BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS
AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

PPTX TO HTML CONVERSION

PŘEVOD PPTX DO HTML

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. HYNEK VILÍMEK

SUPERVISOR
VEDOUCÍ PRÁCE

Prof. Ing. ADAM HEROUT, Ph.D.

BRNO 2016

Abstract

PowerPoint is an excellent tool for creating presentations and people are accustomed to using it. Its only handicap is that it is not installed everywhere and it exists in numerous versions. But there is an application that is installed almost everywhere and that application is the web browser. This work aims to create the PowerPoint presentation viewer for the web browser. With the internet as the environment, it may have a wide range of applications from the content sharing point of view. The solution is a web application that allows to upload the PowerPoint file and then the application displays the content of the file. The application also offers functionality such as navigation between slides and full-screen mode. The rendered slides in the web browser are very similar to the slides in PowerPoint. It does not support advanced features, but it supports displaying text, pictures, video and audio. Further, it supports basic styling options such as colours, margins, position and line height.

Abstrakt

Program PowerPoint je excelentní nástroj sloužící k vytváření prezentací a lidé jsou na něj zvyklí. Jeho nevýhodami jsou jeho absence na některých počítačích a jeho existence v mnoha verzích. Webový prohlízec však tyto nevýhody nemá. Tato práce se snaží využít webové prohlížeče jako programu, který umožní zobrazit PowerPointové prezentace. Díky internetu může mí tato práce široké možnosti použití. Výsledkem práce je webová aplikace, která umožňuje nahrát a spustit PowerPointovou prezentaci. Aplikace umožňuje přepínaní mezi snímky a přepínání prezentace do celoobrazového módu. Výsledné snímky jsou velice podobné originálním snímkům a vytvořené řešení umožňuje zobrazit text, zvuk a video se základními možnosti úprav.

Keywords

Presentation, Slide, PowerPoint, HTML, Client, Server, Conversion, Javascript, Cascading stylesheets, Isomorphic application, NodeJS, ECMA2015, testing, Selenium

Klíčová slova

Prezentace, Snímek, PowerPoint, HTML, Klient, Server, Převod, Javascript, Kaskádové styly, Isomorfní aplikace, NodeJS, ECMA2015, testování, Selenium

Reference

VILÍMEK, Hynek. *PPTX to HTML Conversion.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Herout Adam.

PPTX to HTML Conversion

Declaration

I declare that I have worked on this thesis independently under the supervision of Prof. Ing. Adam Herout, Ph.D. All sources used in the work have been acknowledged and fully referenced in the bibliography.

. . . .
Hynek Vilímek
May 19, 2016

Acknowledgements

I would like to thank my supervisor Prof. Ing. Adam Herout, Ph.D. for his help and my family for their patient and support.

Contents

Chapter 1

Introduction

My work allows people to view the PowerPoint presentation on any computer which has a newer web browser version. This can fulfil the need of portability and flexibility, because no one will be forced to check if PowerPoint is installed and in what version. The application also gives the presentations the share-ability. Each presentation is possible can be addressed by URL and the same is true for each slide.

The solution consists of three parts. The first part extracts the information from the presentation file. This results in the part with a clearly specified input and output. The input is the presentation file and the output is the internal representation of the presentation. This can be easily evaluated by unit tests. The second part interprets the internal representation of the presentation and displays it as the presentation mode in PowerPoint. This part results in the application that takes the internal representation and visually interprets it. It was evaluated by the comparison of the reference image generated by PowerPoint and the image of the actual result. The third part is the part of the application that surrounds the second part and adds features like a full-screen mode, navigation and asynchronous data loading. This par was also evaluated by unit tests.

The existing solutions do not offer the same level of user experience as my application. The first found solution is the web application called *Zamzar*. It offers a PowerPoint to HTML conversion, but it generates a zip archive, which is delivered to the previously filled email. The main difference is in the result of the conversion. It converts slides into images and places text over them. It does not support video, audio and the method of conversion is not suitable for display scaling. This service fulfils the function of partly viewing the content of the presentation file, but clearly does not fulfil the function to view it in the presentation mode. The second solution is from Microsoft itself and it is offered in the package called Office 365. But it also renders slides into images and uses SVG technology to set the correct position. The result of this application is better, but it still renders text into image, which is not optimal solution. In addition, this is a paid solution. The third solution is from Google and it is implemented as part of the Google Drive and Google Slides. These applications are hardly usable, because they have for example problems with fonts and margins. They clearly do not aim to the same use cases, but they aim to fulfil the need of a simple viewer for the PowerPoint presentations and they also allow to create new ones.

My solution is a web application and it consists of a server side part and a client side part. The server side part is responsible for extracting the information from the uploaded file and for their transformation to the internal representation. The client side part visually displays this representation and offers some functionality of the presentation mode from

PowerPoint. It supports timed transitions, full-screen mode with content scaling, navigation between slides and it partly supports animated transitions.

The achieved results are very clear because they can be very easily measured and tested. The accuracy of displaying the slide is very high. The full-screen scaling is also working. Each presentation and each slide has their own URL and that gives many options to transform this application into a start-up. It can be for example the application that allows to remotely control the presentation by smart-phone or the application that allows to distribute the content in the network. The client side application is so called single page application and therefore it delivers convenient user experience, because there is no need to reload the whole application during the navigation process.

Chapter 2

How PowerPoint Works

This chapter covers the key aspects of PowerPoint. The first section 2.1 briefly the beginnings of PowerPoint. It is followed by section 2.2 about the current version of PowerPoint and its possibilities. The last two sections describe how the file format is designed. The third section 2.3 describes the structure of the files and the fourth section 2.4 describes selected parts in detail.

2.1 Brief Information About the Beginnings

The first version of PowerPoint – originally named Presenter, was created by a team around Robert Gaskins in 1987. His idea was to build a tool that allows people to create presentations easily. The targeted platform was Macintosh and he was inspired by already existing programmes on Macintosh such as MacDraw. Initial design counted with text, diagrams, pictures, master slide, sorter, presentation, notes and handouts.

The correct displaying text was the centrepiece of the whole idea. The needed behaviour was to have all options of word processors and also be able to freely position the text on the slide. For that problem, the text box feature was developed. It was a rectangle shaped object and it works like a word processor in a box. It had typefaces, sizes, styles, word wrap, line and paragraph spacing, margins, tabs. Text can be structured in paragraphs or bullet lists.

The next supported feature were diagrams. They were simple graphic shapes which have no fill and colour. They also supported labelling self with the text. It was determined to add additional information to diagrams. Pictures also belong to the key part of the first version. The main reason was to support fully use cases, which could not be fulfilled by PowerPoint itself, but by another program.

Master slide can be imagined like a template or theme for slides. It allows to set a uniform style and to have one place where the style is defined. The title and slide sorters work as overviews of presentations. Finally, slide show and notes were intended to support the presentation [31].

All the ideas behind described parts can be still found in PowerPoint. It shows that the initial design of PowerPoint was correct and that it fulfills the need of making presentations right.

2.2 Current Possibilities and Functions

Even though there are almost 30 years between the first and the current version, the main idea of PowerPoint is still timely. The need of creating presentations exists but the requirements are bigger than before. Nowadays, presentation software should support video files, audio files and offer visual effects, animations, transitions and templates. This section describes how PowerPoint[1] handles these features.

2.2.1 Concepts of Creating the Presentation

Each presentation consists of slides. A slide can be imagined like a container for content. The presentation does not limit the number of slides. A transition is a process when the currently displayed slide should be hidden and the other slide should be displayed. Each slide can have an unlimited number of objects. Object is the plain abstraction of specific content like text, image, sound and video. It is also possible to insert tables, graphs, diagrams, text in WordArt style and equations. Each slide is also linked with a theme that sets the default visual values.

2.2.2 Text and Its Options of Customizations

The text is supported by the feature called text field. It is the rectangle-shaped object with additional related options such as customising the text. Customizations could be applied to the field or just on the text. It is possible to set basic text-related attributes like font family, size, weight, style, colour, many kinds of underline like simple, double and dotted, upper index, lower index and font kerning. There is also a possibility of generic styling and section 2.2.6 describes them. Text can also be structured into lists. Each list contains paragraphs and each paragraph has its own level. Levels can have different indentations, bullets and numbering.

There are also numerous options how to set alignment and flow direction of words and sentences. Text can be aligned horizontally to the left, centre and right and vertically to the top, centre and bottom. The flow direction can be set to horizontal, vertical from bottom to top, vertical from top to bottom or superimposed.

If standard ways of customization are not enough, then it is possible to use WordArt effects. They can be applied to any text and they add a new effect group to the standard ones called transformations. A transformation can, for example, change the line on which the text is aligned by default, to a chosen shape like an ellipse, a circle or a wave.

Animations are also a part of text customization. Their standard options are described in section 2.2.7. But text has also some specific options. It is possible to animate each letter, each word or the whole text independently. If animated object contains more than one paragraph of text, it is possible to animate them as a single object or all paragraphs in parallel or differentiated by level of paragraphs. Paragraph animations can be started in reversed order or automatically after a custom interval.

2.2.3 Images and Their Options of Customizations

Images are inserted as rectangle-shaped objects and as objects, they support all generic customization options described in section 2.2.6. Moreover, there are also additional customizations like image cropping, changing colour saturation, colour shade, the level of

[1] Microsoft Office 365 ProPlus v15.0.4779.1002

acutance, brightness, contrast and graphic filters. Besides such styling, an image can be animated too. The options of animations are described in section 2.2.7.

2.2.4 Audio Content and Its Options of Customizations

Audio content can be recorded inside PowerPoint or included from an external file. When it is inserted, the audio image appears on the slide. The image customization is described in section 2.2.3. Moreover, the audio track itself can be customized as well. It is possible to change its volume and cut the track. An important part of using audio content is also deciding when or how the track will be played. Its start can be set to a specific time or user interaction can start it.

2.2.5 Video Content and Its Options of Customizations

Video content can be included from an external file or from a web source. When the video is inserted, its representing image appears in the slide. In most cases it is the first snapshot of the video, but it can be changed to a different image. This image can be changed or customised as it is described in section 2.2.3.

Besides the standard customization options, there are also video-specific options. These options differ in the source from which the video is loaded. If the video is loaded from a web source, then the video can not be modified. It is played the same way as in a web browser. If the video is loaded from an external source, it is packed with the presentation and it is possible to modify it. The customization of the preview image persists and it is applied during the playback. Besides, it is possible to cut, set gradual fade in or fade out of the sound and set when and how the track will start. The video is played in the shape specified by the preview image. If the video file is from an external source, it can be played in full-screen mode.

2.2.6 Generic Styling Options

These styling options can be applied to every object. These include setting position, dimensions, shape, rotation, filling, border, shadow, reflection, glow, soft border and surround effect. Most of them have several options how to specify the styling. Because there are numerous options how to combine the styles, PowerPoint offers style presets. They help a user to set the desired combination. Each object can also change its geometry from the default. It means that the shape is still rectangular, but the selected geometry crops the content.

2.2.7 Animating of Objects

Animations can be applied to any object. They can be even combined and start off at the exact time or start on the click interaction. It allows playing animations in parallel or serially. They can also have different duration, set a colour after the animation ends and play a sound during it. Each animation has individual settings that are specific to what it performs.

2.2.8 Slide Transitions

Transition is a process of changing the currently displayed slide for another one. The changing can be instant or animated. The transition can be triggered by a user interaction or after a time interval. The animation during the transition can be customised similarly as object animations described in section 2.2.7. Transition can be applied to all slides or explicitly to one slide.

2.2.9 Slide Themes

The theme can be imagined as default visual styles for each object on the slide. It defines 2 fonts with all formatting options, 12 colours and formatting scheme, which defines background fill styles, effect styles, fill styles and line styles. Themes are useful because they allow keeping uniformity of the slides.

2.3 File Format of the Presentation

This section describes the file format of the presentation because the presentation file consists of many files. Each file represents only a part of the presentation and each slide is a composition of several files. The first subsection 2.3.1 introduces used file formats and briefly explains the differences. The second subsection 2.3.2 describes the structure of the presentation file. The third subsection describes how the relations between the files works and the last subsection 2.3.4 describes how the files are composed to represent the complete slide.

2.3.1 Introduction to Used File Formats

PowerPoint used two different versions of file format during its existence. The first version is the binary archive that was internally developed by Microsoft. The second version implements the ECMA-376 standard and therefore is observable by anyone. Microsoft currently prefers this version and it is recommended to use this format before the first one. This fact means that the only version described in this section is the second one.

This file format is named Office Open XML and is also used by other productivity software. It includes spreadsheets, charts and word processing documents. It is a collection of XML files zipped into one archive. The file extension of a presentation archive is *.pptx*. Office Open XML was developed by Microsoft and was standardised to become ECMA-376. The standard covers a huge variety of documents but this thesis only deals with archives containing presentations and only with those parts that are essential for displaying their content [1].

2.3.2 File Structure of the Archive

Significant parts of archive structure are shown in Figure 2.1. All visual content is located in the folder named *ppt*. This folder contains other folders named *_rels*, *media*, *slideMasters*, *slideLayouts*, *slides*, *theme* and file named *presentation.xml*, which is the entry point of the whole presentation. Another content is linked directly to this file or indirectly through files linked directly.

Folder *_rels* contains files that defines relationships between files. Each file from this folder is related to some file that is in the same directory as the folder itself. If the content file does not explicitly defines the name of the relation file, the relation file name has to be same as the content file name. This folder is also located in other folders such as *slideMasters*, *slideLayouts*, *slides* and *theme* and has same importance as in *ppt* folder. The folder named *media* holds all files that were included in the presentation from an external

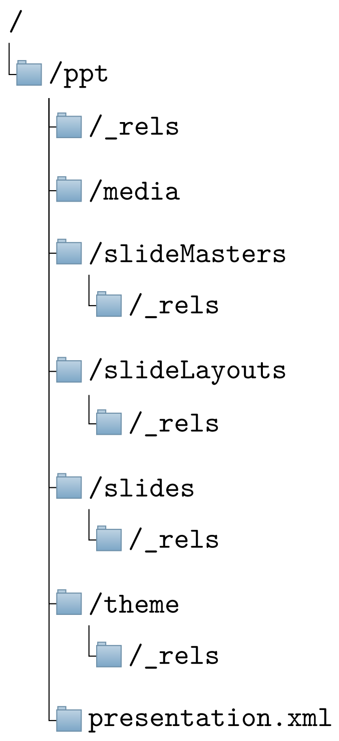


Figure 2.1: The structure of the presentation archive

source: images, audio and video. Folders named *slideMasters*, *slideLayouts*, *slides* and *theme* contain files that define the entire visual appearance of the presentation. Section 2.3.4 describes the files inside these folders [1].

2.3.3 Relationships Inside the Archive

The previous section 2.3.2 describes the structure and shows that the presentation consists of many files which are linked between each other. This required a robust mechanism to preserve order inside the archive. Because the number of identification strings may be very high, each file that has relationships has own set of relationship identification strings. The usage of them is performed by XML attributes defined in the Relationship namespace – Table 2.1. If the file that contains these relationships does not override the name of the referenced file, the referenced file is equally named. It is located in the folder named *_rels*, which has to be in the same directory as the content file is.

These referenced XML files use the Relationship definition namespace – Table 2.1. It determines root node named *Relationships* and its child nodes named *Relationship*. Each of its child nodes has to have 3 attributes named *Id*, *Type* and *Target*. Attribute *Id* refers to identification strings, attribute *Type* refers to XML schema of target file and attribute *Target* contains relative path to target file. This path is relative to the file with relationships and may refer to almost all files in the archive. The rules that specify what references are allowed in relationship file are specific to related file [1].

2.3.4 Composition of the Files that Represent the Slide

The composition begins at the entry point mentioned in section 2.3.2. It is the XML file with the Presentation namespace. – Table 2.1. Its root node is named *presentation* and contains among others an element with name *sldIdLst*. Its child nodes are named *sldId* and they represent slides of the presentation. Each of these nodes has an attribute named *id* from Relationship namespace that is shown in Table 2.1 and it refers to the file that contains data of the slide.

Except that this file contains slide specific content, it has also corresponding relationship file and some of its elements may have attributes that specify the relationship. These attributes are from the Relationship namespace that is shown in Table 2.1. The file is the XML file and the majority of its elements are from the Presentation namespace – Table 2.1. The only required relationship of this file is the relationship to the layout file.

Layout file defines default positioning and appearance of objects on the slide. The number of layout files is not limited and therefore, each slide file can be related to the different layout file. Like others, it is the XML file and the majority of its elements are from the Presentation namespace – Table 2.1. The only required relationship of this file is the relationship to the master slide file.

Master slide file is the base template. It defines objects as placeholders and sets their default appearance like layout file. Thus, it is the XML file and the majority of its elements are from Presentation namespace – Table 2.1. The only required relationship of this file is the relationship to the theme file.

Theme file defines default visual styles like the colour palette, basic fonts and formatting scheme. It is the XML file and the majority of its elements are from the Drawing namespace – Table 2.1. Even the theme file may have defined relationships, but only in a limited way. It may have only relationships to image files.

As the paragraphs above describe, numerous files define the visual appearance of slides. The styles have to be composed into the final state and it has to be specified by the exact mechanism. Its position in structure decides the priority of the style – the closer ones have higher priority than the farther ones.

2.4 Essential Parts of Current File Format

This section describes important parts of ECMA-376 standard for presentations. Mentioned node names and node types aim to help with orientation in ECMA-376 standard. Element definitions in this section contain only parts, which are necessary for visualising. Their exact definition can be found in [1], Annex A. Table 2.1 shows main namespaces that are used.

Name	**URL**
Relationship	http://schemas.openxmlformats.org/officeDocument/2006/relationships
Relationship definition	http://schemas.openxmlformats.org/package/2006/relationships
Presentation	http://schemas.openxmlformats.org/presentationml/2006/main
Drawing	http://schemas.openxmlformats.org/drawingml/2006/main

Table 2.1: Common namespaces from the OOXML format

2.4.1 Theme Container Element

Theme root node name is *theme* and it is from the Drawing namespace. Table 2.1 shows this namespace. Its type is *CT_OfficeStyleSheet* and contains definitions of font, colours and formats. The element structure is shown in Table 2.2.

Name	**Occurrence**	**Type**	**Node name**
Colour scheme	1	CT_ColorScheme	clrScheme
Font scheme	1	CT_FontScheme	fontScheme
Format scheme	1	CT_StyleMatrix	fmtScheme

Table 2.2: Structure of the *theme* element

As it is understandable from names, colour scheme defines colour palette numbering 12 colours and font scheme contains definitions of the major and minor font. Format scheme contains the list of complex styling definitions. These definitions are described in section 2.2.6. These definitions can be referenced by any *shape* element.

2.4.2 Presentation Slide Container Element

Presentation slide root node is named *sld* and it is from the Presentation namespace. Table 2.1 shows this namespace. Its type is *CT_Slide* and Table 2.3 shows its structure.

Name	**Occurrence**	**Type**	**Node name**
Common slide data	1	CT_CommonSlideData	cSld
Slide transition	0-1	CT_SlideTransition	transition
Slide timing	0-1	CT_SlideTiming	timing

Table 2.3: Structure of the *sld* element

Common slide data element is described in section 2.4.5, Slide transition element in section 2.4.11 and Slide timing element in section 2.4.12.

2.4.3 Slide Layout Container Element

Slide layout root node is named *sldLayout* and it is from the Presentation namespace. Table 2.1 shows this namespace. Its type is *CT_SlideLayout* and Table 2.4 shows its structure.

Name	**Occurrence**	**Type**	**Node name**
Common slide data	1	CT_CommonSlideData	cSld
Slide transition	0-1	CT_SlideTransition	transition
Slide timing	0-1	CT_SlideTiming	timing
Header and footer	0-1	CT_HeaderFooter	hf

Table 2.4: Structure of the *sldLayout* element

Description of Common slide data element is shown in section 2.4.5, Slide transition element in section 2.4.11, Slide timing element in section 2.4.12 and Header and footer element in section 2.4.13.

2.4.4 Slide Master Container Element

Slide master root node is named *sldMaster* and it is from the Presentation namespace. Table 2.1 shows this namespace. Its type is *CT_SlideMaster* and its structure is shown in Table 2.5.

Name	**Occurrence**	**Type**	**Node name**
Common slide data	1	CT_CommonSlideData	cSld
Colour map	1	CT_ColorMapping	clrMap
Slide transition	0-1	CT_SlideTransition	transition
Slide timing	0-1	CT_SlideTiming	timing
Header and footer	0-1	CT_HeaderFooter	hf
Text styles	0-1	CT_SlideMasterTextStyles	txStyles

Table 2.5: Structure of the *sldMaster* element

Description of Common slide data element is located in section 2.4.5. Colour map node defines the mapping between colours from theme and colour names used in the slide, slide layout and slide master. Slide transition element is described in section 2.4.11, slide timing element in section 2.4.12, header and footer in section 2.4.13 and text styles in section 2.4.9.

2.4.5 Common Slide Data Element

Common slide data is the main element that contains slide content. This element is named *cSld* and its type is *CT_CommonSlideData*. Its structure is shown in Table 2.6.

Name	**Occurrence**	**Type**	**Node name**
Background	0-1	CT_Background	bg
Shape tree	1	CT_GroupShape	spTree

Table 2.6: Structure of *cSld* element

The background element defines slide background appearance. It may contain styling options. These options are described in section 2.2.6. Shape tree element is the container element for all visual objects. The important parts of its structure are displayed in Table 2.7.

Name	**Occurrence**	**Type**	**Node name**
Non visual group shape properties	1	CT_GroupShapeNonVisual	nvGrpSpPr
Group shape properties	1	CT_GroupShapeProperties	grpSpPr
Shape	0-unlimited	CT_Shape	sp
Group shape	0-unlimited	CT_GroupShape	grpSp
Graphic frame	0-unlimited	CT_GraphicalObjectFrame	graphicFrame
Connection shape	0-unlimited	CT_Connector	cxnSp
Content part	0-unlimited	CT_Rel	contentPart
Picture	0-unlimited	CT_Picture	pic

Table 2.7: Structure of *spTree* element

The non visual group shape properties element specifies identification properties, shape properties and application properties. It means for example definition of a unique identification number, visibility, name and text description of the group. Group shape properties element defines the position, dimensions, filling and effects that are applied to the whole group. Available customizations are mentioned in section 2.2.6. Detailed description of the shape element is located in section 2.4.6. The graphic frame element is used for embedding external objects. It includes for example equations and tables. The element shall optionally have an image, which is used instead of loading actual object data. The connection shape element is intended to connect two shapes. The element specifies the start point, end point and no other points in between. It follows that the desired path might be different depending on the specific needs of the application and therefore it is entirely up to the presentation viewer. The content part element specifies a reference to XML content in a format different from ECMA-376. This can be used for formats such as SVG and MathML and it allows to extend the format. The picture element is used for inserting images, audio and video. Its description is located in section 2.4.10

2.4.6 Shape Element

The shape element is one of the key content holding elements. Its node is named *sp* and the main parts of its structure are shown in Table 2.8.

Name	**Occurrence**	**Type**	**Node name**
Non visual shape properties	1	CT_ShapeNonVisual	nvSpPr
Shape properties	1	CT_ShapeProperties	spPr
Shape styles	0-1	CT_ShapeStyle	style
Text body	0-1	CT_TextBody	txBody

Table 2.8: Structure of *sp* element

The non visual shape properties element defines identification properties, shape properties and application properties. It means for example identification number, visibility, name and text description of the object. Shape properties element specifies the position, dimension, filling and effects that are specific to the shape. Available customizations are mentioned in section 2.2.6. Shape styles element contains definitions of used font colour, fill colour, line colour and effect colour. It can be directly defined in the element or it can contain a reference to the format scheme in related theme element. The text body element is dedicated to holding text with styles and custom formatting. Its description is located in section 2.4.7.

2.4.7 Text Body Element

Text body element is the child node of the shape element and contains text values of the shape and their formatting options. Table 2.9 shows its overview.

Text body properties element defines text anchoring, side insets, rotation and text overflow behaviour. It also specifies how the text should fit into the bounding box. If the text, for example, does not fit in and it should, then line space reduction and font scaling is specified to suit this requirement. Text list styles element specifies how the text lists should be visualized and section 2.4.9 describes these options. The description of text paragraph element is located in section 2.4.8.

Name	Occurrence	Type	Node name
Text body properties	1	CT_TextBodyProperties	bodyPr
Text list styles	0-1	CT_TextListStyle	lstStyle
Text paragraph	1-unlimited	CT_TextParagraph	p

Table 2.9: Structure of *txBody* element

2.4.8 Text paragraph element

The text paragraph structure is shown in Table 2.10.

Name	Occurrence	Type	Node name
Paragraph properties	0-1	CT_TextParagraphProperties	pPr
Regular text run	0-unlimited	CT_RegularTextRun	r
Text line break	0-unlimited	CT_TextLineBreak	br
Text field	0-unlimited	CT_TextField	fld

Table 2.10: Structure of *p* element

The paragraph properties element specifies line spacing, sides spacing, alignment, indentation and level, to which it belongs. It also contains the specification of visualising the bullet of the list. Paragraph may also contain default properties for regular text run element. Regular text run element represents a part of the text within the text body. It is the lowest level of text separation within the text body. Regular text run may have run properties. If it has no properties, then the default run properties are applied. Its properties specify font specific attributes like size, weight, styles and kerning. The whole text is inside the child element named *t*. Text line break element represents newline and it may contain only run properties. Finally, text field element works very similarly to the regular text run. Besides it may have paragraph properties too, which override paragraph properties from the paragraph element.

2.4.9 Text List Styles

The text list styles element is a collection of many paragraph properties elements. The specification of the paragraph properties element is in section 2.4.8. There is one default paragraph properties element and many paragraph properties elements for each level of the list. All these elements are optional.

2.4.10 Multimedia Element

This element represents any form of multimedia data such as image, video and audio. Its type is *CT_Picture* and its structure is shown in Table 2.11.

Name	Occurrence	Type	Node name
Non visual picture properties	1	CT_PictureNonVisual	nvPicPr
Picture fill	1	CT_BlipFillProperties	blipFill
Shape properties	1	CT_ShapeProperties	spPr
Shape style	0-1	CT_ShapeStyle	style

Table 2.11: Structure of *pic* element

The non visual picture properties element specifies identification properties, picture properties and application properties. It is, for example, the identification number, visibility, name and text description of the object. The picture fill element is the primary element for visual appearance. It contains a reference to the image and image customizations. These customizations are described in section 2.2.3. Shape properties element defines basic attributes of the shape and section 2.2.6 describes them. The shape style element is used to hold the reference to the style definition from the format scheme. The format scheme is described in section 2.4.1.

2.4.11 Slide Transition

This element specifies the kind of transition that should be applied. The element contains the specification of the transition like the speed of the execution and the specification of what triggers the execution. The transition can be triggered by user interaction or after a specified time.

2.4.12 Slide Timing

This element specifies all information for controlling animations and timed events within the slide. It includes primary definitions of animations. The important aspect of it is the timeline. It moderates the amount of time from the beginning to the end. The timeline is composed of time nodes that have various types. These types define how time nodes are running. It can be in parallel, sequentially or exclusively. Exclusively running time node means that there can not be another time node that runs too. The next important part of time nodes are conditional properties. They specify conditions that have to be met to continue with animation.

2.4.13 Header and Footer

This element specifies the content of header and footer. Header and footer are special placeholders that should be consistent across all slides. It includes information such as date, time and slide number. The content itself is auto generated and therefore, these features can be only enabled or disabled.

Chapter 3

Client side Technologies

The current chapter is dedicated to the development the application from the perspective of the client. It consists of three sections. The first section is dedicated to building of the user interface. The second section describes how the client environment works and how to develop robust client application. The last section describes libraries which were used.

3.1 Creating a User Interface

User interface of web applications is primarily built using the HTML markup language and Cascading Style Sheets (CSS). This section describes possibilities of this approach from the essential points of view. This section reviews only some attributes of the visualising and presupposes a basic knowledge of the HTML markup language and CSS. This section is mainly based on the sources [27], [28] and [9].

3.1.1 Sizing of an Element

CSS offers several units to specify the length. There are two categories of length units named absolute and relative and the special unit called pixel (px). Pixel unit is the relative unit to the viewing device. For low-resolution devices, 1 pixel is 1 device pixel. For high-resolution devices, 1 pixel is more device pixels. Table 3.1 shows an overview of the absolute length units. The usage of these units results in the same sizing for every device, but only if the output device has a high enough resolution. Table 3.2 shows overview of the relative length units. These units scale with the device and therefore it is recommended to use them in the majority of use cases.

Unit	**Description**
cm	centimetres
mm	millimetres
in	inches (1in = 96px = 2.54cm)
pt	points (1pt = 1/72 of 1in)

Table 3.1: Overview of the common absolute length units

The sizing of every element consists of margin, border, padding and the actual content. Because all HTML elements can be considered as boxes, the model that specifies these properties is called Box Model. Figure 3.1 illustrates this model. The content part is the

Unit	Description
em	Relative to the font size of the element
rem	Relative to the font size of the root element

Table 3.2: Overview of the common relative length units

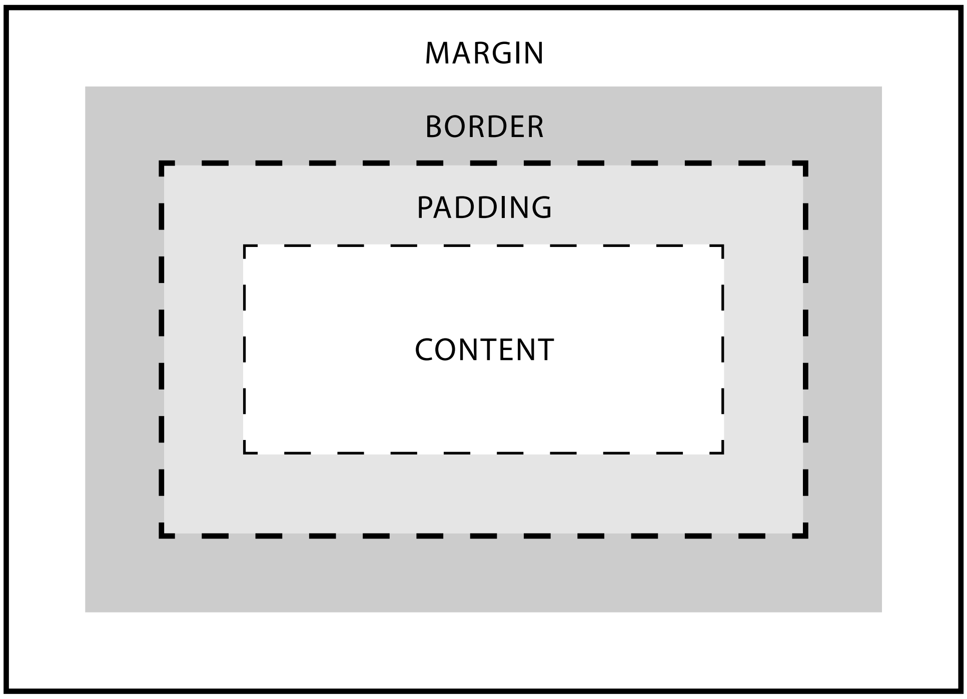


Figure 3.1: Overview of the CSS box model. The size of the element consists of the size of the content, padding, border and margin.

box, where the text, images or other content appear. Padding clears the area around the content and it is transparent. Border wraps around the content and padding. The last part is margin and it is another area that wraps around and it is also transparent. This model is important to set properly sizes of an element. CSS has a property called *box-sizing* that allows to set how the width and height properties are computed. The default behaviour is that the width and height properties include only the content. The second possible behaviour is that the width and height properties include content, padding and border. The difference between the margin property and the padding property is that the margin property can collapse. It means that when two elements have margins that overlapped, the actual distance between them is the minimal possible and not its sum.

3.1.2 Positioning of an Element

CSS offers numerous properties that define the positioning. The first important property is called *display*. This property affects the way how the element will be displayed. Table 3.3 shows the overview of the most used values of this property.

An inline element does not start on a new line and its content defines its width. A

Value	**Description**
inline	Displays an element as an inline element
block	Displays an element as and block element
inline-block	Displays an element as an inline-level block container
flex	Displays an element as an block-level flex container
inline-flex	Displays an element as an inline-level flex container
none	The element will not be displayed

Table 3.3: Overview of the *display* property

block element always starts on a new line. It will take up the full available width if the width is not explicitly specified. A block element can not be inside an inline element. An inline-level block container is an inline element that may have specified a custom width. The described behaviour is shown in Figure 3.2.

A block-level flex container is a new feature from CSS3. The difference between the *flex* and *inline-flex* value is that they set different display behaviour of the container. If the property value is *flex*, the container is a block element. If the property value is *inline-flex*, the container is an inline element. Both of them enable a flex context for all its direct children. The wrapping element is called flex container and the children elements are called items. The flex context includes a whole set of properties and some of them are meant to be set in the container and some of them are meant to be set in the children. The flex context, for example, allows to align items vertically and horizontally, set the direction how the items are positioned, the order of the items and more. Vertical and horizontal alignment is very useful as it allows to align an item to the left, right, top, bottom and middle. The flex context disables the standard positioning properties and therefore they can not be used. The last mentioned value is called *none* and it hides the element completely.

Value	**Description**
static	Elements render in order, as they are in the document
relative	The element is positioned relative to its normal position
fixed	The element is positioned relative to the browser window
absolute	The element is positioned relative to its first positioned parent element

Table 3.4: Overview of the *position* property

The second property is called *position*. Table 3.4 shows an overview of its common values. The first possible value is *static* and it is the default value. The element with this property and value is positioned according to the normal flow of the page. The second value is *relative*. It allows displaying the element relative to its normal position. The positioning is set by properties *top*, *right*, *bottom* and *left*. Other content does not fill the gap between the normal position and the actual position. The next possible value is *fixed*. This value allows setting the position relatively to the browser window. It means that it always stays in the same place even if the page is scrolled. The positioning is set by properties *top*, *right*, *bottom* and *left* and the element does not leave a gap between the normal position and the actual position. The last possible value is *absolute*. An element with this property and value is positioned relatively to the nearest positioned parent. If an absolute positioned element has no positioned parent, it uses the document body and moves along with page scrolling.

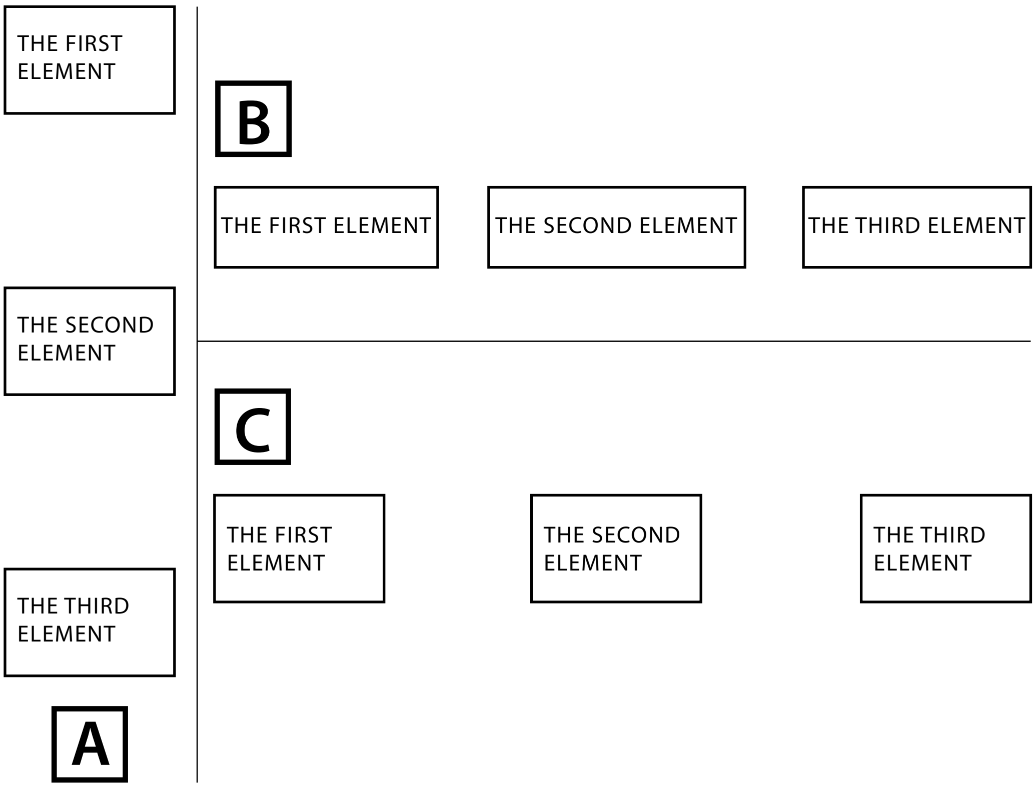


Figure 3.2: How the *display* property affects the positioning. Section A illustrates three block elements. Section B shows three inline elements. Section C shows three inline-block elements.

Value	Description
none	The element is not floated
left	The element floats to the left
right	The element floats to the right

Table 3.5: Overview of the *float* property

Value	Description
none	Allow floating elements on both sides
left	Floating is not allowed on the left side
right	Floating is not allowed on the right side
both	Floating is not allowed on both sides

Table 3.6: Overview of the *clear* property

The last described properties are called *float* and *clear*. Tables 3.5 and 3.6 show overviews of their values. These properties are complementary and allow or forbid floating of the elements to the left or the right.

3.1.3 Typography in the Web Browser

The basic typographic terms are shown in Figure 3.3. The typeface is the letters, numbers and symbols that make up a design of type. Typefaces are usually designed together and they are intended to be used together. Therefore, they group into families. The two main groups of the typefaces are called serif and sans serif. Serifs are small decorative strokes that are added to the end of letter's main strokes. Serif typefaces have letters with serifs and sans serif typefaces do not have letters with serifs. The next term is font weight. It is the relative darkness of the characters in the various typefaces within a font family. The horizontal space between individual characters in a line of text is called kerning. The majority of the characters in a typeface rest on an imaginary line called baseline. Similarly, there is also an imaginary line on the top of the characters and majority of them sits between these two lines. The height from the baseline to this line is called cap height. The parts of the letters that are below the baseline are called descenders. The parts of the letters that are above the cap height are called ascenders. The height of the lower-case letters excluding the ascenders and the descenders is called x-height. The size of the type is the distance from the top of the highest ascender to the bottom of the lowest descender. The last term is called line height. It is the height between two lines of the text [29].

CSS allows setting many of these properties. It is possible to set font, font-size, font-weight, font-style, kerning and line-height. The correct use depends on the selected font in the web browser. If the web browser does not have the current font, the most similar and available font is automatically selected. It is also essential for the font weight and font styles because these properties specify the type of the font. If the selected font does not support specified font weight or font style, their usage do not have the effect.

3.2 Concepts of Developing the Application

It is convenient when development of an application stands on some concepts. It is important to understand these concepts to embrace the possibilities of their usage fully. The

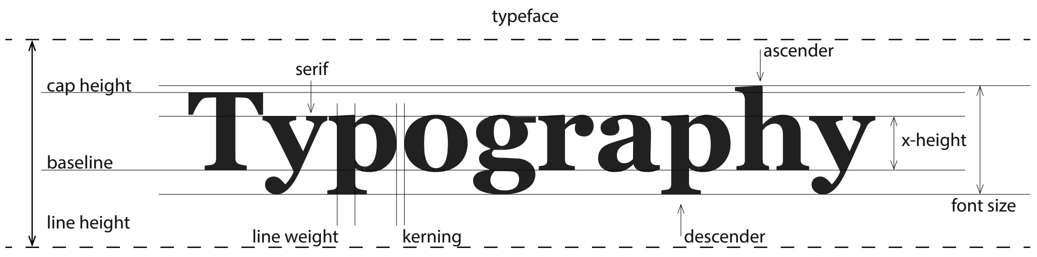


Figure 3.3: Basic fonts terms

first concept is about the environment of the web browser development and section 3.2.1 describes this concept. The second section 3.2.2 describes the application architecture because it affects the entire application. The last section 3.2.3 describes used procedures of the developing.

3.2.1 How Web Browser Works

Web browser allows accessing web pages. The standard web page consists of HTML markup, CSS code, JavaScript code and additional content files such as pictures, video, audio and fonts. When the web browser receives the web page, it starts with parsing, rendering and executing the codes. The browser has a JavaScript engine that interprets the JavaScript code, handles the memory and optimises its performance. The code in the page is executed immediately after its parsing and before parsing the following elements. It can cause problems if the code is somehow interacting with the elements below. Also, the browser waits until the execution is completed. The browser offers a programming interface for HTML, XML and SVG documents called Document Object Model (DOM). This model represents the document and the JavaScript code may change its state. The DOM model specifies numerous events and it is possible to register a JavaScript function that will be run when an event occurs. This approach is convenient because JavaScript engine implements a concurrency model based on the event loop. Figure 3.4 shows this concept.

The first part of the concept is the stack. The stack contains frames. One frame represents one function call. When a function is called, the new frame is created and pushed on the top of the first one. When a function returns, its frame is popped out. The second part of the concept is the heap. The heap is a structure where objects are allocated. It is a part of the memory. The third part is the message queue. It is the list of messages that will be processed. Each message is associated with a function. When the stack is empty, a message is taken out of the queue and processed. The processing consists of calling the associated function and therefore it fills the stack with frames. The stack is empty after the message processing ends. The event loop represents the mechanism where the queue waits synchronously for a message when the queue is empty. When the queue is not empty, the first message is taken out and the engine starts with its processing. Each message is processed completely before any other message. If a message takes too much time to process, it will lead to unresponsive behaviour of the application. When an event occurs and there is an event listener attached to it, the web browser adds a message into the event loop [21].

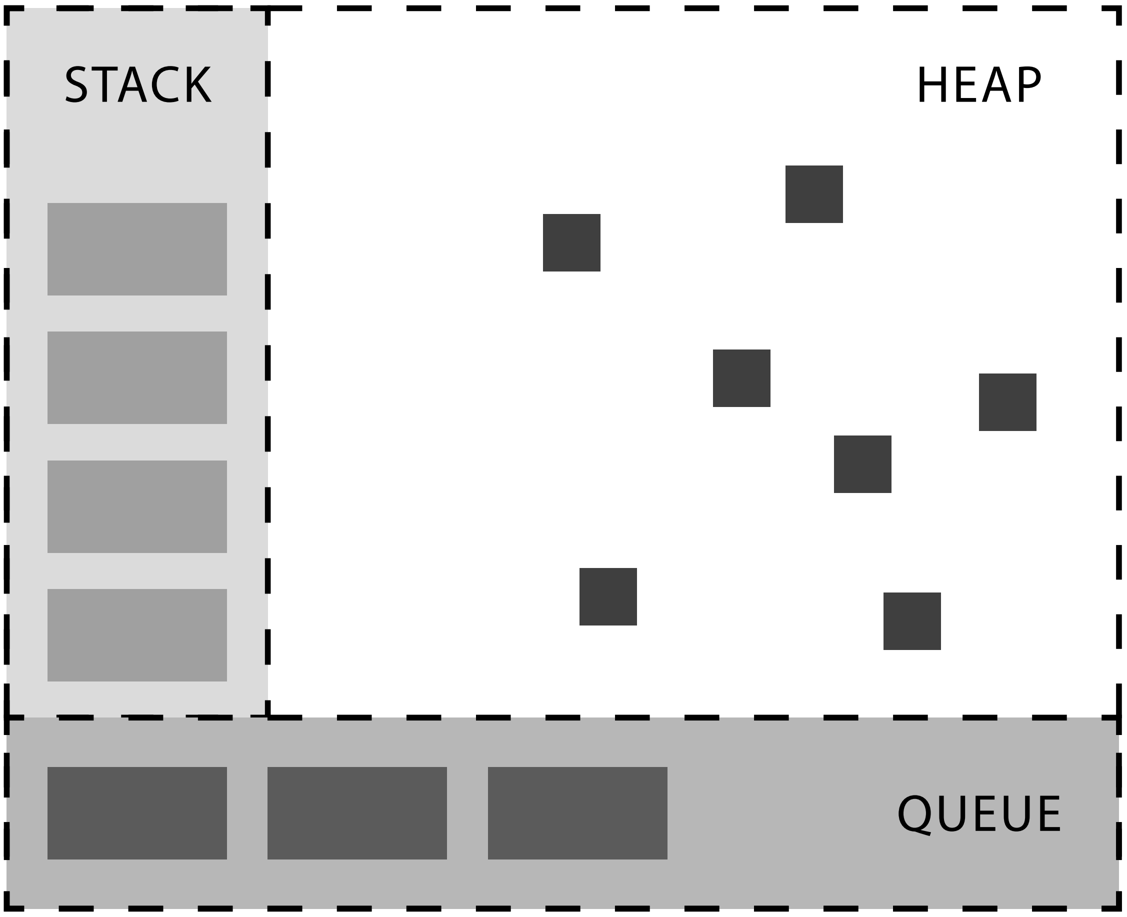


Figure 3.4: How the JavaScript engine works in the web browser. It consists of the stack, heap and queue. The stack holds the current state of the function calls. The heap serves to save the objects. The queue contains messages. The message determines what have to be executed. When the queue is empty, the engine waits until a message is added [21].

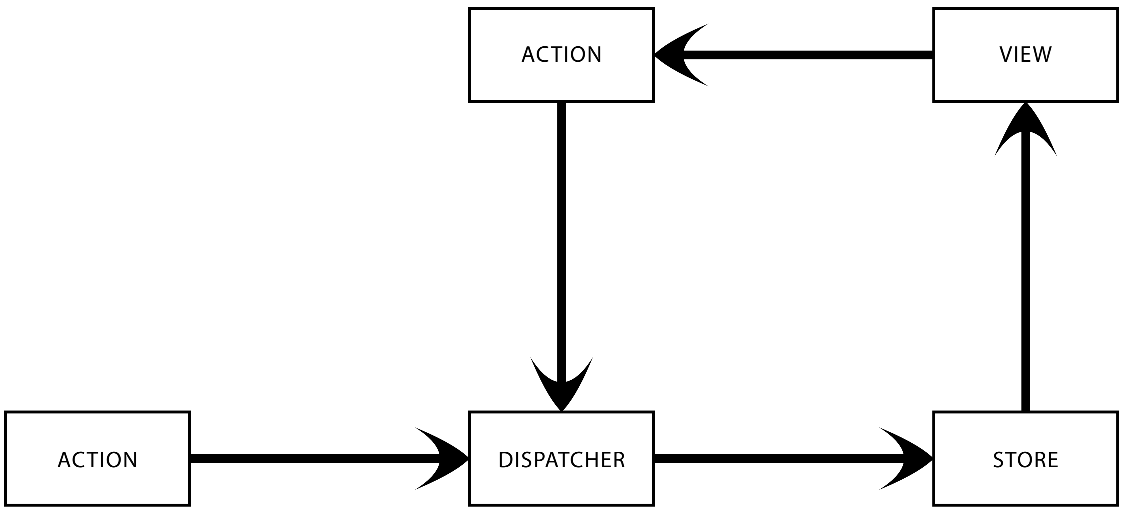


Figure 3.5: Scheme of Flux architecture [4]

3.2.2 The Actual Application Architecture

The actual application architecture is based on the Flux architecture. Flux architecture consists of the dispatcher, the stores and the views and it is shown in Figure 3.5. It is based on the idea of unidirectional data flow. When an interaction appears with a view, the view propagates an action through the central dispatcher to the stores that hold the data of the application and business logic. The change is subsequently propagated to the connected views and they update themselves. The reason for this approach is to have better control of the updates of the application state [4].

The actual application architecture is a little different because it uses Redux library – section 3.3.1, which evolves from the Flux architecture. Figure 3.6 shows the Redux implementation of the Flux architecture.

Redux implementation has only one *Store* object, where the application state is stored. *Store* is an object that allows manipulation with the state. The state is read-only and the only way of mutating the state is to emit an action. An action is a simple object that describes what happened. The state mutating is performed by reducers, functions that take the previous state and action and return the next state. Reducers have to be subscribed to *Store* to process dispatched actions. The way of processing a synchronous action is shown in Figure 3.6. It all starts with an interaction. This can be a user interaction such as a mouse click or it can even be a timed interaction. In the callback function which is hooked to the interaction, an action is created by the action creator function. After that, the dispatcher function dispatches the action. The store automatically passes the action to the reducers. All this results in a new application state and it may lead to redrawing of connected components. While the component is connecting to the *Store* object, it may specify which part of the application state affects the component. This procedure is not applicable to an asynchronous action. This can be supported by an additional library called Redux thunk – section 3.3.1. This library allows action creators to return not only an action object but also a function. The whole asynchronous process can be built from three synchronous actions inside this function. The first action describes the action that

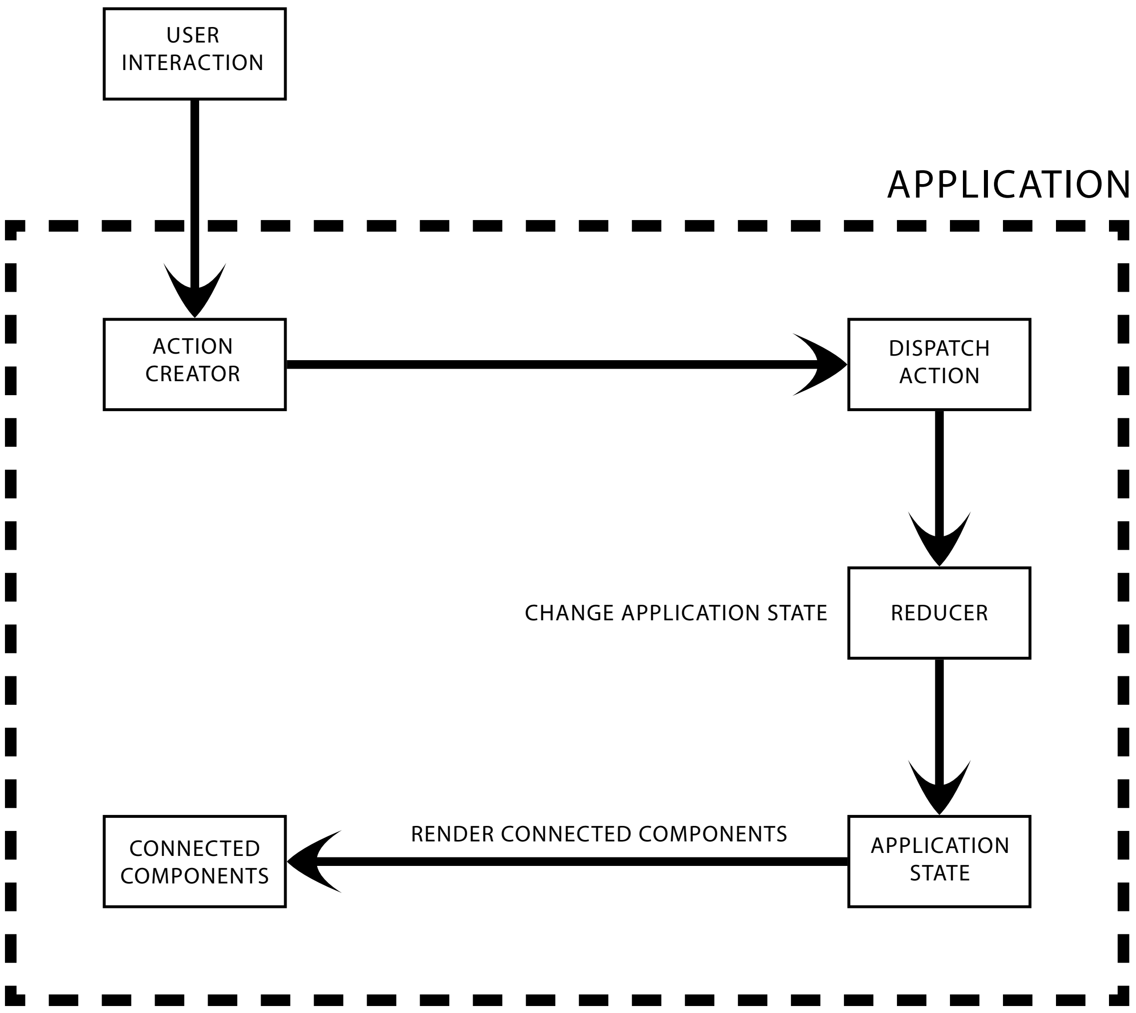


Figure 3.6: Redux implementation of the Flux architecture. An interaction function calls the action creator method and dispatches the returned action. The action is passed to the reducer that performs the changes in the application state. These changes are delivered to the connected components and they reacts to the current application state.

starts the asynchronous action. The second one is fired when the asynchronous action is successful and the third one is fired when the asynchronous action failed.

The difference of the Redux implementation is that the business logic is mainly in the action creators and that the it has a single *Store* object. When the application grows, instead of adding stores, the root reducer is split into smaller reducers independently operating on the different parts of the state tree. The main advantage is that this structure allows tracking every state mutation to the action that caused it. It relieves the developing process because the state can be easily reproduced any time [3].

3.2.3 The Ways How to Write the Application Code

The view part of the application consists of the React components. A component works as an independent unit and it is preferable to split the user interface into numerous components. The user interface is built as the composition of these components. React components are described in section 3.3.2.

The composition of the components follows several rules that improve re-usability and lead to the more readable source code. Components should be as stateless as possible. The state is stored in the *Store* component and there is no real reason to change it. When the component has to interact with the state, it should be connected to the *Store*. The connection should be as close as possible to the actual component and should always be as specific as possible. When this rule is not followed, it may result in performance issues because the connected components will be redrawn every time the specified part of the state changes [3].

	Presentational components	**Container components**
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Generated by React Redux or written by hand

Table 3.7: Features overview of the presentational and container components [23]

The components should be split into presentational and container components. This division is convenient for the cooperation with Redux library and also with other libraries that handle the application state. A summary of their features is shown in Table 3.2.3. This approach is good for the following reasons. Firstly, it offers better re-usability, because the presentational components have no application logic. That allows having more container components with different logic for one presentational component. It also offers a better separation of concerns. It separates application logic from UI components, which is better for testing the components [23].

The functionality can also be shared between the components. This principle is called higher order components and it is based on composition of the components. It allows sharing functionality that somehow interacts with life-cycle hooks across many components. It is a function that takes one parameter, which is the inner component, and the result is a new component, which implements new functionality and renders the inner component. This approach allows to apply a higher-order principle to any component many times [22].

3.3 Overview of Used Libraries and Tools

This section describes the key libraries and their possibilities. These libraries can also be run on the server, but they are primarily for the client. The way how to handle and deploy these dependencies to the client describes section 4.1.3. The first section 3.3.1 describes the Redux library. This library is the key library for the application architecture and it is described in section 3.2.2. The second section 3.3.2 describes the React library. This library is the key library for the application's user interface. The third section 3.3.3 describes the React-router library. This library handles the client application routeing. The last section 3.3.4 describes the Velocity-react library. This library allows animations of React components.

3.3.1 Redux – the Predictable State Container for JavaScript Application

Redux library is built on three fundamental principles. The first is that the state of the whole application is stored in an object tree within a single *Store*. This is useful for universal application because the state can be easily serialised and hydrated into the client. The second principle is that the state is read-only. The only way to mutate the state is to emit an action. An action is an object that describes what happened and optionally carries data. All mutations are centralised and happen one by one in a strict order. Because actions are plain objects, they can be easily logged and stored for debugging or testing purposes. The last principle is that changes are made with pure functions. These functions are called reducers and they take the previous state and an action and return the next state [3].

The standard *Store* object supports only synchronous data flow. Middlewares can enhance the process of dispatching actions. Middleware called Redux-thunk allows to dispatch functions and middleware called Redux-promise allows to dispatch *Promise* objects. Section 4.1.4 describes them [3].

Redux library does not have any relation with React library, but there is other library called React-redux that implements the connection between them. The recommended option is to render special React component called *Provider* once in the root component. It makes the store available to all components in the application without passing it explicitly. When the component has to interact with the application state, it can be connected by the function called *connect*. This function accepts three optional parameters. The first parameter is a function that maps state to the component's props and returns the result. It is then merged with other props and passed to the component. The second parameter is a function that maps action creator functions into the component's props and returns the result. The third parameter is a function that maps the results of the two previous functions and actual props to the component's props [2].

3.3.2 React – the JavaScript Library for Building User Interfaces

React [25] is a library for creating user interfaces. The most common use case with user interfaces is displaying data. React components can be imagined as functions that take in *state* and *props* and render HTML markup. Each update can result in different HTML markup and many DOM operations. Because DOM operations are slow, React internally minimises the number of operations with DOM with a shadow mock DOM. It takes the old one and the new one compares them and computes the minimal number of actual DOM operations. These operations are performed to the real DOM. The shadow mock DOM is used, because it is more lightweight than the actual DOM and therefore the comparison of shadow mock DOMs is faster. Even though React is the client library, it allows rendering the components on the server, too.

React component takes in two parameters affecting the output. The first parameter is named *props* and it can be imagined as parameters of the component. There is no limitation on what can be taken in. It can be a scalar value, a function, an object or even a React component. The second parameter is called *state*. It allows the component to handle its own state. The state is stored in the component. React components should be as stateless as possible. When no other mechanism of handling the application state is used, it is possible to use this parameter. The main difference between the *props* and the *state* is that the *props* are read-only.

React component specification contains one key method called *render*. This method has to return a single child element. It can be either a virtual representation of a native DOM component or another composite component. This method should not modify the component state and interact with the browser. Its output should be the same every time the input parameters are the same. React component specification also contains methods which are executed at specific points during its life-cycle. Their overview is shown in Table 3.3.2.

Method name	**Life-cycle phase**
componentWillMount	Mounting
componentDidMount	Mounting
componentWillReceiveProps	Updating
shouldComponentUpdate	Updating
componentWillUpdate	Updating
componentDidUpdate	Updating
componentWillUnmount	Unmounting

Table 3.8: React component's life-cycle methods

There are three life-cycle phases. The first phase is called Mounting. This phase consists of two methods. The first method is *componentWillMount*. This method is invoked once before the initial rendering occurs both on the client and the server. The second method from this phase is *componentDidMount*. This method is invoked once only on the client and immediately after the initial rendering occurs. At this point of the life-cycle, the underlying DOM is accessible. The second phase is called Updating. This phase consists of four methods. The first method is named *componentWillReceiveProps* and it is invoked when a component is receiving new *props*. This method is not called for the initial render. This method is useful to react to *props* transition before the render. The second method is called *shouldComponentUpdate*. This method is invoked before the rendering but not before the initial rendering. This method decides if the component should be rendered again. The third method is named *componentWillUpdate*. This method is invoked immediately before the rendering and not for the initial rendering. This method is useful for preparation of an update. The last last method is called *componentDidUpdate*. This method is called immediately after the component's updates are flushed to the DOM. This method is also not called for the initial render. The last phase is called Unmounting. This phase contains only one method called *componentWillUnmount*. This method is called immediately before a component is unmounted from the DOM. This method should perform cleanup operations such as invalidating timers.

3.3.3 React-router – the Complete Routing Library for React

React-router library keeps the UI synchronised with the URL. The key component from this library is the *Route* component and it specifies which component has to be displayed according to the URL. If the URL has some parameters, then these parameters are passed to the component. The next component is named *Router*, it consists of the *Route* components and it should be the root node of the application. Other parts of the library allow redirecting the application programmatically to the different URL, creating links to URLs, handling a web browser history and server-rendering [26].

3.3.4 Velocity-react – React components for Velocity.js

Velocity-react library allows animating components with the Velocity.js DOM animation library. It contains two different components which wrap the animations. The first component is named *VelocityComponent*. It wraps around the component that will be animated and allows to specify the animation. The animation is not fired when the component is mounting but it is possible to allow it by the parameter called *runOnMount*. The second component is called *VelocityTransitionGroup*. This component wraps around the multiple components and allows to specify the entering animation and the leaving animation. The entering animation is fired on a child component that is being added. The leaving animation is fired on a child component that is being removed. The component's animations can also be disabled. It is useful for testing purposes [34].

Chapter 4

Server side Technologies

The following chapter describes the core knowledge from the perspective of the server. The first section outlines the basics of Node.js development. It is important to know how the server works so that one understands how to write the code for it. The second section describes the architectural style behind the whole application. The last section describes main libraries that were used to develop the application.

4.1 Concepts of Developing the Application in Node.js

Node.js is a JavaScript runtime built on V8 JavaScript engine. With new releases, it is keeping up-to-date with this engine and it supports new features from the ECMAScript 2015 specification. Sections 4.1.1 and 4.1.2 describe the runtime in detail. The next important tool after the runtime is called Npm. It is the package manager for JavaScript. Section 4.1.3 describes this tool. The last section 4.1.4 describes the way of writing the asynchronous code and the approaches that may help to make it more readable and maintainable.

4.1.1 The Way the Node.js Runtime Works

Node.js is an asynchronous event-driven JavaScript runtime. It has a built-in event loop and majority of the functions work with it. It allows the concurrency with a single thread worker and without threads from the operating system. The asynchronous functions or so-called non-blocking functions have to have a parameter that specifies a function that is called after the operation is executed. For example, when the process reads a file, the process does not wait until the file is read, but it continues with the execution. The process only gives the instructions to the operating system to perform the reading operation. When the reading operation is over, the operating system notifies the process with the result and the process calls back the given function. That function is called the callback function. When the Node.js process starts, it runs all the commands from the given script and after the execution, it enters into the event loop. When there are no more callback functions, it exits the event loop and exits the script [13] [15].

The core of the Node.js application is the event loop. When the Node.js application starts, it initializes the event loop, processes the given script which may make asynchronous calls, schedules timers or calls *process.nextTick()*, then begins processing the event loop. Figure 4.1 shows the phases of the event loop process.

The first phase is called *Timers*. This phase executes callback functions scheduled by *setTimeout()* and *setInterval()* functions. These functions specify the interval after which

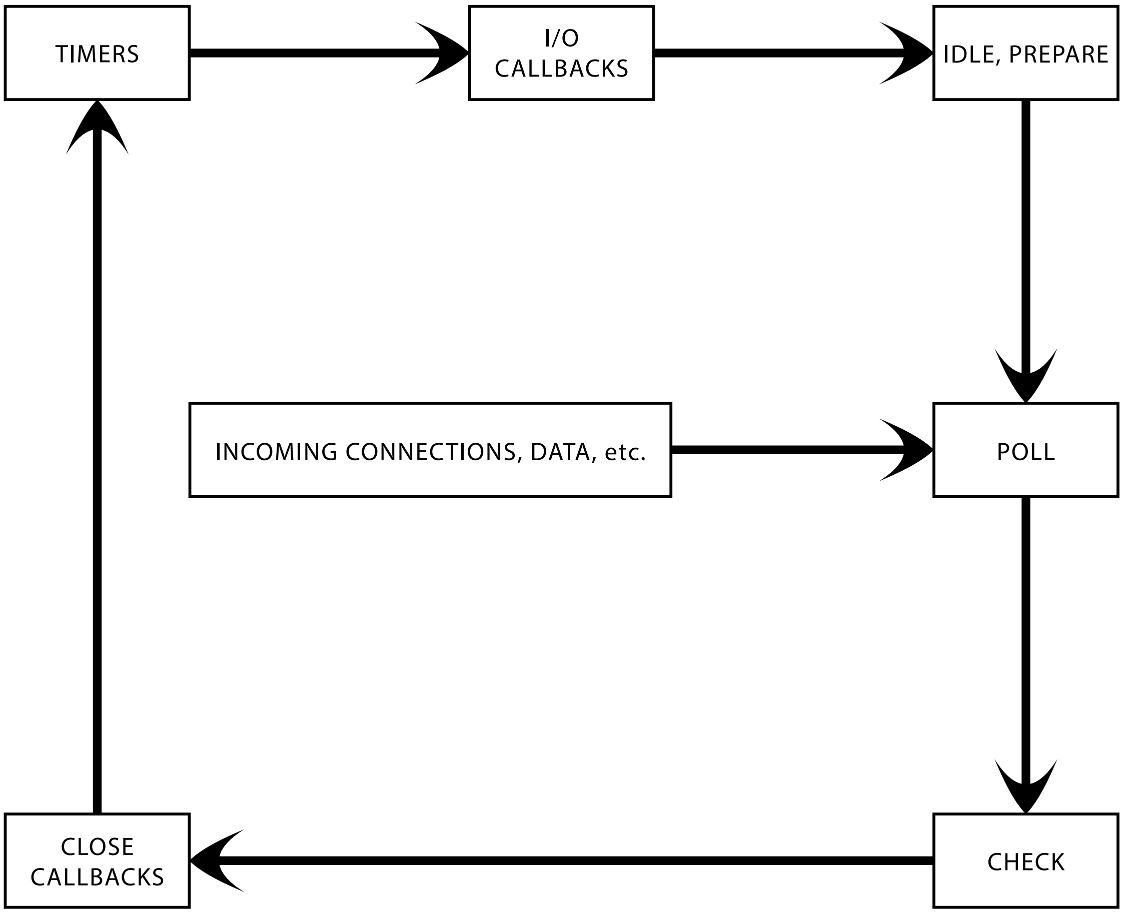


Figure 4.1: The execution phases of the Node.js runtime. Each box represents a phase, where the event loop is. Each phase has a FIFO queue of callback functions to execute. When the event loop enters the given phase, it will perform any operations specific to that phase. Then it executes callback functions from the phase's queue until the queue has been exhausted or the maximum of executed callback functions has been reached. After the execution, the event loop will move to the next phase [14].

the passed callback function may be executed. It is important to differentiate that it does not specify the exact time after the callback function is executed. If the event loop is processing an event during the time, when the callback should be executed, the callback is executed after the event is finished. The second phase is called *I/O Callbacks* and it executes some callbacks for system operations. Its queue is for example used for reporting TCP errors in some operating systems that wait to report the error. The third phase is for internal usage and it is not important for understanding the event loop. The fourth phase is called *Poll* and has two functions. The first one is that it executes the scripts of timers with elapsed threshold. The second one is that it processes its event queue. When the event loop starts with this phase and there are no timers scheduled, there are two options of the sequel. If its queue is not empty, then the event loop will iterate through them and execute them synchronously until the queue is empty or until the predefined limit of the executed events is reached. If its queue is empty, there are two options of the sequel. If scripts have been scheduled by *setImmediate()*, the event pool will end this phase and continue to the phase named *Check*. If scripts have not been scheduled by *setImmediate()*, the event loop will wait for callback functions to be added to the queue then executes them immediately. It can be for example a callback function with an incoming connection. When its queue is empty, the event loop will check for the timers with timed out thresholds. If one or more timers are ready, the event loop will enter *Timers* phase to execute these callbacks. The next phase is called *Check*. This phase allows to a developer to execute a function right after the *Poll* phase. Its queue can be filled by the *setImmediate()* function calls. The last remaining phase is called *Close callbacks*. If a socket or a handle is suddenly closed, the appropriate event will be emitted in this phase. Otherwise, it will be emitted by the call of the *process.nextTick()* function.

The function *process.nextTick()* is important from the point of view of the asynchronous API. It allows executing a callback function after the current operation completes and it is executed in the same phase. It allows putting the asynchronism into synchronous operations. For example, when a port is passed to the server, the server starts to listening on that port immediately. If there is the event that represents the start of the listening, the event will be fired immediately. If the event will be emitted in the callback function passed to *process.nextTick()* function, it starts to behave asynchronously [14].

4.1.2 Handling the Error State

Node.js supports numerous mechanisms for handling errors. The first mechanism is standard *try / catch* construct provided by the JavaScript language. This mechanism works for the synchronous code but does not work for the asynchronous. The reason is that the callback function is executed outside of the context of the *try / catch* construct. An error of the asynchronous operation is passed as the first parameter in the callback function. Therefore, the error handling has to be inside the callback. Even though it is common to handle the error state in the callback itself, it is not convenient because it requires combining the application logic with the error handling logic. The following mechanisms improve the first one. The second mechanism is related with the *Promise* mentioned in section 4.1.4. It allows specifying a callback function that is called when a *Promise* is rejected. It separates the application logic from the error handling. On the other hand, it still does not allow to use *try / catch* construct. The third mechanism is related to the *async* and *await* functions mentioned in the same section. This approach allows usage of the *try / catch* construct. Therefore, it should be optimal to use it [6] [20] [19].

4.1.3 Handling of External Dependencies

The solid handling of external dependencies is the key aspect of the development. Sharing of the code heavily helps with reusing the code and with distributing changes to several projects. The Npm tool fulfills this aspect of developing for JavaScript libraries.

Each project can specify its dependencies. These dependencies are listed in the file named *package.json*. Each dependency is specified by the unique name and by the version specification. The version can by specified with wild-cards that add the flexibility in selecting the version. The installed dependencies are saved in the folder named *node_modules* in the same directory as the file named *package.json* is. Libraries can also be installed globally. These libraries are accessible to all projects. The disadvantage is that their usage partly hides that the project depends on them [8].

The important part of handling external dependencies is resolving them. It works as follows. When the script requires a module, it requires it under the module identifier. If the identifier matches with a native module name, it requires the native module. If the identifier starts with *'/'*, *'../'* or *'./'*, it loads the file on the path specified by the identifier. Otherwise, Node.js tries to load it with the following mechanism. Node.js starts at the parent directory of the current file, adds */node_modules* and attempts to load the module from that location. If it is not found there, it moves to the parent directory until the root of the file system is reached.

The module identifier can match with the file name or the folder name. If it matches with the file name, it will import the file. If it matches with the folder name, the process will get more complicated. If the folder contains the *package.json* file and this file specifies the entry file of the module, it will load this file. If the *package.json* file does not exist, Node.js will try to load the default files called *index.js* or *index.node* inside the directory [11].

4.1.4 Methods of Writing Asynchronous Code

The biggest disadvantage of asynchronism is that it changes how the code is written. Each function that somehow depends on the result of the previous asynchronous function has to be called in the callback function passed to the asynchronous function. The problem appears when there are more asynchronous operations with a dependency to the previous one. It can easily lead to unmanageable code, which is also difficult to read. This state of code is called Pyramid of Doom.

But the newer versions of JavaScript offer methods that allow writing more manageable and readable asynchronous code. The first mechanism is called Promise. It is an object that represents a proxy for a value that is not known when the *Promise* object is created. It requires one parameter in the constructor. This parameter has to be a function that implements the operation. It takes two parameters. The first one is a callback function that is called after the operation is successfully completed and it probably receives the result of the operation in the parameters. The second one is a callback function that is called when the operation failed.

Figure 4.2 shows how the Promise mechanism works. A *Promise* object can be pending, fulfilled or rejected. When it is created, it starts as pending. If the operation succeeds, then it will become fulfilled. If the operation fails, it will become rejected. The Promise mechanism is useful because it allows composition. It is provided by methods *then* and *catch*. These methods accept callback functions as parameters, which are called after the *Promise* object is fulfilled or rejected. As these methods return a *Promise* object, they can be chained.

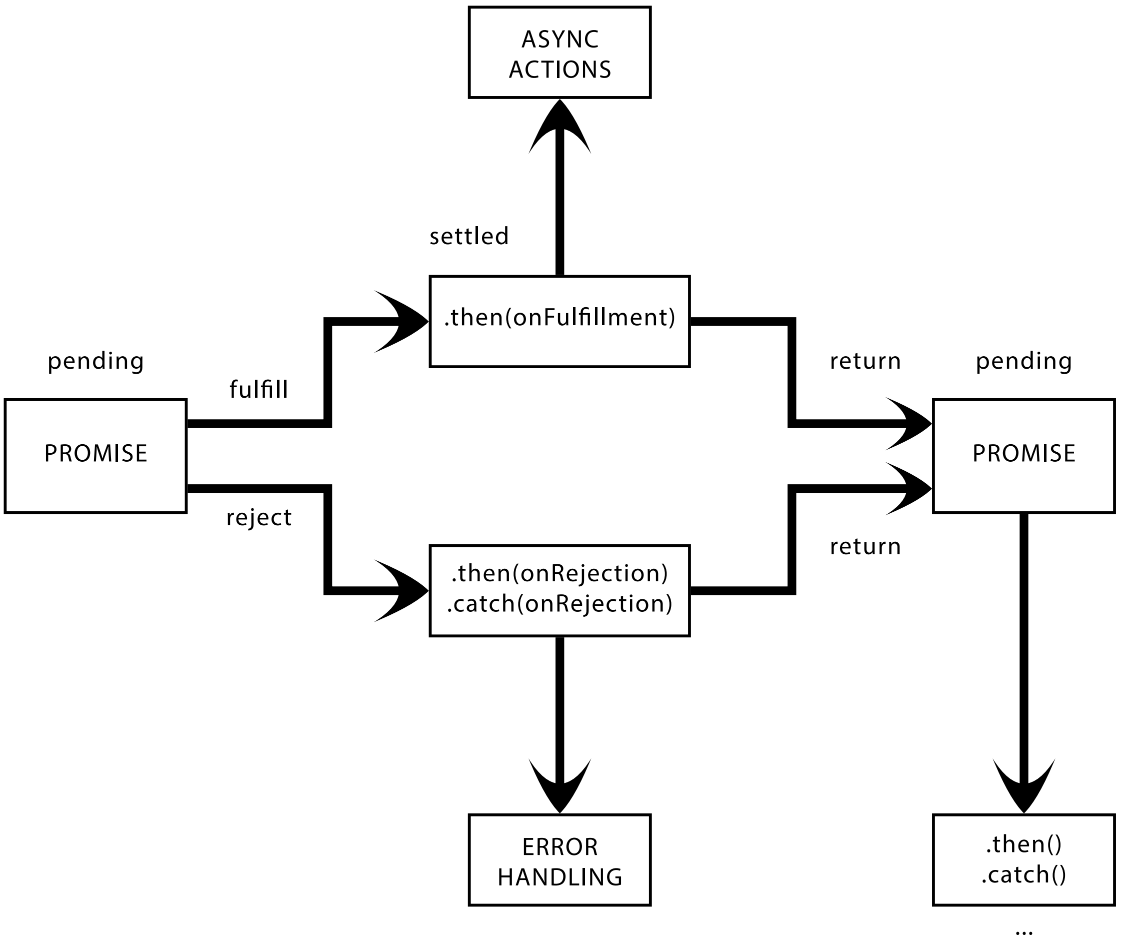


Figure 4.2: How the Promise mechanism works. At the beginning, the *Promise* object is in the pending state. The *Promise* object is fulfilled when the underlying operation is successful or rejected when it is unsuccessful [19].

Method name	**Description**
resolve(value)	Returns fulfilled *Promise* object with the value
reject(reason)	Returns rejected *Promise* object with the reason
race(iterable)	Returns *Promise* object from iterable
all(iterable)	Returns computed *Promise* object

Table 4.1: Methods specified by the Promise API

The API also offers other methods to support writing asynchronous code. Table 4.1 shows the list of them. The first and the second method return a fulfilled or rejected *Promise* object with the value passed as a parameter. The third method is called *race* and it returns one *Promise* object from a given collection. The returned promise can be fulfilled or rejected and it is the *Promise* object that was evaluated faster. The fourth method is called *all*. If all the *Promise* objects in a given collection are resolved, then it will return fulfilled *Promise* object with an array of resolved values from the fulfilled *Promise* objects. If one of the *Promise* objects in a given collection is rejected, then it will return rejected *Promise* object with the reason from the rejected *Promise* object [19].

The second mechanism that improves the coding experience is called Generators [12]. Generators allow representing lazy sequences. It adds the ability to pause the function execution and get the value of a specified command inside the function. The keyword *yield* fulfills this purpose. The generators do not give a way of representing the result of an asynchronous operation. It can be solved by *Promise* mechanism. The implementation is based on the function that wraps the generator function. This function is repeatedly getting the yielded values, trying to resolve them and returns the result [5]. It gets even easier with the ES7 standard and with the new keywords *async* and *await*. The keyword *async* marks the function as asynchronous and allows to use the keyword *await* inside its body. The keyword *await* has to be used before the asynchronous command. These built-in keywords allow writing asynchronous code similarly to synchronous with advantages such as non-blocking operations [6].

4.2 Representational State Transfer

Representational state transfer (REST) is an architectural style that is heavily used by web applications. Many constraints have to be applied to an application to follow this architectural style. The first constraint is to follow the client and server model. It means that the server is not concerned with the user interface or the user state and that allows them to be simpler and scalable. It also means that the client is not concerned with data storage. It allows to develop the server and the client separately, but only if the interface between them remains. This is a second constraint as that it has to have a uniform interface. It means that each resource has to be identified by the request. The URL and request headers are for example convenient to use in the web environment. It also means that the representation returned by the server is not in any way bounded with the actual format. The format of returned representation also has to be negotiable. It also means that the returned representation with meta-data also has to contain enough information to manipulate with the resource. Finally it means that the messages sent between the client and the server have to be self-descriptive. It can be for example in the web environment attained with the URL parameters and request headers. The third constraint is that it has to be stateless.

The communication between the client and the server must not be affected by any context. Each request from the client has to have all the information necessary to service the request. The client holds any other states such as session state. The next constraint is the ability to cache. It means that the responses have to explicitly or implicitly define themselves as cacheable or not. It improves performance and scalability of the application. The last constraint is that the REST application is a layered system. The client cannot recognize if it communicates with the end server or with an intermediary server. It also improves scalability [30].

4.3 Overview of Used Libraries and Tools

This section describes the tools and libraries that were used to develop the application. Each section describes the library and eventually it also represents some other complementary libraries that are primarily used together.

4.3.1 Express – the Web Application Framework

Express is a minimalist web framework for Node.js applications. The main advantage is its minimalism and its ability to be easily composed with other libraries. It allows mounting a handler function to an arbitrary request. It can be for example mounted to a specific URL or HTTP method. This handler receives two objects. The first one is representing a request with received data and the second one is a response object. The essential part of the framework is the middleware concept. Middleware is a function that is invoked by the Express routing layer before the final request handler. This function can also be mounted as the final handler. The only difference is that it exploits the third parameter passed to the handlers. This parameter is a function and its call means the end of the current handler execution. The middleware also can change the request and response object. The handlers can be mounted hierarchically which adds more flexibility to the request processing. The order of the middleware executions respects the order of addition. The first added middleware is executed as the first. Middlewares, for example, handle parsing the request body, cookies, session, cross-site request forgery protection, logging and so on. The biggest advantage of this approach is that it is very easy to insert an application logic into the process of handling the request [17].

4.3.2 Mocha – the Flexible JavaScript Test Framework

Mocha is a test framework that allows testing of synchronous and asynchronous code. The structure of the tests allows flexible nested test composition with all the hooks. Hook is a function that is called before or after the test. It also offers dynamically generated tests or so-called parameterized tests. It means that the test just specifies a collection of the input data and the expected data, then the test iterates over that collection and for each item calls a test. Even though this is achievable by the majority of test frameworks, Mocha achieves it with no special syntax. It improves the readability of the tests. Moreover, it offers standard functionality of a test framework such as reporting the results, tests time-outs and it works with any assertion library that throws an error [18].

4.3.3 Gulp – the Task Manager

Gulp is the task manager that helps with developing applications. It allows specifying tasks that for example process files or run different programs. The commands can be easily chainable which results in better re-usability. There can also be specified dependency between the commands. The file called *gulpfile.js* holds the specification of tasks. They are written in JavaScript and therefore it is easy to use different libraries and it is easy to use concurrency [16].

4.3.4 Webpack – the Flexible Module Bundler

Webpack is a tool that handles all the bundling of the source code for the client. When the client application uses only a few libraries, it is easy to deal with these dependencies by hand. The problem starts when the number of the libraries raises and the dependencies get complicated. Common problems are for example conflicts in the global variable namespace and wrong order of loading. The application structure starts to be fragile and error-prone because the coupling between the libraries is not obvious and therefore one minor change can result in many errors. With the rising number of used libraries, the size of the code that must be transferred to the client grows. Therefore, Webpack allows creating bundles. A bundle is many libraries grouped together. These bundles can be specific for a part of the web application and therefore it can decrease the size of transferred data. The code that is common for all bundles can be automatically extracted into a separate bundle and therefore no unnecessary source code is transferred more than once.

Webpack does not only support JavaScript code bundling, but it allows to bundle stylesheets, images, web-fonts and also allows to compile other languages that are compilable into JavaScript. This functionality is provided by the *loaders*. The use of them can be explicitly stated in the require command or it can be specified in the configuration. The second option is recommended because it allows for applying the loader to numerous require commands. The configuration consists of a regular expression and the name of the loader. If the regular expression is matched with the full file name of the currently processed file, the loader will be applied.

Webpack also allows to extend its functionality and it is achieved by plug-ins. These plug-ins interact with the process of bundling and help to get the optimal result. They can, for example, solve the issues with not completely compatible libraries, optimize the output and support the developing process.

Due to the importance of debugging, Webpack and some additional libraries offer support for the debugging process. Webpack itself supports so-called watch mode. When it starts in this mode, it watches the changes in the source files and if a change occurs, it builds a new version. The second and more advanced tool is called Webpack Dev Server. It is a small Node.js Express server (section 4.3.1) and it serves bundles. Furthermore, it has a little runtime that is connected to the client through the sockets. The server emits information about the compilation state to the client and the client reacts to them. Webpack Dev Server also allows refreshing the web page automatically after the bundling process [10].

4.3.5 Nodemon – Monitor for any Changes

Nodemon library is used for the development only and Node.js based applications. It simply watches the files in the directory where the tool starts. If any files change, it will automatically restart the Node.js application. The advantage of this library is that it does not require any modification of the application itself. It only wraps the application and watches for changes. It can look like this library is primitive, but its usage prevents errors and simplifies the developing process [7].

4.3.6 Webdriver.IO – Selenium tests for Node.js

Webdriver.IO library allows controlling the web application. The orders that can be executed cover a wide range of commands from navigation to URL, reading the web page, filling data to the user actions such as swipe, click and others. The library itself works as a client to the selenium server. The library also contains the test runner. This runner helps with integrating the library into existing tests. It supports all the popular test frameworks including Mocha. A config file configures the runner and specifies everything important to start tests. For example, it specifies paths of the test files, the address of the selenium server, parallelism of the testing process, used browsers, configuration of the plugins and much more [24].

Chapter 5

Application Design

This chapter describes how the application is designed and which ideas are behind this process. The first section 5.1 is dedicated to the application design as a whole and it describes which parts of the application are on the client and which parts of the application are on the server. The second section 5.2 is dedicated directly to the client side part of the application and describes used procedures in detail. The third section 5.3 similarly describes the server-side part of the application. The fourth section 5.4 is dedicated to the building and developing process because it is important to keep the integrity of the source code to have continual development process. The last section 5.5 recaps the application design and mentions known drawbacks. The application design is based on the older version of the Este project [33].

5.1 General View of the Application Design

There are several ideas behind the application design. The first one is that the client has to be as minimal as possible. It is advantageous for many reasons. The main ones are that it reduces the requirements for the processing power on the client and exposes only the necessary parts of the application on the client. The second one is to have a user-friendly web interface. It means that there has to be an option to address the presentation by its name and also slides by some number. It is convenient because it allows to share the presentations through the internet and it extends the usability of the application to more use cases. The last one is that the web client has to be a single page application [32]. The reason is to have the best possible user experience from the application.

The application design aims to fulfill the process behind the use case of running the PowerPoint presentation in a web browser. This process is shown in Figure 5.1. It consists of uploading the presentation file to the server, converting the presentation file into the internal representation, sending this representation to the client side application, visualising this representation and reacting to the user actions such as moving between slides and viewing the presentation in the full-screen mode.

5.1.1 Splitting the Application between the Client and the Server

The functionality is split between the client and the server as follows. The server side of the application is responsible for extracting the information from the uploaded file. It includes numerous reading and parsing files and transforming their content. The output

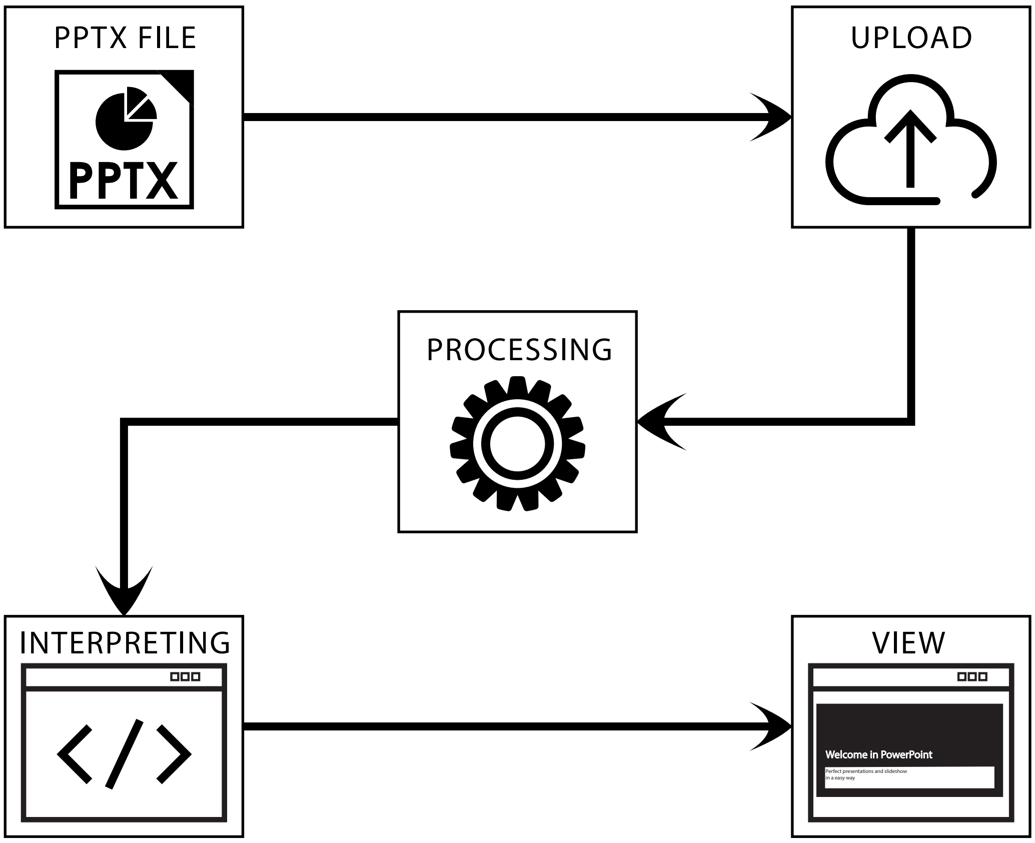


Figure 5.1: The process behind the usage of the application. The input is the presentation file in the OOXML format. This file is uploaded to the server, then the server transforms the presentation into the internal representation, this representation is sent to the client and the client visualises that representation.

of this process is the representation that describes visual appearance of the presentation. This process is described in detail in section 5.3.

The client side of the application is mainly responsible for displaying this representation. Furthermore, it is also responsible for basic client-side functionality such as reacting to the user actions, handling the application state and firing the additional requests to the server. This part of the application is described in section 5.2.

5.1.2 Web Application Interface

The web interface (section 4.2) can be divided into two parts. The first part is RESTful, handles API calls and returns raw data. This part is completely separated from the second one, which is convenient for testing purposes and for implementing other clients. The interface itself is simple. It allows addressing a single slide, the whole presentation directly and it allows to address a media content of the presentation such as images, video and audio. The second part handles the web application calls and it also allows to address directly single slide or the whole presentation. This part is described in section 5.2.2.

5.1.3 User Interface of the Client

The possibilities of creating a user interface for the web clients are large. The first and primarily used option is using HTML markup language with CSS. This is the standard way of creating a user interface and it is also the preferable way. The reason is that it is stable, it has support in the majority of web browsers and with the coming of the HTML5 standard it also supports advanced tasks. Its disadvantage is that the level of customization is still not as high as it is with the following approaches. A user interface can also be completely built with the features from the HTML5 standard called Canvas and SVG. This method gives a higher level of customization, but the downside is that it is not as easy to use as the first one. The third option is to use a third party web browser plug-in. This may look like an ideal solution because it takes the advantages from previous approaches. The problem is that it completely breaks the key benefits of the web environment such as the application on demand, high usability and easy accessibility by the URL. These considerations lead to the combination of the first and the second approach. Standard HTML markup with CSS can completely handle the simpler visualisations and the advanced visualisations can be handled by the SVG and Canvas technologies.

5.2 Client-side Part of the Application

This section is dedicated to the design of the client side part of the application. The first subsection 5.2.1 is dedicated to handling the application state. The second subsection 5.2.2 describes the navigation process. The last subsection 5.2.3 describes designed components and how they fit together.

5.2.1 Handling the Application State and Changes in It

This subsection presupposes the knowledge of the React library (section 3.3.2), Flux architecture (section 3.2.2) and the Redux library (section 3.3.1). This subsection describes how the state container is designed and which redux actions are designed.

The list of the designed fields is shown in Table 5.1. The primary data field of the state container is named *slideshow* and contains data representing the presentation. Its structure

Field name	Type	Description
slideshow	object	the internal representation itself
fullScreen	boolean	the state of the full-screen mode
slideLoading	boolean	the state when the application is requesting a new slide
movingForward	boolean	the state when the transition forwards is executing
movingBackward	boolean	the state when the transition backwards is executing

Table 5.1: Designed data fields in the state container. Values in these fields sets how the user interface looks.

is described in section 5.3.1. The other fields in the state container represent temporally application states, which may occur. The field named *fullScreen* is representing the state of the full-screen mode. It may be only turned off or on. This field is necessary because the application has to be rendered again when the full-screen mode is changed. The next field is named *slideLoading* and represents the state when the application requested the server for the data and the application is waiting for them. The last fields are named *movingForward* and *movingBackward* and represent the state, when the application is performing the transition forwards or backwards between the slides.

Action name	Type	Description
Start slideshow	Synchronous	Fired when the presentation starts
Change full-screen	Synchronous	Change the full-screen mode
Fetch slide	Asynchronous	Request for the slide data
Fetch slideshow	Asynchronous	Request for the whole slideshow data
Start transition forward	Complex	The action is fired when the transition forward starts
Start transition backward	Complex	The is action fired when the transition backward starts
End transition	Synchronous	The is action fired when the transition is finished

Table 5.2: List of the designed actions

The overview of the designed actions is shown in Table 5.2. The first action is *Start slideshow* and it is fired after successfully uploading the presentation when the user wants to start with the presentation. The second action is *Change full-screen* action and it sets the *fullScreen* field in the state container to its inverted value. The next actions are *Fetch slide* action and the *Fetch slideshow* action. Both of them are asynchronous and therefore both consist of three synchronous actions. The starting action sets the flag *slideLoading* in the state container, the success action sets the requested data and cancels the flag and the error action just cancels the flag. The *Fetch slide* action fetches the internal representation of one slide from the presentation and the *Fetch slideshow* action fetches the internal representation of the whole presentation. When the *Fetch slide* action is dispatched and the state container already contains the presentation data, the received data are merged into the current state. This allows to continually update the internal representation in the client. The next important actions are bounded with the slide transitions and are named *Start transition forward* and *Start transition backward*. They check the existence of the next or the previous slide in the state container, fetch the slide data when the state container does not contain it and start the transition between the slides. The last action is called

End transition and it is fired when the transition is finished. The reason for this level of transition process granularity is to have better control of the transition process. The *End transition* action is dispatched either immediately after the transition is finished or after the animation of the transition is finished.

The state container has to have the initial application state. Because of the server-rendering functionality, the initial application state is computed during the first request on the server. Even though it is the relevant solution to leave the computing of the initial state also on the client, it is not desired, because it would mean doubling the number of API requests. The initial state of the application is computed on the server and then it is serialised into the global variable named *___INITIAL_STATE___*. The way of computing the initial state in the server side of the application is described in section 5.3.2.

5.2.2 Navigating the User through the Application

This subsection presupposes the knowledge of the React library (section 3.3.2) and React-router library (section 3.3.3). The application distinguishes several pages and it is possible to address them by the URL. List of the pages is shown in Table 5.3.

Page name	**The URL Postfix**	**Description**
Entry page	/	The entry page of the application
Loader page	/slideshow/<name>	The page that in background loads the presentation data
Slideshow page	/slideshow/<name>/slide/<number>	The page that displays the slides of the presentation

Table 5.3: List of the pages in the application

There are three pages that are important to view the presentation. The first page is called *Entry page* and it contains only one form, which allows uploading the presentation file to the server. The second page is called *Loader page*. When this page is visited, the application starts loading the presentation. The presentation is specified by the parameter in the URL called *name*. When the loading is finished, it offers the option to start the presentation. The last page is called *Slideshow page*. It displays one slide of the presentation, supports navigation in the presentation and it allows to switch the presentation to the full-screen mode. The presentation is specified by the parameter called *name* and the slide by the parameter called *number*. Both parameters are specified in the URL.

The React-router library is used in the entry point of the application. It follows that library handles every navigation action and therefore the components itself have to handle the data fetching. It depends on the current state of the state container and the current action. The server requesting can be performed in the action creators with the use of middleware called *Redux-thunk*. This also brings all the asynchronism into the navigation process. This form of requesting the server is used in the actions, which are performed after the initial request. The examples of that kind of the actions are *Start transition forward* and *Start transition backward*. The server requesting can also be performed in the components themselves if the application takes the advantage of the life-cycle of the ReactJS components. This approach is suitable for the components and the actions that are closely related. The example of that kind of the action is the *Fetch slideshow* action and the *Loader page* component. In this use case, the component pre-loads the presentation before the user can continue to the presentation itself. The component's life-cycle offers two methods to use – *componentWillMount* and *componentDidMount*. The action can be performed in both methods, but the usage of the *componentWillMount* will result in the unnecessary API request during the server rendering. The reason and the description of the server rendering functionality is described in section 5.3.2.

5.2.3 Components Displaying the Presentation

This subsection presupposes the knowledge of React library (section 3.3.2), Redux library (section 3.3.1) and React-router library (section 3.3.3).

An overview of the components is shown in Figure 5.2. The root component is named *Slideshow page*. It is a simple component, which holds the other components together and it receives the current presentation name parameter and the current slide number parameter. The last responsibility of this component is that it specifies the data, which are used for server rendering. Every action that should be fired during the server rendering has to be assign to the component's field called *fetchActions*. The details about this part of the application design are described in section 5.3.2.

The following component is the *Slideshow controller*. Due to its simplicity, it does not follow the presentational and container components principle and it reacts to the application state changes and it also defines visual appearance. It contains one button, which switches the full-screen mode of the presentation.

The next component is called *Interactive slideshow view*. This component is the container component for the *Slideshow view* component with the full-screen and the navigation functionality. The *Slideshow view* component is the presentational component and it is responsible only for displaying the presentation. The implementation of the full-screen and the navigation functionality follows the higher-order components principle and these components are also the presentational components. This separates the application logic from the displaying logic and on top of that, it also helps re-usability.

The *Slide view* component overview is shown in Figure 5.3. It consists of many *Shape view* components and many *Media view* components. The *Shape view* component represents a positioned rectangular object. It may display shape related styles such as background colour. The *Shape view* component may contain the *Text view* component. This component displays text with all the margins and indentations between text runs and paragraphs. The *Media view* component is very similar to the *Shape view* component. It is also a positioned rectangular object. The difference is that it displays media files such as images, video and audio. The video and audio files are not delivered with the internal representation, but they are delivered on demand from the player.

5.3 Server-side Part of the Application

The primary responsibility of the server side of the application is the conversion between the presentation file and the internal representation. The first section 5.3.1 is dedicated to the internal representation and mainly to its structure. The second section 5.3.2 is dedicated to linking and serving the application to the client.

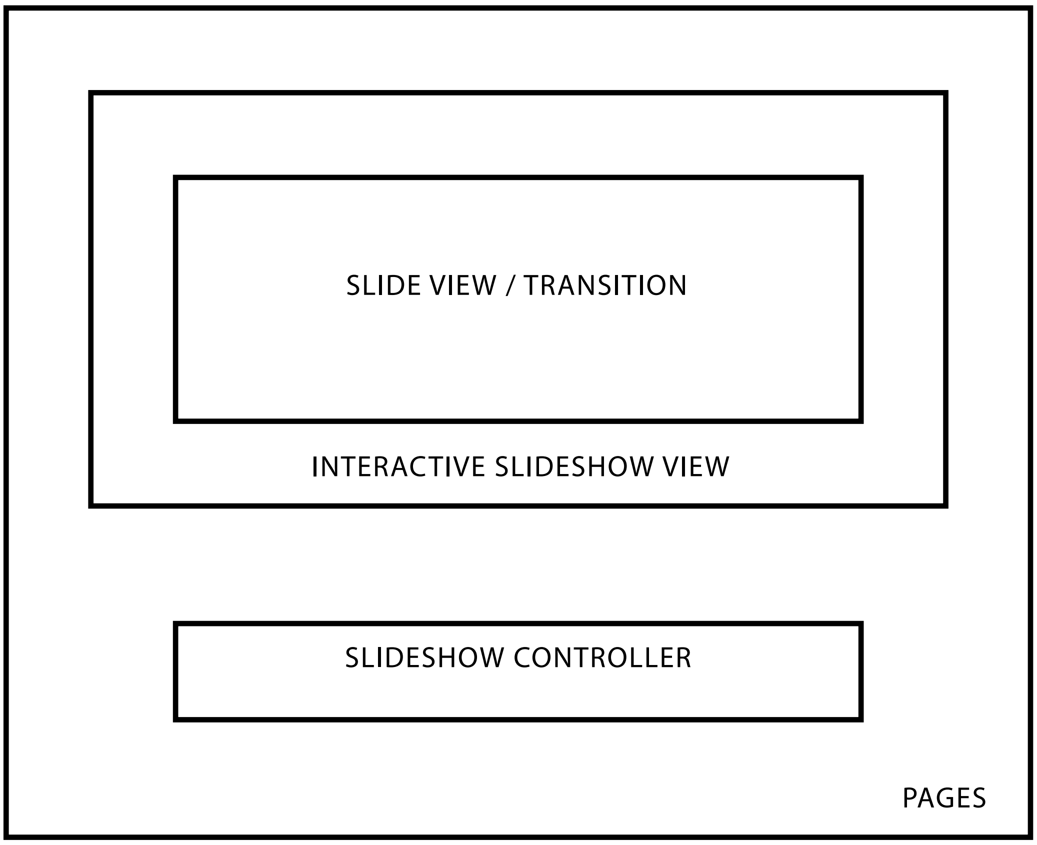


Figure 5.2: Overview of the components composition. The root component is called *Page*. It consists of *Slideshow controller* component and *Interactive slideshow view* component. *Interactive slideshow view* component consists of the *Slide view* component or the *Transition* component. Its content depends on the state of the application. If the application is performing the transition between slides, then the *Transition* component is inside the *Interactive slideshow view*. In other cases, the *Slide view* component is inside the *Interactive slideshow view* component.

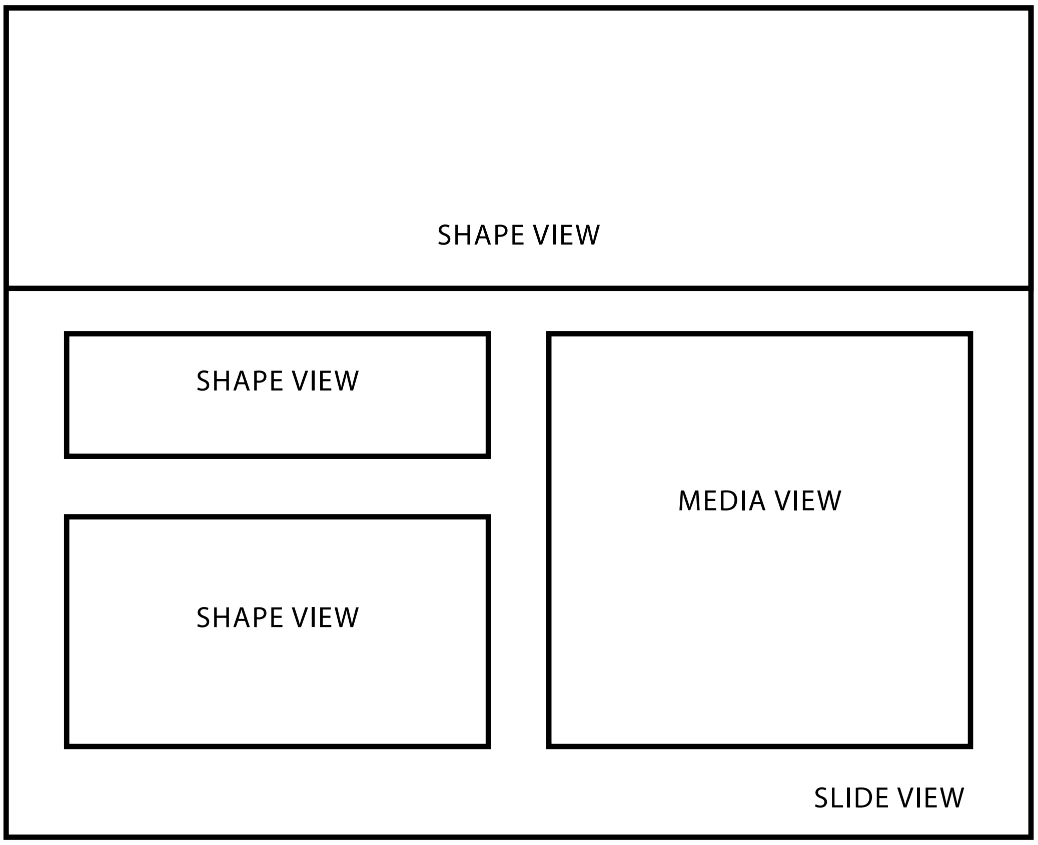


Figure 5.3: The components composition inside the *Slide view* component. In this case the slide data consist of three shapes and one media element. Therefore, the *Slide view* component also consists of three *Shape view* components and one *Media view* component.

5.3.1 Internal Representation of the Presentation

This subsection presupposes the knowledge of the OOXML format (section 2.3 and section 2.4) and the knowledge of the visualizing in the web environment (section 3.1). The structure of this representation is shown in Table 5.4.

Field name	**Type**	**Description**
primaryId	string	Name of the presentation
totalSlidesCount	integer	Total number of slides
width	integer	Original width of the presentation
height	integer	Original height of the presentation
slides	array	Slides specification

Table 5.4: Structure of the internal representation of the presentation

The fields named *width* and *height* hold the original measures of the presentation. The most important field is *slides*. This field contains a collection of the objects where each object is representing one slide. This field can contain fewer objects than the presentation has. The order of these objects is important because it sets the order of the slides.

The structure of the object representing one slide is shown in Table 5.5. The number of objects in the *shapes* field and *media* field corresponds with the number of shapes on the slide and with the number of multimedia objects such as images, video and audio in the slide. Objects from both fields have similar structure and the only reason of the separation is that it follows the OOXML format. The order of these objects in the collection is important because it sets the z-order. The objects on the lower index have the higher z-order than the objects on the higher index.

Field name	**Type**	**Description**
backgroundColor	string	Hexadecimal interpretation of the background colour.
minorFont	object	Specifies default font for the text shapes excluding titles.
shapes	object	A collection of objects. Each object represents one visual object in the slide.
media	object	A collection of objects. Each object represents a picture, video or audio object.
transition	object	Specification of how the transition should be visualized.

Table 5.5: Structure of one object from the *slides* field

The structure of the object from the *shapes* field is shown in Table 5.6. The value in the field named *id* is a unique identification number and it can be used for referencing the shape within an animation. The uniqueness is guaranteed in the context of one slide. Next field is named *type* and its value corresponds with the *type* field from the OOXML format. The field called *properties* contains visual attributes of the whole shape. It includes the definition of position, measures, filling and shape geometry. The last field is named *txBody* and it contains the entire definition of text related content. The structure of the *txBody* field is shown in Table 5.7.

The field called *properties* contains visual attributes such as insets and vertical text alignment. The field named *paragraphs* contains the collection of objects representing one paragraph. The structure of the object from the *paragraphs* field is shown in Table 5.8.

Field name	**Type**	**Description**
id	integer	Identification number of the shape
type	string	Type of the shape
properties	object	Collection of the properties specific to the shape
txBody	object	Object that specifies text content of the shape

Table 5.6: Structure of the object from the *shapes* field

Field name	**Type**	**Description**
properties	object	Visual attributes
paragraphs	object	A collection of objects. Each object represents one paragraph.

Table 5.7: The structure of the object from the *txBody* field

Field name	**Type**	**Description**
properties	object	Visual attributes
runs	array	Collection of objects representing one text run

Table 5.8: The structure of one object from the *paragraphs* field

The field named *properties* contains the visual attributes such as text indentation, paddings and horizontal alignment. The second field called *runs* contains the collection of objects representing one text run. The structure of the object from the *runs* field is shown in Table 5.9.

Field name	**Type**	**Description**
style	object	Visual attributes
text	string	Text content

Table 5.9: The structure of one object from the *runs* field

The first field is named *style* and it contains the visual attributes such as font family, font size and colour of the text. The second field is named *text* and it contains the text itself.

The second field that holds the content in the *slide* object is named *media*. It is the collection of the objects representing multi-media content. Its structure is shown in Table 5.10 and it is similar to the structure of the objects representing shapes.

Field name	**Type**	**Description**
id	integer	Identification number of the shape
properties	object	Collection of the properties specific to the media object
picture	object	The picture specification
video	object	The video specification
audio	object	The audio specification

Table 5.10: Structure of one object from the *media* field

The first field is named *id* and it is a unique identification number. Its uniqueness is guaranteed in the context of one slide. The second field is named *properties* and it contains the visual attributes of the media object. It includes the definition of the position, measures, filling and object geometry. The next field is named *picture* and it contains a specification of the picture. The picture is included in the object for all three types of the media. The *picture*, *video* and *audio* have very similar structure which shown in Table 5.11.

Field name	**Type**	**Description**
url	string	URL where the media content is reachable
name	string	Name of the object
source (image only)	string	The image data itself

Table 5.11: Structure of the object in the *picture*, *video* and *audio* fields

The first field is named *url* and it contains the URL, where the content itself is reachable. The second important field is called *source* and it contains the data of the picture. It allows receiving pictures with an internal representation. However, this approach has some drawbacks. The biggest one is that it forces to save the binary data in text format. It can be achieved only by converting the data with methods such as Base64 but that results in increasing the size.

5.3.2 Link between the Server and the Client

This subsection presupposes the knowledge of the Node.js server (section 4.1), the Express framework (section 4.3.1), the React library (section 3.3.2) and the React-router (section 3.3.3).

The initial server request has to deliver all parts of the application. It primarily means handling the content of the HEAD tag, serving the client application source code and the initial HTML markup. This part of the application is also written as a React component, but this component is rendered to static markup. A part of this markup is used as the mount point of the client application loads up. The markup from the server should be identical to the markup after the client application is initialized.

The client application is wired as the express middleware. It matches all GET requests and it is wired after the more specific middlewares. The generation of the markup is handled by a React component, which is rendered by the server. The processing of the request is passed to the Router component from the React-router library, which resolves which root component should be rendered. This root component can specify additional actions by property called *fetchActions*. These actions are fired during the server-rendering and the application has to wait explicitly for them, because these actions are mostly asynchronous. After all these actions are finished, the application state is serialized into the global variable called *___INITIAL_STATE___*. The client application loads this variable as its initial state during its bootstrapping. Therefore, the initial application state in the client application corresponds to the state after all these actions are completed. The content of the HEAD tag is handled by a library called React-helmet.

This whole process makes the application usable even without JavaScript enabled. The HTML markup in the response contains the same HTML tags as the markup after the client application is bootstrapped. It also makes faster the process of the bootstrapping, because the client application does not have to create all HTML tags.

5.4 Building and Developing the Application

The simplicity, flexibility, usability of the building process and the developing process are essential for the continual development. Splitting the code into modules and handling external dependencies is an important fragment of advanced web development. It is also very convenient to use some tools that improve the developing experience and limit the errors. The first subsection 5.4.1 is dedicated to the building process and it describes its configuration. The second subsection 5.4.2 is dedicated to the developing tools and it describes their usage. The last subsection 5.4.3 is focused on the testing process and how to test the visual appearance of the web application.

5.4.1 Build Tools and Their Usage

The main problem with a structured application is how to resolve dependencies between the files and how to deliver these files to the client. This problem is solved by a library called Webpack (section 4.3.4). The process of the building and the way of the configuration is shown in Figure 5.4.

The configuration itself is a JSON object. This object specifies only one entry point and this entry point is named *main.js*. This file contains references to other files, such as libraries, components and application logic. These files can also contain other references. They could be in numerous file formats and therefore, there has to be a way how to specify methods of processing. This need fulfills Loaders. The current building process uses babel-loader, file-loader, styl-loader and css-loader. The result of the building process is two files. The first one contains the application code and the second one contains the styles specification. The building process as a side effect also emits referenced files such as images and fonts.

5.4.2 Developing Tools and Their Usage

The developing process contains many actions that are repeatable. It is useful to describe these actions in the code. The library, which was chosen for this purpose, is called Gulp (section 4.3.3). In the language of Gulp, these actions are called tasks and therefore Gulp is a Task Manager. The list of the application tasks is shown in Table 5.12.

Task name	**Description**
clean	Remove all files created by build process
build	Build the application
server	Run the server application
server-hot	Run the hot reload server (development mode only)
server-nodemon	Run the server application with Nodemon (development mode only)
default	Alias for the task named server
server-node	Standardly run the server
test	Run all the tests
test:converter	Run the tests of the converter part
test:e2e	Run end to end tests

Table 5.12: List of the application tasks.

The most important tasks from this list are the default task and all the test tasks. The

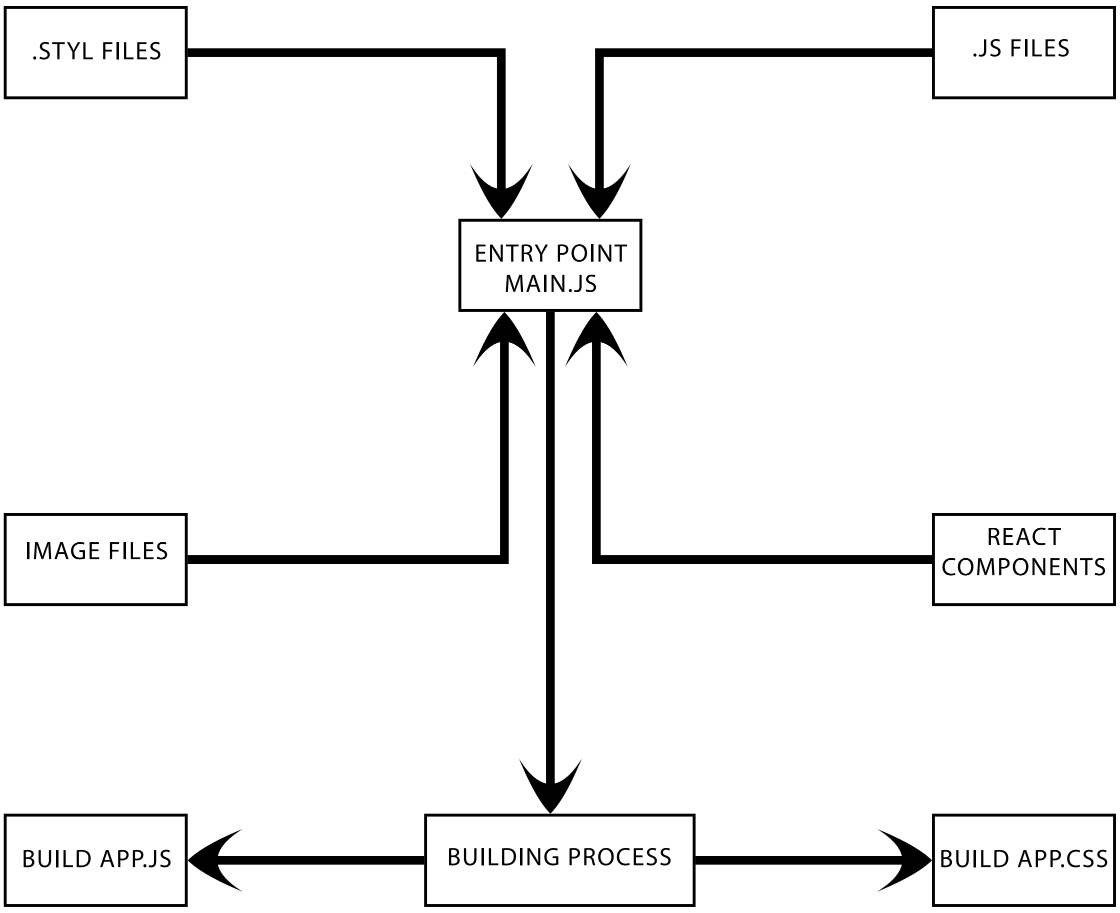


Figure 5.4: The process of building the application. The application consists of one or more entry points. Each entry point can specify numerous dependencies such as other JavaScript files, image files and stylesheets. The output of the process is one JavaScript file, one stylesheet file and files which were used in the JavaScript files or stylesheet files. The JavaScript file is modularized and therefore the global namespace only contains the variables which have been explicitly specified.

default task consists of many subtasks and the result is the running application. It firstly cleans the previous version, builds the current version and then runs the server. It runs the server either in development or production mode. The development mode adds some useful functionality such as automatic application restart due to source code change. These tools are described in sections 4.3.5 and 4.3.4. Optional argument *-p* can force the production mode. The test tasks run the tests and report the results. The reason to have numerous test tasks is to group them by their focus and to have better performance. The *test:converter* task runs the tests that check the validity of the transformation from the presentation file to the internal representation. The *test:e2e* task runs the tests that compare the results of the client application with the reference images. This task allows to run tests for all presentations, for one presentation and also only for one slide from a given presentation. The process behind is described in detail in the section 5.4.3 and its implementation in section 6.2.2.

The next tool that is used is named Nodemon and it is described in section 4.3.5. It allows restarting the server automatically if a change occurs. It assures that the running code is always up to date. The usage of this tool is simple as it only requires to correctly parametrized the command to prevent unintentional restarting. For example, it is convenient to ignore the directory with tests. The task *server-nodemon* runs the application with this tool.

The last tool allows to reload the client application code automatically and therefore it also assures that the running code of the client application is always up to date. It is more than one tool, but the key ones are Webpack and Webpack Dev Server. These tools are described in section 4.3.4. It allows watching changes in the client application code and then building the new application. The task *server-hot* runs the application with this functionality.

5.4.3 Acceptance Testing of the Presentation

The testing of the application is the key aspect that helps to recognize attained results. The automated testing is even better because it allows to check the application repeatedly after some changes, evaluate the results of tests and automatically determine if the modification is an improvement or a deterioration. The essential part of the application is visualising the slide and therefore it is the first part that should be tested. That is achieved with Webdriver.IO library (section 4.3.6). The scheme of how it works is shown in Figure 5.5.

The running test is sending orders to the Selenium server, which fulfills them through the web browser. The orders allow requesting a screenshot of some part of the client application. These screenshots can be compared in order to determine, how much they are identical. Because the testing part of the application is the critical one and should be easy to extend, it is convenient to support the simplest way of writing them as possible. For that reason, the tests follow the convention over configuration principle. The specifics of this implementation are described in section 6.2.2.

5.5 Summary of the Application Design

The application design does not aim to support all features of the PowerPoint fully. It wants to fulfill the standard features and it covers a majority of the features used in basic presentations. Because of that point of view, the application design does not count with

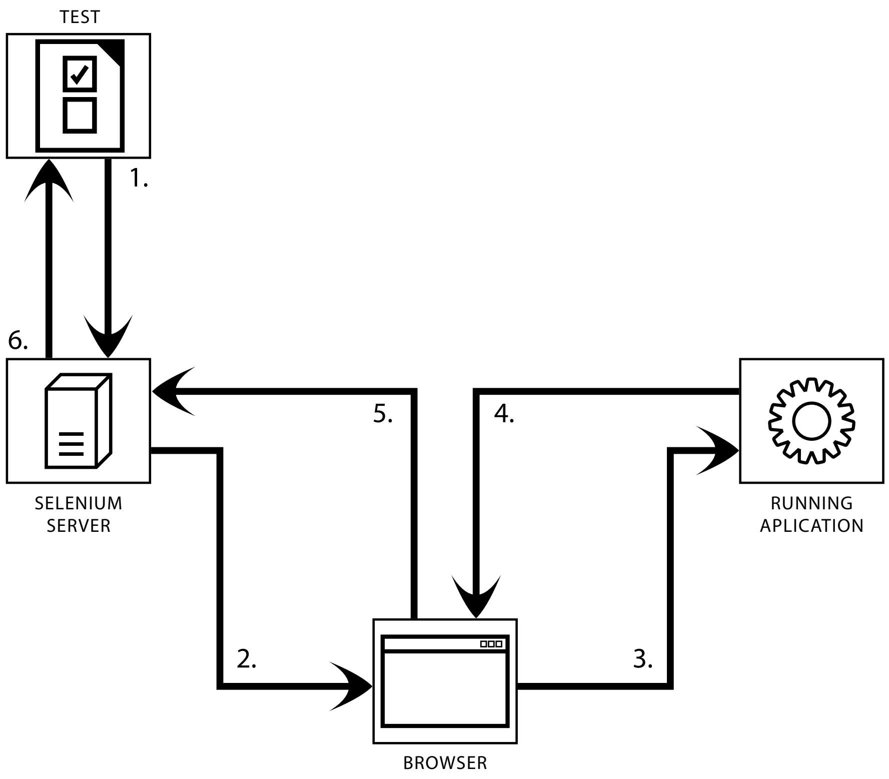


Figure 5.5: The way of running the test. The test is calling the Webdriver.IO API. This API delegates these orders to the Selenium server. The Selenium server runs the previously specified browser and runs the delegated orders such as navigating to the URL. The results of the orders are returned to the test and it evaluates them.

animating objects and with any special visual effects. Furthermore, the design of the animated slide transitions is more proof of concept than a regular solution.

Chapter 6

Implementation and Evaluation

This chapter describes some implementation details of the client (section 6.1) and the server (section 6.2). The last section 6.3 shows and comments attained results.

6.1 Implementation of the Client

The application has to be as small as possible because it is downloaded during the initial request and long download time may deter potential users. The current size is shown in Table 6.1. The Webpack library (section 4.3.4) optimises the size of the production version. The built source code is also compressed before serving it to the client and therefore the actual application size is minimal. On the other side, the size of the development version is big. The reason is that it contains a lot of libraries for development only and that it is not optimised. As it can be seen from Table 6.1, the size of the application does not stand out of the standard sizes.

	Content size	**Actual size**
Development mode	11.1 MB	11.1 MB
Production mode	1.1 MB	336 kB

Table 6.1: The sizes of the client application. The content size is the size after the build process. The actual size is the size of the application that is delivered to the client.

6.1.1 The Difference in the Spacing Specification

Visualisation of the slides contains the correct positioning of the content. The OOXML format specifies only the paddings of elements. HTML markup offers two options how to set spaces between the objects. The first option is to specify margins. The problem with margins is that they are overlapping. The problem is shown in Figure 6.1. When two elements are close enough and their margins overlap, the actual space between them is defined by the larger one. The second option is to use paddings. The difference is that they do not overlap. The actual space between two elements is the sum of their paddings. The OOXML format does not follow the same paradigm and therefore there is no space for margins usage.

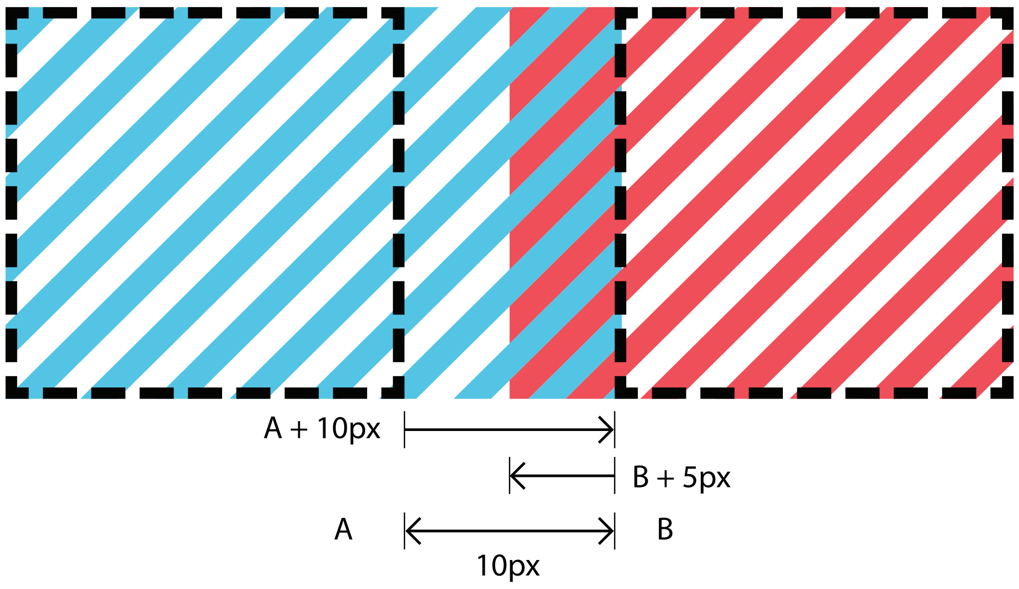


Figure 6.1: Two HTML elements (A, B) with overlapping margins. The bigger margin specifies the gap between them.

6.2 Implementation of the Server

The implementation of the server consists of many asynchronous operations. The process of transforming the presentation file into the internal representation includes many readings of files. The asynchronous code is handled by Promise objects and function generators 4.1.4. The part of the application that transforms the presentation is implemented as an independent module. Table 6.2 shows its interface.

Name	**Type**	**Description**
convertAll	function	Converts the whole presentation
convertSlide	function	Converts single slide from the presentation
MediaDownloader	object	Offers the access to the images, audio and video from the presentation

Table 6.2: Interface of the module that converts the presentation into the internal representation.

The interface consists of two functions and one service. The *convertAll* and *convert-Slide* functions converts the presentation specified by the unique identifier assigned by the application during the upload process. The *MediaDownloader* offers access to media files from the presentation. Each media file is specified by a unique identifier of the presentation and the name of the media file.

6.2.1 Transforming the Presentation to the Internal Representation

The transformation of the presentation can be visualised as two or three step process. The first step of the transformation is parsing. The output of this phase is raw information about the presentation. It does not operate with the relations between the files and therefore, its structure corresponds to the structure in the presentation file. The second and the third step are the phases of the transforming. The reason of splitting the transforming part is to have the option to have multiple different outputs for different clients. The second phase contains the transformations which are common for all clients. For example, this phase is responsible for merging the style definitions into one. The third phase is specific to the web client. Because there is no other client than the web client, this approach is more a proposal than a final solution.

6.2.2 Acceptance Testing and its Implementation

The solid and easy to extend testing environment is important for continual development. The problem is with tools when they do not offer all the desired functionality. The first problem was with the desired interface of the development task (section 5.4.1) and with the actual possibilities of the Webdriver.IO library (section 4.3.6). The desired interface of the task is that it has to allow to run all the tests for all presentations, all the tests for one presentation and only one test of one slide from the presentation. The problem is that the configuration options of this library do not allow it. The paths of the tests can only be specified in the main configuration file. The solution is generating the configuration file at the beginning of the task and then starting the testing part. The other problem was with the reference images. The usage of the library does not count with injecting the reference images. It matches the reference image by the name, but it does not support interlaced images and the error message does not help with solving this issue. The only solution is not to use interlaced images. The configuration file also does not allow to specify different folders for different tests and therefore, the preparation phase also consists of copying the reference images to the correct paths. The next problem was with maximisation of the chrome browser window. It is the problem of the Selenium server and the solution is to use additional parameters when the chrome driver is starting.

The actual implementation follows the convention over configuration principle. This works well with the test generating offered by the Mocha library (section 4.3.2). The test suite is in the separate directory. This directory contains the data of the presentation, the reference images, the history of the accuracy and the actual test. When the test starts, it executes all the preparation tasks. It means copying the presentation files to the server and injecting the reference images. If the conventions are met, these tasks work out of the box. Then the test may only contain the specification of the perceptual threshold mismatch per slide and the calling of the test method.

6.3 Evaluation of the Results

The results of the application can be divided into two groups. The first group represents the static results. These results can be automatically and easily tested and evaluated. The members of this group are for example presentations with a text, an image or an object with a colour. The second group represents the dynamic results. These results can be only partly automatically tested and the evaluation is more on the observer. The members of this group are for example presentations with an audio or a video.

The following sections are referencing to the compilations of the images. The reference images are generated by the PowerPoint itself and therefore it should be the most accurate visualisation. The other images are automatically created as a side product of the testing (section 5.4.3).

6.3.1 Text in the Presentation

The first test slide contains text that is positioned and aligned. Its comparison is shown in Figure 6.2. The text is similarly displayed and the difference is minimal. The second test slide contains the heading text and three paragraphs and its Figure 6.3 shows its comparison. The text is also similarly displayed and the difference is minimal.

The third test slide contains the heading text and three paragraphs. Its comparison is shown in Figure 6.4. These paragraphs are not displayed correctly as the space between them is different than should be. The fourth test slide contains the heading text and one paragraph with increased line height. Its comparison is shown in Figure 6.5. The text is similarly displayed and the difference is minimal.

As the comparisons show, the difference in the text visualisation between the reference image and the actual result exists. On the other hand, the text is in many cases positioned and sized correctly. The visualising of the text has some space for improvement but the achieved results are good.

6.3.2 Image in the Presentation

The first test slide contains one positioned image and its comparison is shown in Figure 6.6. The image is correctly positioned and displayed as it is seen from the comparison. The second test slide contains four positioned images and its comparison is shown in Figure 6.7. All images are correctly positioned, but only three of them are displayed correctly. The bottom right image is not identical. There is no reason from the perspective of the application why this image has some differences. It is probably an external issue caused by the rendering of the web browser.

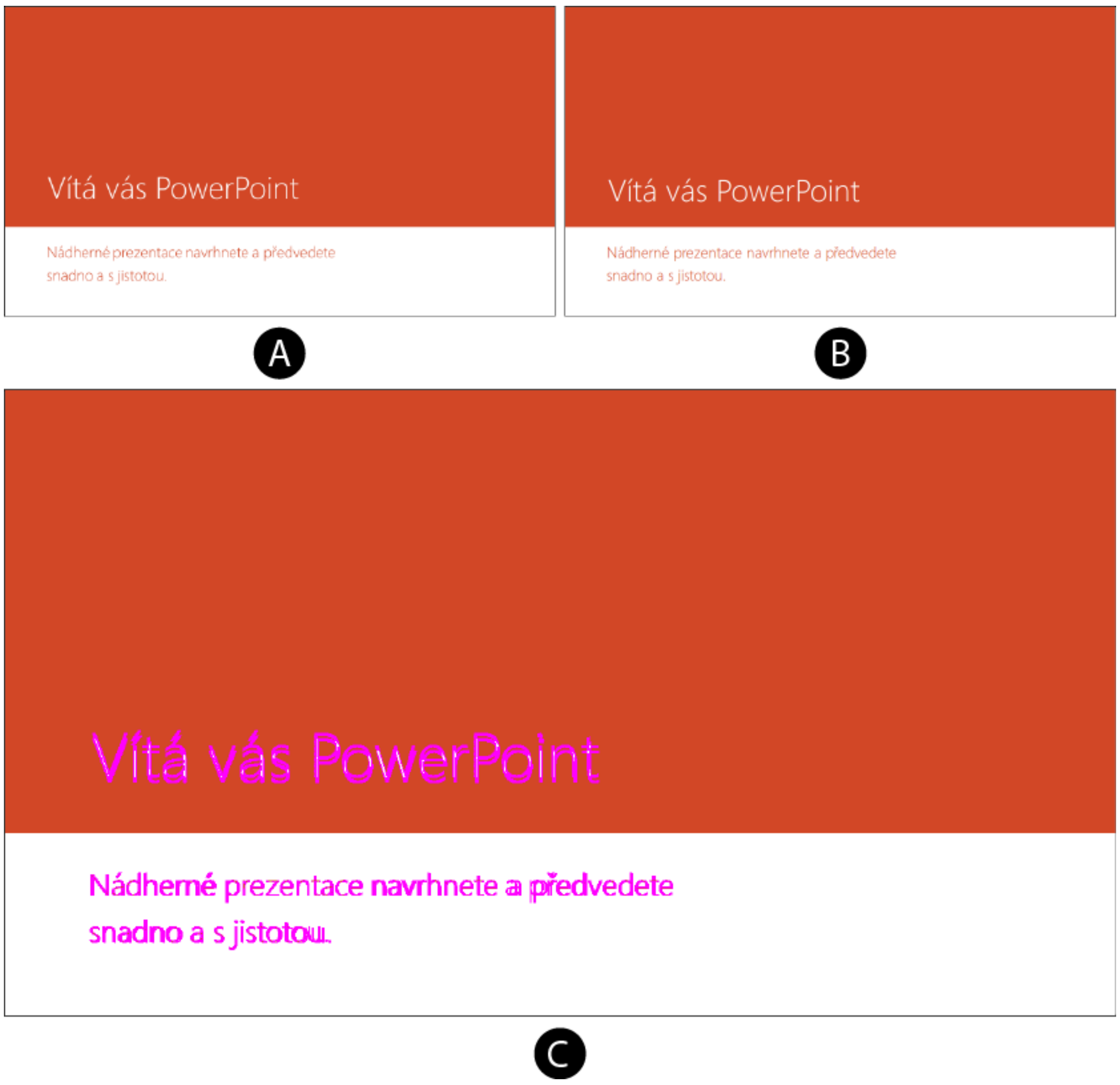
As can be seen from the previous comparison, images are positioned and displayed well. The only issues are not related to the application and are minimal. The achieved results with the image visualisation can also be evaluated well.

6.3.3 Video and Audio in the Presentation

The first test slide contains one positioned video and its comparison is shown in Figure 6.8. The images only display the preview image of the video and therefore its evaluation is very similar to the evaluation from section 6.3.2. The only difference is that the actual image contains a video player bar. The audio is similar to the video in the presentation. It displays the preview image and there is an audio player bar. The streaming of the video and audio is not easily tested but it works well and therefore the achieved results can be evaluated well.

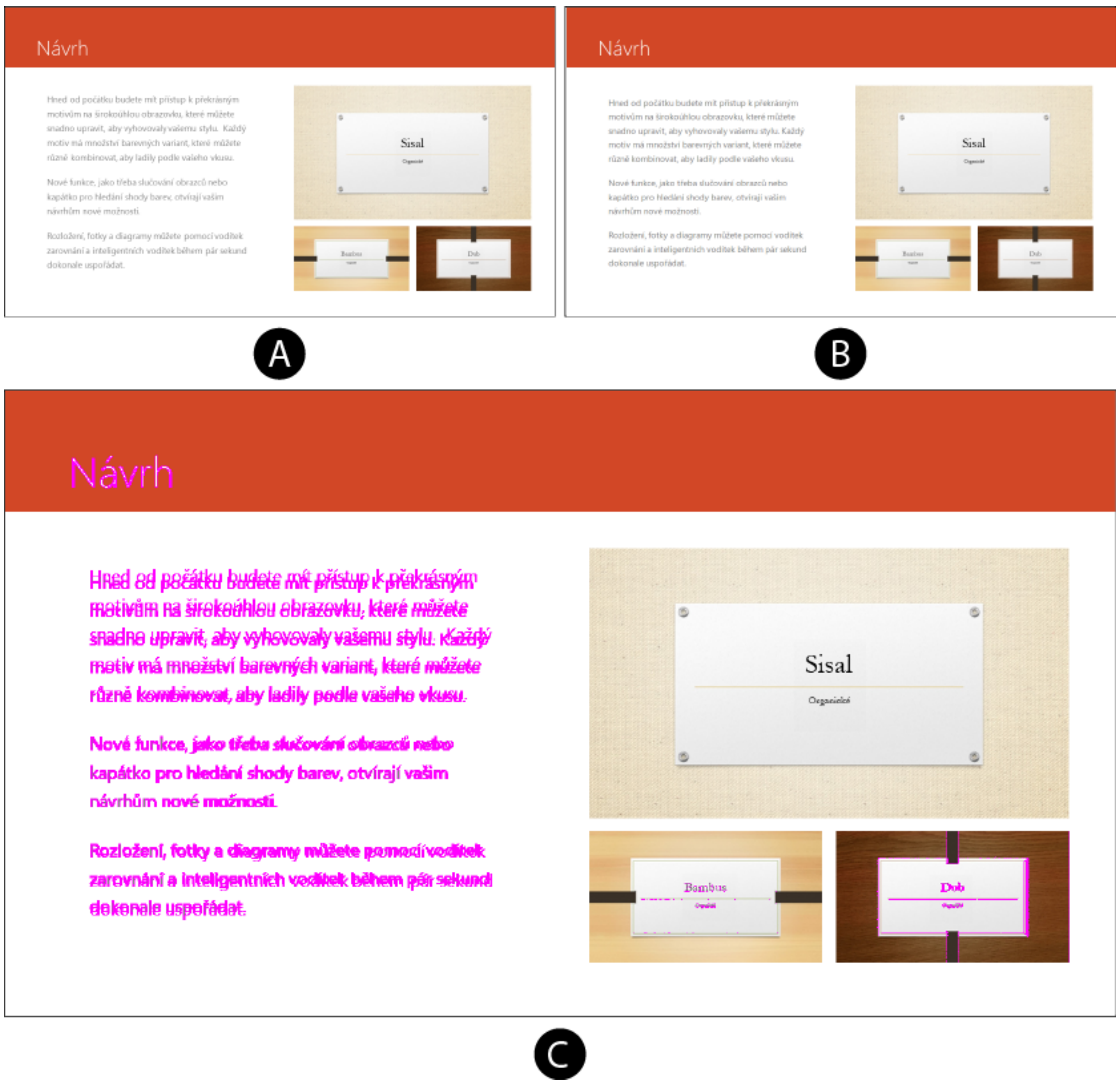
6.3.4 The Summary of the Evaluation

The achieved results can be clearly evaluated because the comparisons are automatically evaluated and because each test also outputs the percentual mismatch value when it fails. The average value of all the mismatches is around 5%. It must be pointed that this value is affected by the actual tests and the value can be very different for different test suites. Therefore, it is important to specify the test suites. They contain only text with standard options such as font, alignment, size and colour. The image, audio and video tests also contain only the image with no special effects. The only supported geometry of the shapes is rectangular. The animations are not implemented and therefore no tests are testing this


Vítá vás PowerPoint

Nádherné prezentace navrhnete a předvedete snadno a s jistotou.

Figure 6.2: Comparison of a slide with text. It contains three images. Image A shows the reference image generated by the PowerPoint itself. Image B is the actual result from the application. Image C is their comparison. Pink elements are the differences between them.



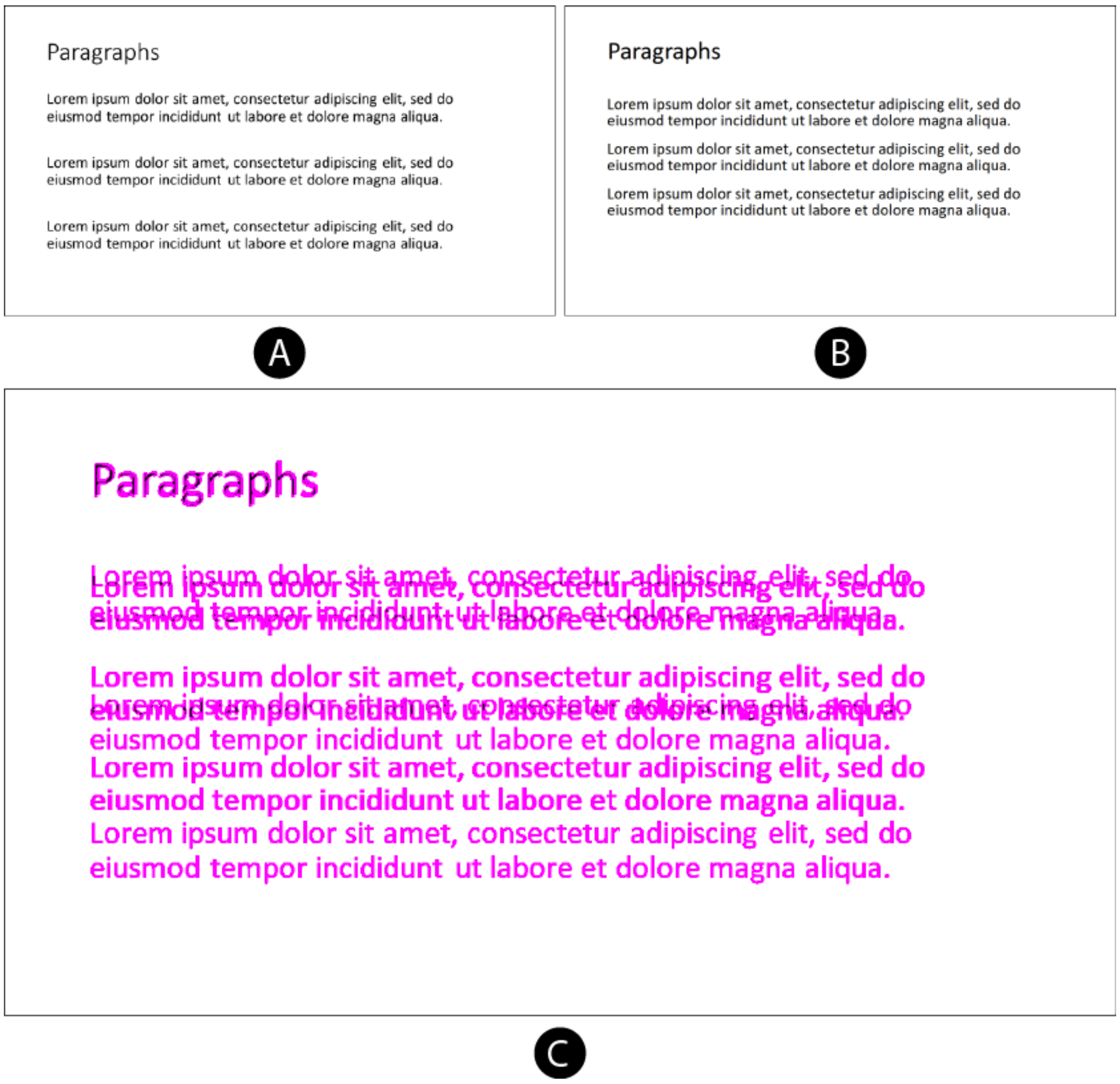
Návrh

Hned od počátku budete mít přístup k překrásným motivům na širokoúhlou obrazovku, které můžete snadno upravit, aby vyhovovaly vašemu stylu. Každý motiv má množství barevných variant, které můžete různě kombinovat, aby ladily podle vašeho vkusu.

Nové funkce, jako třeba slučování obrazců nebo kapátko pro hledání shody barev, otvírají vašim návrhům nové možnosti.

Rozložení, fotky a diagramy můžete pomocí vodítek zarovnání a inteligentních vodítek během pár sekund dokonale uspořádat.

Figure 6.3: Comparison of a slide with text. The explanation of the image is the the same as in Figure 6.2.



Paragraphs

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Paragraphs

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

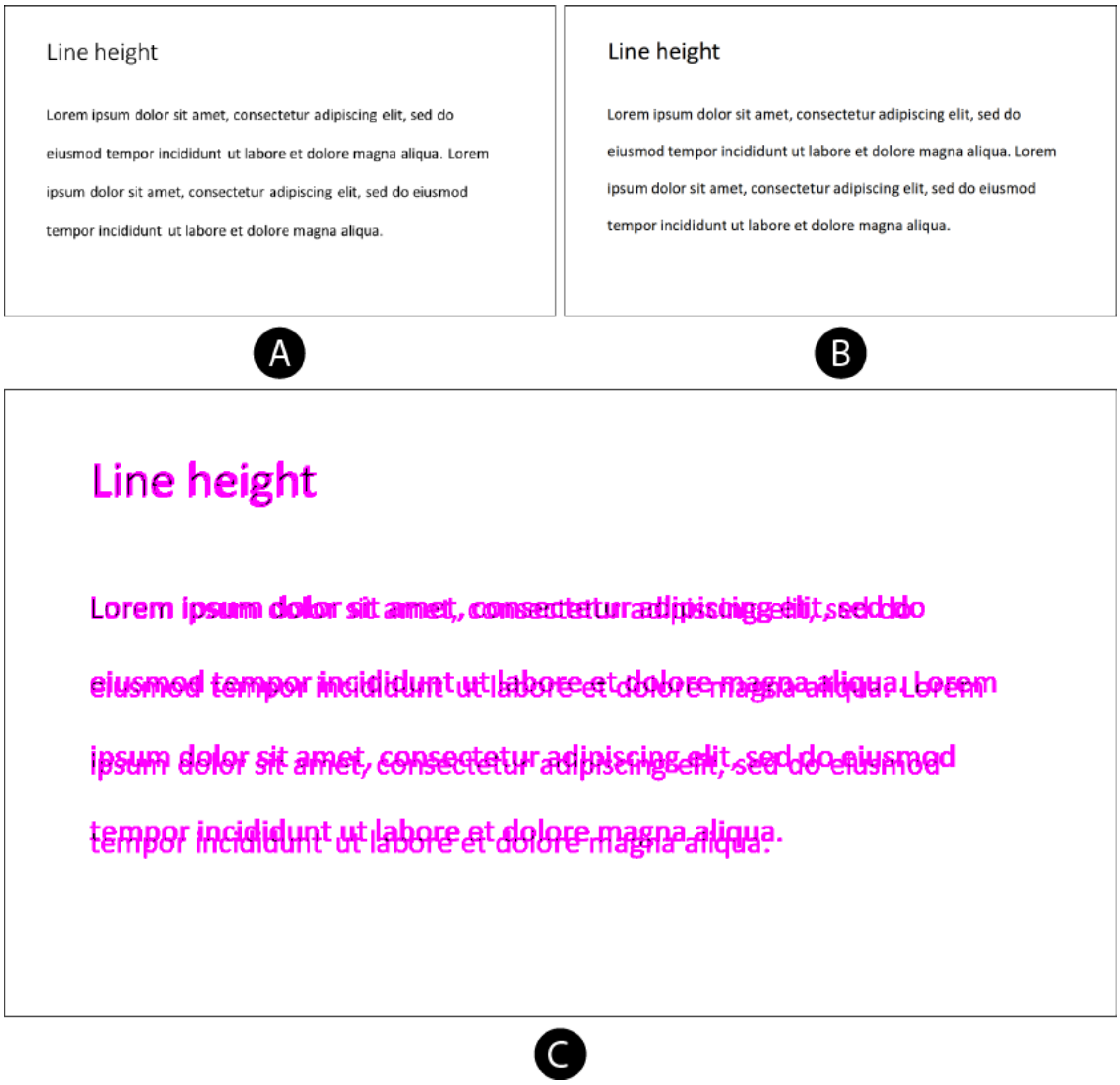
Paragraphs

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Figure 6.4: The comparison of a slide with text. The explanation of the image is the same as in Figure 6.2.



Line height

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

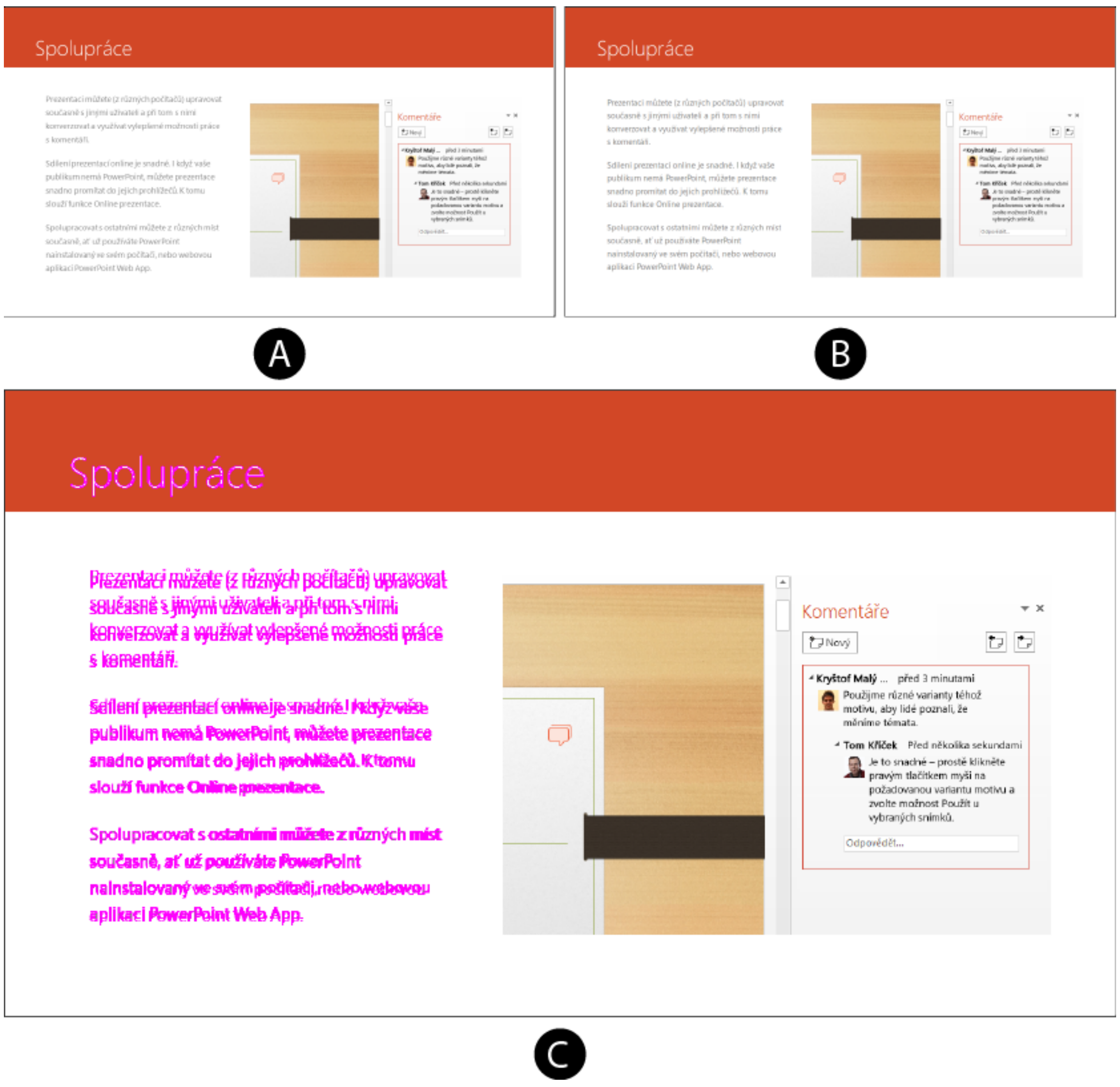
Line height

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Line height

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Figure 6.5: The comparison of a slide with text. The explanation of the image is the same as in Figure 6.2.



Spolupráce

Prezentaci můžete (z různých počítačů) upravovat současně s jinými uživateli a při tom s nimi konverzovat a využívat vylepšené možnosti práce s komentáři.

Sdílení prezentací online je snadné. I když vaše publikum nemá PowerPoint, můžete prezentace snadno promítat do jejich prohlížečů. K tomu slouží funkce Online prezentace.

Spolupracovat s ostatními můžete z různých míst současně, ať už používáte PowerPoint nainstalovaný ve svém počítači, nebo webovou aplikaci PowerPoint Web App.

Komentáře

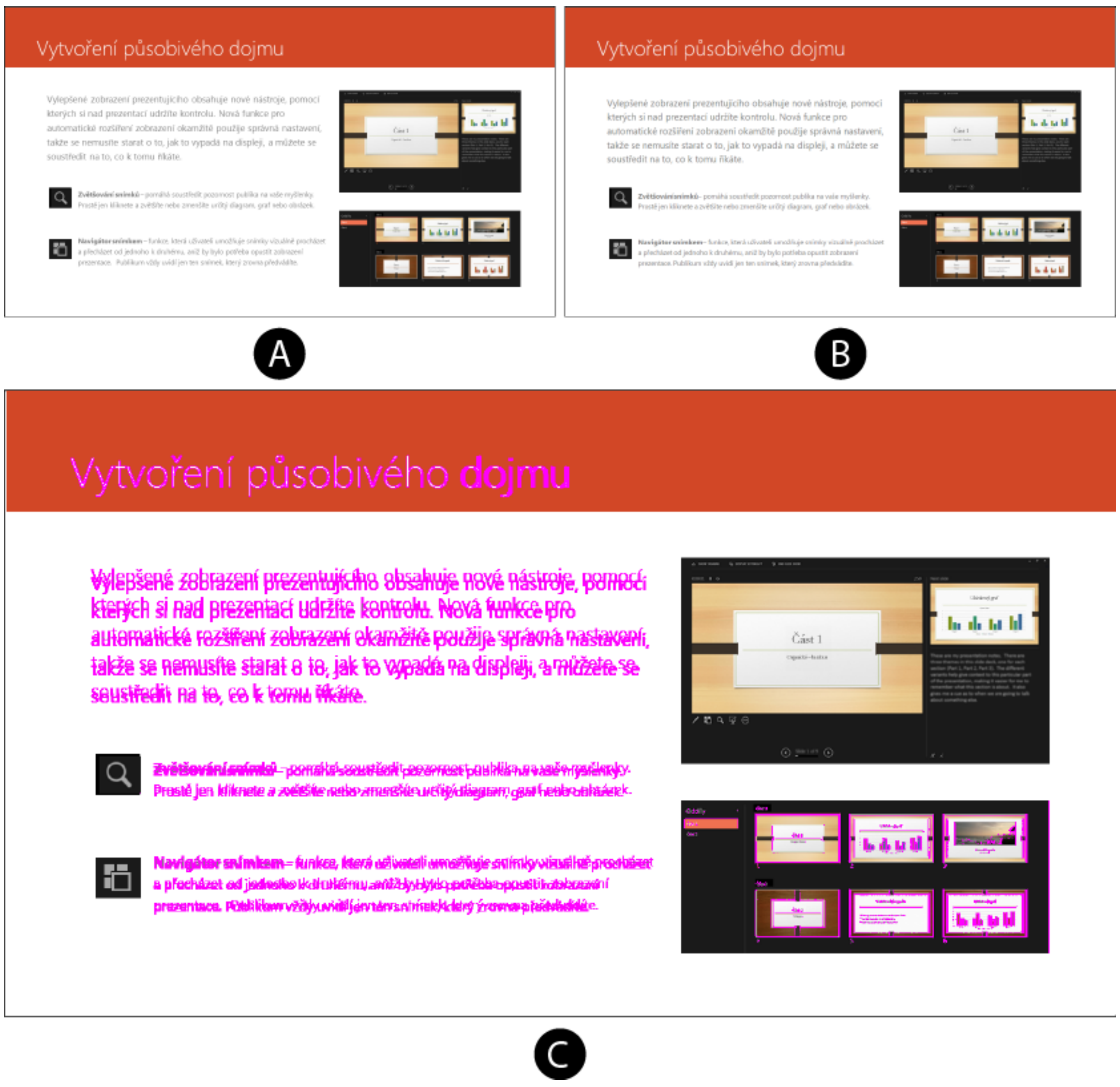
Nový

Kryštof Malý ... před 3 minutami
Použijme různé varianty téhož motivu, aby lidé poznali, že měníme témata.

Tom Kříček Před několika sekundami
Je to snadné – prostě klikněte pravým tlačítkem myši na požadovanou variantu motivu a zvolte možnost Použít u vybraných snímků.

Odpovědět...

Figure 6.6: The comparison of a slide with image. The explanation of the image is the same as in Figure 6.2.



Vytvoření působivého dojmu

Vylepšené zobrazení prezentujícího obsahuje nové nástroje, pomocí kterých si nad prezentací udržíte kontrolu. Nová funkce pro automatické rozšíření zobrazení okamžitě použije správná nastavení, takže se nemusíte starat o to, jak to vypadá na displeji, a můžete se soustředit na to, co k tomu říkáte.

Zvětšování snímků – pomáhá soustředit pozornost publika na vaše myšlenky. Prostě jen kliknete a zvětšíte nebo zmenšíte určitý diagram, graf nebo obrázek.

Navigátor snímkem – funkce, která uživateli umožňuje snímky vizuálně procházet a přecházet od jednoho k druhému, aniž by bylo potřeba opustit zobrazení prezentace. Publikum vždy uvidí jen ten snímek, který zrovna předvádíte.

Figure 6.7: The comparison of a slide with images. The explanation of the image is the same as in Figure 6.2.

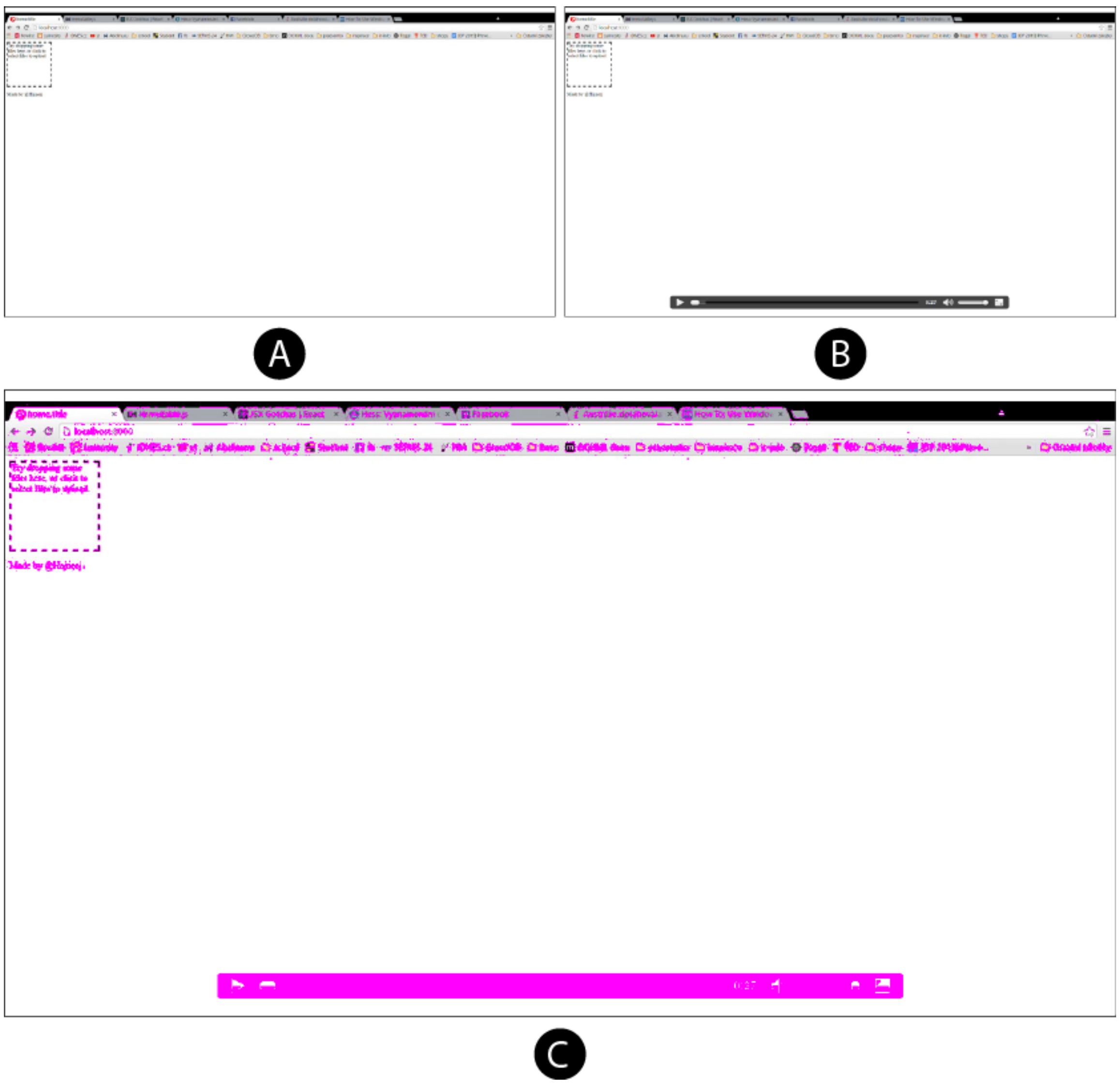

A
B
home.title
localhost:8080
Try dropping some files here, or click to select files to upload.
Made by @lajtoj.
C

Figure 6.8: The comparison of a slide with video. The explanation of the image is the same as in Figure 6.2.

feature. The same applies to the animated transitions. They are currently implemented but it is more a proof of concept than an actual solution. The application supports classic and timed transition between the slides.

Chapter 7

Conclusion

This work shows the application that allows to view PowerPoint presentations in the web browser. This allows to view and share the content of the presentation through the web environment.

The precision of the conversion is the key aspect of this application. If the slides follow basic rules and do not contain special effects, then the 5% difference between the original slide and the actual slide is easily achievable. In addition, the application also contains development tools to iteratively increase the level of precision.

This work shows how modern web applications are built. The web environment is very dynamic and there are numerous tools to choose from and to use. This work shows one approach how the tools can be used and combined together. The application can also be a core of a start-up project, because the application extends the ability of editing the content in the web environment.

Besides the application, the work introduces one approach to building modern web applications. This can be used by web developers to get inspired and use some tools or procedures in their project. The idea of displaying a PowerPoint presentation in the web browser also deserves to be extended in some start-up project.

Bibliography

[1] Ecma-376, 4th edition [online]. http://www.ecma-international.org/publications/files/ECMA-ST/ ECMA-376, Fourth Edition, Part 1 - Fundamentals And Markup Language Reference.zip, 2012 [cit. 2016-01-04].

[2] React redux [online]. https://github.com/reactjs/react-redux/blob/master/docs/api.md, 2014-04-02 [cit. 2016-05-01].

[3] Redux [online]. http://redux.js.org/index.html, 2014-04-25 [cit. 2016-04-30].

[4] Flux [online]. https://facebook.github.io/flux/docs/overview.html, 2014 [cit. 2016-04-30].

[5] Generators [online]. https://www.promisejs.org/generators/, 2015-09-24 [cit. 2016-04-28].

[6] Async functions [online]. http://tc39.github.io/ecmascript-asyncawait/, 2016-01-26 [cit. 2016-04-26].

[7] remy/nodemon [online]. https://github.com/remy/nodemon/, 2016-02-09 [cit. 2016-04-28].

[8] npm documentation [online]. https://docs.npmjs.com/, 2016-02-10 [cit. 2016-04-27].

[9] Web technology for developers [online]. https://developer.mozilla.org/en-US/docs/Web, 2016-03-03 [cit. 2016-04-30].

[10] webpack [online]. https://webpack.github.io/docs/, 2016-03-12 [cit. 2016-04-28].

[11] Modules [online]. https://nodejs.org/api/modules.html, 2016-03-13 [cit. 2016-04-27].

[12] function* - javascript [online]. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*, 2016-03-14 [cit. 2016-04-27].

[13] About node.js [online]. https://nodejs.org/en/about/, 2016-03-19 [cit. 2016-04-26].

[14] The node.js event loop, timers, and process.nexttick() [online]. https://github.com/nodejs/node/blob/master/doc/topics/the-event-loop-timers-and-nexttick.md, 2016-03-19 [cit. 2016-04-26].

[15] Overview of blocking vs non-blocking [online].
`https://github.com/nodejs/node/blob/master/doc/topics/blocking-vs-non-blocking.md`, 2016-03-19 [cit. 2016-04-26].

[16] gulp/docs [online]. `https://github.com/gulpjs/gulp/tree/master/docs`, 2016-04-08 [cit. 2016-04-24].

[17] Express - node.js web application framework [online]. `http://expressjs.com`, 2016-04-16 [cit. 2016-04-27].

[18] Mocha - the fun, flexible javascript test framework [online]. `https://mochajs.org`, 2016-04-16 [cit. 2016-04-27].

[19] Promise - javascript [online].
`https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Global_Objects/Promise`, 2016-04-20 [cit. 2016-04-26].

[20] Errors node.js manual & documentation [online].
`https://nodejs.org/api/errors.html`, 2016-04-20 [cit. 2016-04-27].

[21] Concurrency model and event loop [online].
`https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`, 2016-04-29 [cit. 2016-04-30].

[22] Dan Abramov. Mixins are dead. long live composition. blogpost, 2015 [cit. 2016-04-10].
`https://medium.com/@dan_abramov/mixins-are-dead-long-live-higher-order-components-94a0d2f9e750`.

[23] Dan Abramov. Presentational and container components [online]. blogpost, 2015 [cit. 2016-04-10].
`https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0`.

[24] Christian Bromann. Developer guide. `http://webdriver.io/guide.html`, 2015 [cit. 2016-04-28].

[25] Facebook Inc. Getting started | react [online].
`https://facebook.github.io/react/docs/getting-started.html`, 2013 [cit. 2016-04-28].

[26] Michael Jackson. Introduction [online].
`https://github.com/reactjs/react-router/blob/master/docs/Introduction.md`, 2015 [cit. 2016-04-29].

[27] Matthew MacDonald. *HTML5: The Missing Manual, 2nd Edition*. O'Reilly Media, 2013.

[28] David Sawyer McFarland. *CSS: The Missing Manual, 4th Edition*. O'Reilly Media, 2015.

[29] Jason Pamental. *Responsive Typography: Using Type Well on the Web*. O'Reilly Media, 2014.

[30] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, 2008.

[31] Robert Gaskins. *Sweating Bullets*. Vinland Book, 2012. ISBN 978-0-9851424-1-4.

[32] Jose Maria Arranz Santamaria. The single page interface manifesto [online]. `http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php`, 2015-09-21 [cit. 2016-03-15].

[33] Daniel Steigerwald. Este - dev stack and starter kit [online]. `https://github.com/este/este`, 2015 [cit. 2016-01-10].

[34] Twitter. Readme [online]. `https://github.com/twitter-fabric/velocity-react`, 2015, [cit. 2016-04-29].

Appendices

List of Appendices

Appendix A

Content of the CD

Content of the enclosed CD is shown in Table A.1.

Name	**Description**
src-app/	All the source code of the application
src-text/	All the source code of the thesis
readme.txt	Basic description of the application
install.txt	Installation instructions
master-thesis.pdf	The thesis
poster/	Poster of the thesis
video/	Video about the thesis

Table A.1: Content of the enclosed CD.