model is a component that represents data and business logic of our application. We can say that it is a central component of the MVC pattern and it should not depend on the view or the controller. In Figure 2.6, the model component is called as “Business Services”. Struts2 does not define any specific ways of creating models, however, it is considered good practice to use Inversion of Control (IoC) and Dependency Injection (DI) in their implementation. For more information on IoC and DI, please refer to the official documentation [11].

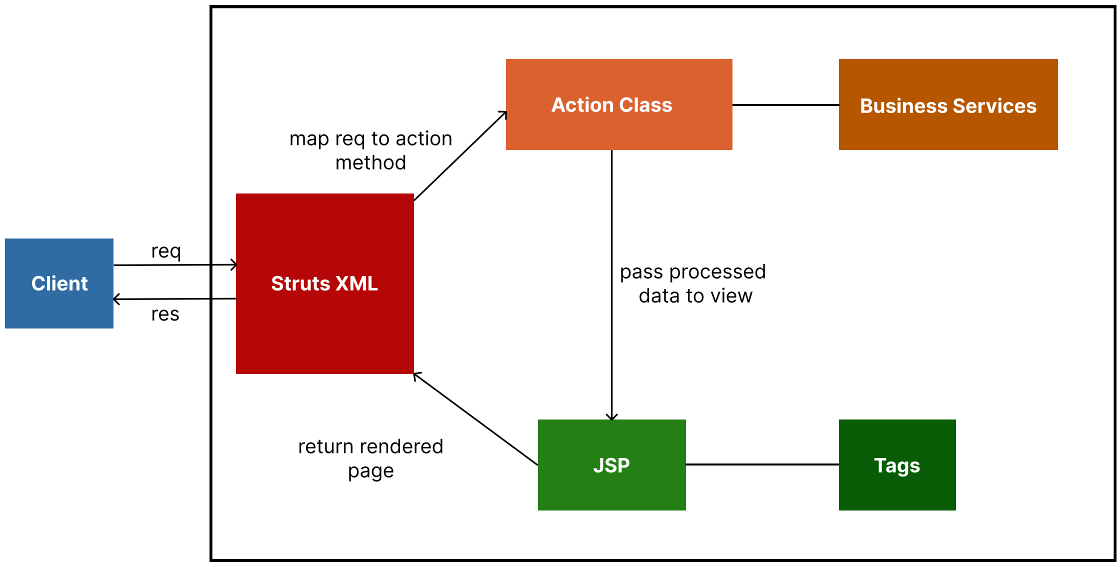


Figure 2.6 The MVC pattern in Struts2

In Listing 2.4 we can see an example of the `UserService` class that can be considered as a model class in the MVC pattern. This class includes methods for implementing business logic, such as `getFullName(User user)` and `isAdult(User user)`, as well as methods for data persistence, such as `saveToDatabase(User user)` and `loadFromDatabase()`.

2.4.2 Struts2 Views

We can say that the view in the MVC pattern is any component that is responsible for rendering the UI and displaying data to users. There can be several different views for the same data, and the MVC views usually depend only on the models, but not on the controllers. Struts2 provides support for several view technologies, including FreeMarker, Velocity, Thymeleaf and JSP, which is one of the most widely used. We will use JSP as the view technology in our Struts2 application.

JSP technology allows us to dynamically generate HTML or other types of text-based documents by embedding Java code directly into HTML pages. In the internal workings, JSPs are compiled into Java Servlet classes at some point, which means that they can be cached and reused instead of processing templates at runtime for every new request, as Thymeleaf does [12]. On the other hand, it means that JSPs cannot be served without an application server.

As shown in Figure 2.6, JSP pages receive data for rendering from *action classes*, which play the role of controller components in MVC terminology and will be described in the next subsection. We can also see on the diagram that JSPs have an association to something called *tags*, which are libraries that allow us to encapsulate complex logic and functionality into reusable elements that can be used inside our JSP pages. For more information on JSP, please check the official documentation [13].

Listing 2.5 demonstrates an example of a simple JSP page which embeds Java code to display the current date using the `java.util.Date` class. A more complicated and realistic looking example of a JSP page will be provided in the next subsection once we define action classes in Struts2.

```java
public class UserService {
    // Business logic
    public String getFullName(User user) {
        return user.getFirstName() + " " + user.getLastName();
    }

    public boolean isAdult(User user) {
        return user.getAge() >= 18;
    }

    // Data persistence
    public void saveToDatabase(User user) {
        // Code to save the user data to a database
        // ...
    }

    public User loadFromDatabase(Long userId) {
        // Code to load the user data from a database based on the userId
        // ...
    }
}
```

■ **Code listing 2.4** Example of a model class in MVC

```html
<!DOCTYPE html>
<html>
<head>
    <title>Hello JSP</title>
</head>
<body>
    <h1>Hello, World!</h1>
    <p>This is a simple JSP example.</p>
    <p>Today's date is: <%= new java.util.Date() %></p>
</body>
</html>
```

■ **Code listing 2.5** Example of a simple JSP page

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.5//EN"
        "http://struts.apache.org/dtds/struts-2.5.dtd">
<struts>
    <package name="default" extends="struts-default">
        <action name="login" class="com.myapp.actions.LoginAction">
            <result name="success">/index.jsp</result>
        </action>
    </package>
</struts>
```

Code listing 2.6 Example of a View-Controller in Struts2: `struts.xml`

2.4.3 Struts2 Controllers

The MVC pattern defines controllers as components that are responsible for handling of user input, managing application state, and coordinating the interactions between models and views. Controllers act as intermediaries that receive input from users, update the model accordingly, and then update the view to reflect the changes in the model. This way, MVC controllers usually depend on views and models.

In Struts2, controllers are defined by using so-called *action classes*. Action classes must either implement the `Action` interface or extend the `ActionSupport` class. By doing so, they will be obliged to implement the `execute()` method, which is the action method that will be called by default when the controller is triggered. Although we can define several different action methods for a single action class to be invoked, we will assume for now that we have an action class with one default action method.

Every action method returns a string value that represents its resulting state, this value can later be used for mapping to some JSP page depending on the result of the action. As we already mentioned before, action classes provide JSP pages with data for rendering. This can be achieved by defining instance variables with getters and setters inside our action classes. Using the defined getters, our resulting JSP pages will be able to access the corresponding instance variables.

Below, we can see an example of code with a Struts2 action and view. To transform this example into a MVC pattern example, we would need to add some model classes and use them in our actions, but we decided to leave them out for simplicity's sake. Listing 2.6 demonstrates an example of the `struts.xml` file, which is a configuration file used by Struts2. We can configure many things in our `struts.xml`, but one of its main purposes is to define the mapping of application endpoints to action classes and the mapping of actions results to corresponding views. Listing 2.7 and Listing 2.8 complete our example with the view and action class code.

2.5 Tools used for metrics collecting

We have already defined metrics from the client's perspective in Section 1.5.1 and from the server's perspective in Section 1.5.2, but we also need to define the tools we will use to collect these metrics. In addition, we need to think about the environment in which we will collect the metrics, because it is important that both implementations run under the same “fair” conditions so that the results are not affected by other tasks running on the host machine.

```
<!DOCTYPE html>
<html>
<head>
    <title>Struts Example</title>
</head>
<body>
    <h1>Hello, Struts!</h1>
    <p>Message: <s:property value="message" /></p>
    <form action="/myApp/login" method="post">
        <input type="submit" value="Login" />
    </form>
</body>
</html>
```

Code listing 2.7 Example of a View-Controller in Struts2: `index.jsp`

```
public class LoginAction extends ActionSupport {
    private String message;

    public String execute() {
        message = "Welcome to Struts!";
        return SUCCESS;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Code listing 2.8 Example of a View-Controller in Struts2: `LoginAction.java`

Client's perspective

As it was mentioned in Section 1.5.1, we will use Chrome DevTools kit [5] to collect metrics from the client's perspective because it is an open-source tool that can provide us with most of the information we need for our comparison. Google Lighthouse tests run in a headless browser environment, but we still should not overload our machine with heavy external tasks while collecting metrics to avoid a shortage of system resources that could affect the results.

We will also use the React Developer Tools [14] browser extension because it can provide us with important information about our React application, such as components' re-rendering time.

Server's perspective

There are many tools that can be used for servers load and performance testing, for example Gatling, LoadRunner and WebLoad, but in this thesis we will use Apache JMeter [15] as it is a powerful, extensible, and flexible open-source testing tool with good plugin support. One of its plugins is called PerfMon [16] and it allows us to monitor server resources usage, including CPU load. This is exactly what we need to collect the metrics that we have defined. It is important to mention that for the PerfMon plugin to work properly, we need to run a special ServerAgent application on the monitored systems.

Since configuring Apache JMeter is not a trivial task, we need to take a closer look at its concepts. Apache JMeter introduces the following terms to create test scenarios:

Test Plan is the main building block of a JMeter test script. Basically, it is a container that holds all the information about actions that need to be performed during testing. In our case, we will have 2 test plans: for our implementations in React and Struts2, respectively.

Thread Group represents a group of virtual users that will execute a series of testing actions against our tested web application. In other words, it simulates user behavior and generates load on the application. This load can be configured using the following parameters:

- **Number of threads** defines the number of virtual users that will execute the defined test plan. We set this value to 1500 users.
- **Ramp-up period** represents how long it will take JMeter to "ramp-up" to the full number of threads. In other words, this parameter defines the interval with which our threads will start. We set the ramp-up period value to 60 s, which means that the interval between creation of new threads is $60/1500 = 0.04$ s.
- **Loop count** defines the number of times a Thread Group will execute its set of testing actions (so-called *samplers*, which are defined below). We set this value to 1, which means that every virtual user will execute his testing action only once.

Samplers generate requests to the tested web application, recording the response time and other performance metrics. In our case, a sampler will send GET requests to the book reading page. The sampler's "Retrieve All Embedded Resources" option is enabled to retrieve some of the external resources included in our web page, such as CSS and JavaScript files.

Environment isolation

In order to provide some level of isolation, we will containerize both of our client implementations along with the ServerAgent application using Docker containers. Unfortunately, Docker containers share the host machine's CPU resources with other applications, therefore we will not have truly isolated environments. However, if we execute the performance tests multiple times with approximately the same number of external tasks running on the host machine, then the average value of the metrics will be sufficient for our comparison.

Chapter 3

Implementation

In this chapter, we will go into more detail about the development process of our online books/comics reader system. First, we will discuss the server implementation, looking at the main technologies we used in the development process and the components that are included in each layer of our back-end application. After, we will mention the decisions we made regarding the tools and libraries used to implement the client applications in React and Struts2. We will also describe the main components contained in each implementation according to their design, which we defined in the previous chapter.

3.1 Server implementation

3.1.1 Technology decisions

Java 17 was chosen as the implementation language for our back-end application because it is the latest released version of Java with long-term support.

Spring Framework was chosen as the core tool for our application. It provides us with such essential features as DI and IoC, not to mention that Spring Framework Ecosystem has many subprojects that can help us solve any task a regular back-end developer may face. For instance, we will use the Spring Boot tool as it has an embedded Tomcat server, which relieves us of having to worry about manual application deployment.

PostgreSQL was chosen as a persistent data storage because it is a relational database that will be enough for the purposes of our system.

Hibernate was chosen as an Object Relational Mapping (ORM) framework that provides a convenient way to map Java objects to relational database tables and perform database operations using Java code, without writing explicit Structured Query Language (SQL) queries.

Maven was used as a dependency management tool in our project.

3.1.2 Persistence layer

The persistence layer of our server application will consist of *Entity Objects* (also known as *entities* or *domains*) and *Data Access Objects (DAOs)*, which are commonly referred to as *repositories*.

Entities are objects that are stored in some persistent storage, in our case they are Java objects that are mapped to database tables. Esentially, entities are implemented based on domain model

```java
public interface BookJpaRepository extends JpaRepository<Book, Long> {
    Collection<Book> findBooksByNameContaining(String name);
}
```

Code listing 3.1 Example of a DAO implementing `JpaRepository`

diagrams, the one for our system is illustrated in Figure 1.5. That being said, we will have two entities in our persistence layer: `Book` and `Page`.

DAOs operate with entities and provide us with access and retrieval functionality, such as querying, inserting, updating, and deleting data in the persistent storage. They often encapsulate the logic for interacting with the underlying database or storage system, and provide an abstraction layer between the application's business logic and the actual storage medium. Spring Data, which is part of Spring Framework Ecosystem, provides us with a quick way to define DAOs using the `JpaRepository` interface. By implementing this interface and specifying the model type with its identifier type, Spring will automatically generate CRUD operations on the specified entity. Also, we can now define new queries in a declarative way, without having to write SQL queries explicitly. Listing 3.1 shows us an example of a DAO object for a `Book` entity. As we can see, besides CRUD operations, that DAO provides us with a method for books retrieving by their names, and this method is defined in a declarative way. For more information about Spring Data JPA, please refer to the official documentation [17].

3.1.3 Application layer

At the application layer, we will implement the business logic of our system by creating *service classes*. Based on the use cases, defined in Section 1.4.2, all the business logic of our application will only concern books and pages, that is why it makes sense to create our service classes in two separate Java packages: `service.book` and `service.page`.

However, there is one service class that is too general to be created inside `service.book` or `service.page` package, and it is called `ImageService`. Basically, this service works as a simple image server. By using its `getImageFromPath(path)` method, we can get an image resource that is located under the given path, which is relative to some base directory. The path of the base directory can be configured in the `application.properties` file. The implementation of `ImageService` is shown in Listing 3.2.

It is considered good practice to define the business logic of our applications using Service Provider Interface (SPI) as it helps us to create easily replaceable service modules. This way we can have several different implementations of a single SPI, and we can easily switch between them in case if other parts of our system depend on the SPI, but not on its concrete implementation. Listing 3.3 and Listing 3.4 demonstrate the SPIs for book and page services, respectively.

3.1.4 Presentation layer

The presentation layer of our application is responsible for handling incoming HTTP requests from users, triggering the required business logic by calling the appropriate method of our service classes, and returning the result to users. It is worth to mention that we need to distinguish between the data we receive as request parameters and the data we operate with in our service classes. There are several reasons for distinguishing them. The first reason is that it is usually unreal to get a complete model object as a request parameter because users simple do not have all the necessary data to create a model object. The second reason is that sometimes we do not want to expose the entire model object in our response. That is why we

```java
@Service
public class ImageService {
    @Autowired
    private ResourceLoader resourceLoader;

    @Value("${bi-bap.images.defaultBaseDirPath}")
    private String imagesBaseDirPath;

    public Resource getImageFromPath(String path) {
        return resourceLoader.getResource("file:" + imagesBaseDirPath + path);
    }

    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }
}
```

■ **Code listing 3.2** Code of the `ImageService` class

```java
public interface BookSPI {
    Collection<Book> getAll();

    Collection<Book> getAllContainingName(String name);

    Book getById(Long id) throws NoEntityFoundException;

    Book create(Book book, Collection<Resource> bookImages) throws
    ↪  EntityStateException;

    Book update(Long id, Book book) throws EntityStateException;

    void deleteById(Long id) throws NoEntityFoundException;
}
```

■ **Code listing 3.3** Code of the `BookSPI` interface

```java
public interface PageSPI {
    Page getById(Long id) throws NoEntityFoundException;

    Page getByNumber(Long bookId, Long pageNumber) throws
     ↪  NoEntityFoundException;

    Collection<Page> getPageBundle(Long bookId, Long bundleNumber);

    Collection<Page> getPageBundleContaining(Long pageId);

    Long getPageBundleNumberContaining(Long pageId);

    Long getNumberOfPageBundlesByBook(Long bookId) throws
     ↪  NoEntityFoundException;

    Page create(Page page) throws EntityStateException;

    Collection<Page> persistImages(Collection<Resource> images);

    void removeImagesByBookId(Long id);

    Page update(Long id, Page page) throws EntityStateException;

    void deleteById(Long id) throws NoEntityFoundException;
}
```

Code listing 3.4 Code of the `PageSPI` interface

```java
@RequiredArgsConstructor
@CrossOrigin
@Controller
public class ImageController {

    private final ImageService imageService;

    @GetMapping("/images")
    public ResponseEntity<Resource> getImage(@RequestParam("path") String
    ↪  path) {
        String decodedPath = URLDecoder.decode(path, StandardCharsets.UTF_8);
        Resource resource = imageService.getImageFromPath(decodedPath);
        if (!resource.exists()) {
            throw new NoEntityFoundException("File not found: " +
            ↪  decodedPath);
        }

        return ResponseEntity.ok()
                .header(HttpHeaders.CONTENT_DISPOSITION,
                    "attachment; filename=\"" + resource.getFilename() + "\"")
                .body(resource);
    }
}
```

Code listing 3.5 Code of the `ImageController` class

will use *Data Transfer Objects (DTOs)*, which are basically miniature versions of our models, and *mappers*, which allow us to convert DTOs to models and vice versa.

Spring Web, which is part of Spring Framework Ecosystem, provides us with annotations for defining presentation layer components that are called *controllers*. Controllers consist of handler methods that are mapped to our API endpoints and are called when a particular endpoint is triggered. For more details about Spring Web, please check the official documentation [18].

Listing 3.5 shows us an example of the `ImageController` class. That controller has only one handler method called `getImage(String path)`. This handler method is triggered when users send GET requests to the "/images" resource along with the `path` parameter. That parameter will later be decoded and passed to the service method, which will try to find the image resource according to the given path. If there is no resource under the given path, then an exception is thrown[1], otherwise an image resource object is returned in a response entity.

Exception handling in controllers

Instead of handling all possible exceptions within individual controllers, we can use a more general approach by utilizing the `@RestControllerAdvice` annotation. It allows us to define a single class with methods that handle specific exceptions across multiple controllers, providing centralized exception handling logic.

In Listing 3.6 we can see an example of global controllers exception handling by using `GlobalExceptionHandler`, which is a `@RestControllerAdvice`-annotated class. It contains two exception handlers, `handleException()` and `handleNotFoundException()`, which handle `Exception` and `NotFoundException` respectively. These exception handlers can have custom

[1]The mechanism for handling exceptions in our controllers will be described later in this section

```java
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception ex) {
        String errorMessage = "An error occurred: " + ex.getMessage();
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
         ↪  .body(errorMessage);
    }

    @ExceptionHandler(NotFoundException.class)
    public ResponseEntity<String> handleNotFoundException(NotFoundException
     ↪  ex) {
        String errorMessage = "Resource not found: " + ex.getMessage();
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(errorMessage);
    }
}
```

Code listing 3.6 Example of global exception handling for controllers

error handling logic to return appropriate HTTP response codes, error messages, and other error details to the client.

Controllers code generation from the OpenAPI specification

As we could notice from the previous examples, the declaration of controllers strongly depends on how the API endpoints are defined. We will use the code generation capabilities of the OpenAPI Specification that we defined in Section 2.2.1. It will free us from the need to declare our handler methods manually, all we will need to do is to implement them. Code generation from the OpenAPI specification is managed by the `openapi-generator` Maven plugin. For more details, please check the official documentation [19].

3.2 React implementation

3.2.1 Technology decisions

TypeScript was chosen as the programming language for our React client application. The main benefit that it brings comparing to plain JavaScript is type checking at compile-time, which can save us a lot of time and effort.

Bootstrap was chosen as the main UI library as it has a lot of predefined components and styles for creation of responsive websites. We will use the React-Bootstrap library that provides a set of pre-built React components that are based on Bootstrap.

React-Snap library was used to pre-render our React components into static HTML files. The generated HTML files can be used to speed up the JSP pages creation process for our Struts2 client implementation.

3.2.2 Directories structure

As we mentioned in Section 2.3.1, the core concept of React applications is components. However, we still need to structure our components into some hierarchy. In this subsection, we will describe the hierarchy of components in our React application.

src/index.tsx is the entry point of our application. This file is responsible for rendering of the root component, which is called `<App/>`, and mounting it to the DOM of our resulting HTML file.

src/App.tsx contains the definition of the `<App/>` component. It uses the React-Router library, which provides a way to handle client-side routing in a declarative manner, allowing us to define the routes of our application as view components and render them based on the Uniform Resource Locator (URL) of the current page.

src/views directory contains the view components of our application. Basically, we can say that every view component represents some web page, which consists of smaller components that can be used in multiple different view components.

src/components/ directory contains the smaller components that can be used on different web pages of our website.

src/generated-sources/ directory contains the automatically generated utility files. We will discuss the generation of these files in more detail in the following subsection.

3.2.3 API components generation

Just like we managed to generate server controllers based on our OpenAPI specification in Section 3.1.4, we can generate the React components used for accessing the server API. We will use the `openapi-generator-cli` tool to do this. As a result, the tool will generate API and DTO objects for the resources defined in our OpenAPI specification, these objects will be placed in the `src/generated-sources/openapi/` directory.

In Listing 3.7 we can see a simplified example of the `<LibraryPage/>` view component that uses a `BooksApi` object generated from the OpenAPI specification to fetch available books and pass them to the `<BooksList/>` component, which will take care of the rendering of these books.

3.2.4 React components pre-rendering

As we already mentioned in Section 3.2.1, the `React-Snap` library can be utilized to pre-render our React components into static HTML files. It uses a headless browser environment to crawl all the accessible routes, starting with the root component. For this to work, we need to add a postbuild script in our `package.json` file and change the way our application is booted in the `src/index.tsx` file. For more details, please check the official documentation [20].

3.3 Struts2 implementation

3.3.1 Technology decisions

Java 17 was chosen as the programming language for our Struts2 application. The reason was the same as for the choice of Java version for our server implementation in Section 3.1.1: it is the latest released version of Java with long-term support.

```
const LibraryPage = () => {
    const [books, setBooks] = useState<BookDTO[]>([]);

    const booksApi = new BooksApi();

    useEffect(() => {
        booksApi.getBooks()
            .then((books: BookDTO[]) => setBooks(books))
            .catch(() =>
            ↪   log.debug('Sth went wrong while fetching available books'));
    }, []);

    return (
        <Body>
            <Row className="w-100 justify-content-center">
                <Col className="col-lg-8 col-md-10 col-sm-12">
                    <BooksList books={books}/>
                </Col>
            </Row>
        </Body>
    );
}

export default LibraryPage;
```

Code listing 3.7 Example of the generated `BooksApi` object usage

Struts 2.5.30 was chosen as the main framework for our client implementation. So far, no publicly known vulnerabilities have been found in it and it does not contain any features that we would miss compared to newer versions of the Struts framework.

Spring Framework was used mainly because of its IoC and DI features, which will be used to create the MVC model components in our Struts2 application.

Tomcat 9 was chosen as the application server for our project. We originally wanted to use Tomcat 10, as it is the latest released version, but we soon discovered that it was incompatible with the Struts framework. The thing is that even the latest version of Struts[2] still depends on some packages from Java 8. However, Tomcat 10 supports only Java 11 or later. That means that in order to use the Struts framework in our application, we are forced to use Tomcat 9 or lower. This is a known issue [21], but it is still not fixed in the latest versions of the Struts framework.

3.3.2 Directories structure

As we already know from Section 2.4, the Struts2 framework adheres to the MVC architectural pattern. The following subsection describes where the individual components of the MVC pattern are located in our Struts2 project.

src/main/webapp/ directory contains the web application's static resources, such as HTML files, CSS files, JSP files, JavaScript files, images, and other client-side assets. In other words, this directory contains the MVC view components of our application, except the DTOs, which are located in the `src/main/java/cz/cvut/fit/bap/ponomkyr/struts/dto` directory. It is worth to mention that the `src/main/webapp/index.jsp` file is used as the entry point of our Struts2 application.

src/main/java/cz/cvut/fit/bap/ponomkyr/struts/service/ directory contains the MVC model components of our Struts2 application. Like the service classes from the application layer of our back-end application, which were described in Section 3.1.3, our models will implement the corresponding SPIs for better modules' replaceability.

src/main/java/cz/cvut/fit/bap/ponomkyr/struts/action/ directory contains the action classes of our application, which along with the `src/main/resources/struts.xml` file can be interpreted as the MVC controller components of our application.

3.3.3 Spring configuration

There are two different approaches to configure the Spring framework: annotation-based configuration and XML-based configuration. Annotation-based configuration is considered to be more readable and easier to maintain than XML-based configuration, that is why we used the annotation-based approach to configure the Spring framework for our back-end application. However, some difficulties were encountered when we tried to use the annotation-based approach to configure the Spring framework for our Struts2 application. No mention of the annotation-based approach was found in the official documentation, only the XML-based configuration was used [22], that is why the Spring configuration for our Struts2 framework is provided in the `src/main/resources/applicationContext.xml` file.

[2]As of today, it is Struts 6.1.1

Chapter 4

Testing

Testing is the crucial part of software development, even if we are only implementing a sample system. That is why we will describe how our server and client applications were tested in this chapter.

4.1 Server application testing

We need to test the server application to make sure that both of our client implementations receive the correct data using the provided API. To achieve this, we covered our server implementation with *unit tests*.

Unit testing is a software testing technique that checks individual units or components of a software system in isolation from the rest of the system. The purpose of unit testing is to verify that each unit of code performs as expected and meets its design specifications. Unit tests provide us with an efficient way of testing the isolated parts of our system, even though they do not verify the way these parts interact with each other.

All classes from the presentation and business layers, which form the core of our server application, were covered by unit tests. It is considered good practice to test all layers of our application, but in the presentation layer we tested only the `ImageController`[1] class, because it could contain an obvious and dangerous vulnerability. We needed to make sure that users could not access any file with sensitive data on our system by passing its absolute path to the controller. For example, a user could easily try to request the `/etc/passwd` file, but his request had to be declined.

Listing 4.1 shows an example of unit tests for the `ImageController` class that cover the scenarios when we request for an existing and a non-existing image resource. Also, in Listing 4.2 we can find a definition of the auxiliary `NamedByteArrayResource` class, which is used during testing of the `ImageService` class in Listing 4.3.

4.2 Client applications testing

One of the important things to remember when testing our client implementations is that we have defined a non-functional requirement in Section 1.4.1 that says that both of our client implementations must have the same appearance and functionality. These two points will be discussed below.

[1]For recalling the logic of the `ImageController` class, please check Section 3.1.4

```java
@WebMvcTest(ImageController.class)
class ImageControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private ImageService imageService;

    @Test
    void getImage_shouldReturnImageResource() throws Exception {
        String path = "/path/to/image.jpg";
        Resource resource = new NamedByteArrayResource(new byte[]{1, 2, 3},
        ↪   "image.jpg");
        when(imageService.getImageFromPath(path)).thenReturn(resource);

        MvcResult result = mockMvc.perform(get("/images").param("path", path))
                .andExpect(status().isOk())
                .andReturn();

        assertThat(result.getResponse().getContentAsByteArray())
                .isEqualTo(new byte[]{1, 2, 3});

        assertThat(result.getResponse()
        ↪   .getHeader(HttpHeaders.CONTENT_DISPOSITION))
        ↪   .isEqualTo("attachment; filename=\"image.jpg\"");
    }

    @Test
    void getImage_shouldThrowExceptionWhenFileNotFound() throws Exception {
        String path = "/path/to/nonexistent/image.jpg";
        when(imageService.getImageFromPath(path)).thenReturn(new
        ↪   ClassPathResource("nonexistent"));

        mockMvc.perform(get("/images").param("path", path))
                .andExpect(status().isNotFound());
    }
}
```

Code listing 4.1 Example of unit tests coverage for the `ImageController` class

```
static class NamedByteArrayResource extends ByteArrayResource implements
↪  WritableResource {

    private final String filename;

    public NamedByteArrayResource(byte[] byteArray, String filename) {
        super(byteArray);
        this.filename = filename;
    }

    @Override
    public String getFilename() {
        return filename;
    }

    @Override
    public OutputStream getOutputStream() throws IOException {
        return null;
    }
}
```

Code listing 4.2 Definition of the auxiliary `NamedByteArrayResource` class

4.2.1 The same appearance

As it was mentioned in Section 3.2.4, we managed to pre-render our React components into plain HTML and CSS files. These generated files were later used to create the JSP files in our Struts2 client implementation. That being said, we can state that both of our client implementations have the same appearance.

4.2.2 The same functionality

To make sure that both of our client implementations have the same functionality, we performed manual testing. During this manual testing, we checked all the use cases of our system that were defined in Section 1.4.2.

```java
@SpringBootTest
class ImageServiceTest {

    @Autowired
    private ImageService imageService;

    @MockBean
    private ResourceLoader resourceLoader;

    @Value("${bi-bap.images.defaultBaseDirPath}")
    private String imagesBaseDirPath;

    @BeforeEach
    void setup() {
        imageService.setResourceLoader(resourceLoader);
    }

    @Test
    void getImageFromPath() {
        String path = "/Dune/image.jpg";
        Resource resource = new ByteArrayResource(new byte[]{1, 2, 3},
        ↪  "image.jpg");
        Resource emptyResource = new ClassPathResource("nonexistent");

        when(resourceLoader.getResource(any())).thenReturn(emptyResource);
        when(resourceLoader.getResource("file:" + imagesBaseDirPath +
        ↪  path)).thenReturn(resource);

        assertEquals(imageService.getImageFromPath(path), resource);
    }


    @Test
    void getImageFromPath_shouldThrowExceptionWhenTryingToGetExternalFile()
    ↪  throws Exception {
        String path = "/etc/passwd";
        Resource resource = new ByteArrayResource(new byte[]{1, 2, 3},
        ↪  "passwd");
        Resource emptyResource = new ClassPathResource("nonexistent");

        when(resourceLoader.getResource(any())).thenReturn(emptyResource);
        when(resourceLoader.getResource(path)).thenReturn(resource);

        assertEquals(imageService.getImageFromPath(path), emptyResource);
    }
}
```

Code listing 4.3 Example of unit tests coverage for the `ImageService` class

Chapter 5

Comparison based on the collected metrics

In this chapter, we will collect the metrics defined in Section 1.5.1 and Section 1.5.2. As a test scenario for both implementations, the loading of the book reading web page will be analyzed. For fairness, browser caching will be disabled during metrics collection. With the collected metrics, we will be able to do the comparison of CSR and SSR approaches.
First, we will look at the results from the client's perspective. Afterward, we will look at the results from the server's perspective. Finally, we will be able to draw some conclusion about the conditions under which one approach performs better than the other one.
The containerized versions of our client applications, which will be used for metrics collection, are available in the "metrics_capturing" *branch of the project's GitLab repository.*

5.1 Collected metrics from the client's perspective

In this section, we will collect and analyze all the metrics from the client's perspective that were defined in Section 1.5.1. The results are shown in Table 5.1 and some comments on the individual metric results will be provided below.

Metric name	React	Struts2 with JSP
First Contentful Paint	1.8 s	2.4 s
Largest Contentful Paint	6.3 s	3.7 s
Total Blocking Time	0 ms	0 ms
Speed Index	1.8 s	2.4 s
Packets Size	480 kB	270 kB

Table 5.1 Collected metrics from the client's perspective

First Contentful Paint time result is better in case of the React implementation. This may be caused by internal optimizations of React.

Largest Contentful Paint time result is now significantly better in case of the Struts2 implementation. The explanation of this observation comes from the basic idea of CSR: the entire page rendering is a client-side responsibility and it costs some time, while in case of SSR clients receive already rendered pages.

Total Blocking Time metric results are the same in case of both implementations. The zero value of this metric means that we do not have any long-running tasks that take more than 50 ms to execute in both of our implementations. Generally, it is a desirable result, and in our case it may be caused by the specific requirements of our system, which were not that complex in terms of performance.

Speed Index metric result is better in case of our React implementation, which means that its perceived loading speed is higher than in case of the Struts2 implementation.

Packets Size metric results show us that we send significantly less data to our clients in case of the Struts2 implementation, rather than in the React implementation. This can be explained by the fact that the received React bundle contains all the information needed to render any page of our application, while our Struts2 implementation only provides us with the data of one specific page.

It is also important to mention that the re-rendering time of the book reading page in React took us only 3.2 s, which is an even better result on the Largest Contentful Paint metric than the one shown in Table 5.1. This is explained by the fact that we do not need to send additional requests to the server to render next pages, all the necessary rendering instructions are already contained in the bundle we received in response to our very first request.

Implication

Based on the results from the client's perspective, we cannot say that one of our client implementations has significantly better performance on all the defined metrics. On some metrics our Struts2 implementation shows better results (Largest Contentful Paint and Packets Size), and on some other metrics our React implementation performs better (First Contentful Paint and Speed Index).

5.2 Collected metrics from the server's perspective

Following the collection of client's metrics in Section 5.1, in this section we will collect and analyze the metrics from the server's perspective, which were defined in Section 1.5.2. As it was mentioned in Section 2.5, we will use the Apache JMeter tool for our purposes. A more detailed look at its configuration is also provided in Section 2.5.

The results from the server's perspective are provided in Table 5.2, while some comments can be found below.

Metric name	React	Struts2 with JSP
Average CPU Usage	7.22%	14.64%

Table 5.2 Collected metrics from the server's perspective

Average CPU Usage metric result is significantly better in case of the React implementation. To get more representative data, we ran our JMeter tests several times and calculated an average value based on these results. Moreover, the React results start to appear even better when we realize that with the served React bundles users will be able to render any page of our web application without having to send additional requests to the React server, while the Struts2 implementation only provides users with the rendered data of a single web page.

Implication

Looking at the results from the server's perspective, we can say that the React implementation of our online books/comics reader system seems to be much more advantageous. Not only was the server load significantly lower when processing the same test scenarios, but the React implementation also provided us with a bundle containing all the necessary data to render any page of our web application, while the Struts2 implementation returned us only a single rendered web page. This means that with the CSR approach we would reduce the load on the server not only in terms of performance, but also in terms of the number of incoming requests.

5.3 Final comparison

After implementing our system using both CSR and SSR technologies, and collecting the predefined metrics based on these implementations, we can finally make some comparison between these two approaches. Since it would be completely wrong to say that one approach is always better than the other one, our final comparison will be based on criteria that we can follow to determine what approach is better according to our needs.

Initial page load time

In case if we need to have a really short initial page load time, then the SSR approach will be a better choice than CSR. However, we need to keep in mind that the significant difference in load time can only be observed when the website is accessed for the first time. One of the real-world examples, where we can use SSR to significantly change the UX of our users in terms of the website's initial page load time, are landing pages. Otherwise, websites will most likely consist of several web pages, so a short initial page load time can easily be counterbalanced by a longer subsequent page load time.

Subsequent page load time

If we are interested not only in the initial page load time, but rather in the average load time of all our web pages, then the CSR approach will be a better option for us. To render subsequent pages in case of CSR, there is no need to send additional requests to the server, as the user has already received a bundle with the code necessary to render any page of our web application. This way, it is only needed to re-render the relevant React components with possibly making some AJAX calls to the API, which takes less time than requesting an entirely new page in case of the SSR approach.

Search Engine Optimization (SEO) rankings

In case if the SEO ranking of our website is a crucial point to us, then we need to be aware of the possible SEO struggles that may be faced in the web applications developed using the CSR approach. Crawler bots, which are responsible for assigning the SEO rankings to websites, usually expect to receive an already rendered web page from the server. This means that websites created using CSR may appear incomplete to crawler bots, which will most likely degrade their SEO ranking. If it is a big deal for us, then we should consider using SSR or some other approach instead.

Accessibility

Accessibility of the CSR approach can be a concern as it uses JavaScript to render content on the client side, which can cause delays and potential issues for users who rely on screen readers or who have other accessibility needs. We can say that the SSR approach has better accessibility, because in this case users receive already rendered HTML pages without need to run JavaScript code on the client side. However, even in case of SSR, it is important to follow the best practices[1] to ensure that our web application is accessible to all possible types of users.

Interactiveness

Since the SSR approach does not assume that any JavaScript code is executed on the client side, applications based on this approach will end up being less interactive. As a result, there are a lot of features that would not be possible to implement without using CSR techniques. For example, we can mention the *Data Layer* tracking, which is a popular technique used by web analytics to collect the information about users' interactions with a website, such as pageviews, clicks, and form submissions.

Transparency of source code

In some cases, we may wish not to share the source code of our website rendering logic with clients. If this is an important point to us, then chances are that we will prefer to use the SSR approach. In SSR, we generally do not share any JavaScript code containing rendering logic. Instead, our clients only receive the results of rendering. It is important to mention that many CSR frameworks use various techniques to complicate reading of the source code contained in the served bundles. However, we can also get around it with help of special tools called *code beautifiers*.

Server performance

If performance of the server that hosts our client application is crucial for us, then we should consider using the CSR approach. Not only will the number of requests to the server be lower, but also its workload will not be so high as in case of the SSR implementation.

[1] For more details, please visit `https://www.w3.org/standards/webdesign/accessibility`

Conclusion

The goal of this thesis was to compare Client-Side Rendering (CSR) and Server-Side Rendering (SSR) approaches to client-side web applications development. React and Struts2 with Java Server Pages (JSP) technologies were selected as representatives of each approach. It was necessary to define our own metrics and implement a sample online books/comics reader system using each approach to do the comparison.

At first, we did the analysis that consisted of describing the basic concepts of CSR and SSR, studying the already existing comparisons, defining our own metrics and defining the requirements for a sample system.

After that, the design of our system was carried out. This required a creation of wireframes for our application, a description of the server-side and client-side applications' architecture and concepts of the chosen technologies. As the last topic in the design chapter, we discussed the tools and environments that would be used to collect our metrics.

Based on the defined design, our system was implemented. The code was covered with unit tests and the manual testing was performed to ensure that our system behaves as expected and that both of our client implementations have the same appearance and functionality.

Finally, we had two different client implementations of the system that allowed us to collect the defined metrics and make our own comparison, which is exactly what was done in the last chapter of this thesis.

It is worth to mention that this work does not take into account the existence of the "symbiotic" approaches to client-side web applications development, such as Next.js, which allows us to create React-based web applications with server-side rendering and static website generation support. A study of that third approach to client-side web applications development could be done in the future.

Bibliography

1. Q-SUCCESS. *Usage statistics of PHP for websites* [online]. Q-Success, 2019-11 [visited on 2023-04-13]. Available from: `https://w3techs.com/technologies/details/pl-php`.
2. ISKANDAR, Taufan Fadhilah; LUBIS, Muharman; KUSUMASARI, Tien Fabrianti; LUBIS, Arif Ridho. Comparison between client-side and server-side rendering in the web development. In: *IOP Conference Series: Materials Science and Engineering*. IOP Publishing, 2020, vol. 801, p. 012136. No. 1.
3. GOOGLE. *Performance audits* [online]. Google [visited on 2023-04-13]. Available from: `https://developer.chrome.com/docs/lighthouse/performance/`.
4. BEKE, Mathias. *On the Comparison of Software Quality Attributes for Client-side and Server-side Rendering*. June, 2018.
5. GOOGLE. *Chrome DevTools* [online]. Google [visited on 2023-04-16]. Available from: `https://developer.chrome.com/docs/devtools/`.
6. GOOGLE. *Lighthouse* [online]. Google [visited on 2023-04-13]. Available from: `https://developer.chrome.com/docs/lighthouse/`.
7. SOFTWARE, SmartBear. *OpenAPI Specification* [online]. SmartBear Software, ©2023 [visited on 2023-04-16]. Available from: `https://swagger.io/specification/`.
8. SOURCE, Meta Open. *Learn React* [online]. Meta Open Source, ©2023 [visited on 2023-04-18]. Available from: `https://react.dev/learn/`.
9. FOUNDATION, The Apache Software. *Getting started* [online]. The Apache Software Foundation, ©2000-2022 [visited on 2023-04-19]. Available from: `https://struts.apache.org/getting-started/`.
10. GUINDON, Christopher. *Jakarta Servlet 5.0: The Eclipse Foundation* [online]. Eclipse Foundation [visited on 2023-04-19]. Available from: `https://jakarta.ee/specifications/servlet/5.0/`.
11. JOHNSON, Rod; HOELLER, Juergen; DONALD, Keith; SAMPALEANU, Colin; HARROP, Rob; RISBERG, Thomas; ARENDSEN, Alef; DAVISON, Darren; KOPYLENKO, Dmitriy; POLLACK, Mark, et al. *Spring Framework: The IoC container* [online]. VMware, Inc., ©2004-2016 [visited on 2023-04-19]. Available from: `https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html`.
12. ROSENCRANTZ, Niklas. *What kind of solution is Thymeleaf?* [online]. Stack Overflow, 2016 [visited on 2023-04-20]. Available from: `https://stackoverflow.com/questions/38806245/what-kind-of-a-solution-is-thymeleaf`.

13. ORACLE et al. *The java EE 5 tutorial, JavaServer Pages Technology* [online]. Oracle, 2007-09 [visited on 2023-04-20]. Available from: `https://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html`.
14. SOURCE, Meta Open. *React developer tools* [online]. Meta Open Source, ©2023 [visited on 2023-04-30]. Available from: `https://react.dev/learn/react-developer-tools`.
15. *Apache JMeter - Apache JMeter™* [online]. Apache Software Foundation, ©1999–2022 [visited on 2023-04-23]. Available from: `https://jmeter.apache.org/`.
16. POKHILKO, Andrey et al. *Servers Performance Monitoring* [online]. jmeter-plugins.org, ©2009-2023 [visited on 2023-04-23]. Available from: `https://jmeter-plugins.org/wiki/PerfMon/`.
17. GIERKE, Oliver; DARIMONT, Thomas. *Spring Data* [online]. VMware, ©2008-2022 [visited on 2023-04-24]. Available from: `https://docs.spring.io/spring-data/jpa/docs/current/reference/html/`.
18. WEBB, Phillip; SYER, Dave; LONG, Josh; NICOLL, Stéphane; WINCH, Rob; WILKINSON, Andy, et al. *Spring Web* [online]. VMware, 2023 [visited on 2023-04-26]. Available from: `https://docs.spring.io/spring-boot/docs/current/reference/html/web.html`.
19. *Plugins: Openapi generator* [online]. OpenAPI-Generator Contributors, 2023 [visited on 2023-04-26]. Available from: `https://openapi-generator.tech/docs/plugins/`.
20. *React-Snap* [online]. npm, Inc., 2019 [visited on 2023-04-29]. Available from: `https://www.npmjs.com/package/react-snap`.
21. BERRE, Daniel Le. *Support for JEE 9+* [online]. ASF JIRA, 2021 [visited on 2023-04-27]. Available from: `https://issues.apache.org/jira/browse/WW-5141`.
22. FOUNDATION, The Apache Software. *Spring and struts 2* [online]. The Apache Software Foundation, ©2000-2022 [visited on 2023-04-29]. Available from: `https://struts.apache.org/getting-started/spring`.

Contents of enclosed CD

`implementation`	source codes of the online books/comics reader system
`bi-bap_api`	source code of the server application
`bi-bap_react`	source code of the client application written in React
`bi-bap_struts`	source code of the client application written in Struts2
`thesis`	the thesis text directory
`thesis-sources`	source code of the thesis
`thesis.pdf`	the thesis text in PDF format