Michael Steinberger, BSc

Design of a Web Visualization Concept for the PUMA Automation System

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Electrical Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach

Institute of Technical Informatics

Dr. Ludwig Bloder, AVL List GmbH

Graz, December 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.
The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Acknowledgments

Firstly, I would like to thank my thesis advisor Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach of the Institute of Technical Informatics at Technical University of Graz, for his comprehensive support of my master's thesis.

In addition, I would also like to thank Ludwig Bloder as my supervisor during my work at AVL List GmbH, especially, for his highly valuable feedback during the entire work on my thesis. I would also like to thank Herwig Schelch, Werner Fuchs and Matthieu Clauet, who, among other things, made it possible for me to write my thesis at the AVL List GmbH.

Furthermore, I would like to thank the team at AVL List GmbH for accepting me very well in their team and supporting me whenever I had problems. Especially, I want to express my special thanks to Harald Rosenberger for our great discussions. In addition, I would like to thank Clemens Meinhart and his team for their support when I had problems on the server-side. Last but not least, I would like to thank Wolfgang Neubauer, who gave me insights in the current visualization of the PUMA automation system.

Michael Steinberger
Graz, December 2016

Abstract

The automation system AVL PUMA Open 2™ is a software suite for powertrain development. The software suite offers a set of tools for parameterization, monitoring and the control of powertrain testbeds. As an interface between the automation system and the testbed operator, the desktop application PUMA Operator Interface (POI) provides all functionality of the automation system via a Graphical User Interface.

In the course of this master's thesis, the desktop application POI was extended by a web visualization. The current HTML5 standard and a state-of-the-art JavaScript framework was used for the visualization. A Single-Page Application (SPA) hosts the visualization and is structured by controls, which are implemented following the Web Components standard.

An infrastructure was established to give the testbed operator the possibility to customize the visualization in a form that maps best to the current testbed setup. Thereby, the infrastructure supports the assignment of data retrieved from the automation system to a control during the runtime of the SPA. Furthermore, simple presentation logic can be implemented by the testbed operator for example to indicate when a measurement value of the automation system reached a limit. In addition, a library was established that provides a set of Web Components to allow a reuse within multiple web applications.

This work presents a realistic solution and a feasibility study of several applications for a web visualization, which can serve as a basis for a further development towards a remote monitoring of the testbed status. Performance tests present the processor load and memory usage of the web-based solution to ensure no performance impacts of the web visualization on the automation system.

Kurzfassung

Das Automatisierungssystem AVL PUMA Open 2™ ist ein Software-Paket für die Entwicklung von Antriebssträngen und umfasst Software-Werkzeuge für die Parametrierung, Überwachung und Steuerung von Prüfständen in der Automobilindustrie. Im Form eine Desktop-Applikation ermöglicht das PUMA Operator Interface (POI) durch eine grafische Benutzeroberfläche die einfache Überwachung und Steuerung des Automatisierungssystems.

Im Zuge dieser Diplomarbeit wurde die Desktop-Applikation POI durch eine Web-Visualisierung erweitert. Dabei wurde der aktuelle HTML5 Standard und ein JavaScript Framework verwendet. Die Visualisierung ist in eine Single-Page Webanwendung (SPA) eingebettet und wird durch Controls strukturiert, welche den Web Components Standard implementieren.

Eine Infrastruktur wurde umgesetzt um den Prüfstandsingenieur eine Anpassung der Visualisierung an den aktuellen Aufbau des Prüfstands zu ermöglichen. Dabei wurde die Möglichkeit geschaffen einen Messwert des Automatisierungssystems durch ein Control anzeigen zu lassen und während der Laufzeit der SPA diese Zuweisung dynamisch zu verändern. Weiters kann einfache Präsentationslogik implementieren werden um zum Beispiel das Erreichen des Limits eine Messgröße darzustellen. Zusätzlich wurde die Basis einer Control-Bibliothek für die Wiederverwendung in weiteren Webanwendungen geschaffen.

Diese Arbeit präsentiert eine realistische Lösung und eine Machbarkeitsstudie von mehreren Anwendungsfällen für eine Web-Visualisierung, welche als Basis für zukünftige Entwicklungen in Richtung eine Fernwartung von Prüfständen dienen kann. Performance-Messungen stellen die Prozessorlast und den Speicherbedarf der Web-Visualisierung dar, um sicherzustellen dass die Web-Visualisierung keinen Einfluss auf das Automatisierungssystem hat.

Contents

List of Abbreviations **iii**

1 Introduction **1**
1.1 The PUMA Automation System 1
1.2 Graphical User Interface of the PUMA Automation System 3
1.3 Web Interface for PUMA Resources 5
1.4 Extending the Visualization of the PUMA Automation System 6

2 Related Concepts for a Web Visualization **11**
2.1 Single-Page Application Frameworks as a Basis for Web Development . . 11
2.2 Change Detection in Single-Page Applications 27
2.3 Dependency Injection in Single-Page Application Frameworks 34

3 Infrastructure for the Web Visualization **39**
3.1 Structure of a Web Window 39
3.2 Startup Process of a Web Window 43
3.3 The Building Blocks of the Infrastructure 49
3.4 Data Source Assignment 55
3.5 Custom Presentation Logic 61

4 Visualization Library for Web Applications **68**
4.1 Structure of the Library 68
4.2 The Alphanumeric Control as a Model Example for a Control of the Visualization Library 70
4.3 The Xt-Control as an Implementation of a Multi-Series Line Chart . . . 71

5 Performance Evaluation **73**
5.1 Evaluation Setup 73
5.2 Alphanumeric Test Cases 75
5.3 Framework Evaluation 84
5.4 Hardware Rendering 86
5.5 Xt-Control Test Cases 87

5.6 Test Cases on a Mobile Device 88

6 Conclusion **94**

7 Future Prospects of the Web Visualization **97**

Bibliography **99**

List of Abbreviations

ADE	Advanced Design Editor
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASAccessServer	Automation System Access Server
AVL	Anstalt für Verbrennungskraftmaschinen List
Blob	Binary Large Object
CAN	Controller Area Network
CEF	Chromium Embedded Framework
COM	Component Object Model
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DIP	Dependency Inversion Principle
DOM	Document Object Model
DSL	Domain-Specific Language
ES6	ECMAScript 2015 Version 6
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IE	Internet Explorer

List of Abbreviations

IoC	Inversion of Control
IPC	Inter-Process Communication
JSON	JavaScript Object Notation
POI	PUMA Operator Interface
LCIE	Loosely-Coupled Internet Explorer
MFC	Microsoft Foundation Class
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
MVW	Model-View-Whatelse
OOP	Object-Oriented Programming
RAM	Random-Access Memory
REST	Representational State Transfer
RIA	Rich Internet Application
RxJS	Reactive Extensions for JavaScript
SoC	Separation of Concerns
SPA	Single-Page Application
SVG	Scalable Vector Graphics
UI	User Interface
URI	Uniform Resource Identifier
XML	Extensible Markup Language
WPF	Windows Presentation Foundation

1 Introduction

1.1 The PUMA Automation System

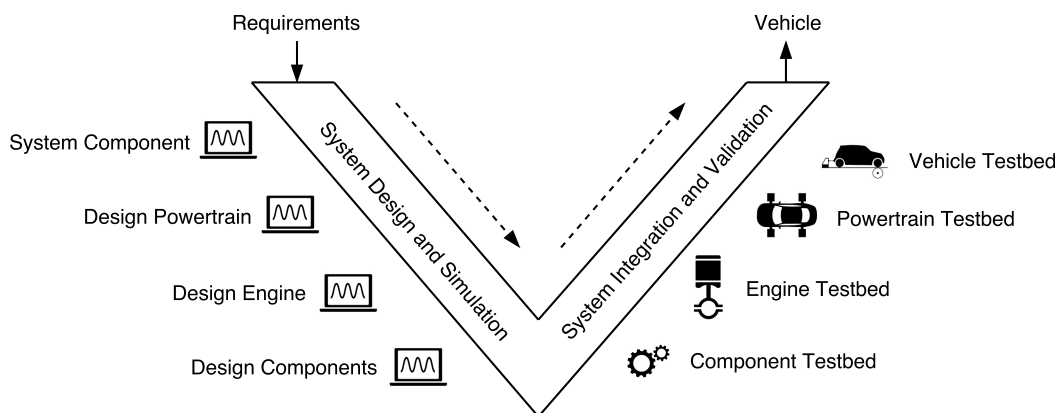

Figure 1.1: Testbed Types used during the Vehicle Development Cycle described by the V-Model, based on Paulweber and Lebert (2014)

As presented in Figure 1.1, the development cycle in the automotive industry can be described by the V-Model. In the beginning of the system design and simulation phase, the overall requirements of the vehicle are broken down into requirements of the components of the vehicle, which form the basis of the subsequent design phase. After the design process, all design decisions are evaluated in the system simulation phase by Software-in-the-Loop (SiL) or Model-in-the-Loop (MiL) systems. Through the increased complexity of state-of-the-art vehicles caused by new powertrain concepts such as hybrid powertrains or e-mobility with reduced CO_2 emissions, the influence of every design decision on the overall system needs to be evaluated early in the development process. For this reason, every design phase has its counterpart in form of a testbed type in the system integration and validation process along the right arm of the V-model and thus, enables to test every component in parallel to the development of other components.

Thereby, depending on the testbed type, the missing components of the system need to be simulated, for example for a powertrain testbed the wheels, the chassis of the car and the maneuvers, which would otherwise be carried out by the test driver. Because of the increased system complexity trough the interaction between intelligent sub-systems such as driver assistance systems,as stated in Paulweber and Lebert (2014), the test cycles should be based on real load profiles instead of synthetical load profiles commonly used in SiL or MiL systems.

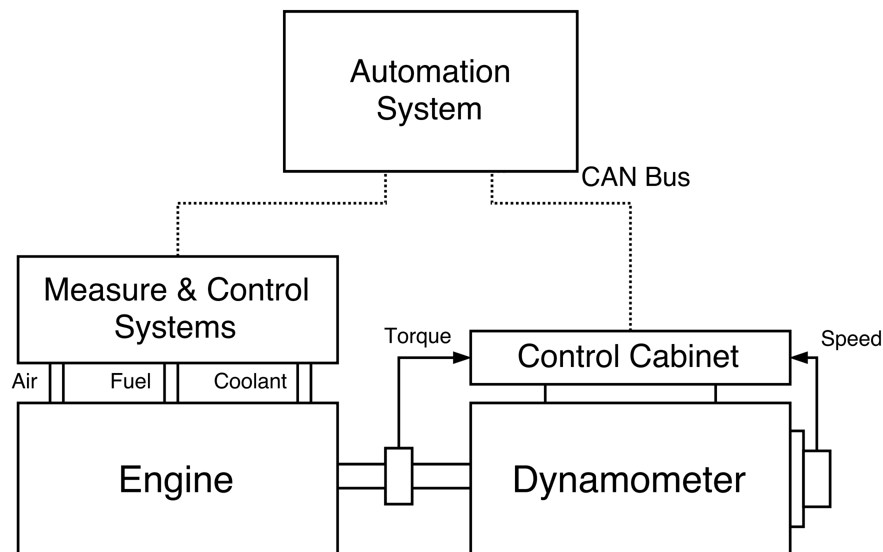

Figure 1.2: Overview of a Combustion Engine Testbed, based on Fang and Lahdelma (2016)

An exemplary setup of an engine testbed for the development of a combustion engine is shown in Figure 1.2. In such a setup, a dynamometer acts as a controllable load for the engine under test. For driving a specific test cycle, the automation system, such as the AVL PUMA Open 2™, sends Controller Area Network (CAN) messages, which define the current torque and speed to the control cabinet. The control cabinet brings the dynamometer in the specified state and returns the current torque and speed values to close the control loop. During the execution of the test cycle, measure systems report the fuel consumption or intake air mass flow rate of the engine to the automation system via the CAN bus. In addition, control systems regulate important parameters such as the temperature of the coolant.

The software suite AVL PUMA Open 2™ offers tools for the parameterization, monitoring and the control of all testbed types as shown in Figure 1.1. By the use of parameter sets,

the devices, for example the dynamometer, connected at the testbed can be configured. Parameter sets define a mapping between the measure or control values of the testbed and unique channels, which are known in the automation system, for example the actual rotational speed of the dynamometer is mapped to the channel *act_dyno_speed*, which can be accessed inside the automation system by its unique identifier. A graphical test run editor facilitates the creation of test cycles and monitors the current status of the testbed during the execution of a test run. In addition, the automation system allows the manual operation of the testbed (AVL List GmbH, 2016a; AVL List GmbH, 2016b).

1.2 Graphical User Interface of the PUMA Automation System

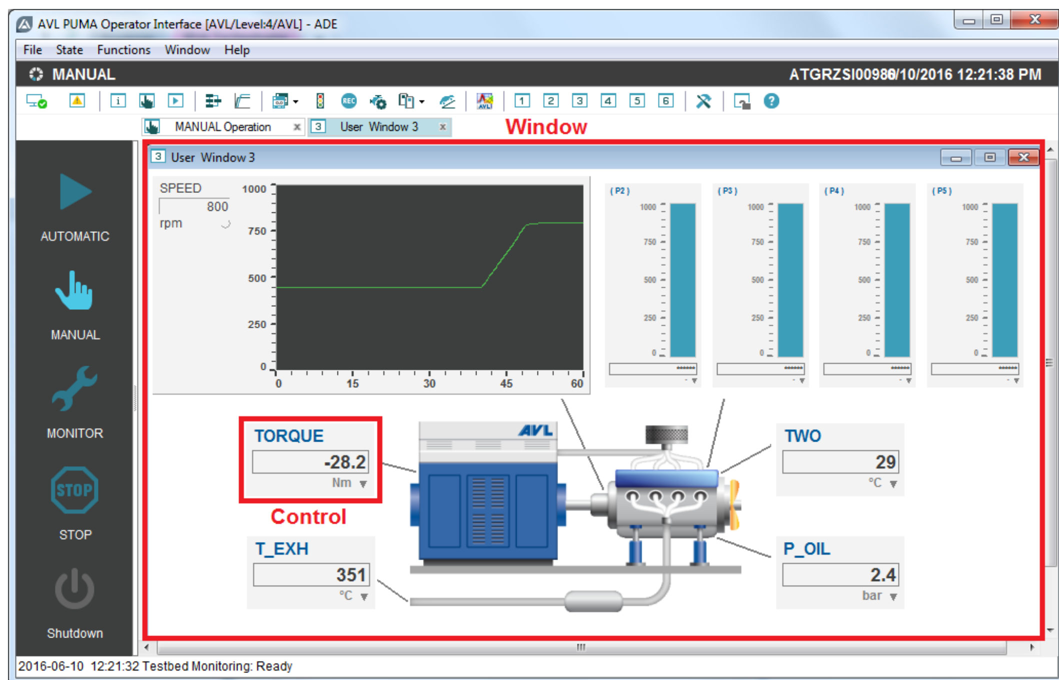

Figure 1.3: Visualization of the Testbed Status within the PUMA Operator Interface

As an interface between the automation system and the testbed operator, the software suite offers the PUMA Operator Interface (POI). The POI desktop application provides

all functionality of the automation system via a Graphical User Interface (GUI). An exemplary visualization of the POI is shown in Figure 1.3. The POI is structured into multiple windows, which host controls visualizing the testbed status. A testbed operator can choose between a predefined set of windows to display the current status of the testbed channels. During the execution of a test run, the operator can change the assigned channel of a control through a browser dialog, which opens with a double-click on the control. In addition, the testbed operator can bring the POI in a specific DESIGN mode. In this mode, the layout of the window can be changed, custom windows can be created, and controls out of a set can be added to the window. In this way, every testbed operator can define a custom visualization, which suits best the current test run and the real testbed setup. In Figure 1.3, a predefined window is used to visualize channels describing the testbed status such as the current torque and speed of the dynamometer.

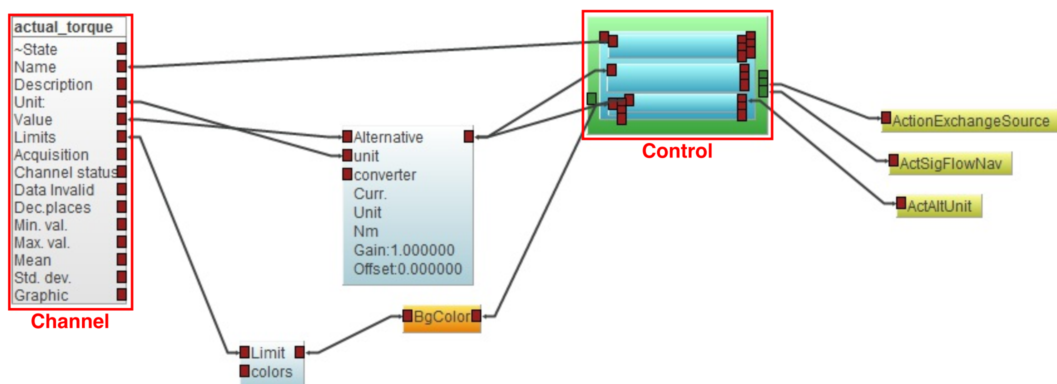

Figure 1.4: Unit Conversion and Limit Monitoring with the Advanced Design Editor

Furthermore, the Advanced Design Editor (ADE) can be opened for every control in the window. Inside the ADE, every visual element or event of a control, for example the text field or the click event of an alphanumeric control, is hereby accessible from a block with multiple in- and outputs (see Figure 1.4). To bind channel properties to the control, a data source block can be inserted and its properties can be routed in a flow-based programming manner to the inputs of the control, for example the channel name to the text field of an alphanumeric control. Figure 1.4 shows also the possibility to insert further functional blocks between channel properties and control inputs to implement custom presentation logic, for example to monitor minimum and maximum values of a channel, that is limit monitoring.

1.3 Web Interface for PUMA Resources

The ASAccessServer is implemented as a Representational State Transfer (REST) interface for a web-based access of PUMA resources. Fulfilling the requirements of the REST architectural style stated in Fielding (2000, Chapter 5), the Automation System Access Server (ASAccessServer) acts stateless and returns the requested PUMA resource independently of preceding Hypertext Transfer Protocol (HTTP) requests.

In Figure 1.5, the data structure inside the PUMA automation system and the relation to the data sources provided by the ASAccessServer are presented. Hereby, the following objects are known inside the PUMA automation system:

- **Channel:** A channel reflects a measure or control value of the automation system, for example the current speed of the dynamometer.
- **Quantity:** A quantity maps to a channel of the automation system and extends the data provided from the channel with additional meta data.
- **Activation Object:** An activation object can be used to trigger events on the automation system, for example to start the measurement of a specific channel.

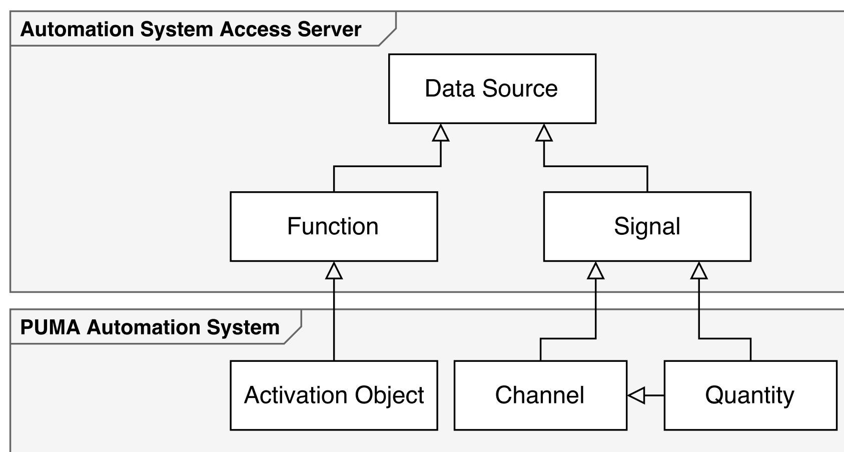

Figure 1.5: Data Structure of the Automation System Access Server

The REST interface of the ASAccessServer abstracts both channel and quantity resources by the data source type Signal. In addition, the activation object can be accessed from the interface as a Function data source. For a more efficient retrieval of channel or quantity values with a higher update rate up to 10kHz, an additional data source, the

XtSignal will be implemented. The XtSignal data source would return, in contrast to the single value of a Signal data source, a vector of values for each HTTP request.

Following the REST architectural pattern, each data source type, for example a Signal data source, can be obtained as a server resource by using either a GET or POST HTTP verb with an Uniform Resource Identifier (URI) (Pautasso, Zimmermann, and Leymann, 2008). The complete Application Programming Interface (API) is defined in a TypeScript definition file for data format consistency between the ASAccessServer and any client which draws the PUMA resources from this API (Microsoft, 2016b).

1.4 Extending the Visualization of the PUMA Automation System

1.4.1 Motivation for a Web Visualization

This work presents a concept for a web visualization for the PUMA automation system. The following points outline the need of an extension of the current visualization with a web visualization.

Remote monitoring: A common requirement of the visualization of the PUMA automation system is to diagnose the actual state of the testbed from a remote location. The current visualization in the POI only supports to monitor the actual PUMA state from a remote location through a Virtual Network Computing (VNC) connection to the testbed computer. The necessary VNC connection complicates, especially on a mobile device, a quick overview of the PUMA state or a parallel monitoring of several testbeds. In addition, a remote support in case of a software error can currently only achieved through the tedious analysis of a large number of log files transmitted in a single archive file.

The introduction of a web visualization can simplify the remote monitoring of the actual state of the testbed. Through the established REST interface provided by the ASAccessServer, the data from a testbed can be accessed easily from any remote location and presented by the web visualization. In contrast to the current visualization, no extra software, such as the PUMA Operator Interface, needs to be installed on the device, which enables the remote monitoring. Furthermore, with a web visualization no restrictions exist anymore in terms of installed operating system or hardware equipment of the device.

In addition, the remote diagnosis can ease the support in case of a software error and the parallel tracking of the state of several testbeds. Especially for large test fields, the requirement exists to monitor the health and current test duration of all testbeds of the test field from a remote location or on a mobile device. In case of an error in one of the testbeds, the testbed operator can be informed instantly through the web visualization and hence, costly idle times of the testbed can be avoided. Furthermore, it is often necessary to monitor the current testbed status without disturbing the testbed engineer, who works on the testbed using the POI. Here, the web visualization can enable the parallel monitoring of the testbed state and it can provide a different representation of the testbed showing other control or measurement values as defined in the visualization layout of the POI.

State-of-the-art visualization: In addition to the requirement for remote monitoring, an increasing demand of the customer exists for a state-of-the-art visualization of the actual testbed state. The current visualization, that is the POI, was developed using the Microsoft Foundation Class (MFC) library, which has already experienced with the Windows Forms and Windows Presentation Foundation (WPF) libraries two successors for creating user interfaces for the Windows operating system using the .NET framework. In contrast to the MFC library, especially the latest WPF library allows an alignment of the User Interface (UI) to current styling standards, such as introduced with the Microsoft design language, and to a styling which requires more advanced criteria to meet the corporate identity. The introduction of a web visualization can offer the possibility to style the visualization towards a state-of-the-art UI and a consistent user experience for all applications offered by the AVL List GmbH.

Extensibility: Notwithstanding the above, the integration of additional controls in the POI is difficult and requires expert knowledge, which is currently only possessed by a single developer in the AVL List GmbH. Furthermore, software developers with the required C++ background to extend the current visualization are increasingly difficult to find. The introduction of a web visualization based on the latest HTML5 technology for creating user interfaces can solve this problem. The development of a web visualization is well documented in the web and developers are easier to find for JavaScript development as for more complex languages as C++. Furthermore, a large number of frameworks and libraries exist for creating web visualizations, which can be used to decrease the development effort and costs.

1.4.2 Applications of the Web Visualization

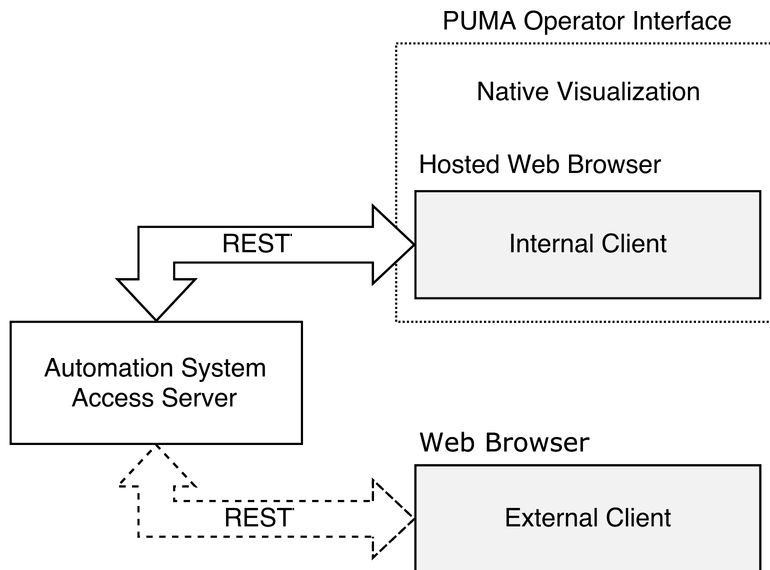

Figure 1.6: Applications of the Web Visualization

For the above mentioned reasons, the POI visualization introduced in Section 1.2 shall be extended by a web-based solution. This solution will cover two applications: (i) an internal and (ii) an external client. For both clients, the technology to describe the view, that is the content of the web page, can be restricted to the current HTML5 standard.

The internal client shall act as an extension of the current visualization inside POI. For this purpose, a web browser will be hosted inside the POI desktop application to display the current status of the testbed using web technology. The existing REST interface of the ASAccessServer will give the internal client full access to all resources of the testbed such as the current channel values (see Figure 1.6). The external client will monitor the status of the testbed outside the POI. The scope of application shall include a simple testbed status overview showing the state of the automation system and additional information such as the currently loaded parameter sets. A possible case of application could be to monitor from a remote location via a mobile device the current state of a test run or the total time of the test run execution. The same technology shall be used as for the internal client. For the external client, a restricted interface to the ASAccessServer will be used (depicted as a dashed line in Figure 1.6).

1.4.3 Requirements of the Web Visualization

Infrastructure for the Web Visualization

An infrastructure needs to be established to fulfill the requirements of the web visualization. The infrastructure should realize a seamless integration of a web visualization inside the POI. For this reason, the web visualization shall be organized in independent windows, which hold all relevant information (for example the list of needed channels) and shall be hosted by the ASAccessServer. The controls embedded inside a window shall be part of a common library. The resources needed for the execution of the infrastructure (JavaScript files, HTML files, etc.) shall be obtained from one common place to guarantee that the infrastructure and common library can be updated without the adaption of an existing window.

Similar to the current visualization, the infrastructure for the web visualization needs to support the assignment of channels of the automation system to a control. During the runtime of the visualization, the infrastructure must allow to change the assigned channel of a control via a browser dialog and to persist this change across sessions. Through the existing REST interface of the ASAccessServer, the actual state of the channels shall be periodically retrieved. The infrastructure shall manage the efficient retrieval of needed channels and avoid duplicate HTTP requests if the same channel is assigned to multiple controls. The retrieval of the ASAccessServer resources shall be encapsulated inside an separate module in the infrastructure to hide the used transport protocol, that is HTTP, and to simplify a future replacement of the protocol, for example by the WebSocket protocol (Fette and Melnikov, 2011).

A possibility to add presentation logic by the testbed operator, as currently implemented in the ADE, shall be evaluated. In contrast to the ADE, the presentation logic shall be implemented with a code snippet written in a syntax similar to JavaScript. The code snippet shall be executed without any involvement of the ASAccessServer. Furthermore, the infrastructure should support to insert existing windows inside a new one. The overall impact of the web visualization on the automation system in terms of processor load and memory usage shall be evaluated on a regular basis during the development of the infrastructure.

Visualization Library for Web Applications

A concept for a visualization library for web applications, that is the internal and external client presented in Figure 1.6, shall be presented. A first foundation block of the library

should consist of several controls, which can run independently or within the infrastructure for the web visualization inside the POI.

Each control of the library must follow an interface to enable the assignment of multiple channels to an input or output of the control, respectively. The concept of the library should support to include third-party controls inside the library and to add a wrapper for each control to unify the interface of all controls. The library should support also internationalization, that is to translate texts depending on the current environment.

2 Related Concepts for a Web Visualization

2.1 Single-Page Application Frameworks as a Basis for Web Development

2.1.1 Introduction to Web Application Models

In the classical web application model presented in Garrett (2005), the major part of the logic runs on the server. The web browser, that is the client, receives as a response to each user action one complete Hypertext Markup Language (HTML) file describing the current view and additional Cascading Style Sheets (CSS) files for the styling of the web application. A client that implements the classical web application model is described in literature also as a thin client (Shklar and Rosen, 2009). As can be seen in Figure 2.1, any user interaction is first processed inside the server in an UI generation block. Then, the necessary data to generate the view is obtained from services. Finally, the view described by a HTML file and the CSS file are sent as a response to the UI of the web browser. Thereby, a major drawback is that the user needs to wait after each action for a response of the server, which is potentially delayed by the network latency and the processing of the user interaction in the UI generation block (Morales-Chaparro et al., 2007). Furthermore, for every received HTML file from the server the entire web page must be reloaded and redrawn in the UI of the web browser and causes in this way a chopped user interaction.

For modern web applications, the Asynchronous JavaScript and XML (AJAX) web application model describes a method to utilize the JavaScript engine as a fundamental element of all modern web browsers, and to modify asynchronously only parts of the UI without an entire page reload (Flanagan, 2011). In this way, a more responsive web application also known as a Rich Internet Application (RIA) can be developed that mimic the user experience known from desktop applications (Rossi et al., 2008, Chapter 2). The

AJAX engine as a pendant to the UI generation block in the classical web application model is moved to the client and retrieves for each user action only parts of the view definition in form of a XML file.

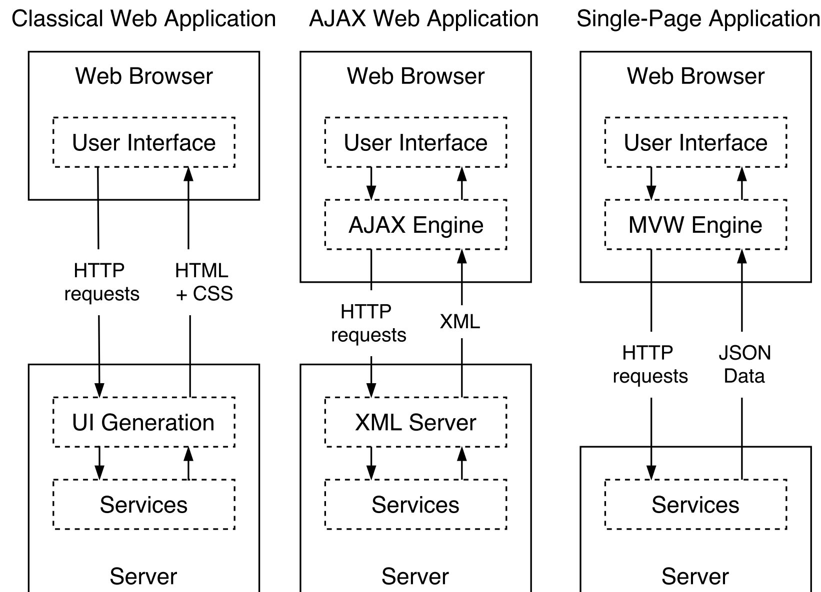

Figure 2.1: The Evolution of Web Application Models, adapted from Petitit and Tricot (2014)

A dominant type of web application that extend the AJAX web application model is a Single-Page Application (SPA). During the entire lifetime of a SPA, no page reload is needed and all necessary resources are retrieved in a single page load (Mikowski and Powell, 2013). The content of a SPA, that is the view and all application logic, is described in one HTML file retrieved during the initial page load, whereby in state-of-the-art web applications techniques exist to accelerate the initial page loading, for example asynchronous or lazy script loading (Barker, 2012). Single-page applications are often implemented by following a derivation of the architectural Model-View-Controller (MVC) pattern. All derivation of the MVC pattern are often summarized by the term Model-View-Whatelse (MVW) pattern (see also Section 2.1.2). As can be seen in Figure 2.1, in a SPA the simple AJAX engine is replaced by a more complex MVW engine, which processes the user actions and generates the view out of the data previously retrieved from server. Nowadays, the XML file format used in early implementations of the AJAX model, is replaced by the JavaScript Object Notation (JSON) file format. Because of the fact that the JSON file format is based on the specification of the JavaScript programming

language, JSON data can be easily converted to a JavaScript object and hence, simplifies the process to parse the received data in the MVW engine. In contrast to the classical web application model, in SPA applications the server is principally responsible to host all relevant resources for the web application, such as JavaScript, HTML and CSS files, and to respond to data requests of the client.

2.1.2 Patterns in Single-Page Applications

For a modern SPA, the application logic, that is for example the processing of the user action and the retrieval of data from the server, becomes increasingly complex and the asynchronous behavior inherited from the AJAX model makes the organization of the logic difficult. For this reason, several patterns evolved for general-purpose applications were adapted to structure a web application in favor to make the application better organized and more maintenance-friendly.

The Model-View-Controller Pattern

The MVC architectural pattern was first used on server-side mentioned in Freeman (2014), however, can also be exploited to fulfill the Separation of Concerns (SoC) design principle in complex client-side web applications (Microsoft Patterns and Practices Team, 2009, Chapter 2).

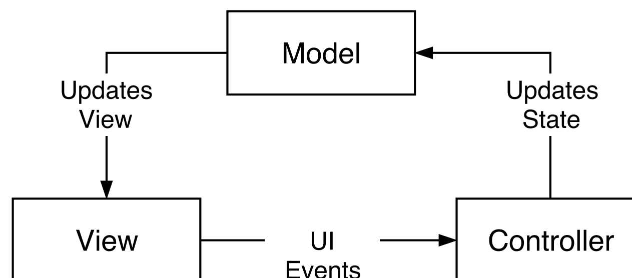

Figure 2.2: Components of the Model-View-Controller Pattern, adapted from Eckstein, Loy, and Wood (1998) and Elliott (2014)

As presented in Figure 2.2, the MVC pattern, introduced in Krasner and Pope (1988), is separated into three main building blocks to decouple the data model from the business and presentation logic. The three components can be described as follows:

- **Model:** All data which is retrieved from the server is bundled and managed by the Model component and encapsulates, thereby, how the data is retrieved from the server (Osmani, 2012, Chapter 11). The Model component is often implemented by an object that reveals also available operations on the data specific to the current domain of the application, denoted in Evans (2014) as Domain Model.
- **View:** The View component is the visual representation of the model and gets updated whenever the state of the model changes. Hereby, the view can also implement the Observer pattern to update itself when the model has changed (Gamma et al., 1995). Furthermore, simple presentation logic resides in the view, for example to change the color of an element in the view.
- **Controller:** The Controller processes all UI events triggered by the View component, for example the click event of a button defined in the view, and updates furthermore the model. An important point is here that the Controller is independent of the used technology for the view presentation, for example in a SPA the click event handler is not allowed to access directly the Document Object Model (DOM).

The Model-View-Presenter and Model-View-ViewModel Pattern

As already mentioned in Section 2.1.1, several architectural patterns were derived from the MVC pattern and are summarized by the MV* or MVW pattern. The two most wide-spread adaptations of the MVC pattern for web applications are the Model-View-Presenter (MVP) and the Model-View-ViewModel (MVVM) pattern.

In the MVP pattern (see Figure 2.3), the Controller component is replaced by a Presenter, which acts as a layer between the model and the view. In opposite to the MVC pattern, where the view is updated directly by the model, the Presenter component incorporates the entire logic to adapt the data of the model for the view and to handle all UI events triggered by the view. Through the simplicity of the View component, which provides an interface to the Presenter, the view is also referred to as a Passive View (Fowler, 2002). By decoupling the view from the model and the relocation of the presentation logic from the view to the Presenter, MVP applications become better testable by unit-tests because the Presenter can be used as a complete mock of the UI (Osmani, 2012). Furthermore, the presentation logic inside the Presenter can be reused across applications independent of the application specific implementation of the view.

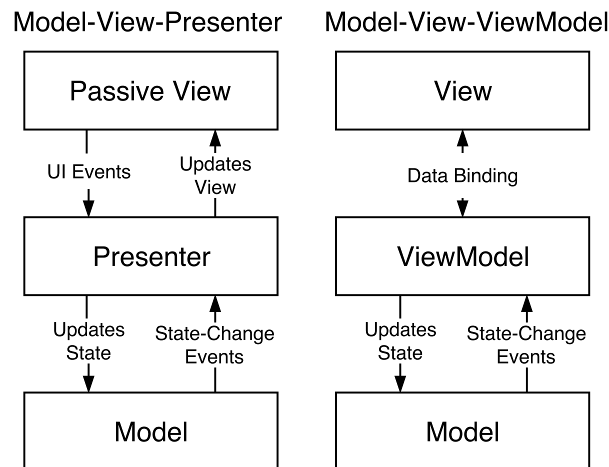

Figure 2.3: Variations of the Model-View-Controller Pattern, adapted from Osmani (2012)

The separation between the view and model implemented in the MVP pattern comes with an additional implementation overhead to mediate between the view and the model and further, to update the Passive View through its interface. The MVVM pattern can be seen as a combination of the MVC and MVP pattern and tries to simplify the development process by replacing the Controller or Presenter with the ViewModel component. Thereby, the ViewModel acts as a subset of the Model component and adds additional functionality to the view for manipulating the model. In contrast to the Passive View of the MVP pattern, the MVVM view knows about the ViewModel and is actively bound to the ViewModel via two-way data binding implemented mostly by the Observer pattern. Through the two-way data binding the ViewModel is even easier to unit-test than the event-driven implementation of the MVP Presenter.

As described in Boduch (2016), web applications based on the architectural MVW pattern can suffer from the following problems:

- **Complexity:** The SoC principle introduced by the MVW pattern can cause unnecessary complexity and can lead to reduced productivity when several components such as the model and the view needs to be modified when a simple feature is added.
- **Scalability:** Web applications following the MVW pattern consist usually of multiple views each visualizing the data stored in models. Furthermore, the application state can be distributed in several models, which are all modified by different instances, for example Controller components. In this way the application can

easily grow as such that dependencies between the MVW building blocks can become hardly traceable and new features are difficult to implement.

- **Cascading Updates:** When a web application grows as highlighted in the above mentioned point Scalability, an user action in a view can update several models at the same time and vice versa, the models can trigger the views to update. This multi-directional data flow can cause cascading updates through several models and Views and lead to an unpredictable data flow.

The Flux Pattern

The Flux architectural pattern tries to overcome the mentioned problems of the MVW pattern by introducing a Dispatcher component that manages the application state changes in form of an Action entity and replaces in this way, the multi-directional with a strict uni-directional data flow (Gackenheimer, 2015, Chapter 5).

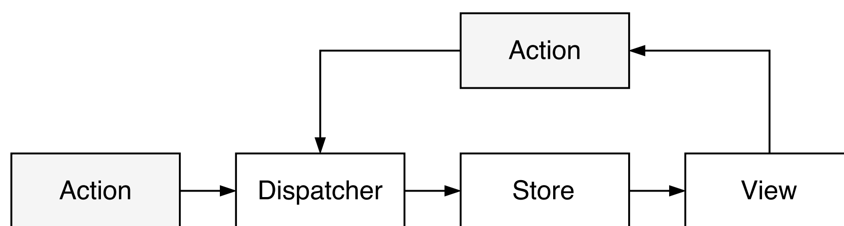

Figure 2.4: Components of the Flux Pattern, adapted from Facebook (2016a)

The following components shown in Figure 2.4 form the Flux pattern (Facebook, 2016a):

Action – Representing a Model Change: Actions are the starting points of the Flux architectural pattern and describe a single functionality of the web application, for example, to add an item to a list displayed by the view. In addition to the type of an Action described by a unique string, an Action holds the new data as payload. An Action can be either triggered by a server-side event, for example during the initialization of the web application, or by an user action in the view.

Dispatcher – Handling any Model Change: The Dispatcher acts as a central hub in the application and thus, every change must go through the Dispatcher via an Action. Every Action is forwarded to a Store by the Dispatcher, for example the content of the new list item is forwarded to a *ListStore* holding the state all list items. Furthermore, the Dispatcher can manage dependencies between Stores, for example when additional content such as an image from another Store must be added first to every list item. In this case, the Dispatcher first requests the image for example from a *ImageStore* holding all images of the application and then forwards the appended content to the *ListStore*.

Store – Holding all models: The Store component is the single place where the current state of the Flux application is persisted and can be compared to the model of the MVW pattern. Every Store registers a callback to the Dispatcher that is executed when an Action is dispatched to the Store.

View – Displays the Model: In the Flux pattern, the view is separated into two components: (i) a simple View component that follows the Web Components standard described in Overson and Strimpel (2015), and (ii) a Controller-View, which acts as a parent of several View components. The Controller-View listens similar to the Presenter in the MVP pattern, for updates in the relevant Store and forces any hierarchical nested View component to update via a defined interface. As mentioned above, an user action emitted in a View component is redirected first to its Controller-View parent and is then forwarded as an Action to the Dispatcher. The View component can be seen as a Passive View of the MVP pattern, which knows nothing about the Domain-Model retrieved from the Store component and can be reused across applications.

By the introduction of the uni-directional data flow and the management of all user actions by one instance the Dispatcher, the mentioned problem of cascading updates in the MVW pattern can be solved by the Flux pattern. Furthermore, new features can be easier implemented through an additional Store, Action, View component, whereby, only the Dispatcher component must be adapted when a component was added.

2.1.3 Features of Single-Page Application Frameworks

As stated in Fain, Rasputnis, and Tartakovsky (2014) and Kuuskeri (2011), especially for large-scale web applications the general limitations of the JavaScript programming language will quickly become obvious and hence, the use of a SPA framework can simplify the development of a reliable and well-tested web application. Furthermore, the use of a

SPA framework reduces the development effort of a RIA, which incorporates one of the presented patterns, such as the MVC pattern (see Section 2.1.2).

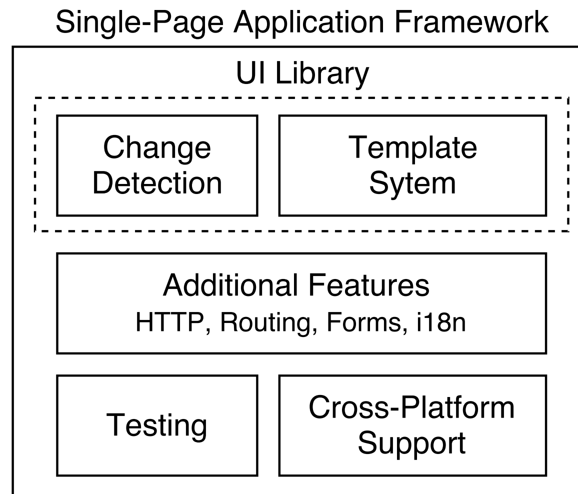

Figure 2.5: Features of a Single-Page Application Framework

In Figure 2.5, the main features of a SPA framework are presented. Several libraries, here termed as UI libraries, exist which only provide the functionality of a SPA framework that is responsible for the View component and consist mainly of the two blocks: (i) the change detection and (ii) the template system.

Change Detection and Data Binding

Whenever the model of a SPA application changes, the view must be updated to reflect the current state of the model. In the context of web applications, the view is represented by a tree of visual elements called the DOM, which is constructed out of multiple HTML files by the rendering engine of the web browser. Because of the tree-like structure of the DOM, for any modification of the view, for example the creation of an additional visual element, in the layout process an entire DOM tree traversal is necessary to take the new element into account. This tree traversal results in an expensive operation in terms of performance (Stefanov, 2010, Chapter 8). For this reason, SPA frameworks try to optimize any DOM update with algorithms grouped under the term change detection (see Section 2.2). Furthermore, several frameworks provide a functionality to bind the view to a subset of a model and vice versa, to update the model based on user actions, described

in Gechev (2016a) as Data Binding, and facilitate in this way the implementation of the MVVM pattern.

Template System and Reusable Components

As described in the above section, several SPA frameworks offer the functionality to let the framework handle the update of the DOM when the model changes and to react when the user interacts with view. This Data Binding functionality is often provided by a Domain-Specific Language (DSL) which exchanges or extends the HTML, that usually describes the view (Gechev, 2016a). Hereby, the DSL utilizes a template engine, where the properties of the model can be accessed in the view definition by a specified syntax, for example by double curly braces (Mustach Team, 2016). In addition, the DSL provides special constructs defined as HTML attributes, to implement simple presentation logic inside the view definition, for example to hide a DOM element depending of the state of a model property or to show multiple DOM elements sourced from a collection in the model.

Furthermore, SPA frameworks give the possibility to structure the view of a web application into reusable components and thus, help to improve the maintainability of every SPA. Moreover, the components often provide the possibility to define custom presentation logic and fulfill parts of the Web Components standard. For an optimized user experience, for example to reduce the initial page load time, the used SPA framework exposes in many cases the framework-specific life-cycle-hooks of the View component. Thereby, a third-party JavaScript library can be loaded asynchronously in the life-cycle-hook that maps to the initialization of the View component, which is the consumer of the library, to reduce the initial page load time.

Cross-Platform Support

In general a SPA is used across multiple platforms or devices, which leads to the requirement to adapt the view of the web application to the current available screen size and its functionality to the platform-specific operating system. Several solutions exist in combination with a SPA framework to reuse the same code base across multiple platforms, that is on desktop and mobile devices. In this context, the term Renderer is used to describe the part of the SPA framework which is responsible to draw the actual view, for example the DOMRenderer uses the DOM API to render the view of the SPA in the web browser (Jbanov and Bosch, 2016; Hors et al., 2000). Through the encapsulation

of the Renderer in a separate component, it is possible to exchange the Renderer by a custom Renderer specific to the current environment.

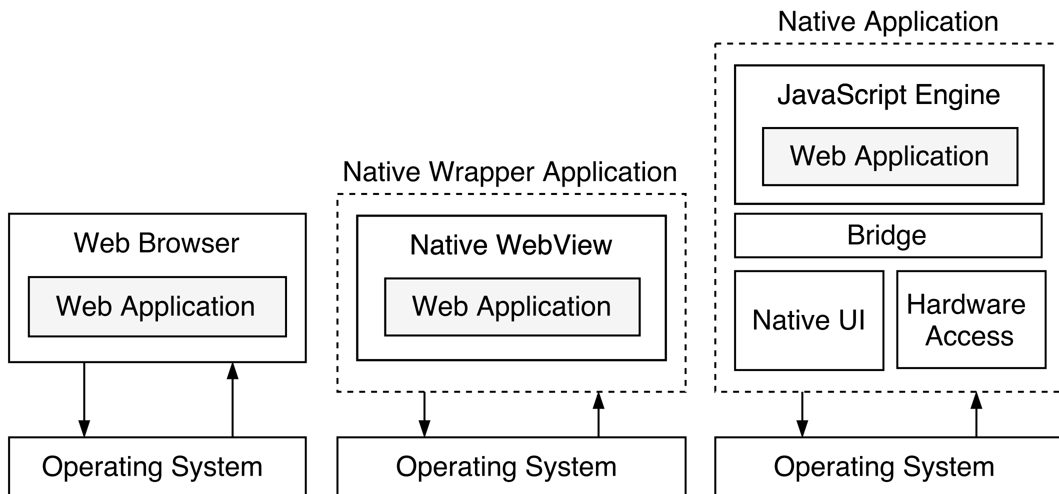

Figure 2.6: Possibilities to run a Web Application on a Mobile Device, adapted from Looper (2015)

In contrast to run a web application inside a web browser (see Figure 2.6), a SPA can be wrapped in a native application by hosting it into a Web View, which is a native control that can load and visualize web content (Looper, 2015). For a performance compared to native applications, SPA frameworks offer a bridge to access the native UI and hardware of the device. This bridge depicted in Figure 2.6 acts as a layer between the JavaScript engine, where the actual logic of SPA is interpreted, and the device-specific API. Furthermore, it offers a custom implementation of the Renderer to draw the view of the SPA by native UI components. In this way, the web application appears in terms of performance and styling similar to a native application, whereby the same code base is used as when the application is hosted in the web browser. Furthermore, the full hardware can be accessed by the web application, such as the camera or the compass of the device.

Testing and Debugging

The JavaScript language is interpreted and compiled through Just-In-Time (JIT) compilation during runtime and hence, no possibility exists to detect potential errors before

the deployment of the web application. Furthermore, a state-of-art web application incorporates a large fraction of open-source JavaScript packages, which can vary widely between their versions and can compromise the functionality of the web application. For this reasons, it is import that the functionality of a SPA is completely covered by tests and all tests are run on a regular basis, for example after new features were added or packages were updated.

Many SPA frameworks support this established practice, known also as Test-Driven Development described in Fain, Rasputnis, and Tartakovsky (2014, Chapter 7), and offer framework-specific test utilities to facilitate the testing of the SPA. Every test setup usually consists of the following parts:

- **Test Framework:** With a test framework, tests can be written and combined in a test suite. Following the Behaviour-Driven Development process defined in (Craven, 2016a) and (Craven, 2016b), the tests are written as user stories in a business-specific language.
- **Test Utility:** Especially for Unit-Testing, SPA frameworks usually offer helper functions packed in a test utility to initialize framework-specific components in a test environment outside of the web browser.
- **Test Runner:** A test runner allows to repeat a test suite defined in the test framework multiple times across several web browsers. Furthermore, it can be used to automate the test execution, for example whenever a file changes, and to visualize the test results via the command line.
- **Assertion Library**: An assertion library enables the test framework to check if the test result equals the specified value in the test. Thereby, the developer can choose between several assertion syntaxes, such as `value.should.be`, `expect(value).to.be`, or similar as in the C language: `assert(testValue == value)`.
- **Mocking Framework:** When testing a component independent of the web application, all possible dependencies of the component need to be brought into a specific state. Furthermore, some dependencies might not be available in the test environment, such as the communication to the server through HTTP. For this reasons, a mocking framework provides methods to replace specific functions or a complete object, namely Stubs or Mocks, of the web application when testing the component (Fowler, 2007). Furthermore, the execution of a specific function can be observed by using a Spy, for example the applied arguments or the number of calls of the function.
- **Web Browser Automation:** A common requirement is to test a web application in parallel across multiple browsers. To ease the life of a developer and to have consistent test cases especially for Integration-Testing, a solution exists to control

web browsers through an WebDriver API. Thereby, the WebDriver API abstracts the diversity of the the built-in automation support of every web browser.

Additionally, most of the SPA frameworks come with a browser extension for the Google Chrome and Firefox web browser to debug the running web application.

Additional Features

SPA frameworks usually include libraries, which cover frequently needed functionality for example a wrapper of the XMLHTTPRequest object, which is supported in every web browser, for HTTP requests or for internationalization (i18n) to translate texts on the client-side depending on the current environment. Furthermore, SPA frameworks support the navigation through the web application denoted as client-side routing to mimic the behavior of multiple views known from the classical web application model.

2.1.4 Overview of Single-Page Application Frameworks and User Interface Libraries

Backbone Framework

The Backbone framework was released 2010 and is one of the first JavaScript frameworks for developing a SPA. Developed by Jeremy Ashkenas, the framework is preferred against other SPA frameworks because of its small file size, which enables a fast initial page load time (Fender and Young, 2015). The Backbone framework provides several framework-specific JavaScript objects that can be extended via the prototype-inheritance in JavaScript. The combination of the extended objects forms the SPA using the MVP pattern (see Figure 2.7 and Section 2.1.2). The following list gives an overview of all Backbone objects:

- **Backbone.Model:** The Backbone.Model object is used to store the state of the application and represents the Model component of the MVP pattern. Furthermore, data-related logic, such as data validation or conversion, can be implemented in the Backbone.Model object.
- **Backbone.Collection:** The Backbone.Model object can be used to group several Backbone.Models and to host logic, which can be applied on the collection of models.

- **Backbone.View:** The Backbone.View object simplifies the application development by splitting the DOM into multiple logical blocks (De, 2014). In the Backbone.View the major part of the application logic resides, such as the binding between the Backbone.Model or Backbone.Collection and the DOM. The Backbone.View can be seen as the Presenter component of the MVP pattern.
- **Backbone.Router:** The Backbone.Router enables client-side routing as known from the classical web application model and maintains the application state when the URL changes in the web browser.

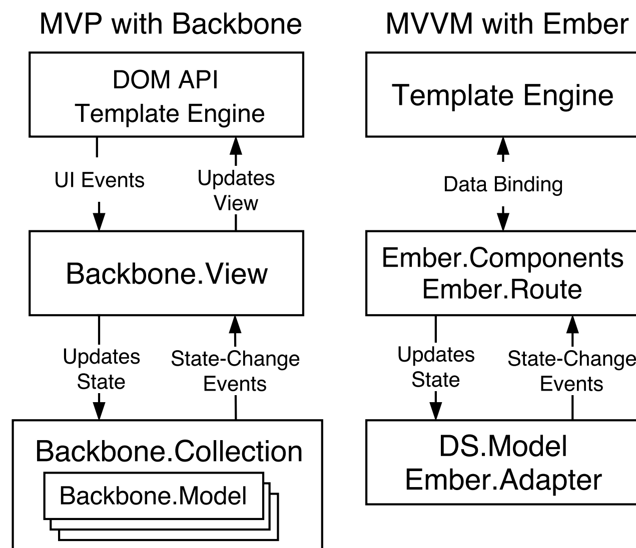

Figure 2.7: The MVP pattern with the Backbone and the MVVM pattern with the Ember Framework

As described in more detail in Section 2.2.1, the Backbone framework provides an event system to notify the Backbone.View when the model stored in the Backbone.Model has changed. The developer itself is responsible to implement how the view gets updated when the model changes. The actual View component in the MVP pattern is left open by the Backbone framework, which means that the developer can choose how to render the current state of the Backbone.Model or Backbone.Collection object to the DOM, for example with a template engine or directly via the DOM API provided by every web browser.

Ember Framework

The Ember Framework origins from SproutCore developed by Yehuda Katz and Tom Dale and was first released in 2011. The functionality is based upon the Convention Over Configuration paradigm and utilizes the navigation functionality of the web browser, that is the URL (Kelonye, 2014). Similar as in the Backbone framework, Ember provides specific JavaScript objects, which can be extended to form the SPA. The main building blocks of the Ember framework can be summarized as follows:

- **Ember.Router:** The Ember.Router objects can be used to map a specific URL to a route handler implemented in a Ember.Route object. Whenever the web browser navigates to this URL, the configured route handler is called.
- **Ember.Route:** In the route handler implemented in a Ember.Route object, the ViewModel of the MVVM pattern can be implemented. The main application logic is implemented in the route handler. The data binding between models and the view is configured in the route handler and facilitated by a template engine. Hereby, the Ember framework uses the template engine to update only parts of the DOM when the model defined in the route handler changes.
- **Ember.Components:** Ember.Components fulfill the Custom Elements section of the Web Components standard and thus, accomplish to reuse presentation logic across Ember applications. Every Ember.Components consists of a custom template and presentation logic. Through the assignment of a custom tag, a Ember.Components can then be reused by its tag in multiple template files similar as every HTML tag defined by the W3C.
- **DS.Model:** All application data is stored in DS.Model objects from the Ember framework. After extending the DS.Model object, it can be accessed by every Ember.Route object via an global store object of the Ember framework. Ember provides basic functionality to retrieve data via HTTP from a server through an Ember.Adapter object using the adapter pattern and to cache the response of previous HTTP operations. The default HTTP protocol can be easily replaced via a custom implementation of the Ember.Adapter.

As can be seen in Figure 2.7, the Ember framework enforces the MVVM pattern through the data binding between the Model component and the View component (see also Section 2.2.2). In contrast to the Backbone framework, were the developer can choose how to update or describe the View component, Ember only supports the HTMLBars template engine used for the data binding between ViewModel and View component. To manage dependencies in an Ember application, the framework includes a simple dependency injection container which enforces the Interface Injection pattern described Section 2.3.2.

React Library

The React library was initially developed by the Facebook Inc. to implement parts of the in-house web page, such as the newsfeed or advertisement and was released open-source in 2013 (Gackenheimer, 2015). In contrast to the Backbone or Ember framework, the React library can only be seen as an UI library as presented in Section 2.5 and has a single goal to structure complex UIs in web applications by reusable components. Although React does not assume which architectural pattern the underlying SPA uses, it was initially designed to implement the View component in the Flux pattern. For this reason, most of the single-page applications based on the React library assemble also other libraries developed by the same company, for example the Flux library as a foundation to implement the Flux pattern or the ReactTestUtils library for the efficient testing of React SPAs.

Appart from the various of libraries to add functionality of a React empowered application, the React library consists mainly of two parts:

- **React.Component:** Similar as the Ember.Component, the React library provides a possibility to reuse presentation logic and small parts of the view definition across the web application. Hereby, every React.Component has an own template, which defines a part of the view. Through custom properties, the state of the React.Component can be set from outside similar as for every HTML tag. For the definition of the template, the React library extends the JavaScript language with a syntax based on XML and accomplishes, in this way, that the view part can be defined directly in the same file as the JavaScript code for the presentation logic.
- **ReactDOM:** The ReactDOM is the React specific implementation of the Virtual DOM described in Section 2.2.4. Whenever the state of an implementation of a React.Component changes, the Virtual DOM starts to render the actual state of the component to the DOM.

AngularJS and Angular 2 Framework

The AngularJS framework is developed by Google Inc. and is similar as Ember, a framework for the development of single-page applications. Released in 2010, the AngularJS tries to ease and accelerate the development of SPAs, which use one of the MVC-based patterns summarized by the MVW. The developers of AngularJS chose the approach to extend the HTML as the view definition of the SPA by framework-specific DSL denoted as directives, contrary to React where the JavaScript is extended by a XML-like syntax for the view definition. The main building blocks of an AngularJS SPA can be listed as follows:

- **Directive:** As stated above, an AngularJS Directive can be used to extend the functionality of HTML, for example to show or hide elements depending on a model property, or to encapsulate presentation logic similar as in React.Components. For this reasons, AngularJS allows to define a directive with a unique tag and its own HTML template file, that can be inserted in other templates across the application.
- **Controller:** The AngularJS controller origins from the Controller component in the MVC pattern and can be used to implement business logic for the actual view.
- **Service:** In contrast to the controller, an AngularJS Service is a singleton class used to implement business logic independent of the actual view and maps to the Model component of the MVW pattern, more specifically to a Domain Model. The logic in the Service can be reused in several controllers and provides often data relevant operations, for example data access from the server or data validation.

A major advantage that brings the AngularJS framework, is the possibility to bundle all relevant blocks, for example several Directives and Service implementations into one module. In this way, a well-designed and encapsulated AngularJS module can be reused across several SPA applications. In addition, the AngularJS offers a dependency injection container described in Section 2.3.3 to handle the initialization and injection of the singleton Services. Furthermore, a test utility is provided by the AngularJS to facilitate the testing of AngularJS specific components in a test environment. Similar as in the Ember framework, a router module enables to mimic the separation of the SPA into multiple pages known from the classical web application model.

In 2016, with Angular 2 a second version of the AngularJS framework was released. Although Angular 2 is based on the same principles of its predecessor, the main API of the building blocks changed without backwards-compatibility. Hence, any existing AngularJS code cannot be integrated in an Angular 2 and vice versa without major changes. Angular 2 was developed in TypeScript and suggests to align to its language choice for a full benefit of all features which comes with the TypeScript language, such as static-typing or metadata reflection. In addition to an improved dependency injection and change detection, Angular 2 merged the functionality of AngularJS directives and controllers in an Angular 2 Component. This Angular 2 Component follows the Web Components standard and allows to encapsulate (i) the view definition, (ii) the styling and (iii) the presentation logic of a subset of the DOM from the rest of the SPA.

2.2 Change Detection in Single-Page Applications

As already mentioned in Section 2.1.3, the update of the view, that is the DOM, to represent the actual state of the model is the most expensive operation in terms of performance for a SPA. For this reason, SPA frameworks implement algorithms to minimize the number of DOM operations or even, to let the developer specifically control when the DOM gets updated. The algorithms to optimize the DOM manipulations, described in Parviainen (2015), have evolved over time and will be presented in the following sections.

2.2.1 Manual Re-Rendering of the Document Object Model

The first frameworks for the development of SPA, for example the Backbone framework, introduced to move the model of the application from the server to the client and thus, simplified the implementation of the MVW pattern on the client-side for more responsive web applications (see Section 2.1.1). Hereby, framework-specific model objects can be used to track changes in the model by subscribing event-handlers to the change event of the model following the Publish-Subscriber pattern (Hohpe and Woolf, 2003, Chapter 4). In the Backbone framework, this functionality is maintained in every model instance through a list of event-handlers subscribed on the change-event of the model. Whenever the model changes, this list is used to notify all event-handlers. Although the framework notifies every event-listener when the model has changed, the actual manipulation of the DOM, for example to find and update the DOM element which displays the model value, needs to be implemented and thus can be highly optimized by the developer. The optimization done by the developer is depicted in Figure 2.8 with a dotted line.

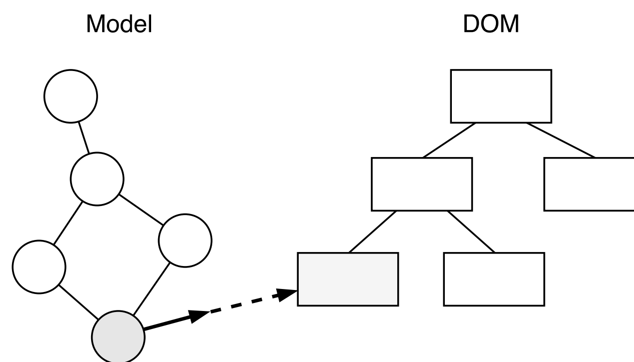

Figure 2.8: Manual Re-rendering and Data Binding, adapted from Parviainen (2015)

The manual re-rendering approach could be mapped to the MVP pattern described in Section 2.1.2: the Model component is provided by the framework itself, whereby, all logic for the mapping between the model and the view needs to be implemented in the Presenter component by the developer. The view acts as a Passive View and is entirely controlled by the Presenter via its interface, that is the DOM API.

2.2.2 Data Binding between the Model and the View

As soon as features are added to a SPA and the number of elements in the view of the application increases, the logic for the binding between the model and the view in the manual re-rendering approach causes a growing complexity of the code base. In order to ensure a clean and maintainable web application, successors of the first SPA frameworks tried to abstract the DOM manipulation by a data binding mechanism. This data binding mechanism is achieved when the model is stored in framework-specific data objects, for example in a DS.Model object in the Ember framework (see Section 2.1.4). Then, the framework itself manages the update of the DOM when a model property changes. The specification of the DOM element which needs to be updated, or, in other words, the binding between the DOM element and the model is described in a markup file inspired by HTML and enabled through template engines, such as Handlebars or HTMLBars. The dotted line in Figure 2.8 is, when using for example the Ember framework, implemented entirely by the framework itself and thus web applications which use the data binding mechanism follow the MVVM pattern (see Section 2.1.2).

As stated in Parviainen (2015), a drawback of this approach is, that in order to let the framework handle the DOM update, framework-specific objects need to be used to describe the models in the application. Hence, all external data received for example from the server needs to be filled first in a framework-specific model object, that is the DS.Model object in the Ember framework, to enable the data binding.

2.2.3 Dirty Checking and JavaScript Zones

Similar as in the Data Binding approach, the AngularJS SPA framework tries to solve the increased complexity of the manual re-rendering and forwards again the responsibility of the actual DOM manipulation to the SPA framework. In contrast to the Ember framework, where the view and the data binding is described in a custom markup language format, the AngularJS framework uses a HTML file extended by a framework-specific syntax denoted as template, to specify the view and its binding to the model. The DSL of every

template is compiled by the AngularJS framework at startup to a template function, which can be used later to update the DOM when the model changes.

Another advantage of the AngularJS framework is, that every object can act as a model and can be bound to the view. This fact is archived by creating a watcher function for every model, which compares the current model with its previous state and updates the DOM element through the generated template function when a change was detected (see Figure 2.9). The process to compare the previous and new state of the model implemented in the watcher function is denoted also as Dirty Checking. Because of the fact that every JavaScript object can be used as a model, no Publish-Subscriber pattern can be implemented as in the Backbone framework. For this reason, the Dirty Checking of all watcher functions, denoted as digest loop, must be started whenever a event happens which could alter the model state, for example a click-event in the DOM (Gechev, 2016a, Chapter 2).

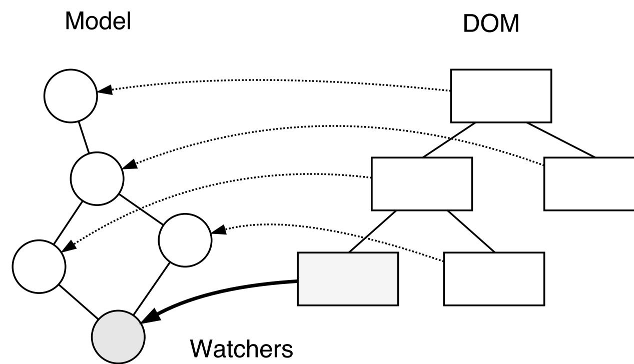

Figure 2.9: Dirty Checking of the DOM, adapted from Parviainen (2015)

A drawback, which comes with the generic object that can be used for the Model component, is that it can be hard for the AngularJS framework to detect if an event outside of the framework, for example an HTTP response, could have modified a model. Although AngularJS provides wrappers, for example for the HTTP functionality to detect whenever a response arrived, and a method to explicitly trigger the digest loop, it can become a tedious task for a complex web application to detect all possible cases where a model could change.

For this reason, the evolution of the AngularJS framework Angular 2 reworked its change detection system. As stated in Archibald (2015), every window of the web browser with the same origin shares the event loop of the JavaScript engine. In this event loop, all

asynchronous operations are, depending of its type or priority, queued either as tasks or microtasks and executed sequentially by the JavaScript engine. Through the sequential processing in the event loop and the asynchronous behavior of the task, the execution context, such as the state of the variables where the task was initially triggered, is lost when the task is executed. The Zone.js library, described in Pisman (2016), adds this missing execution context for asynchronous operations, such as HTTP requests, and allows to persist the context of the function in which the HTTP request was initially triggered inside a Zone object. The corresponding function which usually triggers the asynchronous operation is wrapped hereby by a custom Zone proxy, which holds a reference to the Zone object and thus, the execution context of the function where the asynchronous operation was initiated. The Angular 2 framework uses the concept of persisting the execution context and provides an implementation of the Zone object. This custom Angular 2 Zone is spawn over the whole web application and facilitates to detect whenever a asynchronous operation was completed in the event loop of the window. In this way, the change detection system of Angular 2 can detect whenever an asynchronous operation has completed and could have altered the model state.

2.2.4 Modeling the Document Object Model

Another approach to detect changes in the model and to optimize DOM operations is to maintain an in-memory copy of the entire DOM denoted as Virtual DOM. The Virtual DOM acts hereby as the View component as for example in the MVW pattern, however, can be used independently of the underlying pattern of the web application. An implementation of a Virtual DOM is currently the main part of the React library and is often used to realize the Flux pattern.

The main idea behind the in-memory representation of the DOM by a JavaScript data structure is to reduce the number of DOM manipulations and merge several DOM accesses in one operation, that is a patch. Whenever a model changes the complete view is constructed again by a Virtual DOM. As shown in Figure 2.10, an algorithm computes the difference of the new Virtual DOM at time T_2 to the old representation of the DOM at time T_1 and updates only the changed elements in the DOM in a single patch. The algorithm to compute the difference of the new Virtual DOM and the old Virtual DOM, denoted in React (2016) as Diffing Algorithm or more general as Reconcilation, has a complexity in order of $O(n^3)$ (Bille, 2005), where n stands for the number of DOM elements. In a state-of-the-art web application a DOM tree can consist of up to 10,000 DOM elements which would take the Diffing Algorithm about 17 minutes when computing the difference of both Virtual DOM trees on a 1GHz processor. For this

reason, several optimizations and assumptions were made in the Diffing Algorithm of the React library to reduce at the end the complexity to an order of $O(n)$.

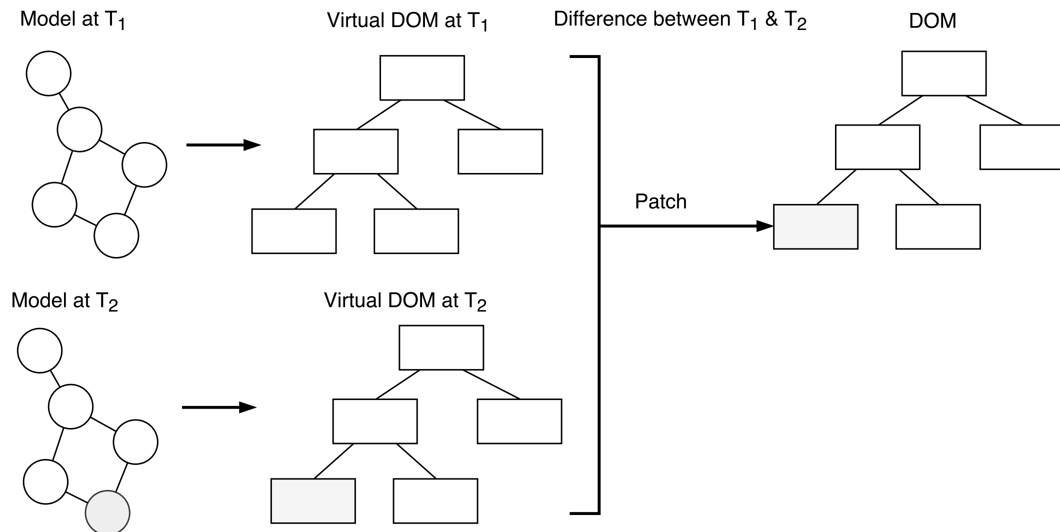

Figure 2.10: Virtual DOM as a Model of the DOM, adapted from Parviainen (2015)

Restrict Diffing Algorithm to the same Tree Level

In a web application, it can rarely happen that a element of the Virtual DOM is moved during the reconstruction of the Virtual DOM to a complete different tree level. As mentioned in Chedeau (2013), the only possibility that an element is moved across the tree is a drag-and-drop action, which is an uncommon behavior in most web applications.

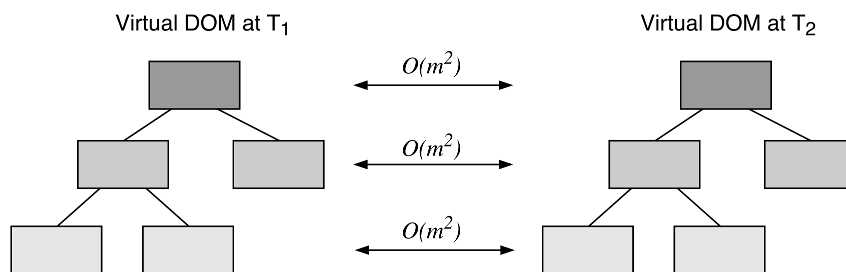

Figure 2.11: Comparing each Tree Level of the Virtual DOM, adapted from Chedeau (2013)

Hence, the Diffing Algorithm can be restricted to compare only elements which are on the same tree level counted in Figure 2.11 by the variable m. Through this assumption, when summing up the effort to compute the differences in every tree-level, that is $O(m^2)$, the complexity can be decreased to an order of $O(n^2)$.

Label Tree Elements with unique Identifiers

A problem arises when for example a new element is inserted in the middle of a list of elements. As depicted in Figure 2.12, the Diffing Algorithm can not find out where to insert the new element when it compares both Virtual DOM levels by each other. To circumvent this problem, the Virtual DOM implementation in the React library assigns a key property to every element in the Virtual DOM and makes in this way a list comparable between two Virtual DOM representations (Chedeau, 2014). Through this method, the complexity of the Diffing Algorithm could be reduced further to a linear dependency on the number of elements in the tree (React, 2016).

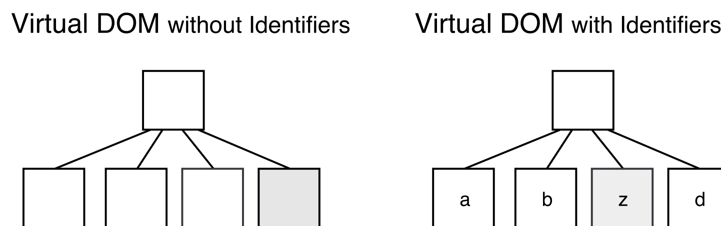

Figure 2.12: Unique Identifiers for each DOM Element, adapted from Chedeau (2013)

As described in Section 2.1.4, a web application built with React is usually a composition of multiple React components, which are again a conglomerate of DOM elements. The React Library further optimizes its Diffing algorithm with the assumption that when a React component has been replaced by another component the algorithm will not compare both components and re-render the whole new React component with its subtree of elements.

Selective Subtree Rendering

In the React library, every React component provides a method to the rest of the application to set the current state of the component, which will induce the React library

to mark this component as dirty. At the end of the JavaScript event loop, described in Daggett (2013, Chapter 5), the React library will apply the Diffing Algorithm on all components that were previously set to dirty and reduces in this way, the number of performance costly DOM manipulations by merging all operations in a single patch. When a component is set to dirty, React will perform its Diffing Algorithm on the component and to its children forming a subtree. However, React introduced a possibility to control if the whole subtree shall be compared to the previous version of the subtree with another method in the React component. Inside this method, the new state of the component set by the application is passed to the method and can be evaluated by a custom logic. Depending of the boolean result of this logic, the subtree will be compared by the Diffing Algorithm or not. Through this approach, the web application performance can be optimized further by the developer, whereby it must be considered that the execution of the custom logic for the evaluation shall take less time as the update of the whole subtree would have taken.

2.2.5 Immutable Objects for performant Change Detection

As mentioned in the above sections, the change detection system in a SPA framework often includes the task to compare JavaScript objects and to detect in this way if the model in the application has changed and hence, the view must be updated. Especially, when comparing deep object structures, for example in the Dirty Checking approach, the comparison between JavaScript objects can become a performance bottleneck in the change detection system and can degrade the responsiveness of the overall web application.

The change detection system as the Dirty Checking in the AngularJS and Angular 2 framework or the Diffing Algorithm in the React library can be improved when using immutable objects to hold the state of the application, for example in the model in a MVW pattern or in the Store component using the Flux pattern. In contrast to a regular JavaScript object, an immutable object cannot be changed after it is created and whenever the object is mutated a new object needs to be initiated with a new reference pointing to the allocated memory. Because of this fact, for example the implementation of the logic if a subtree should be re-rendered described in Section 2.2.4 can be highly optimized when only the objects references need to be compared to detect if the data has changed. Furthermore, the Dirty Checking approach in the Angular framework can be improved by avoiding deep comparison between current and previous state of the component by using immutable data structures.

Although, it seems that using immutable data structure seem will bring a huge drawback in terms of memory consumption, structural sharing allows to reuse parts of the memory, which can be implemented through vector trie or hash mapped tries (Bagwell, 2001). As the current JavaScript standard does not natively support immutable data structures, JavaScript libraries, such as Immutable.js, exist to use immutable objects to store the application state. The Immutable.js library originated from the React library and uses already vector tries to make arrays and hash maps tries to make sets or maps more memory efficient (Facebook, 2016b).

2.3 Dependency Injection in Single-Page Application Frameworks

2.3.1 The Dependency Inversion Principle

As stated in Jansen (2016), the growing complexity of web applications based on JavaScript will make it necessary to consider software architecture design patterns and principles that origins from Object-Oriented Programming (OOP) languages. Five well-known principles for OOP languages often abbreviated in the term SOLID and introduced by Martin (2000) can lead to a more maintainable and extendable application. The statement of each principle can be summarized as follows:

- **Single Responsibility Principle:** Every class is responsible of only a single functionality decoupled from other requirements of the application.
- **Open Closed Principle:** The functionality of every module can be easily extended through additional features without any modification of the existing source code.
- **Liskov Substitution Principle:** A class derived from a base class can be replaced by any other class derived from the same base class without impairing the functionality of a possible user of the base class.
- **Interface Segregation Principle:** Specific interfaces are preferred over one general purpose interface to avoid side effects between interface realizations.
- **Dependency Inversion Principle (DIP):** Modules, which hold the business logic, are decoupled from low level modules, which for example manage I/O operations.

Since JavaScript supports prototype-based inheritance and the latest JavaScript standard ES6 even simplified the prototype syntax with the introduction of the class keyword known from other OOP languages, all five principles can be applied when developing web applications with JavaScript. The term module in the Open Closed Principle can be

mapped to the latest specified JavaScript ES6 module, which is not yet supported natively by all modern web browsers, however, its functionality can be emulated when using one of the JavaScript module formats currently in use, such as CommonJS (Rauschmayer, 2016, Chapter 16). When using the TypeScript transpiler which adds static typing to the JavaScript language and compiles to ES5 JavaScript, similar interfaces can be declared for the Interface Segregation Principle as known from other OOP languages such as C#.

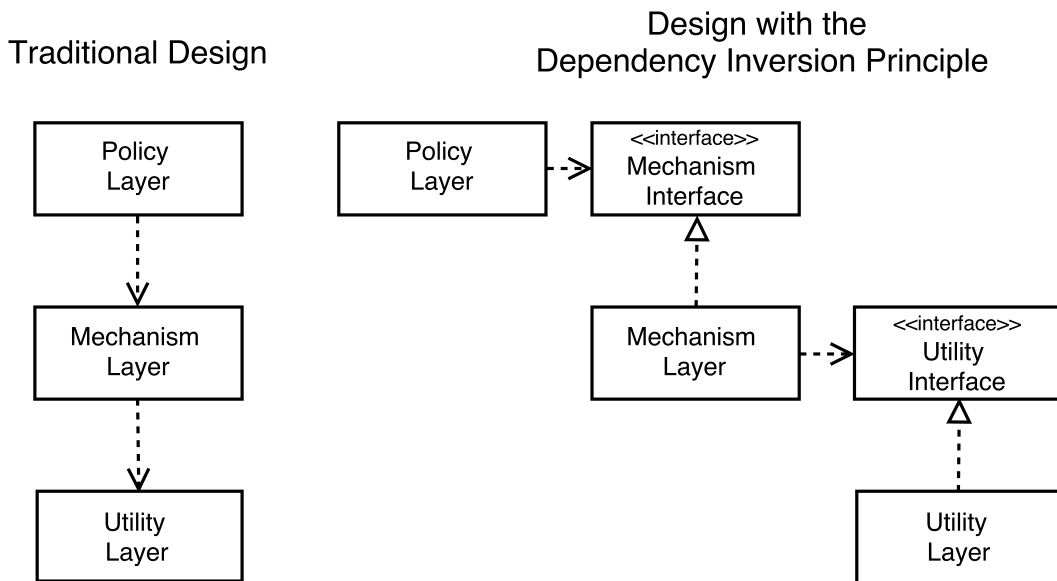

Figure 2.13: Dependency Inversion Principle, adapted from Martin (1997)

The major part of functionality of a state-of-the-art web application is provided by third-party libraries, which are often deployed via a remote repository and developed under an open-source license. Although the open-source license model brings the advantage that the JavaScript libraries are maintained by a large distributed community, it comes with the disadvantage that their functionality and exposed API can vary highly during the development time. Thus, the architecture of web applications needs to take the rapid change of the third-party libraries into consideration to guarantee a stable functionality of the web application.

Especially, the DIP is important to decouple the business logic of the web application from third-party libraries. As shown in Figure 2.13, the left structure shows the traditional design approach without using the Dependency Inversion Principle, where each layer

provides a set of functionality through a defined interface (Martin, 1997). A problem when using this structure arises when for example a class in the Utility Layer is modified, which causes that every dependent class in the Mechanism Layer must be adapted. Through the inversion of the dependencies by interfaces defined by the layer which actually uses this interface (see right structure in Figure 2.13), a class in a lower level can be altered without a change in the business logic in higher level module. Because of the fact, that a third-party library used here in the Utility Layer cannot be modified to implement the interface dictated by the Mechanism Layer, Martin (1997) proposed a further abstraction. Through the introduction of an adapter described in Gamma et al. (1995, Chapter 4), the interface can be implemented by an intermediate layer between the third-party library and the Utility Layer and further decouples any changes in the third-party library from the web application.

2.3.2 Dependency Injection and Inversion of Control

In addition to the DIP, a web application can be further stabilized through Dependency Injection, which separates the initialization of dependencies from the actual usage of the dependencies. In this way, the class itself is no more responsible to initialize all its dependencies and informs instead a global authority about its needed dependencies (Martin, 2008, Chapter 11). As stated in Fowler (2004), several methods exist how the global authority often denoted as dependency container injects the dependency into the target class. The most commonly used methods are summarized as follows:

- **Constructor Injection:** All dependencies described through interfaces are listed as parameters of the constructor in the target class, which can then be stored for any later usage into class fields. In the dependency container, a configuration specifies which implementation of the interface listed as dependency should be injected into the constructor.
- **Setter Injection:** Instead of a constructor, the class exposes the dependency in form of a setter. This setter can be used by the dependency container to inject the dependency into the class, similar as for the Constructor Injection.
- **Interface Injection:** The constructor and setter in the above injection methods are further abstracted by a method defined in an interface for each dependency. When the class needs that dependency, it implements the corresponding interface with the specified method, which is then called by dependency container with the dependency as a parameter.

All three methods have in common that the class has forwarded the actual initialization of the dependencies to the dependency container. The principle to move the initialization of

the dependency from the class to the dependency container is also denoted as Inversion of Control (IoC). Because of the fact that the dependency container or IoC container has the single task to initialize and inject all dependencies, the IoC principle can be see also as an implementation of the Single Responsibility Principle (Schuchert, 2013).

For web applications based on JavaScript, libraries exists that provide an IoC container and often SPA frameworks support dependency injection. When using the TypeScript language, decorators known from OOP languages can be used to add metadata to a JavaScript class and mark in this way dependencies of a class that need to be injected by the dependency container. In future, the use of transpilers such as TypeScript can become no longer necessary when the Metdata Relection API, that include decorators and was proposed by the latest ES7 JavaScript standard, gets implemented by all web browser manufacturers. Furthermore, the first implementations of IoC containers such as in the AngularJS framework, facilitates strings to identify each dependency in the configuration of the dependency container. Especially for large-scale applications, string identifiers can lead to naming conflicts between dependencies. For this reason, modern IoC containers, such as InservifyJS, support to identify each dependency by a Symbol, which is an unique and immutable data type introduced with the ES6 JavaScript standard and currently supported by all web browsers (Jansen, 2016; Knol, 2013).

2.3.3 Dependency Injection in the Angular 2 Framework

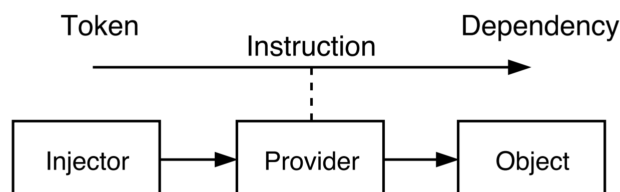

Figure 2.14: Dependency Injection in the Angular 2 Framework, adapted from Precht (2016)

As a main feature of the rapid developing of a robust SPA, the Angular 2 framework improved the dependency injection functionality implemented already in its predecessor AngularJS. The dependency injection container uses hereby the Constructor Injection method to inject dependencies in other Angular 2 Components or Services. Therefore, the following building blocks form the Angular 2 dependency injection:

- **Injector:** The injector is provided by the Angular 2 framework and exposes an API to initialize and inject dependencies.

- **Provider:** As presented in Figure 2.14, a provider is the mapping between the injector and the actual object that is initialized by the injector and describes how the dependency should be initialized and injected as an object to the target class. The provider uses hereby a certain token to identify each object when it is initialized.
- **Dependency:** In the Angular 2 framework, a dependency is an Angular 2 Service holding the main business logic in the SPA.

As stated above, several configuration options exists in the provider for the dependency injection. When several Angular 2 Services exists in an application which implement the same interface, through the provider it can be defined which Service is actually initialized and injected (Precht, 2016). Furthermore, the Angular 2 IoC container is implemented as a hierarchical dependency injection system, which means that every Angular 2 Component modeling the view in form of a tree, has its own IoC. In this way, a specific Angular 2 Service can be injected in the whole application or only in one specific Angular 2 Component. In addition, it can be defined if the dependency shall be created only once, described in the term singleton scope, or if for each target class a new instance shall be created, denoted as transient scope. Similar as in the IoC container of the InverisfyJS library, in the Angular 2 provider several Services which implement the same interface can be mapped by a single provider, denoted as multi-provider. The Angular 2 Component that needs the configured Service as a dependency choose itself which implementation of the interface should be injected. This method is denoted also as contextual binding (Jansen, 2016).

A major advantage of the dependency injection in the Angular 2 framework is the increased decoupling between the Services and Components and the simplified testing. Because the initialization of the dependency or more specifically the Angular 2 Service is moved to the injector, any Service can be replaced easily by a mock-service when testing the functionality of another Component or Service without the need to adapt the Component or Service under test.

3 Infrastructure for the Web Visualization

3.1 Structure of a Web Window

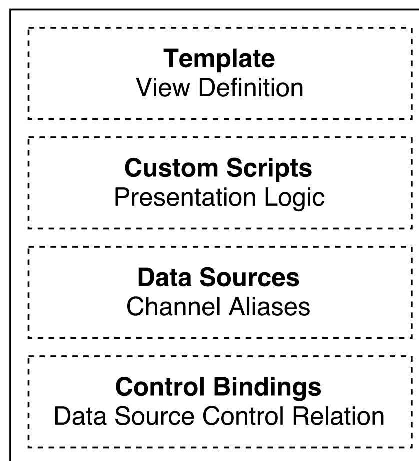

Figure 3.1: Structure of a Web Window

Similar to the visualization in the POI, the web visualization is structured into multiple windows, whereby each window consists of several controls showing the current testbed status and contains all information needed for its visualization. As a consequence, the deployment of a Web Window is simplified by one HTML file. To separate between the information bundled inside the file, a Web Window is divided into four main blocks identified by an explaining HTML tag, for example `<data-sources>` for the Data Sources block (see Figure 3.1).

3.1.1 Template

The Template block describes the view of the Web Window and is rendered in the web browser in the same manner as any HTML file. All controls from the visualization library (see Chapter 4) can be included with their custom tags, for example `<avl-alpha-num-control>` for an alphanumeric control, inside the Template block. An example of a Template block with three alphanumeric controls from the visualization library and its rendered visualization in the web browser is shown in Figure 3.2.

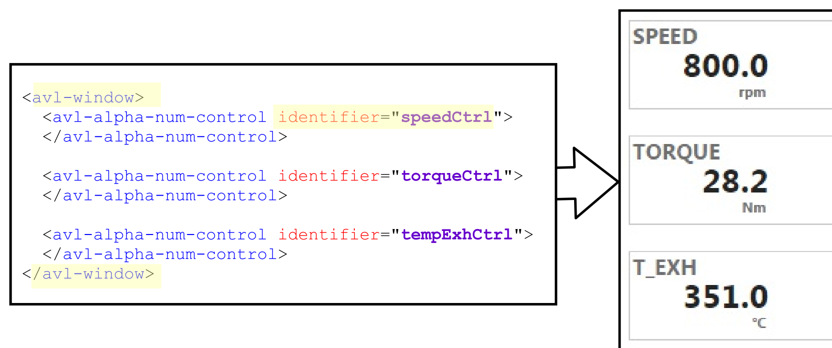

Figure 3.2: Example of a Template Block with three alphanumeric Controls

When a data source obtained from the ASAccessServer shall be visualized by a control of the visualization library, or that is to say, the infrastructure, as described in the following sections, manages the binding between the control and one or more data source(s), two conditions must be met in the Template block: (i) the control needs to be enclosed by a `<avl-window>` tag, and (ii) an unique identifier must be specified for the control. In Figure 3.2, for the first alphanumeric control the identifier `speedCtrl` was defined, which is later accessible inside the infrastructure by this identifier.

HTML-compliant Template

The use of the Angular 2 framework provides an extra set of Angular-specific expressions for dynamic template compilation (see Section 2.1.4 on page 25). A downside, which comes with the additional expressions of the Angular 2 template syntax, is that the template is not HTML compliant anymore before the Angular app is bootstrapped and Angular compiles the template in valid HTML (see Eisenberg (2016) and Group (2016)). This fact causes problems when the content of the Web Window is parsed from the

Extensible Markup Language (XML)-Parser in the ASAccessServer for persisting any changes of the Web Window content (see Section 3.4.2).

To benefit from the Angular 2 expression inside the template block and ensure that the Template block can be parsed from the XML-Parser, the following points must be observed when defining the Template block:

- The entire content of the Web Window, must be enclosed by a root node, for example a `<div>` element.
- Every HTML tag must be closed, either with an explicit closing tag, for example `<div>` and `</div>`, or with a self-closing tag, for example `<input/>`.
- Custom HTML tags must have and separate closing tag, for example `<avl-alpha-num-control></avl-alpha-num-control>`.
- The Angular property binding expression `[propertyName]` must be defined in the bind-syntax: `bind-propertyName`.
- The Angular event binding expression `(eventName)` must be defined in the on-syntax: `on-eventName`
- The hash symbol for an Angular template reference variable `#variable` must be replaced by `ref-`: `ref-variable`.
- Every Angular template reference variable must be initialized with an empty value: `ref-variable = ''`.

3.1.2 Custom Scripts

Inside the Custom Scripts block, presentation logic can be defined which is executed on the client-side managed by the infrastructure. On top of the Custom Scripts block in a separate metadata section, all input data sources obtained from the ASAccessServer for the execution and outputs created by the presentation logic can be defined. In a second section, the actual logic, which should be executed, can be either included from an external JavaScript file or defined inside the enclosing tags of this section. For a detailed description of the Custom Script block and an example implementation see Section 3.5.

3.1.3 Data Sources

In the Data Sources block, all data sources needed for the visualization of the Web Window are listed. Similar to the ADE inside the POI in which each channel obtained

from the PUMA automation system is represented by an individual block (see Figure 1.4 on page 4), this list defines which data sources are requested initially by the infrastructure from the ASAccessServer.

Each entry in the list of data sources is characterized by the following three properties:

- **Id:** The identifier of the data source used to obtain the data source from the REST interface of the ASAccessServer, for example `TORQUE` for the actual torque of the dynamometer.
- **Type:** The type of data source defined by the interface description of the AS-AccessServer, for example Signal or Function (see also Section 1.3).
- **Alias:** An unique alias to identify the data source inside the Web Window.

Because of the fact, that the identifier of the data source obtained from the ASAccess-Server can adjust to the current environment in which the Web Window is used, for example `Istmoment` for the German instead of `TORQUE` for the English environment, an unique Alias was introduced for each data source. In this way, only the Data Sources block or more specifically the Ids in the Data Sources block, must be adapted to any different namespace of the environment in which the Web Window could be used.

3.1.4 Control Bindings

In the Control Bindings block, the mapping between data sources listed in the Data Sources block and the controls of the Template block is defined. As described in Section 3.1.1, to each control of the visualization library inside the Template block an unique identifier must be assigned.

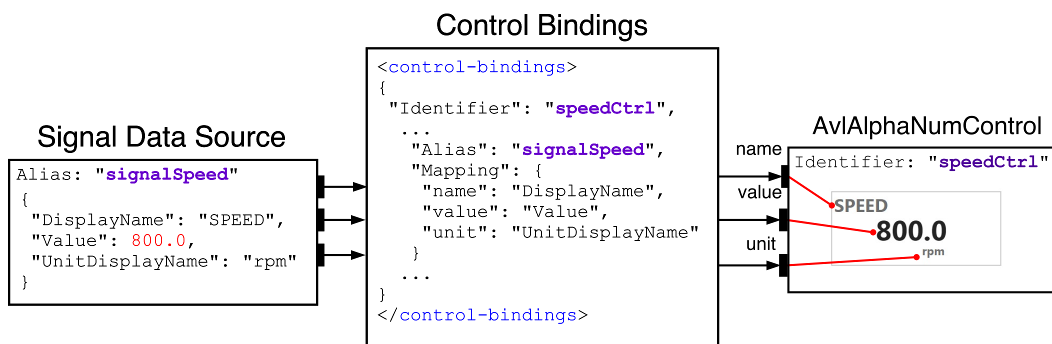

Figure 3.3: The Control Bindings Block defines the Mapping between Control Inputs and Data Source Properties

In Figure 3.3 an exemplary Control Bindings block is depicted: the data source with the alias `signalSpeed` specified in the Data Sources block, is bound to the alphanumeric control with the identifier `speedCtrl`. Furthermore, a mapping between each property of the data source and an input of the control can be specified. In contrast to the Control Bindings block shown in Figure 3.3, more than one data source can be mapped to one control.

3.1.5 Sub-Windows

Listing 3.1: The existing Web Window `barchart` and gauge are inserted into a new Web Window

```
1 <avl-window>
2   <avl-sub-window identifier="wnd1" path="barchart">
3   </avl-sub-window>
4   <avl-sub-window identifier="wnd2" path="gauge">
5   </avl-sub-window>
6 </avl-window>
```

In the POI the functionality exists to reuse multiple visualizations, in other words, the testbed operator can insert any existing window inside a new one. In this way, for example a tab control is currently implemented in the POI, whereby each tab hosts and displays a separate window file. A similar concept can be used to insert an existing Web Window in the actual Web Window file with the custom tag `<avl-sub-window>`. In Listing 3.1, an example for inserting two sub-windows in a Web Window is presented. With the custom HTML attributes, `identifier` and `path`, an unique identifier for the inserted window and the file path to the Web Window, which exists relative to the `wndfiles` directory (see Section 3.2.1), need to be defined.

3.2 Startup Process of a Web Window

3.2.1 Deployment of a Web Window

For the web visualization, every Web Window file, as described in Section 3.1, and all relevant files for the visualization need to be hosted and served by a web server to be later accessed from a web browser (Shklar and Rosen, 2009, Chapter 7). To access the PUMA resources and the Web Window from the same server, the web service of the ASAccessServer is configured to host and serve the files to any internal client. For this reason, in the web service of the ASAccessServer virtual directories can be

initialized, which act as an interface between a physical directory of the machine, where the ASAccessServer runs, and the REST interface. In this way, every HTTP GET request accessing the ASAccessServer with a specific URI path returns the corresponding file in the configured physical directory (Templin, 2007).

To separate between the files needed for the visualization and the actual Web Window files, the following two virtual directories are configured:

- **wndweb**: The `wndweb` virtual directory returns all files, for example JavaScript files or stylesheets, needed to enable the visualization of a Web Window and is set to the `common` folder under the program data path of the ASAccessServer web service, that is `C:\ProgramData\AVL\ASWebService\wndweb\common`.
- **wndfiles**: All Web Window files are hosted in the physical directory to which the `wndfiles` virtual directory points. This physical directory exists next to the `common` folder, that is `C:\ProgramData\AVL\ASWebService\wndweb\wndfiles`.

Through the configuration of both virtual directories, it was achieved that a Web Window file can be accessed by a simple HTTP GET method, for example the request with the URI `http://localhost:8890/wndfiles/demo.html` returns the `demo.html` Web Window file, when the ASAccessServer runs on the same machine and the port number is set to 8890 (Eastlake and Panitz, 1999). Furthermore, any updates on the JavaScript files for the visualization of the Web Window, for example if a control was added to the visualization library, can be rolled out separately to the `wndfiles` folder without touching any of the Web Window files.

3.2.2 Retrieval of a Web Window

Because of the specific structure of a Web Window and the separation of the storage location of the Web Window files (`wndfiles` folder) and the files to enable the visualization of the Web Window (`common` folder), an infrastructure was established for retrieving and further processing a Web Window until it is rendered in the web browser. The process of the retrieval of a Web Window, outlined in Figure 3.4, can be described in the following steps:

1. User enters URI for Web Window

When a user wants to display a Web Window in the web browser, the user enters in the address bar of the web browser the URI of the `wndweb` virtual directory and specifies with

the query parameter of the URI the name of the Web Window, which should be opened. The URI `http://localhost:8890/wndweb/?demo` opens, for example, an existing Web Window with the filename `demo.html` in the `wndfiles` virtual directory or program data folder, respectively. As described in Berners-Lee, Fielding, and Masinter (2005, chapter 3), the query parameter specified with the question mark at the beginning of the Web Window name is stored by the web browser in the search property of the location object location.search and is in this way persisted across the entire retrieval sequence of the Web Window (Hickson et al., 2016).

2. Web Service returns the Index-File

In Step 1., no path argument was specified for the request to the `wndweb` virtual directory, hence, the web service of the ASAccessServer returns the configured default file, that is the `index.html` file (Microsoft, 2016a). When the web browser receives the `index.html` file, it parses through the content of the HTML file and requests every external file, for example a JavaScript or a stylesheet file, from the same web service or more specifically from the `wndweb` virtual directory (Flanagan, 2011, Section 13.3.4). In the `index.html` file for the web visualization, the following external JavaScript files are referenced by a `<script>` tag, which are obtained from the `wndweb` virtual directory:

- **polyfills.js:** Includes all JavaScript code to implement features, which are not supported in the web browser in which the Web Window is loaded. For example, additional code to support the custom decorators of the Angular 2 framework such as the @Component decorator in web browsers, which does not support metadata reflection standardized in ES7 (see Section 2.1.4).
- **vendor.js:** Includes all vendor files from the Angular 2 framework and all dependencies, such as the RxJS library.
- **libs.js:** Includes all external JavaScript libraries loaded as global modules, such as the charting library FusionCharts.js.
- **avllib.js:** Includes all files of the infrastructure for the web visualization and the visualization library described in Section 4.
- **main.js:** The main entry point for the web visualization, which initializes the AVL Window Manager and bootstrapps the Angular 2 framework described in Section 2.1.4 and in the next point.

The listed JavaScript files are a product of the specific build process for the web visualization by using the webpack module bundler.

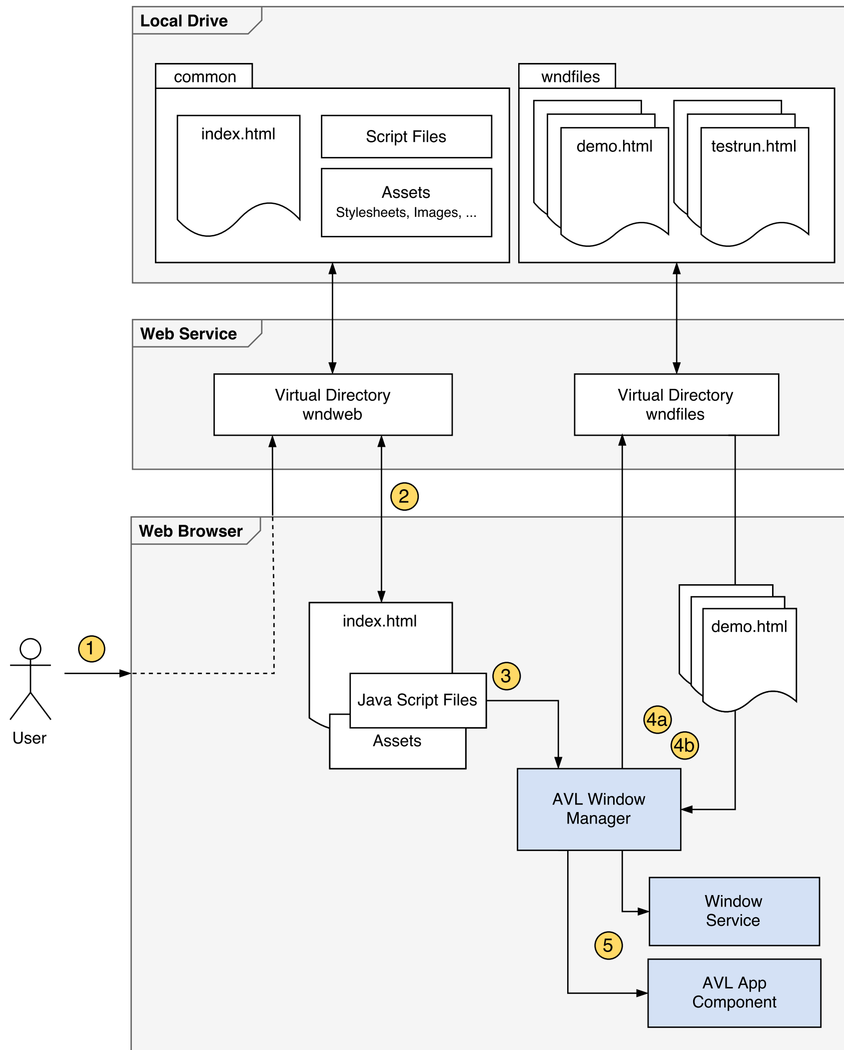

Figure 3.4: Sequence for retrieving a Web Window

3. Initialize AVL Window Manager

When all external files are obtained from the `wndweb` virtual directory, a JavaScript Promise described in Kambona, Boix, and Meuter (2013) guarantees that before the Angular app is bootstrapped (see Section 2.1.4), the AVL Window Manager is initialized. In this way, it can be guaranteed that after the Web Window is parsed and the JSON

formatted content is extracted, the Angular root component (see Section 3.3.1) only obtains the pure HTML defined in the Template block.

4. Request Web Window(s)

4a. Fetch Root Window
On initialization of the AVL Window Manager, the query parameter, for example `demo`, is read out of the location object and the Web Window file with the filename specified in the query parameter, that is `demo.html`, is requested from the `wndfiles` virtual directory of the ASAccessServer web service. The Web Window file opened by the user specified initially with the query parameter is described here as the root window.

When the requested root window file is received from the web service, a separate component the AVL Template Parser extracts the content of the Web Window. The content describes all relevant information of a Web Window such as the Data Sources or Control Bindings block, except the Template which describes the view of the Web Window. Furthermore, all external script files for the custom presentation logic described in Section 3.5 are requested.

4b. Request any Sub-Windows
In Section 3.1.5 the possibility was presented to insert an existing Web Window, that is a sub-window, in a new Web Window. Figure 3.5 shows how a sub-window is loaded and inserted in the Web Window when it was specified with the `<avl-sub-window>` tag, as described in Section 3.1.5, inside the root window.

Firstly, all path attributes of the `<avl-sub-window>` tags in the root window Template block are collected. As specified in the path attribute of the sub-window, every Web Window file is requested then from the ASAccessServer web service similar as the root window file. For each received sub-window file, the content is extracted same as for the root window file and any external script file for the custom presentation logic is requested. In a separate step, all identifiers specified in the Control Bindings block and all aliases in the Data Sources block are prefixed with the identifier defined as a custom attribute of the corresponding sub-window. This step guarantees that when the content of the sub-window is merged later on with the content of the root window, the data binding and the needed data sources for the inserted sub-window remain unique and no naming conflicts occur with the content of the root or other inserted sub-windows.

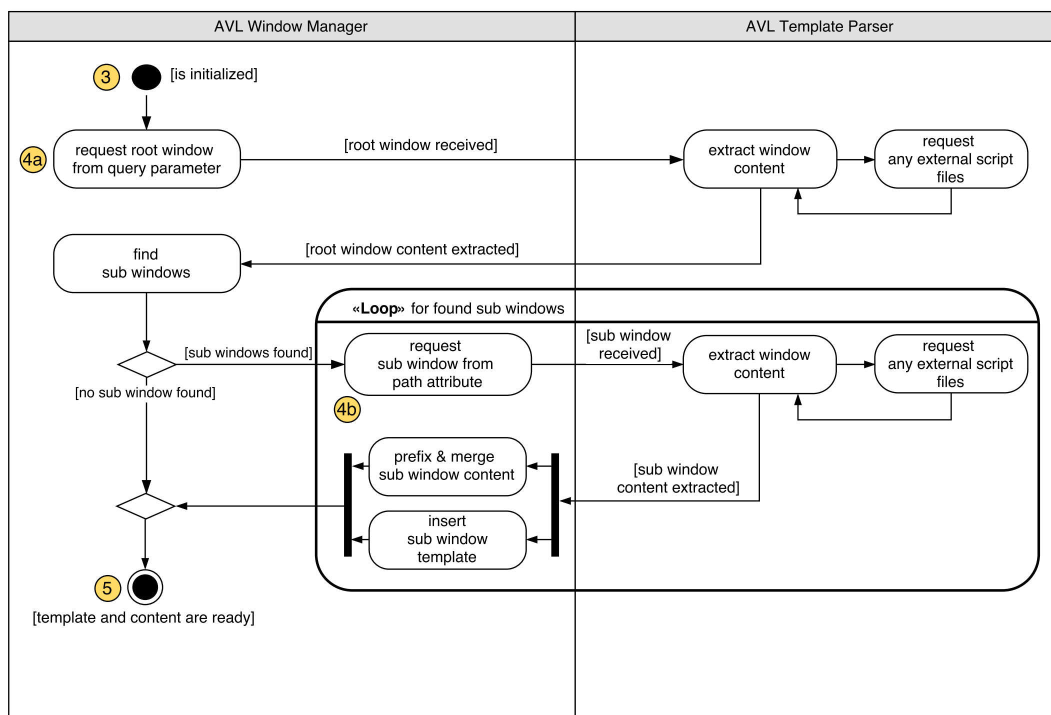

Figure 3.5: Sequence Diagram for loading multiple Sub-Windows

Similar as for the content, all identifiers of the controls from the visualization library in the Template block of the received sub-window file are prefixed with the identifier to match the previously renamed Control Binding entries. Then, the sub-window tag inside the Template of the root window is replaced by the received and adapted Template block of the sub-window.

5. Provide Content & Template of the Web Window

At this point, the content of all sub-windows defined in the root window is merged with the content of the root window. The content and the Template are provided then to the Window Service or the AVL App Component, respectively, both described in the following sections.

3.3 The Building Blocks of the Infrastructure

As described in Section 2.1.4, an Angular app can be modularized into separate building blocks called Angular modules. Apart from the AVL Window Manager and the AVL Template Parser, which are implemented as ES6 classes, the rest of the infrastructure for the web visualization is bundled into the Angular module AVL Base Module (Ecma International, 2015, Section 14). Following the SoC principle stated in Microsoft Patterns and Practices Team (2009, Chapter 2), the AVL Base Module is split into three main layers each providing specific functionality to the infrastructure for the web visualization. Figure 3.6 on page 50 shows the class diagram of the infrastructure with the following three layers:

- *Backend Access*,
- *Data Source Registry*, and
- *Control Registry* layer.

The above mentioned three layers form the layered architecture of the infrastructure, as presented in Microsoft Patterns and Practices Team (2009, Chapter 3), and guarantee a further important principle that a lower layer, for example the *Backend Access* layer, does not depend on any upper layer such as the *Data Source Registry* layer. In this way, the *Backend Access* layer could be reused in other applications or components. Every layer consists of one or more Angular Service(s) as an implementation of the Singleton pattern (see Stefanov (2010, Chapter 6) and Section 2.1.4), which provide the specified functionality of the layer, and are described in the following sections.

3.3.1 AVL App Component

The AVL App Component is the root Angular Component and accesses the Template block from the AVL Window Manager after the Angular 2 framework is bootstrapped. All functionality which is neither part of the infrastructure nor of the visualization library and needs to be accessible by all loaded Web Windows needs to be implemented in the AVL App Component.

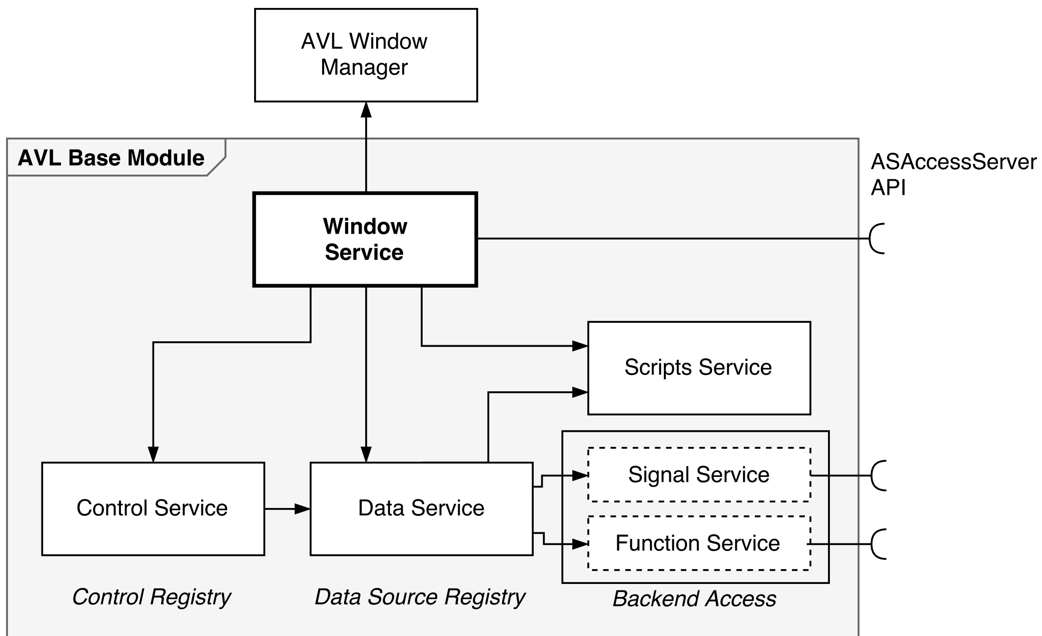

Figure 3.6: Class Diagram of the Infrastructure Services

3.3.2 Window Service

The Window Service acts independent from the three layers mentioned above and as the key link between the AVL Window Manager and the services responsible for the data source handling, such as the Control Service, Data Service and Scripts Service. Therefore, the Window Service forwards the content of the Web Window, that is the content of the Data Sources, Control Bindings and Custom Scripts block extracted by the AVL Window Manager, to the respective service. For example, the content of the Custom Scripts block is passed on to the Scripts Service as described in more detail in Section 3.5.2.

Another important task of the Window Service is to serialize the content received from the AVL Window Manager in more efficient dictionary data types, for example the content of the Data Sources block is sorted in a dictionary by the Alias property. Apart from the interface to the AVL Window Manager, the Window Service provides an interface to persist changes in the Control Bindings and Data Sources block by the ASAccessServer, for example when another data source as initially defined in the Web Window file shall be displayed by a control (see also Section 3.4.2).

3.3.3 Control Registry Layer

The Control Registry Layer is implemented by the Control Service as an Angular Service and is responsible to assign data sources as defined in the Control Bindings block to each control. To bind the data sources to a specific control, the Control Service needs a reference to every control inside the Web Window and realizes therefore the Registry pattern described in Fowler (2002, Chapter 18). The data source specified in the Control Bindings block are obtained hereby with the Alias from the Data Service. A main component that enables the assignment of data source properties to an input or output of a control is the AVL Window Component, which is also a part of the AVL Base Module.

AVL Window Component

The AVL Window Component is an Angular Component and can be inserted into the Template block of a Web Window with the tag `<avl-window>` as shown in Figure 3.2 or Listing 3.1. The property decorator `@ContentChildren` of the Angular framework is used, as described in Gechev (2016b), to get all controls of the visualization library defined as the content of the AVL Window Component and accessible from the DOM. Every control received from the property decorator is later registered to the Control Service to have a reference of every control and the inputs and outputs of the control.

3.3.4 Data Source Registry Layer

The Data Source Registry Layer is implemented by the Data Service as an Angular Service and is responsible to provide a common interface for all types of data sources. The Data Service holds a list of all data sources needed in the infrastructure and is based on the content of the Data Sources block. In addition, the list of data sources maintains a list of local data sources, more specifically Script data sources, which are created by the custom presentation logic (see Section 3.5). The used term local denotes that the data source is created entirely inside the infrastructure without a request to the ASAccessServer.

For each entry in the list of data sources, the Data Service requests a data source from the Backend Access Layer by the Id of the data source, that is depending on the data source type either from the Signal Service or Function Service. This data source is persisted then for later accesses of the same data source in a dictionary accessible by the Alias of the data source. An example for an entry in the Data Sources block shown in Listing 3.2,

a Signal data source with the Id `act_dyno_speed` is obtained from the Signal Service and is stored in the a dictionary accessible by the Alias `signalSpeed`. In this way, the Data Service acts as a local cache of data sources and an abstraction layer for obtaining data sources from the Backend Access layer and further from the ASAccessServer.

Listing 3.2: Example of a Data Source Definition

```
1 {
2  "Alias" : "signalSpeed",
3  "Type" : "Signal",
4  "Id" : "act_dyno_speed"
5 }
```

3.3.5 Backend Access Layer

The Backend Access Layer provides an interface to PUMA resources retrieved from the ASAccessServer to the rest of the infrastructure. For each data source type, for example the data source type Signal representing a PUMA resource channel or quantity (see Section 1.3), a separate Angular Service is implemented. Thereby, the access of each data source can be separated from the rest of the infrastructure and future data source types, for example the data source type SignalXt, can be added in a modular way without touching the running code base. Additionally, the separation of the data source access to the rest of the infrastructure hides implementation details, such as the used protocol or the specific API of the ASAccessServer. For both implemented services of the Backend Access Layer, the HTTP library of the Angular framework is used.

Signal Service

The access of a Signal data source from the ASAccessServer API is separated into two HTTP requests, which return the actual value and the metadata of the Signal data sources. In this way, the API reflects the different rate of change of the metadata, for example the name of a testbed channel, and the value, for example the actual rational speed of the dynamometer. The Signal Service merges the two separate requests and hence, data structures for the metadata and value of a Signal data source and simplifies thereby the access of a Signal Data Source.

The access of Signal data source inside the Signal Service can be described in three main steps:

1. request the metadata,
2. request the actual value of the Signal data source, and finally,
3. merging both the metadata and the value to provide the Signal data source to the infrastructure.

1. Request Metadata
To retrieve the metadata of a Signal data source, the ASAccessServer offers two HTTP requests: (i) to retrieve metadata of a single Signal data source and, (ii) to get the metadata of all Signal data sources with an id beginning with at least three specified characters. The Signal Service gives access to each of the requests to provide parts of the infrastructure, for example the Signal Browser (see Section 3.4.2), with the metadata of a Signal data source and caches the result of metadata queries of both requests.

When providing a Signal data source, that is the metadata and value of a Signal data source, the Signal Service first reads out the cache filled by any previous metadata queries of the infrastructure and returns the found metadata. When, for example at startup of the infrastructure, multiple metadata queries are initiated at once, it can happen that for the same Signal data source id a request to the ASAccessServer was already triggered, however no response for this request has been received yet and hence, the corresponding metadata could not be found in the cache. In order to avoid in this case, multiple requests for the same metadata, an additional cache filled with metadata Observables, which represent an reference of the asynchronous result of an HTTP request.

2. Request Value
For a more efficient access of the actual value of a Signal data source, the ASAccessServer provides an HTTP POST request to retrieve an array of actual values from multiple Signal data sources, whereby a list of Signal ids, for example `act_dyno_speed` for the actual speed of the dynamometer, is supplied in the body of the request. When a new value of a Signal data source is queried by the infrastructure, the id is added to a global list of all needed Signal data sources in the Signal Service, which is used for each request.

Long polling, as described in Fain, Rasputnis, and Tartakovsky (2014, Chapter 8), is used to update the Signal data source values on a regular basis. For this purpose, as shown in Figure 3.7, the `interval` operator of the Reactive Extensions for JavaScript (RxJS) library is used to create an Observable which periodically emits an ascending integer. The `switchMap` operator is chained to the `interval` operator to trigger on each emission of an integer, an HTTP POST request with the currently needed Signal ids in the body of the request.

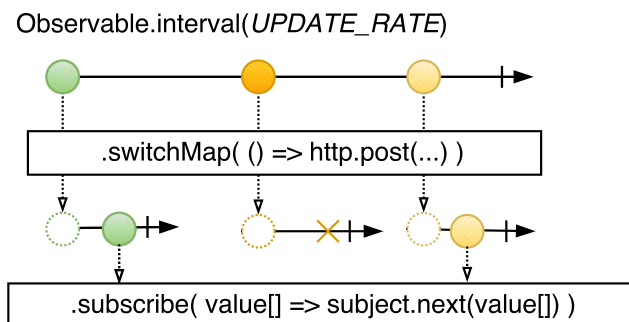

Figure 3.7: Periodic Requests for the actual Value of all Signal Data Sources

In the event of a delay of a response, for example due to the network latency, which exceeds the interval rate (`UPDATE_RATE`), the `switchMap` stops the latest request and initiates a new request. In this way, it can be guaranteed that always the actual value of the Signal data source is forwarded to the infrastructure and no queuing of delayed Signal values occurs. By subscribing on the created Observable, the overall sequence is started, and on each received HTTP response, a Subject is used to forward the retrieved array of Signal values with the operator `next` to the proceeding step.

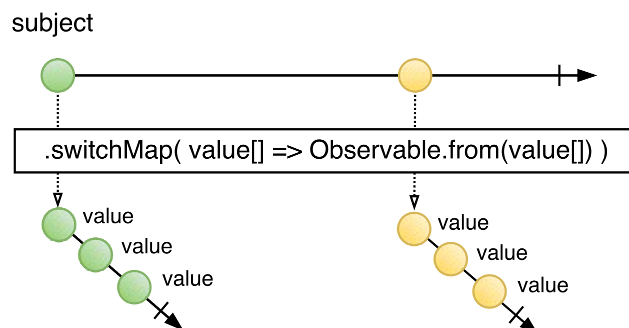

Figure 3.8: Flattening the Array of Values to a Stream of Values

Because of the fact, that the HTTP request of the ASAccessServer returns for each request an array of Signal values, the `switchMap` operator is used again to transform the retrieved array to a Observable stream of separate values. Hereby, the operator, as shown in Figure 3.8, is chained to the Subject used to retrieve the array, as described above.

3. Merge Metadata & Value
When one of the requests or more specifically the Observable for the metadata and the value, emits an item the `combineLatest` operator is used to merge always the latest received metadata and value to a Signal data source. This Signal data source can be accessed by a method in the Signal Service with a the Signal id.

Function Service

Similar as in the Signal Service, the ASAccessServer offers two HTTP requests of a Function data source: (i) to retrieve the metadata of a Function data source, for example the needed input and output parameters of an activation object (see Section 1.3), and (ii) to trigger an Function data source with the provided input parameters, specified in the metadata of the Function data source. To retrieve the metadata of a Function data source, the same cache strategy is implemented as for the metadata in the Signal Service.

3.4 Data Source Assignment

3.4.1 Initial Assignment of a Data Source

The sequence of the initial assignment of all data sources to the controls as specified in the Control Bindings block is shown in Figure 3.9 and can be described as follows:

1. Set Content to Services

As described in the Section 3.3.2, the Window Service retrieves the list of data sources defined in the Data Sources block of the Web Window and fills the list of data sources in the Data Services. Similarly, the content of the Control Bindings block is set to the Control Service.

The Data Service holds now a dictionary with all needed data sources of the infrastructure. The Control Service holds a mapping between the inputs and outputs of all controls referenced in the Template block and the data source properties listed in the Data Sources block.

2. Get Signal/Function Data Source

When a dictionary entry of the needed data source in the Data Service is created, depending on the data source type either from the Signal or the Function Service, a data source Observable is requested, which is then persisted again in a separate dictionary for each data source type. The Signal Service or Function Service makes an HTTP request to the ASAccessServer, as described in Section 3.3.5, and returns an Observable.

The Data Service holds now a dictionary with Observables for the data sources defined in the Data Sources block of the Web Window.

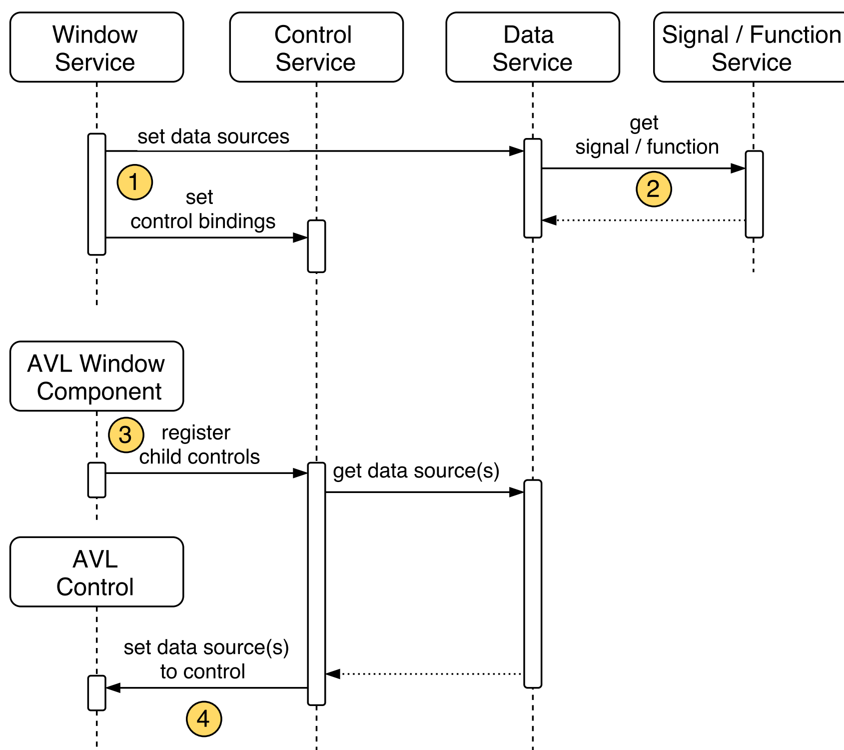

Figure 3.9: Initial Assignment of a Data Source to a Control

3. Register Controls

Inside the ngAfterViewInit life-cycle hook of the AVL Window Component (see Section 3.3.3 and 2.1.4), all child controls of the AVL Window Component from the

visualization library are registered to the Control Service.

When a control is registered to the Control Service, it searches in the dictionary, sorted by the identifier of the control and set by the Window Service in Step 1., for the Control Binding of the registered control. All data sources specified in the found Control Binding are then requested from the Data Service accessible by the Alias of the data source.

The Control Service knows now which Data Source property shall be bound to which registered control and holds an Observable of all referenced data sources.

4. Bind Data Source to Control

When the Data Service returns a data source Observable, the Control Service searches for the reference of the control in the dictionary of controls in the Web Window filled by the AVL Window Component. Then, a dynamic subscriber is created, which sets whenever a data source item is emitted by the Observable of the data source the inputs of the control to the actual property values of the data source item.

3.4.2 Dynamic Assignment of a Data Source

One requirement of the infrastructure was to allow to modify the visualization during the runtime, this means that the user can change the assigned data source to a control during the runtime. The term runtime states here that the Web Window file already exists and is loaded in the web browser as described in Section 3.2. To give the user the possibility, to change the assigned data source to a control different as defined initially in the Control Bindings and Data Sources block of the Web Window, for each data source a Browser component exists. This Browser component can be added in a modular way to every control of the visualization library defined in the Template block, enclosed by the starting and closing tag of the corresponding control similar as for the AVL Window Component.

Signal Browser

Listing 3.3 shows how to add a Signal Browser component implemented as an Angular Component (see Section 2.1.4) to an alphanumeric control, which is displayed, when the snippet is added to Template block of the Web Window in the web browser, as presented in Figure 3.10.

Listing 3.3: Adding a Signal Browser to the Alphanumeric Control

```
1 <avl-alpha-num-control identifier="ctrlSpeed">
2   <avl-signal-browser identifier="ctrlSpeed">
3   </avl-signal-browser>
4 </avl-alpha-num-control>
```

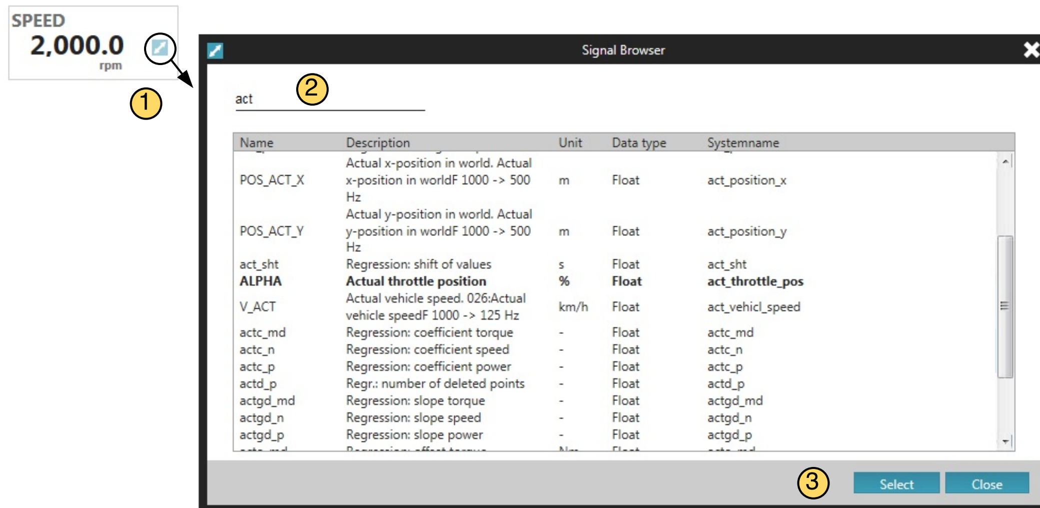

Figure 3.10: Signal Browser to change the assigned Signal Data Source

The process to change the assigned Signal data source of a control can be split in three steps, described as follows:

1. User opens the Signal Browser Dialog

When adding a Signal Browser to a control, an icon appears on the right side of the control (see Figure 3.10), which opens a Signal Browser instance for this control. When the Signal Browser was opened, as presented in the sequence diagram in Figure 3.11, the Signal Browser searches for previously cached metadata for Signal data sources in the Signal Service as described in Section 3.3.5. In addition, the metadata of all created local data sources, that is Script data sources, for the custom presentation logic are obtained from the Data Service. In this way, the user can also assign a locally defined data source in the Custom Scripts block to a control during runtime. The retrieved metadata is then displayed in a table inside the Signal Browser.

2. User searches for Signals

In a separate text input control, which is part of the visualization library, the user can

search for metadata of Signal data sources, which are currently not used within the infrastructure and therefore are not cached by the Signal Service. When the user enters at least three characters, the corresponding request of the ASAccessServer is used in the Signal Service to get all metadata for Signal data sources with the beginning characters. Furthermore, the list of Script data sources are filtered by the entered characters and merged with the retrieved metadata from the Signal Service. In Figure 3.10, the list in the Signal Browser shows all Signal data sources beginning with 'act', such as the actual vehicle speed or the actual throttle position, after the user entered the three characters.

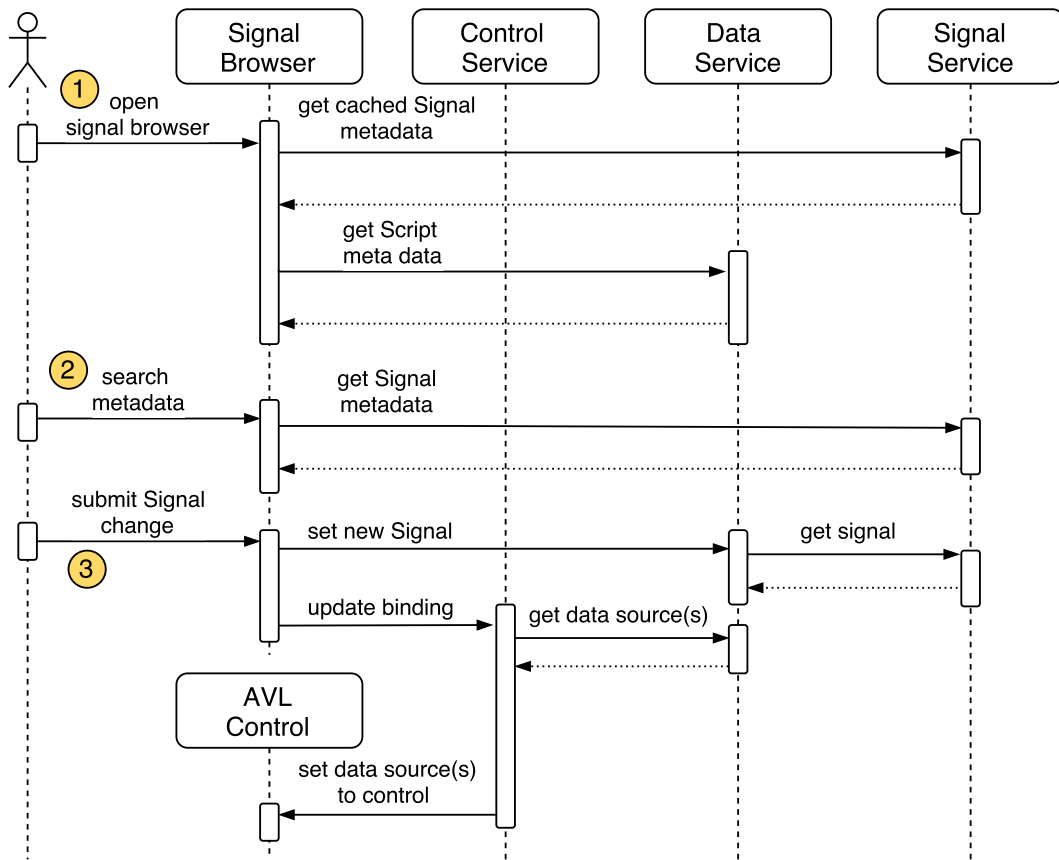

Figure 3.11: Dynamic Assignment of a Data Source to a Control

3. User submits a Change
When the user selects a Signal or Script data source in the list of metadata and submits the selection with the Select button, the Signal Browser creates a new Signal data source in the Data Service. Hereby, the Data Service adds this Signal data source to the list of data sources and requests a new Observable for the Signal data source from the Signal Service. Furthermore, the Signal Browser adapts the Control Binding of the control according to the change in the bound data source, this means that the mapping between data source properties and control inputs remains unchanged and only the Alias field in the Control Binding, as described in Section 3.1, is updated to the new Signal data source. When the Control Binding of the control is adapted inside the Control Service, the data source is bound to the control inputs in the same way as described in Section 3.4.1.

Persisting the Changes of the Web Window

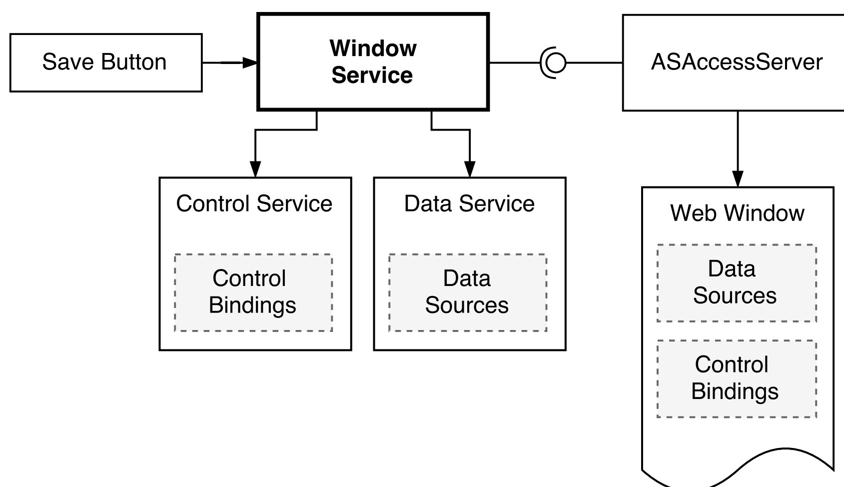

Figure 3.12: Persisting Changes of the Web Window

When the user changed for example the channel, which shall be displayed by an alphanumeric control via the Signal Browser, a requirement of the infrastructure is to persist these changes. For this reason, the web service of the ASAccessServer provides an interface to write back the changes to the Web Window file. As mentioned in Section 3.3.2, the Window Service requests this interface in a public method to save the actual Control Bindings of the Control Service and list of data sources stored in the Data Service (see Figure 3.12). When this method is called, the Window Service accesses the

current version of all Control Bindings and the list of data sources, packs both in a valid JSON format for an HTTP request, and triggers the corresponding POST request of the ASAccessServer with the name of the Web Window as an URI parameter. The web service of the ASAccessServer parses then the Web Window file specified with the URI parameter, for example `demo.html`, for the Control Bindings and Data Sources block specified by their unique HTML tags, and writes the body of the corresponding request inside both blocks.

To allow the user to save any changes the user has made, a Save Button was implemented as an Angular Component and can be inserted in every Web Window with the custom tag `<avl-save-button>`. The Save Button triggers, when clicking on the used HTML button control, the public method of the Window Service to persist the content of a Web Window as describe above.

3.5 Custom Presentation Logic

3.5.1 Defining the Presentation Logic

Similar to the ADE mode in the POI, custom presentation logic can be defined inside a Web Window. Therefore, in the Custom Scripts block (see Section 3.1.2) script snippets, which are stored in the same `wndfiles` folder as the Web Window files, can be included and a mapping between inputs and outputs of the script snippet and data sources, known inside the infrastructure, can be defined.

In Listing 3.4, a definition of custom presentation logic for a Web Window is shown, split into two sections: (i) the metadata for the presentation logic, and (ii) a reference to an external script snippet file. To reuse script snippets in multiple Web Windows, a generic namespace is used in every script snippet for the definition of inputs and outputs, for example `Input1` to `InputN` for N inputs of the script snippet. For this reason, when including the script snippet in a Custom Scripts block, a mapping must be defined between the generic input and output names of the script snippet and data sources of the infrastructure.

Listing 3.4: Definition of a Presentation Logic inside a Web Window

```
<custom-script>
  <meta-data>
  {
    "InputMapping" :  {
      "Input1" : "act_dyno_speed"
    },
    "OutputDeclarations" : [
      {
        "Id" : "scriptSpeedLimit",
        "Description" :  "Speed is above 1000rpm",
        ...
      }
    ],
    "OutputMapping" : {
      "Output1" : "scriptSpeedLimit"
    }
  }
  </meta-data>
  <code path="indicateLimit"/>
</custom-script>
```

This mapping must be defined on top of each custom scripts block in a separate metadata section, which consists of the following three parts:

- **InputMapping:** Defines a mapping between inputs of the script snippet, for example `Input1`, and a data source known inside the infrastructure. The data source is defined hereby with the Id, for example `act_dyno_speed` for the actual speed of the dynamometer (see Data Sources block in Section 3.1).
- **OutputDeclarations:** Lists all new data sources, which are created by the Custom Scripts block, and adds metadata to the data sources, for example a description of the new data source, which can be later on displayed in the Signal Browser (see Section 3.4.2).
- **OutputMapping:** Defines a mapping between outputs of the script snippet, for example `Output1`, and a data source declared in the OutputDeclarations section.

The external script snippets can be included with the `<code>` tag, where the custom attribute `path` defines which script snippet file shall be used. For example, the script snippet presented in Listing 3.5 is included in the Custom Scripts block shown in

Listing 3.4, which provides the basic functionality to monitor the maximum value of a data source, here the actual speed of the dynamometer.

Listing 3.5: Reusable Script-Snippet for a Custom Presentation Logic

```
1 var indicateLimit = (Input1.Value > 1000);
2 var Output1 = {
3   IsAboveLimit: indicateLimit
4 };
```

3.5.2 Running the Presentation Logic

Similar as described for inserting already existing Web Windows in a new Web Window (see Section 3.2.2), the AVL Template Parser requests all external script snippet files and extracts the metadata section from each Custom Scripts block of the loaded Web Window (see also Figure 3.5). When all external script snippets were received, the AVL Template Parser uses the InputMapping and OutputMapping of the metadata section to customize the imported script snippet for the current Web Window, which means that all generic variables inside the script snippet are replaced by the data source Id defined in the InputMapping or OutputMapping, for example `Input1` is replaced by `act_dyno_speed`.

In case that the Custom Scripts block was defined in a sub-window file, all variables are prefixed in a separate step with the unique identifier of the sub-window similar as for the Control Bindings and the Data Sources block as described in Section 3.2.2. For all proceeding steps, the content of the Custom Scripts block is bundled to a ScriptDefinition object, which consists of all inputs (data source ids in the InputMapping), outputs (OutputDeclarations) and the actual code of the presentation logic.

For the execution of the code for the custom presentation logic, an additional Angular Service, the Scripts Service is implemented. The Scripts Service retrieves in the same way as the Data Service, the content of all Custom Scripts blocks of the Web Window from the AVL Window Manager via the Window Service and manages the execution of the scripts in separate Web Worker threads. As reported in Flanagan (2011, Section 13.3.3), the execution of JavaScript in the web browser is strictly single-threaded and every JavaScript is run in the UI thread. The Web Worker feature as part of the HTML5 standard, enables to swap computationally expensive logic into a separate Web Worker context and background thread (Chromium Team, 2016). In this way, the UI thread can remain responsive and will not be blocked by any long-running logic (Oakley and Pulimood, 2012).

For this purpose, for every ScriptDefinition obtained from the Window Service, the subsequent steps are processed:

1. The code of the ScriptDefinition is parsed by the Acorn JavaScript parser[1] for a first rough validation of the imported script snippet.
2. The Binary Large Object (Blob) API is used to swap the execution of the evaluated JavaScript code in a separate Web Worker thread (Ranganathan, Sicking, and Kruisselbrink, 2016). With a message bus, that is the Channel Messaging API defined in Group (2016, Section 9.5), data is passed between the UI and the Web Worker thread.

Listing 3.6: Code which runs in a separate Worker Thread for the Presentation Logic

```
1  self.onmessage = function(e){
2    var act_dyno_speed = e.data[0];
3    //******** START: User defined code ********//
4    var indicateLimit =
5      (act_dyno_speed.Value > 1000);
6    var scriptSpeedLimit = {
7      IsAboveLimit: indicateLimit
8    };
9    //******** END: User defined code **********//
10   var scriptResult = {
11     scriptSpeedLimit: scriptSpeedLimit
12   };
13   postMessage(scriptResult);
14 }
```

3. The imported script snippet is surrounded by code to access the retrieved data from the message bus and to send the result data back to the UI thread. In Listing 3.6, the acutal code, which runs inside on Web Worker is presented. In Line 2, the input parameter data of the `onmessage` event callback is accessed and stored in a local variable defined by the inputs of the ScriptDefinition. In addition, all script snippet outputs are merged into one result object, which is posted back to the UI thread with the `postMessage()` method (see Line 10-13).
4. Already created data sources with the same id as defined in the inputs of the ScriptDefinition are searched in the Data Service. When no data source was found, a new data source with the specified id is created in the Data Service (see Section 3.3.4).

[1]Acorn: Open source JavaScript parser: https://github.com/ternjs/acorn

5. A new data source is created in the Data Service for each entry in the outputs of ScriptDefinition.
6. The `combineLatest` operator is applied to all data sources listed in the inputs of the ScriptDefinition, similar as for the metadata and the value of a Signal data source described in Section 3.3.5. In the subscriber, which is chained to the result of the merged input data sources, the current input data is posted to the Web Worker thread with the `postMessage()` function.
7. To receive the result object from the Web Worker thread, an Observable is created from the `onmessage` event of the Web Worker thread with the `fromEvent` operator. The previously merged result object received from the Web Worker thread is separated again and forwarded to the created Script data source in the Data Service via a Subject (see also `next` operator in Section 3.3.5)

3.5.3 Editing the Presentation Logic

To allow the user to change the custom presentation logic defined inside the Web Window or in an external script snippet file, the Script Editor component is implemented using the Angular `@Component` decorator (see Section 2.1.4). Similar as the Signal Browser, described in Section 3.4.2, the Script Editor consists of an button and a modal dialog, which opens when the user clicks on the corresponding button. When the Script Editor opens, all ScriptDefinition entries of the current Web Window are obtained from the Script Service and the metadata of all data source ids listed in the inputs of the ScriptDefinition are acquired from the Signal Service.

The modal dialog of the Script Editor, shown in Figure 3.13, is divided into the following parts:

1. Outputs

A list box is filled with all ScriptDefinition objects received from the Scripts Service and shows for each ScriptDefinition the name of all defined data sources in the outputs of the ScriptDefinition. The name was hereby initially defined in the OutputDeclarations in the metadata section of the Custom Scripts block (see Section 3.5.1).

2. Inputs

The names of all inputs of the ScriptDefinition object, which is selected in the Outputs list box, are listed here. The name of the data source is hereby obtained, as described above, from the Signal Service, when the Script Editor opens.

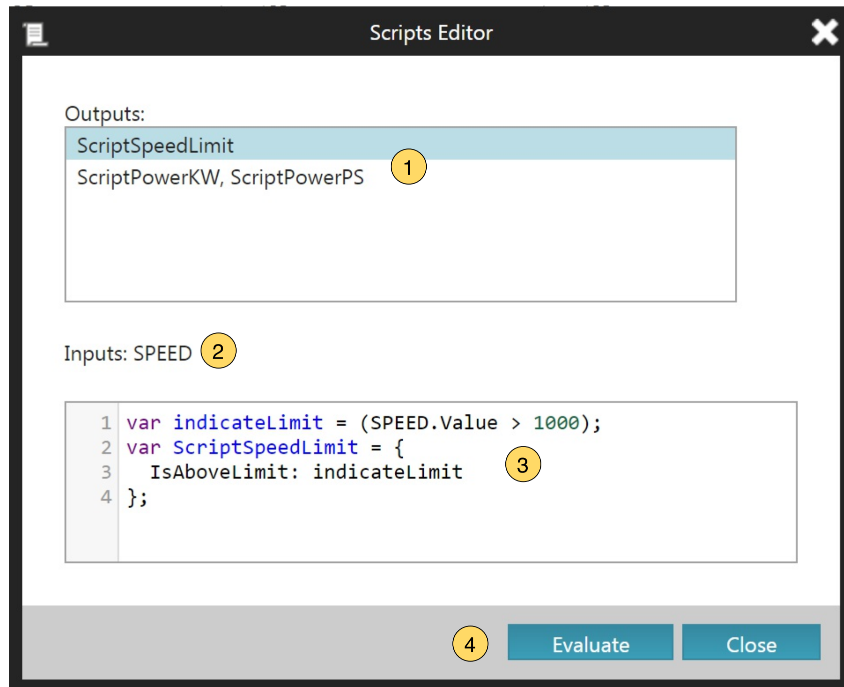

Figure 3.13: Scripts Editor for the Custom Presentation Logic

3. Code

The CodeMirror[2] text editor is hosted in the Script Editor to show the JavaScript code, which was requested from the external script snippet file, with syntax highlighting and to allow to adapt the presentation logic. Before the script snippet is loaded inside the CodeMirror text editor, all data source ids are replaced by the name of the data sources obtained from the Signal Service and from the OutputDeclarations similar as for the Outputs and Inputs section. When comparing, the code displayed by the CodeMirror

[2] CodeMirror: JavaScript text editor `https://codemirror.net/`

text editor in Figure 3.13 and the original script snipped file shown in Listing 3.5 on page 63, it can be seen that all data source ids, such as `act_dyno_speed`, are replaced by their names, that is `SPEED`. In this way, the user always sees the names of the used PUMA resources of the current environment, for example `TORQUE` for the English and `Istmoment` for the German environment.

4. Update Code

When the user submits any changes in the code editor with the Evaluate button, the previously data source names are translated again to their ids and stored back to the Scripts Service, where the corresponding web worker is updated with the modified code (see Section 3.5.2).

4 Visualization Library for Web Applications

4.1 Structure of the Library

Similar as the infrastructure for the web visualization described in Chapter 3, the visualization library is bundled in a single Angular 2 module. In this way, it is guaranteed that the visualization library can be reused in multiple applications by simply inserting the module into an Angular 2 application. An important point here is that the Angular 2 module of visualization library is independent of any other Angular 2 module such as the module for the infrastructure.

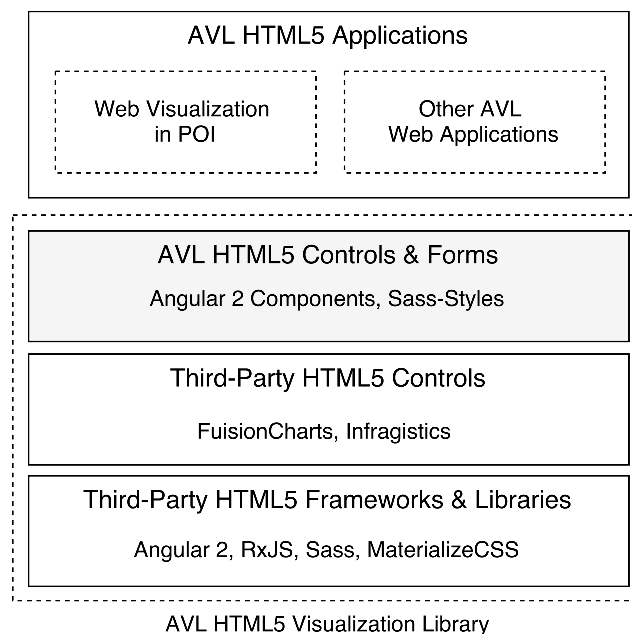

Figure 4.1: Library Stack of the AVL HTML5 Visualization Library

The layered structure of the visualization library is presented in Figure 4.1 and can be summarized as follows:

- **AVL HTML5 Applications:** As already mentioned, the visualization library itself is independent of any other Angular 2 modules and thus can be used by any other web application which is based on the Angular 2 framework, for example the web visualization in the POI. The corporate-wide use of the same library for web applications ensure the same functionality of controls across all web applications. Furthermore, it decreases the development time of a new web application.
- **AVL HTML5 Control & Forms:** The visualization library provides currently two basic blocks: controls and forms, both implemented as Angular 2 components. Whereby, controls only incorporate the View definition, styling and control-specific logic, forms include further logic to access any necessary data from a server. All controls and forms of the visualization library provide a common interface in form of an Angular 2 component to other web applications. In this way, any changes in the third-party libraries are decoupled from the business logic in the web application (see Section 2.3.2). In addition, to the style definition of each control, several common style definitions are deployed by the library using Sass-Styles, which extend the CSS syntax towards a better management of multiple stylesheet files. These common style definitions allow a consistent corporate styling across all web applications, which uses the visualization library.
- **Third-Party HTML5 Controls:** Third-party libraries mainly based on JavaScript and the jQuery DOM manipulation library are used in the visualization library with an Angular 2 component wrapper, for example the multi-series line chart of the FusionCharts library is wrapped by a XtControl Angular 2 component.
- **Third-Party HTML5 Frameworks & Libraries:** At the bottom of the library stack, all frameworks and libraries are listed which enable the overall web application, for example Angular 2 for the SPA development, RxJS for the asynchronous HTTP functionality and MaterializeCSS for a responsive design of the web application. Every web application which consumes the visualization library needs to align with the used technology stack listed here to ensure the best overall performance of the application.

4.2 The Alphanumeric Control as a Model Example for a Control of the Visualization Library

The alphanumeric control is used here as a model example of a control of the visualization library. Every control of the visualization library implemented as an Angular 2 component consists of three files, following the Angular 2 style guide (Angular Team, 2016) (see also Figure 4.2):

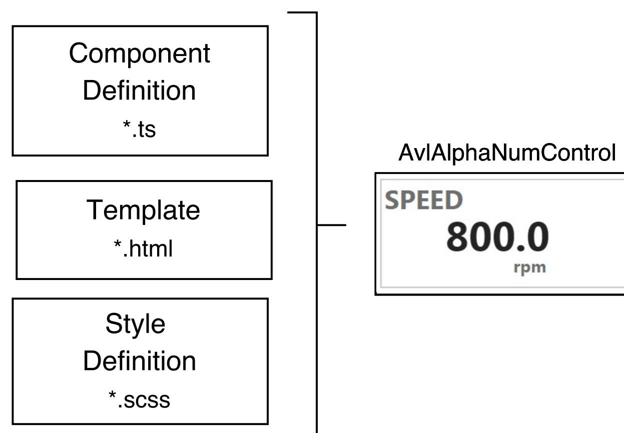

Figure 4.2: Library Stack of the AVL HTML5 Visualization Library

- **Template:** The template is a HTML file extended by Angular 2 keywords and describes the view of the control, that is the composition of elements represented later in the DOM.
- **Style Definition:** In a Sass-Style, all relevant style definitions of the visualization library are referenced and additional control-specific style rules are added.
- **Angular 2 Component Definition:** The Angular 2 component definition consists of a TypeScript file, which references the template and the style definition file. Furthermore, control-specific logic such as presentation logic can be added in the component definition. Every control of the visualization implements the AvlControl interface, which enables the use of every control in applications such as for the extended POI visualization described in Chapter 3. As an example of an Angular 2 component definition, the definition for the alphanumeric control is shown in Listing 4.1.

Listing 4.1: Angular 2 Component Definition of the Alphanumeric Control

```
1  @Component({
2    selector: 'avl-alpha-num-control',
3    templateUrl: './avl_alpha_num_control.html',
4    styleUrls: [
5      './avl_common_styles.scss',
6      './avl_alpha_num_control.scss'
7    ]
8  })
9  export class AvlAlphaNumControl implements AvlControl
10 {
11   @Input() identifier: string;
12   @Input() name: string = 'Name';
13   @Input() value: number = 0;
14   @Input() unit: string = 'Unit';
15
16   ...
17 }
```

4.3 The Xt-Control as an Implementation of a Multi-Series Line Chart

In addition to the alphanumeric control, a xt-control was implemented to show the evolution of data sources in form of a multi-series line chart retrieved, for example, from the ASAccessServer API. Currently, two technologies exist in a state-of-the-art web browser to implement such as multi-series line chart: (i) the Scalable Vector Graphics (SVG) and (ii) the Canvas technology. To examine the advantages and disadvantages of both technologies, two xt-controls were implemented each using one of the listed technologies. Similar as for the alphanumeric control, both implementations of the xt-control are implemented as an Angular 2 Component and provide the same interface in form of a component definition to the web application. As can be seen in Listing 4.2, currently, both xt-controls only support to draw the evolution of two data sources provided by the two input parameters `value1` and `value2`. Similar as for the alphanumeric control, the unit of the data source and a descriptive name of the data source can be displayed by the xt-control.

As already mentioned, the xt-controls use the SVG and the Canvas technology. Therefore, the third-party library CanvasJS [1] for the Canvas technology and the FusionCharts [2] library for the SVG technology were wrapped by the corresponding Angular 2 Component. To indicate when the wrapped multi-series chart needs to be updated, a `timeStamp` input parameter was introduced. Whenever this input parameter changes, the current data source values given by the other input parameters are forwarded to the corresponding API call of the wrapped library.

Listing 4.2: Angular 2 Component Definition of the Xt-Control

```
@Component({
  selector: 'avl-xt-control',
  template:  './avl_xt_control.html',
  styleUrls: [
    './avl_common_styles.scss',
    './avl_xt_control.scss'
  ]
})
export class AvlXtControl implements AvlControl {
  @Input() identifier: string;
  @Input() timeStamp: number = 0;
  @Input() name1: string = 'Name 1';
  @Input() value1: number = 0;
  @Input() unit1: string = 'Unit 1';

  @Input() name2: string = 'Name 2';
  @Input() value2: number = 0;
  @Input() unit2: string = 'Unit 2';

 ...
}
```

In addition to the described alphanumeric and xt-control, the visualization library currently consists of two standalone forms, which can be used independently of other Angular 2 modules for the visualization, as described in Chapter 3, to display the current state of the automation system and act thereby as implementation of an external client presented in Section 1.4.

[1] CanvasJS: HTML5 JavaScript charting library http://canvasjs.com
[2] FusionCharts: JavaScript charts for web and mobile http://fusioncharts.com

5 Performance Evaluation

5.1 Evaluation Setup

For the evaluation of the performance impact of the web visualization on the automation system, several measurements were run on a Dell Latitude E6530 laptop with an i7-3740QM processor, which has four physical cores running at 2.70GHz. Enabled by the Intel®Hyper-Threading Technology described in Intel (2016), the execution resources of one physical core can be shared within two independent processors each holding an architectural state and denoted as logical processors (Marr et al., 2002).

The PUMA Open™2 (Release 3) automation system is installed on the same laptop, which requires the installation of the INtime for Windows real-time operating system. The INtime operating system is configured through the installation of the automation system to occupy one logical core of the processor. Hence, only seven of the eight logical cores of the i7 processor were available for the performance evaluation.

All measurements were evaluated on the following web browsers, based on the three most used web browsers for the Windows operating system during 2015 and 2016 (StatCounter, 2016):

- Google Chrome v50,
- Mozilla Firefox 45.0.2,
- Internet Explorer (IE) 11.

With the Windows Performance Monitoring Tool, which is integrated into the Windows operating system, the processor load and the memory usage of the relevant processes of the above listed web browsers were recorded over a time span of ten minutes for each measurement (Microsoft, 2016c). The processor load of the specific browser process was hereby measured by the *% Processor Time* performance counter of the Window Performance Monitoring Tool, which is the percentage of time one logical processor was busy servicing the measured process (Microsoft TechNet, 2016).

The memory usage is described by the *Private Working Set* performance counter of the process and stands for the total physical memory, that is the Random-Access Memory (RAM), used by the process (Russinovich, Solomon, and Ionescu, 2012, Chapter 10). The time between two consecutive sampled values were set to the smallest available timespan of one second. For all evaluated web browsers, the hardware acceleration was enabled by default, this means that the Graphics Processing Unit (GPU) can be utilized by the web browser to accelerate the compositing and rasterization of the Web Window (see Section 5.4). The term compositing describes here the process to calculate the actual size and position of all elements described by the DOM.

As described in Reis and Gribble (2009) and The Chromium Projects (2016b), the Google Chrome web browser based on the open-source Chromium web browser, spawns a separate process for the UI frame of the web browser (browser process), for example the navigation bar, and for each browser tab a separate process to render the content of the tab itself (render process). In addition, a process is started by the Google Chrome web browser for the GPU accelerated rendering (GPU process, see Wiltziusa, Kokkevis, and Chrome Graphics team (2014)). A similar process model, denoted as Loosely-Coupled Internet Explorer (LCIE) in Zeigler (2008) and Crowley (2010, Chapter 1), is implemented for the IE, in which the browser window in this context named as the UI frame resides in one common process (browser process) and each tab runs in a separate process (render process). In the version of the Firefox browser used for the performance measurements, the same process is used for the actual browser UI and the content of the tabs (Mozilla Developer Network, 2016). Future versions of the Firefox browsers will align with the process model of the IE and the Google Chrome browsers known under the Firefox Electrolysis architecture (Dotzler and Lassey, 2016).

In order to collect the full activity of all web browsers, the performance counters of the following processes were recorded by the Windows Performance Monitoring Tool:

- Google Chrome: chrome.exe (browser process), chrome.exe (render process), chrome.exe (GPU process),
- Mozilla Firefox: firefox.exe (browser process),
- IE11: iexplore.exe (browser process), iexplore.exe (render process).

In addition to the three listed browsers, the IE was integrated into the POI using the Web Browser Control provided through a Component Object Model (COM) interface (Shdocvw.dll) described in Crowley (2010, Chapter 11), Microsoft Software Developer Network (2016b) and Microsoft Software Developer Network (2016a). Similar as for the standalone web browsers, the processor load and memory usage of the relevant process were recorded (iexplorerhost.exe), here abbreviated as Hosted IE. When using the Web Browser Control the following registry keys must be set to guarantee that the current

version of the Internet Explorer is used and the hardware rendering is enabled (Microsoft Software Developer Network, 2012; Microsoft Software Developer Network, 2014).

- FEATURE_BROWSER_EMULATION: 0x11000 for IE11,
- FEATURE_GPU_RENDERING: 1 to enable hardware rendering.

For a fair comparison between all web browsers and the hosted IE, the processor load and memory usage of all processes of the corresponding web browser are summed up. In addition, similar test cases as described in the next sections are setup with the native visualization for the current visualization in the POI (see Section 1.2). Hereby, the processor load and private working set of the relevant process (poi.exe) was recorded using the same settings of the Windows Performance Monitoring Tool. For all test cases for the POI, the values are simulated with the Multi-Function-Generator (MFG) parameter block of the PUMA automation system. The reserve channels RESCHA40-RESCHA48 are set to generated sawtooth signal of 50Hz frequency to guarantee that every 10Hz all displayed values change.

The test setup can be summarized as follows:

- **Hardware:** Dell Latitude E6530 laptop (i7-3740QM processor, 2.7GHz),
- **Metrics:** Processor Load [%], Private Working Set [MB],
- **Test duration:** 10 minutes, 1 sample per second,
- **Browsers:** Google Chrome, Mozilla Firefox, IE11, Hosted IE11,
- **Current Visualization:** Comparison with the POI.

5.2 Alphanumeric Test Cases

For a first benchmark, the processor load and memory usage for a Web Window with 20 alphanumeric controls of the visualization library were measured. The PUMA Open™ 2 provides 9 reserve channels (RESCHA40-RESCHA48), which are not mapped by default to a real measurement value and can be set through an user configuration to a generated signal within the automation system, for example to a signal with a sine or rectangle waveform. These 9 channels are visualized by the 20 alphanumeric controls in the first benchmark, whereby the channels are displayed in order: 7 control pairs show RESCHA42-RESCHA48 and two triplets show RESCHA40 and RESCHA41.

As presented in Figure 5.1, the infrastructure with the layered architecture described in Chapter 3 was used and the update rate for the long polling in the Backend Access

layer (described in Section 3.3.5) was set to 100ms (10Hz). First performance measurements with the real ASAccessServer API implementation have shown that it can not be guaranteed that on each HTTP request, triggered by the client in the Backend Access layer, the ASAccessServer responds within the time interval. In order to avoid random measurement errors caused by these delays and to test only the client-side performance of the web visualization, the ASAccessServer API was mocked by a Node.js® test server [1], which generates random values for the 9 Signal data sources on each HTTP request (see Section 1.3). In this way, it can be guaranteed that for each request a new value is returned and no delays occur caused by the internal implementation of the ASAccessServer. For a comparison between the web visualization and the current visualization in the POI, the processor load and memory usage of the POI process was recorded. Hereby, the POI showed similar as in the test case for the web visualization a window with 20 alphanumeric controls showing the values of the reserve channels as described above.

The alphanumeric base setup can be summarized as follows:

- **Web Window:** 20 alphanumeric controls,
- **Data Source:** 9 Signal data sources, random values on each request, 10Hz polling rate,
- **Server:** Node.js® test server.

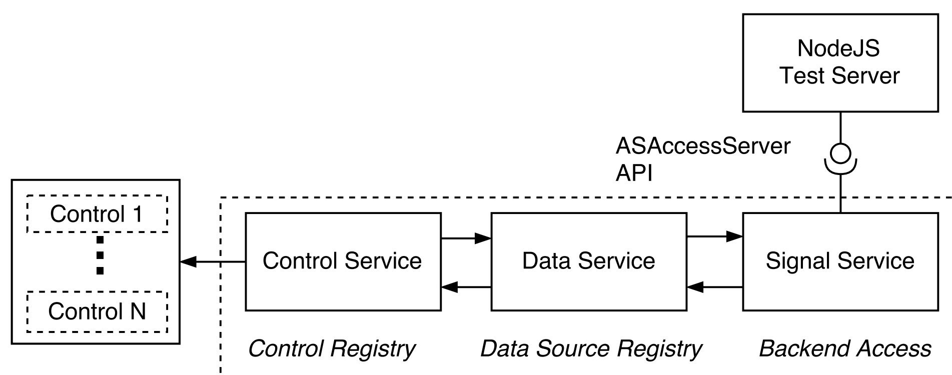

Figure 5.1: Base Setup for the Evaluation of the Web Visualization

As can be seen in Table 5.1 on page 77 for the base setup with 20 alphanumeric controls showing 9 Signal data sources with an update rate of 10Hz, the hosted Internet Explorer

[1] Node.js® JavaScript runtime engine: `https://nodejs.org`

performs best in terms of processor load (5.57% of one logical core) and memory usage (53.19MB RAM usage). This fact can be traced back that for the hosted Internet Explorer, that is the COM Web Browser Control, only the main functionality without the User Interface, for example the address bar, is used.

When comparing the figures of the current visualization (POI column) with the figures of the web visualization expressed by all other columns, the major drawback in terms of processor load of the web visualization is outlined. Hereby, the rendering technology in every web browser cannot compete with the rendering through the Windows API used by the current visualization in the POI.

5.2.1 Increasing the Update Rate

To evaluate the impact of the update rate on the processor load and memory usage of the web visualization, the update rate was increased starting with 3Hz up to the rate of the base test setup of 10Hz. When comparing the evolution of the processor load of the four web browsers in Figure 5.2, it can be seen again that the hosted IE outperforms for all update rates and shows a trend with a lower ascent towards higher update rates than the other web browser. Furthermore, the update rate of the visualization in the POI was similarly adapted in the poi.ini configuration file. Thereby, the processor load caused by the POI process remained stable over all test cases.

Table 5.1: Processor Load [%] as a Function of the Update Rate

Update Rate [Hz]	POI	Chrome	IE	Firefox	Hosted IE
3	0.02	4.88	5.52	4.74	2.05
5	0.12	7.91	9.46	10.41	4.75
8	0.02	13.01	15.64	14.78	5.89
10	0.01	16.45	19.23	18.14	5.57

The overall measurement results including the memory usage are summarized in Table 5.1 and Table 5.2. The memory usage of all web browsers and the POI process remain hereby almost constant and has only small fluctuations between the measurements. Similar as for the processor load, the hosted IE performs best and needs about the half of the RAM as the standalone IE. Especially, the lower memory usage can be explained as follows: The hosted IE only spawns a single process representing the content of the web page (render process), instead of several processes as for example the Google Chrome web browser does.

Table 5.2: Private Working Set [MB] as a Function of the Update Rate

Update Rate [Hz]	POI	Chrome	IE	Firefox	Hosted IE
3	43.68	211.87	109.95	224.39	45.80
5	43.57	213.23	127.89	197.21	51.74
8	43.94	210.14	118.94	198.39	51.90
10	43.63	209.94	134.22	200.64	53.19

5.2.2 Increasing the Number of Controls

In addition to the update rate, the number of alphanumeric controls visualized by one Web Window was increased and again the processor load and memory usage were collected. Figure 5.3 presents the linear rise of the processor load for all web browsers when increasing the number of controls inside one Web Window.

Similar to the measurement with the increasing update rate, the hosted IE utilizes the least processor time of all web browsers for the 20 controls and 40 controls test case. A further increase of the number of alphanumeric controls shows that the Firefox has about half of the processor load as the Internet Explorer (see also Table 5.3). Especially, the difference in the processor utilization of the web browsers when increasing the number of windows can be explained by the differences in the implementation details of the rendering engine of each web browser, that is Gecko for the Firefox, Blink as a fork from WebKit for the Google Chrome and Trident for the IE and hosted IE web browser (Crowley, 2010; The Chromium Projects, 2016a).

Table 5.3: Processor Load [%] as a Function of the Number of Controls

Controls	POI	Chrome	IE	Firefox	Hosted IE
20	0.01	16.45	19.23	18.14	5.57
40	0.04	24.55	35.41	23.60	20.36
80	1.35	35.88	64.53	33.49	50.64
100	3.92	47.66	81.47	38.96	64.07

When comparing the figures of the private working set used by the web browsers in Table 5.4, the memory usage fluctuates slightly between the measurements and the hosted IE needs again less RAM because of its simple process model.

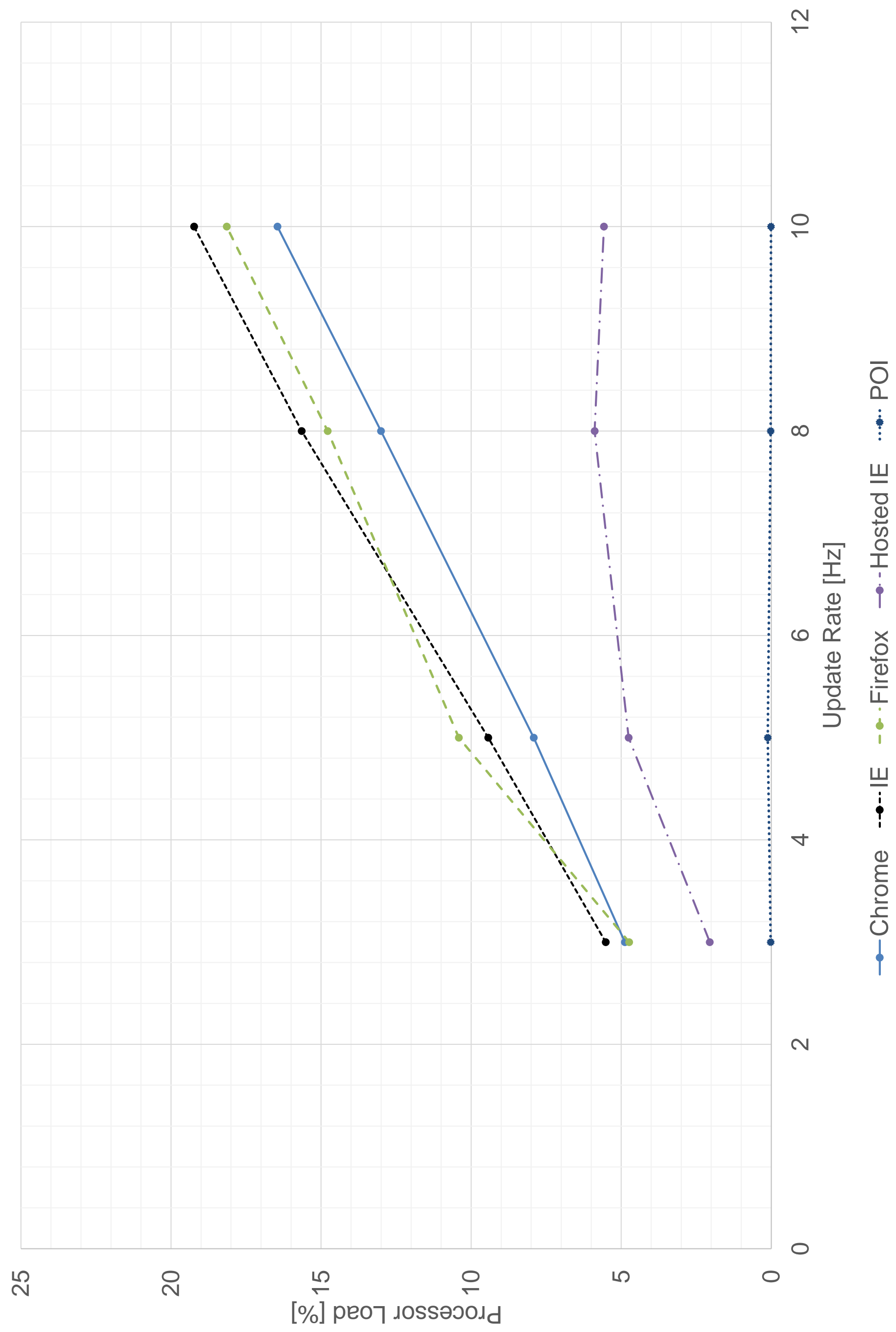

Figure 5.2: Processor Load as a Function of the Update Rate

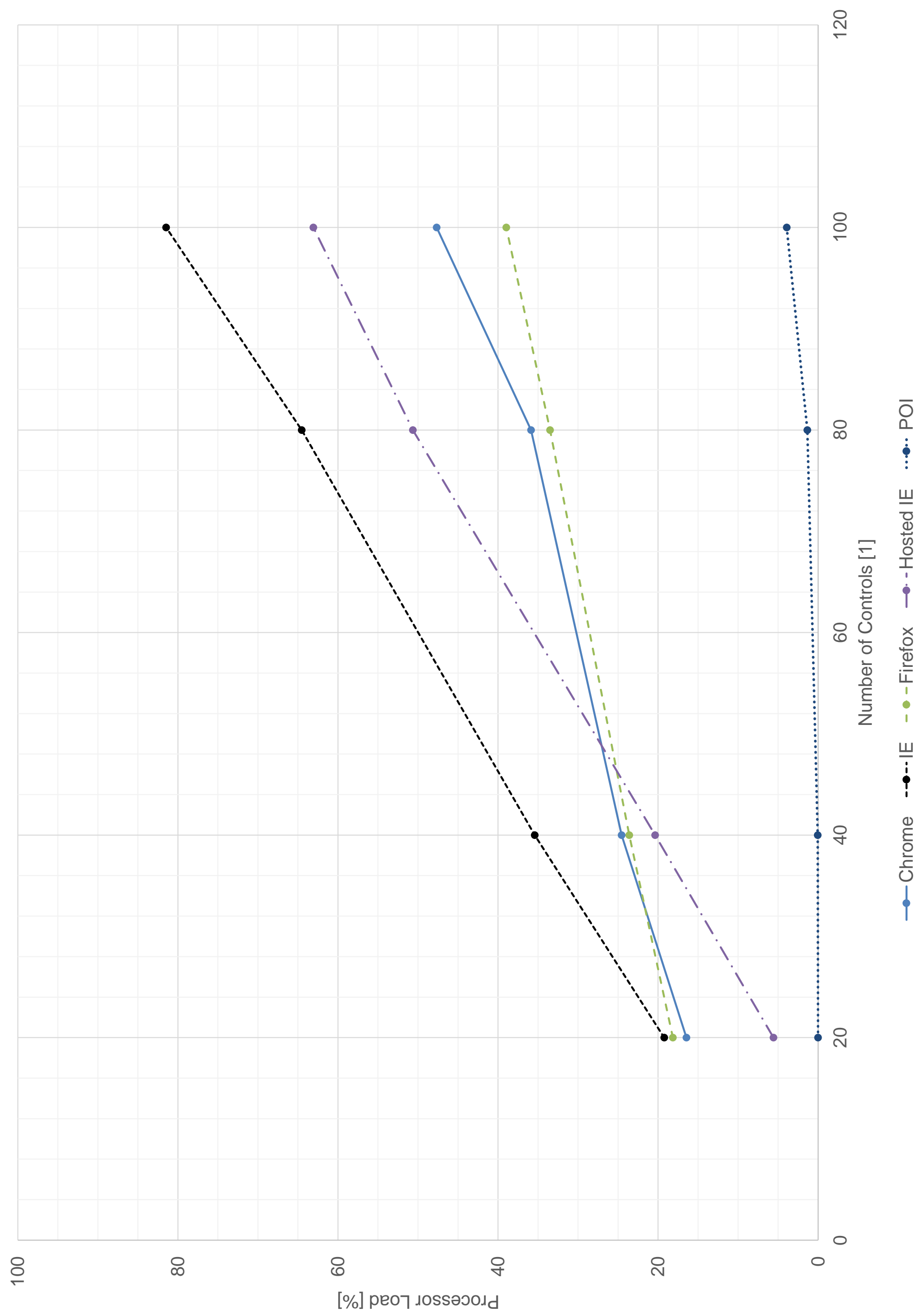

Figure 5.3: Processor Load as a Function of the Number of Controls

Table 5.4: Private Working Set [MB] as a Function of the Number of Controls

Controls	POI	Chrome	IE	Firefox	Hosted IE
20	43.63	209.94	134.22	200.64	53.19
40	52.28	226.33	109.80	189.79	65.70
80	52.35	238.66	120.32	211.51	78.65
100	51.41	248.65	76.33	249.58	65.27

5.2.3 Root-Cause Analysis

For a more detailed evaluation of the performance impact, a root-cause analysis was conducted in which parts of the infrastructure were either bypassed or modified towards a deeper understanding of the presented figures in Section 5.2.1 and Section 5.2.2. In a first step, the same base setup as described above was used, however, both the Data Service and Control Service were bypassed (see Figure 5.4). In this way, the impact of the dynamical binding of data sources to controls, described in Section 3.4.2, can be evaluated.

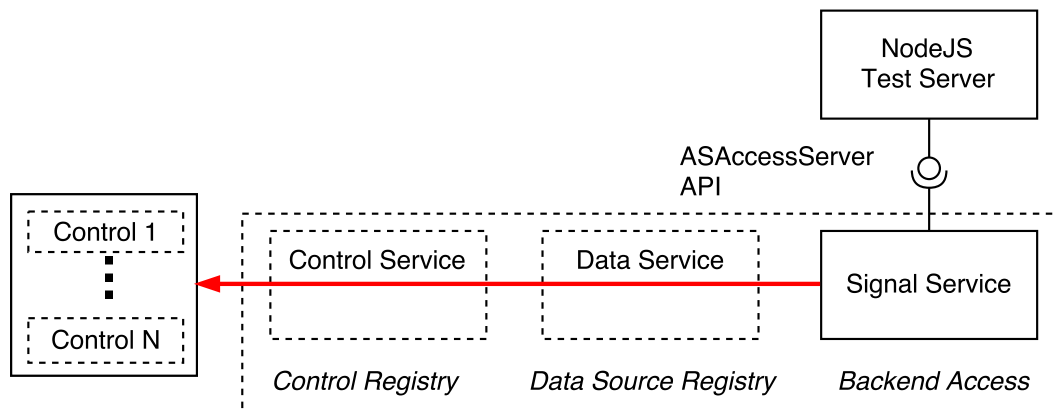

Figure 5.4: Bypassing the Control Service and Data Service

As a second test case, again the same setup was used, however, the Node.js® test server was omitted and the same 9 Signal data sources were generated on the client-side inside the Signal Service of the infrastructure (see Figure 5.5). In this way, the performance impact of the long polling in the Signal Service (see Section 3.3.5) can be evaluated.

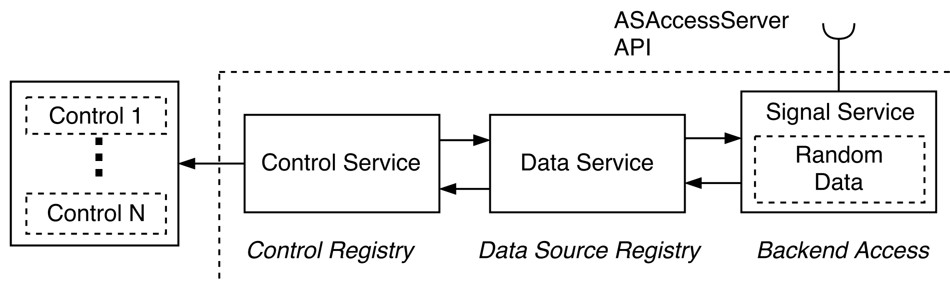

Figure 5.5: Random Data generated inside the Signal Service

Finally, the same base setup was used, however, instead of random values the Node.js® test server responded to each request with a constant value. Through this test case, it can be seen which part of the processor load is caused by the re-rendering of the DOM with the changed Signal data source values.

The three test cases can be summarized as follows:

- **Dynamic Binding:** Base test setup,
- **Static Binding:** Bypassed Control Service and Data Service,
- **No Long Polling:** Random values are generated inside Signal Service (no client-server interaction),
- **Constant Data:** Server responses with constant values.

All three test cases were conducted with 20 (base setup) and 100 alphanumeric controls inside a Web Window. For both test cases, the figures from the IE and the hosted IE deviate from the figures of the Chrome and the Firefox web browser. This fact can be explained through some functionality standardized in ECMAScript 2015 Version 6 (ES6) that is in contrast to the Chrome and Firefox web browser not implemented by the IE and hence, in the hosted IE, and must be provided by less performant polyfill implementations in JavaScript. For example the ES6 Map collection used often in the Data Service or Control Service to hold all data sources or a reference to all controls needs to be provided by a JavaScript Object implementation.

When taking the difference between the figures of the Dynamic Binding and Static Binding test case in Figure 5.6 and Figure 5.7, it can be concluded that about 5-8% of the processor load is caused by the dynamic assignment of data sources to control inputs implemented in the Control Service and the Data Service (Chrome: 5% for 20 and 6.7% for 100 controls, Firefox: 8.43% for 20 and 8.3% for 100 controls).

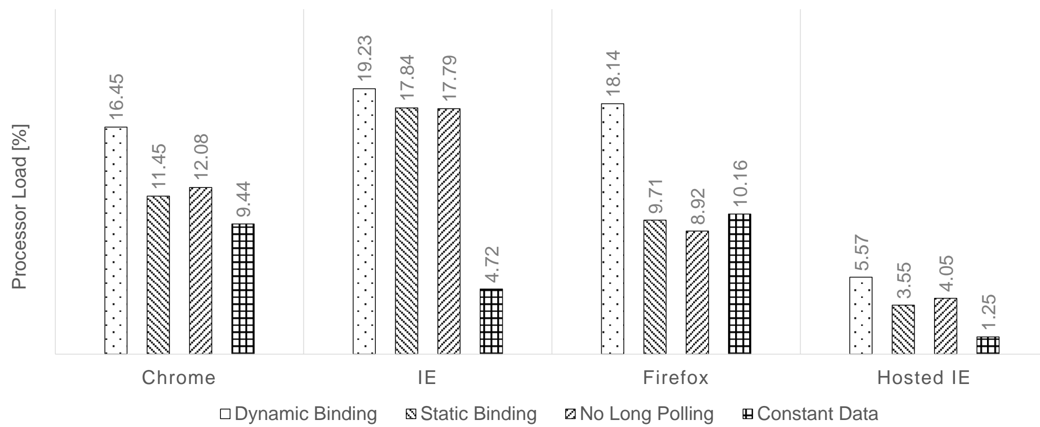

Figure 5.6: Root-Cause Analysis for 20 Controls

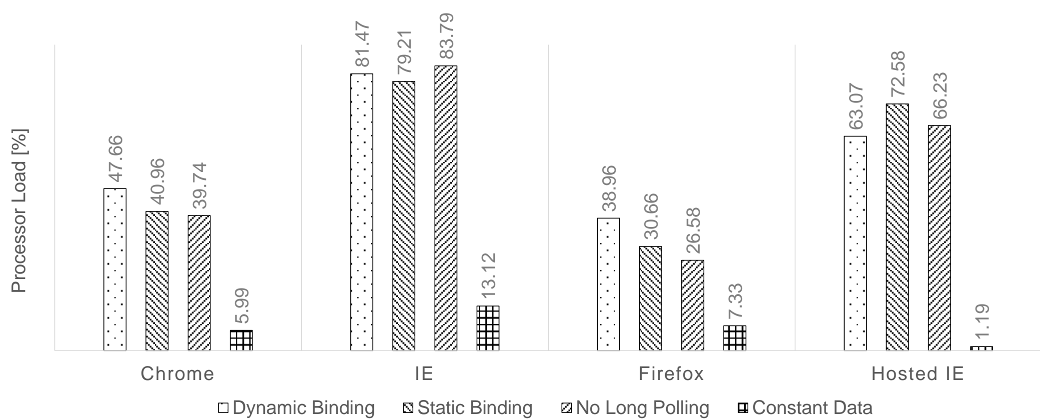

Figure 5.7: Root-Cause Analysis for 100 Controls

Almost the same portion of the processor load can be assigned to the long polling in the Signal Service, when setting the figures for the Dynamic Binding and for the No Long Polling test case to each other in relation (Chrome: 4.37% for 20 and 7.92% for 100 controls, Firefox: 9.22% for 20 and 12.38% for 100 controls).

Due to the change detection system of the Angular framework (see Section 2.1.4 and 2.2.3), for the Constant Data test case the received data from the server is compared with the previous data stored into the Angular Component tree and hence, the DOM is

not updated. This causes the minor fraction of the processor load presented in Figure 5.6 and Figure 5.7 when only constant data is sent by the server and outlines, when the figures are compared with a test case in which the data source values changes periodically, the part of the web visualization which causes the most processor load, the re-rendering of the DOM with new data source values.

5.3 Framework Evaluation

Further measurements were conducted to evaluate the impact of the used framework on the processor load of the automation system. For this reason, the infrastructure, as presented in Chapter 3 was re-implemented using the React JavaScript library developed by the Facebook team (see Section 2.1.4). The alphanumeric control of the visualization library (see Chapter 4) was re-implemented by a React Component and all Angular Services such as the Data Service or Control Service were implemented by ES6 classes to use the services in combination with the React library.

In addition, a basic example was implemented in JavaScript without any external library or framework based on the native web browser API and shall outline the processor load of the three basic operations without a framework or library:

- **HTTP:** Trigger the HTTP request and process the response of the server.
- **Control search:** Find the DOM element of the control in the DOM tree.
- **Control update:** Update the part of the DOM to visualize the new content by the found control.

The alphanumeric control was constructed in memory by the DOM API defined in Hors et al. (2000), for example `document.createElement()`, when the page was initially loaded and was reused to fill the DOM with 20 controls or 100 controls. To identify later each control inside the DOM, an unique identifier was assigned as a HTML attribute to the control when it is created. For a basic HTTP functionality, the XMLHttpRequest object was used available in all web browser, that is to retrieve the same 9 Signal data sources via the ASAccessServer API with an update rate of 10Hz as in the base setup described in Section 5.1. On each received response from the Node.js® test server, the DOM element representing a control was found in the DOM by the `document.getElementById()` method. The received value was then updated to the control or DOM element, respectively, setting the value to its `element.textContent` property.

In Figure 5.8 and Figure 5.9 the measurement results of the base setup are presented, where it can be seen that no major difference especially for the Chrome web browser

between the Angular framework and React library in terms of processor load exists. When comparing the figures from the Angular framework or the React library with the pure JavaScript implementation, the figures confirm the conclusion drawn from measurements in Section 5.2.3, that most of the processor load is caused by the rendering of the DOM.

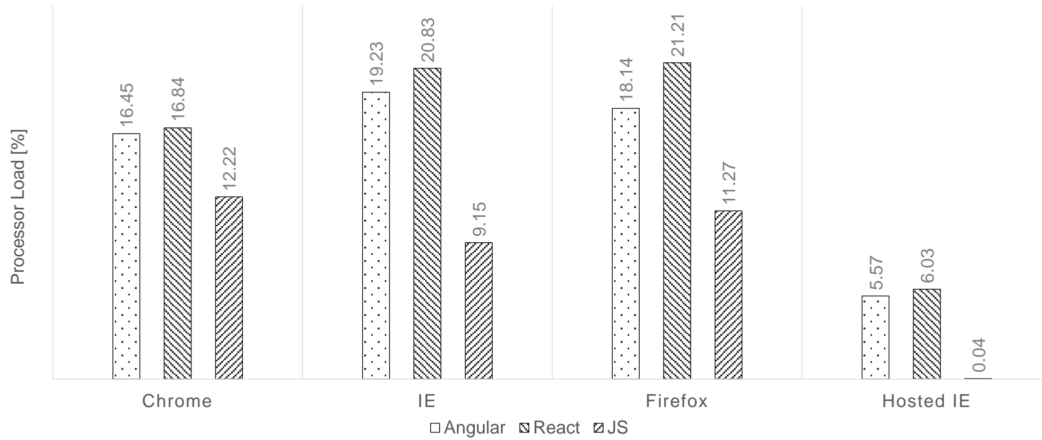

Figure 5.8: Comparison between Frameworks with 20 Controls

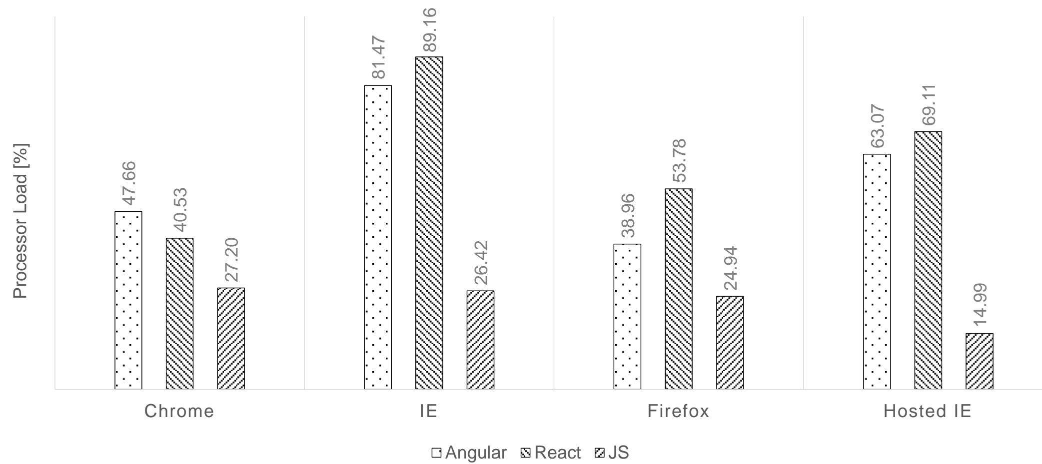

Figure 5.9: Comparison between Frameworks with 100 Controls

5.4 Hardware Rendering

For an evaluation of the influence of the hardware rendering, that is the compositing and rasterization supported by the GPU described in Garsiel and Irish (2011), on the processor load, the hardware rendering was turned off for all web browsers and the same base test was run for 20 and 100 alphanumeric controls (see Section 5.1). In the Firefox, Chrome and IE web browser, an explicit setting exists to turn off the hardware rendering and to use software rendering instead. For the hosted IE, the relevant registry key described in Section 5.1 must be set to zero.

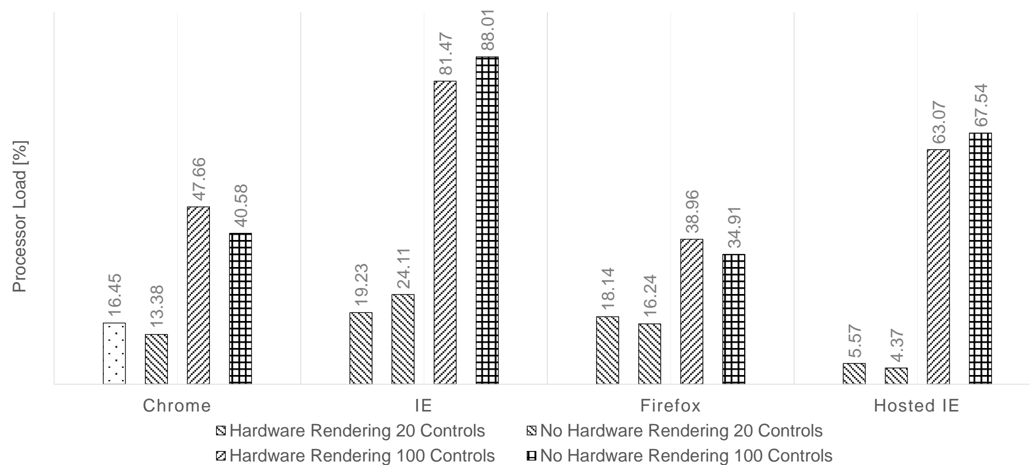

Figure 5.10: Influence of Hardware Rendering on the Processor Load

As can be seen in Figure 5.10, for the Chrome and the Firefox web browser the hardware rendering brings no benefits in terms of processor load, quite the contrary, the processor is used from two up to seven percent less when the software rendering is used instead of the hardware rendering. This can be explained as follows: For each new Signal data source value only the DOM text element of the alphanumeric control needs to be updated and hence, the process of compositing and rasterization offloaded to the GPU can not be fully utilized. In this way, especially for the Chrome web browser, it needs more processor load to keep the separate GPU process alive and enable the Inter-Process Communication (IPC) between the GPU process and the render process than to rasterize the updated bitmap entirely in the Central Processing Unit (CPU) (The Chromium Projects, 2016c; Wiltziusa, Kokkevis, and Chrome Graphics team, 2014).

The Internet Explorer, particularly, can exploit the support of the hardware and uses about five percent less of the processor when the hardware rendering is turned on, which can be a result of the better integration of the platform specific Microsoft Windows DirectX graphics API used to facilitate the graphic card (Internet Explorer Team Blog, 2010).

5.5 Xt-Control Test Cases

To evaluate the performance impact of a xt-control from the visualization library (see Chapter 4.3), a test setup was run similar to the alphanumeric test cases described in Section 5.1. Therefore, the Node.js® test server is again polled by the Signal Service in the Backend Access layer with an update rate of 10Hz.

In the visualization library described in Chapter 4.3, two xt-controls were implemented using the SVG and the HTML5 Canvas drawing technology. Both xt-controls visualizes the evolution of two Signal data sources obtained from the Node.js® test server via the Control Service and Signal Service. Therefore, the data source values are first stored in a ring buffer with a fixed size of 100 samples inside both xt-controls. Then, the control is updated with the content of the ring buffer. Furthermore, the hardware rendering was disabled similar as in Section 5.4 to present the impact of the hardware acceleration on the processor load for both xt-controls.

Similar as for the alphanumeric test cases, in the POI desktop application a window was created showing a multi-series line chart with the same two channels as in the web visualization. The number of displayed values were set again to 100 samples and the update rate of the POI was configured to 10Hz. The POI process showing the multi-series line graphic caused a processor load of 0.01% and needed 55.90MB RAM.

In Figure 5.11 the processor load of the xt-control using SVG for drawing the two Signal data sources is compared against the xt-control using a HTML5 Canvas element. The xt-control using the SVG technology adds an additional SVG element for every update of a Signal data sources, for example a circle for a data point, and hence, a DOM element needs to be created and added to the existing DOM elements for each request, that is every 100ms. In addition, all existing SVG DOM elements must be updated to visualize in combination with the new DOM element the evolution of the Signal data source.

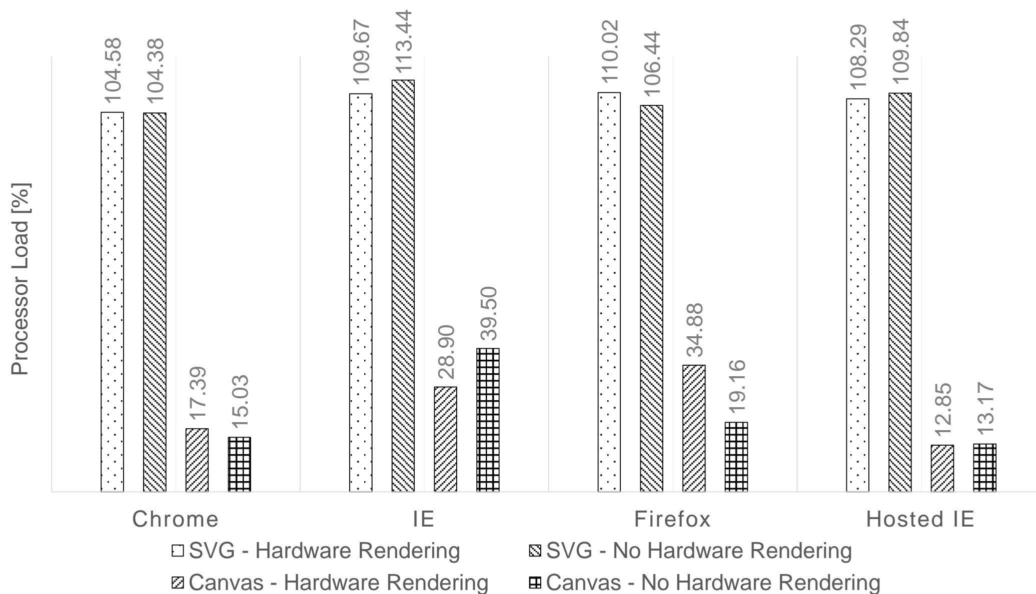

Figure 5.11: Difference between SVG and Canvas Technology

In contrast to that, the Canvas xt-control exists only of one HTML Canvas element providing a raster graphics image to which the evolution of the Signal data source is drawn pixel wise. The impact of the computational expensive DOM operations for the SVG xt-control in relation to the simple Canvas painting, can be seen in Figure 5.11, where the SVG xt-control needs up to six times more processor time than the Canvas xt-control. Additionally, similar as in the test case described in Section 5.4, the hardware rendering could not improve the processor load. Again, only the Internet Explorer can exploit the hardware rendering and hence, the processor was utilized about 5-10% less when the hardware acceleration was turned on.

5.6 Test Cases on a Mobile Device

In addition to the test cases described above, similar test cases were run on a mobile device to evaluate the performance impact of the web visualization when it is used for remote monitoring of the actual testbed state. For this reason, the base test case with 20 alphanumeric controls described in Section 5.2 was run on a Huawei Honor 5X smartphone with a Qualcomm Snapdragon 616 processor with 1.6GHz (8 cores) and

2GB RAM. The latest Android 6.0 Marshmallow operating system was installed on the used smartphone and all test runs were performed with the Google Chrome for Android v54 and the Mozilla Firefox for Android 50 web browser. The Android Debug Bridge (ADB) tool was used to record the total processor load and the memory usage of all relevant processes of the web browser with a sampling interval of one second. Same as for the test setup on the desktop computer, the performance figures of all processes of each web browser were summarized and averaged over a time period of 10 minutes. On the Android operating system, the resident set size (RSS) stands for the RAM used by each process and was here used to outline the actual memory usage of the Chrome and Firefox web browser.

The test setup can be summarized as follows:

- **Hardware:** Huawei Honor 5X Smartphone (Qualcomm Snapdragon 616 processor, 1.6GHz),
- **Metrics:** Processor Load [%], Resident Set Size [MB],
- **Test duration:** 10 minutes, 1 sample per second,
- **Browsers:** Google Chrome for Android, Mozilla Firefox for Android.

5.6.1 Alphanumeric Test Cases

In addition to the base test case, the number of controls per web window and the update rate was increased. In Table 5.5 and Figure 5.13, the evolution of the processor load when increasing the update rate is shown. Hereby, similar as in the test cases performed on a desktop computer the processor load increases linearly with the processor load. Because of the fact that on a mobile phone only one browser tab can be simultaneously active, a maximum processor load of all test cases of about 12% is acceptable for the remote monitoring. As already seen in the previous test cases, the memory usage expressed by the RSS figures fluctuates slightly between the test cases and does not indicate the actual working load of the test case.

Table 5.5: Processor Load and Resident Set Size as a Function of the Update Rate

Update Rate	**Processor Load [%]**		**RSS [MB]**	
	Chrome	Firefox	Chrome	Firefox
3	2.17	3.16	330.75	297.10
5	4.02	6.85	340.23	265.44
8	5.45	9.89	412.25	283.56
10	9.56	11.81	394.34	280.21

As can be seen in Table 5.6 and Figure 5.14, the processor load of both web browsers increases linearly until the test case with 40 controls for the Firefox and with 80 controls for the Chrome web browser. When increasing the number of control per web window further, the processor load saturates at about 15%. This can be explained as follows: When the number of controls exceeds a certain limit, the JavaScript engine cannot update all controls during the 100ms time span until the next request is triggered. Therefore, not every control is updated during the time span given by the 10Hz update rate. Because of the fact, that the most expensive operation in terms of performance is the re-rendering of the DOM or here, the DOM elements showing the actual value of the data source, the processor load increases not further when additional controls are added.

Table 5.6: Processor Load and Resident Size as a Function of the Number of Controls

Controls	**Processor Load [%]**		**RSS [MB]**	
	Chrome	Firefox	Chrome	Firefox
20	9.56	11.81	394.34	280.21
40	11.46	14.96	527.63	293.25
80	15.00	15.29	659.02	267.66
100	15.17	15.10	714.13	299.02

5.6.2 Xt-Control Test Cases

In addition to the alphanumeric test cases, the same test cases using the multi-series line charts of the visualization library are performed on a smartphone. Therefore, the two implementation of the multi-series line chart were configured to show the evolution of two data sources obtained from the test server with an update rate of 10Hz.

Although Figure 5.12 shows no major difference in terms of performance between the two implementations using the SVG and Canvas technology, the same problem exists as in the alphanumeric test cases for the xt-control using the SVG technology. Similar as for the test cases with more than 40 alphanumeric controls, the JavaScript engine cannot update the position of all 100 displayed data points of the SVG xt-control during the 100ms time frame. This causes a sluggish visualization of the data source evolution in the xt-control using the SVG technology. In contrast to that, the simple pixel drawing of the xt-control using the Canvas technology updates every 100ms and shows a continuous representation of the data source evolution.

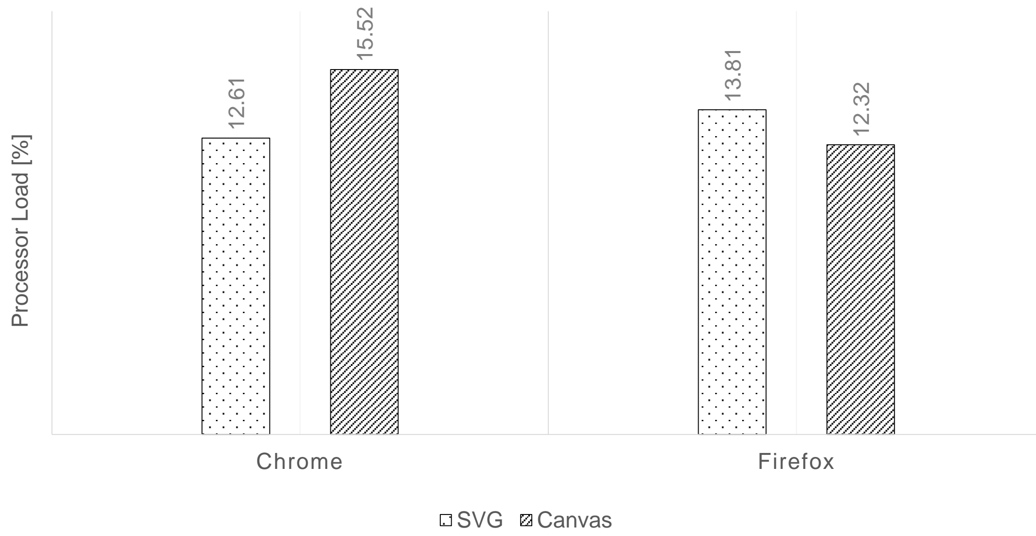

Figure 5.12: Difference between SVG and Canvas Technology on a Mobile Device

Both test cases using the Chrome and the Firefox web browser show with an maximum processor load of about 15% that the xt-control using the Canvas technology can be used on a mobile phone for remote monitoring of the current testbed state.

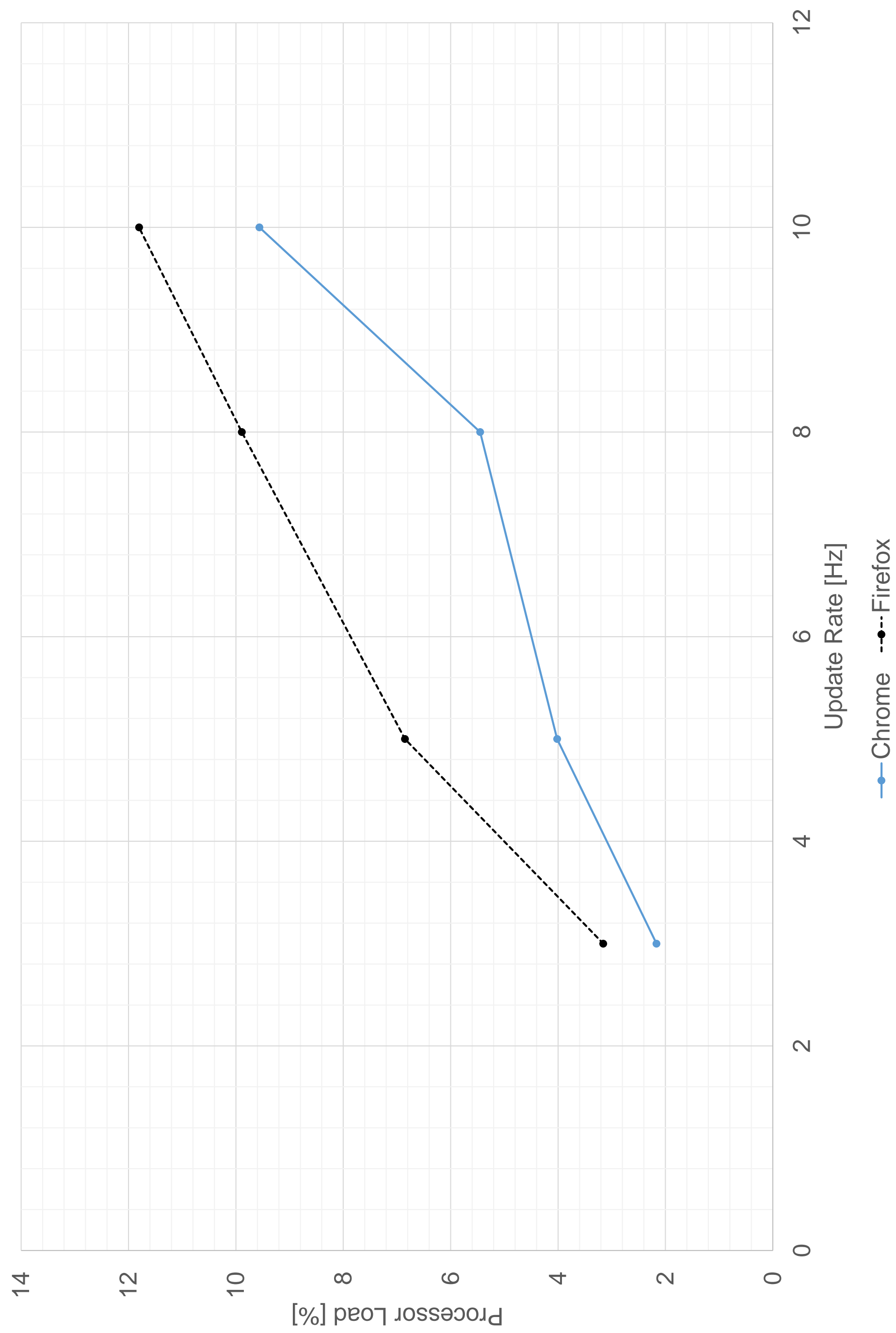

Figure 5.13: Processor Load as a Function of the Update Rate

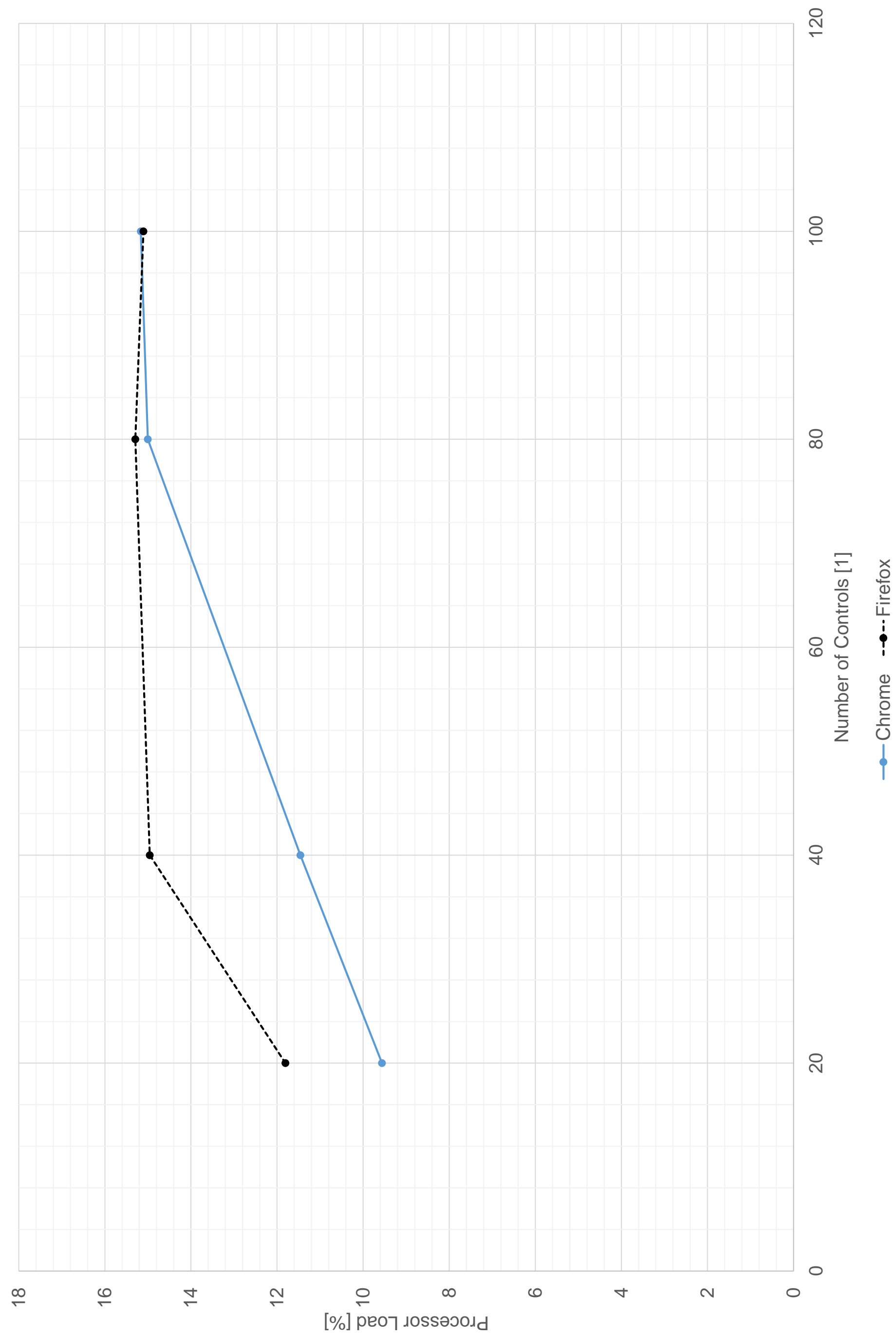

Figure 5.14: Processor Load as a Function of the Number of Controls

6 Conclusion

In the course of this work, a state-of-the-art SPA framework was used to extend the UI of the POI with a web visualization. An infrastructure was established that enables to structure the web visualization similar as in the POI in separate windows each stored in a single file and denoted here as a Web Window. The concept of independent Web Windows enabled the reuse of existing visualizations across multiple testbeds. Furthermore, through the implemented infrastructure the testbed operator can adapt the visualization described by a Web Window to the current testbed setup and test run. In this sense, the testbed operator can assign different data sources retrieved from the automation system to a control inside the Web Window.

In addition, a solution was realized to run simple presentation logic inside the infrastructure for example for limit monitoring or unit conversion. To avoid a blocking of the single-threaded JavaScript execution, all presentation logic was swapped out in a separate Web Worker thread. In addition, predefined presentation logic stored in JavaScript files can be included, which allows a reuse of logic such as unit conversion across several Web Windows. Modal dialogs were implemented to change the assigned data source to a control and to view or edit the custom presentation logic during the runtime of the SPA. In addition, all changes done by the testbed operator can be stored back into the corresponding Web Window by a dedicated API of the automation system server. A functionality was implemented to insert existing Web Windows inside a new Web Window to realize for example a tab control where each tab shows one Web Window. An important point to note is that every Web Window is independent of the enabling infrastructure and thus, gives the possibility to extend the functionality or replace the enabling technology for example the used SPA framework without an adaption of the Web Windows designed by the testbed operator.

As a basis for the web visualization, a concept for a visualization library was established for web applications based on the Angular 2 SPA framework. A layered architecture was proposed to enable the inclusion of third-party controls inside the visualization library. The Dependency Inversion Principle described in Section 2.3.1 was facilitated to decouple changes of the third-party controls from the web application, which acts as the consumer of the library. The current state of the visualization library includes nine controls, such as

an alphanumeric, a bar chart, a custom button, a gauge, a custom text input, a warning indicator, for example for limit monitoring, and two xt-controls, for multi-line charts using the SVG or the HTML5 Canvas technology. In addition, two forms were implemented to show either the current testbed status or a rendered picture of a POI window without the need of an additional infrastructure.

For an impact analysis of the web visualization on the testbed, several test cases were run and the processor load and the private working set was measured. For the base test case with a 10Hz update rate and 20 alphanumeric controls showing 9 channels of the testbed, the Firefox, Chrome, and Internet Explorer (IE) needed between 15-20% processor load of one logical core. The Web Browser Control denoted here as hosted IE, which provides a COM interface for parts of the Internet Explorer, consumed about 6% processing time of one logical core (see also Section 5.2). Further test cases showed a linear dependency of the processor load on the update rate and the number of controls per Web Window. For all test cases, the private working set remained almost stable at about 200MB for the Firefox and Chrome web browser and at about 60-120MB for the IE and the hosted IE. The stable amount of the private working set could be traced back that every web browser allocates at startup a dedicated amount of memory independent of its actual workload. When comparing the figures of the web visualization with the native visualization of the POI (see Table 5.1 and Table 5.3 on page 77 and 78), the web browsers showing the web visualization cause up to 100% more processor load as the native visualization. In terms of memory usage, the native visualization showed similar figures as for the hosted IE (see Table 5.2 and Table 5.4).

Additionally, a comparison between an implementation of the infrastructure using Angular 2 and React showed no major performance difference of the change detection system explained in Section 2.2 for all test cases. A root-cause analysis revealed that between 5-8% of the processor load is caused by the dynamic assignment of data sources implemented in the infrastructure and almost the same amount is needed for the long polling of the automation system server. Through a comparison of a test case with retrieved constant data, it could be concluded that the major part of the processor load is caused by the re-rendering of the DOM. Furthermore, the impact analysis highlighted that the support of the GPU for the rendering of the DOM cannot improve the processor load. Contrarily, the separate thread spawn for the hardware rendering caused an increase of the total processor load. The implementation of the multi-line chart using the SVG and Canvas technology demonstrated the major advantage of the later technology in terms of processor load (see Section 5.5).

As an overall conclusion of the performance measurements, the increase of the processor load when using the web visualization instead of the native visualization needs to be

taken into consideration. Although a reduced processor load of the web visualization can be expected when using a more realistic update rate of about 5Hz, the same performance figures cannot be expected as for the native visualization in the POI. Therefore, in a first phase the web visualization can only, as originally planned, act as an extension of the current visualization. Hereby, the web visualization could provide, for example, a state-of-the-art representation of parts of the overall visualization of the automation system. Furthermore, it must be taken into consideration that the technology enabling the web visualization was initially designed for a simple exchange of information in form of hypertext documents and not for real-time monitoring applications. For this reason, the same concepts as currently implemented in the native visualization cannot be integrated to the web visualization without considering the resulting performance impact based on the measurements presented in this work. To prevent any performance impact on the automation system, the infrastructure could limit the number of controls per Web Window and the number of Web Windows, which can be opened in parallel.

Finally, the current proposed infrastructure in combination with the visualization library simplifies the monitoring of the testbed status from a remote location and eases in this way the support in case of an error of the automation system. Furthermore, the build setup established for the infrastructure and gathered knowhow can be reused for future projects and thus, accelerates the development of a web application across the AVL List GmbH.

7 Future Prospects of the Web Visualization

The current solution implemented in this master's thesis is proposed to work as an extension of the visualization of the POI. Thus, the controls, forms and modal dialogs of the visualization library and infrastructure are implemented for the use on a desktop computer and thus, are not optimized for smaller screen sizes. Future developments should address this issue to have a better visual experience on mobile devices. In addition to the necessary adaptations of the modal dialogs for smaller screen sizes, a hybrid web application could be implemented mentioned in Section 2.1.3 and could benefit in this way from the same JavaScript code base as for the web visualization in the POI. Such a hybrid web application could track the status of several testbeds in a test field and could allow to notify the testbed engineer immediately if an error has occurred. In this way, costly idle time of testbeds could be avoided.

Currently, a predefined set of Web Windows exists and the testbed operator can either adapt the visualization during the runtime or edit the Web Window file in a text editor. For a more user-friendly approach to create custom visualizations, a designer could be implemented which provides an UI to create a Web Window. The designer could include a tool bar, which lists all controls of the visualization library, and a region where controls out of the tool bar can be placed. Furthermore, the designer could support the assignment of real data sources retrieved from a testbed to the controls and the generation of a Web Window file out of the design process.

In addition, the current implementation lacks the integration of a test framework described in Section 2.1.3 and the complete coverage of the functionality by system and unit tests. Future works could add a test framework to the infrastructure and the visualization library and implement system and unit tests towards a test-driven development.

Through the established retrieval of JavaScript third-party modules via the node package manager [1] and the inherited dependency management, it is difficult to trace back the

[1] Node Package Manager: `https://npmjs.com`

exact version of all modules at a specific point of time. This fact could lead to problems when it is necessary to restore a specific state of the code base to fix errors in a deployed version of for example the visualization library. More advanced mechanisms of dependency management introduced by the yarn package manager [2] could solve this problem.

Although the Web Browser control denoted in this work as hosted IE performed well in performance tests, the underlying Internet Explorer does not support important parts of the new HTML5 features and web standards, such as the Web Components standard. Furthermore, the development of the Internet Explorer is discontinued and security updates will not be deployed in near future by Microsoft. For this reasons, the hosted IE should be replaced by a state-of-the-art web browser that can be embedded in desktop applications, such as the Chromium Embedded Framework (CEF), in favor of a better support of all web standards inside the POI. In addition to the better support of web standards, the use of the CEF could help to improve the processor load and memory usage of the web visualization. Because of the fact that the latest version of the Chromium web browser integrated in the CEF supports the SharedWorker standard, parts of the enabling infrastructure, such as the long polling of the ASAccessServer in the BackendAccess layer, could be moved to one shared thread. These parts could then be shared across all open Web Windows and hence, the overall processor load of the web visualization could be decreased.

Furthermore, the Chromium web browser wrapped by the CEF could be configured in a way that similar as the hosted IE only one process is started instead of the three processes (see Section 5.1). This configuration would decrease the memory usage of the Chromium web browser to the same amount as for hosted IE or the native visualization in the POI. Another optimization in terms of processor load could be achieved through the replacement of the long polling in the BackendAccess layer by the WebSocket protocol. Hereby, the ASAccessServer could notify any client, that is the web visualization, when a value of a PUMA resource has changed, and hence, the processor load caused by the long polling in any client could be avoided.

Currently, Signal data sources provided by the ASAccessServer, as depicted in Figure 1.5 on page 5 can be obtained from any client which is in the same network. Each activation object needs to be explicitly configured if it should be provided through the Function interface of the ASAccessServer. Future developments could implement authentication and authorization methods to enable the access of PUMA resources also from devices, which are not in the same network as the PUMA testbed.

[2] Yarn: Reliable and secure package manager `https://yarnpkg.com/`

Bibliography

Angular Team (2016). *One Framework. - Angular 2*. URL: https://angular.io/ (visited on 11/18/2016).

Archibald, Jake (2015). *Tasks, microtasks, queues and schedules*. URL: https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/.

AVL List GmbH (2016a). *Product Brochure Puma Open 2*. URL: https://avl.com/documents/10138/2095827/Product+brochure+Puma+Open+2 (visited on 11/18/2016).

— (2016b). *Solutions Catalogue 2016*. URL: https://avl.com/html/static/emag/Solutionskatalog2016_EN/index.html (visited on 11/18/2016).

Bagwell, Phil (2001). "Ideal Hash Trees." In: 1195.

Barker, Tom (2012). "Web Performance Optimizations." In: *Pro JavaScript Performance*. Springer, pp. 109–137. ISBN: 978-1-4302-4749-4.

Berners-Lee, Tim, Roy Thomas Fielding, and Larry Masinter (2005). *Uniform Resource Identifier (URI): Generic Syntax. IETF RFC 3986*. URL: https://tools.ietf.org/html/rfc3986 (visited on 11/18/2016).

Bille, Philip (2005). "A survey on tree edit distance and related problems." In: *Theoretical Computer Science* 337, pp. 217–239.

Boduch, Adam (2016). *Flux Architecture*. Packt Publishing. ISBN: 978-1-78646-581-8.

Chedeau, Christopher (2013). *React's Diff Algorithm*. URL: http://calendar.perfplanet.com/2013/diff/ (visited on 11/18/2016).

— (2014). *React Architecture*. URL: http://blog.vjeux.com/2014/javascript/react-architecture-oscon.html (visited on 11/18/2016).

Chromium Team (2016). *Blink Workers - Current architecture, implementation and future project ideas for Blink Workers*. URL: https://docs.google.com/document/d/1i3IA3TG00rpQ7MKlpNFYUF6EfLcV01_Cv3IYG_DjF7M/edit#heading=h.7smox3ra3f6n (visited on 11/18/2016).

Craven, Adam (2016a). *Fundamentals of Enterprise-Scale Behaviour-Driven Development (BDD)*. URL: http://methodsandtools.com/archive/entreprisebdd.php (visited on 11/18/2016).

— (2016b). *The Behaviour Specification Handbook*. URL: https://devzone.channeladam.com/library/bdd-behaviour-specification-handbook/ (visited on 11/18/2016).

Crowley, Matthew (2010). *Pro Internet Explorer 8 Development*. Apress. 446 pp. ISBN: 978-1-4302-2853-0.

Daggett, Mark E. (2013). *Expert JavaScript*. 1st. Apress. ISBN: 978-1-4302-6097-4.

De, Swarnendu (2014). *Backbone.js Patterns and Best Practices*. Packt Publishing. ISBN: 978-1-78328-357-6.

Dotzler, Asa and Brad Lassey (2016). *Mozilla Blog - What's Next for Multi-process Firefox*. URL: https://blog.mozilla.org/futurereleases/2016/08/02/whats-next-for-multi-process-firefox/ (visited on 11/18/2016).

Eastlake, D. and A. Panitz (1999). *Reserved Top Level DNS Names*. URL: https://tools.ietf.org/pdf/rfc2606.pdf (visited on 11/18/2016).

Eckstein, Robert, Marc Loy, and Dave Wood (1998). *Java Swing*. O'Reilly. ISBN: 978-1-56592-455-0.

Ecma International (2015). *ECMAScript® 2015 Language Specification. Standard ECMA-262 6th Edition*. URL: http://ecma-international.org/ecma-262/6.0/ (visited on 11/18/2016).

Eisenberg, Rob (2016). *On Angular 2 and HTML Spec Compliance*. URL: http://eisenbergeffect.bluespire.com/on-angular-2-and-html/ (visited on 11/18/2016).

Elliott, Eric (2014). *Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries*. 1st. O'Reilly. ISBN: 978-1-4919-5029-6.

Evans, Eric (2014). *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, LLC. ISBN: 978-1-4575-0119-7.

Facebook (2016a). *Flux Online Documentation*. URL: https://facebook.github.io/flux (visited on 11/18/2016).

— (2016b). *Immutable.js - Immutable collections for JavaScript*. URL: https://facebook.github.io/immutable-js/ (visited on 11/18/2016).

Fain, Yakov, Victor Rasputnis, and Anatole Tartakovsky (2014). *Enterprise Web Development*. O'Reilly. 609 pp. ISBN: 978-1-4493-5681-1.

Fang, Tingting and Risto Lahdelma (2016). "Optimization of combined heat and power production with heat storage based on sliding time window method." In: *Applied Energy* 162, pp. 723–732. DOI: 10.1016/j.apenergy.2015.10.135.

Fender, Joe and Carwin Young (2015). *Front-End Fundamentals*.

Fette, I. and A. Melnikov (2011). *The WebSocket Protocol*. URL: https://tools.ietf.org/pdf/rfc6455.pdf (visited on 11/18/2016).

Fielding, Roy Thomas (2000). "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis.

Flanagan, David (2011). *JavaScript: The Definitive Guide*. O'Reilly. XVI pp. ISBN: 978-0-596-80552-4.

Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison Wesley. ISBN: 978-0-321-12742-6.

— (2004). *Inversion of Control Containers and the Dependency Injection pattern*. URL: http://martinfowler.com/articles/injection.html%5C#ServiceLocatorVsDependencyInjection (visited on 11/18/2016).

— (2007). *Mocks Aren't Stubs*. URL: http://martinfowler.com/articles/mocksArentStubs.html (visited on 11/18/2016).

Freeman, Adam (2014). *Pro AngularJS*. 1st. Apress. ISBN: 978-1-4302-6448-4.

Gackenheimer, Cory (2015). *Introduction to React*. Apress. ISBN: 978-1-4842-1245-5.

Gamma, Erich et al. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0-201-63361-0.

Garrett, Jesse James (2005). *Ajax: A New Approach to Web Applications*. URL: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/ (visited on 11/18/2016).

Garsiel, Tali and Paul Irish (2011). *How Browsers Work: Behind the scenes of modern web browsers*. URL: https://html5rocks.com/en/tutorials/internals/howbrowserswork (visited on 11/18/2016).

Gechev, Minko (2016a). *Switching to Angular 2*. Packt Publishing. ISBN: 978-1-78588-620-1.

— (2016b). *ViewChildren and ContentChildren in Angular 2*. URL: http://blog.mgechev.com/2016/01/23/angular2-viewchildren-contentchildren-difference-viewproviders/ (visited on 11/18/2016).

Group, Web Hypertext Application Technology Working (2016). *HTML Living Standard*. URL: https://html.spec.whatwg.org/ (visited on 11/18/2016).

Hickson, Ian et al. (2016). *HTML5, A vocabulary and associated APIs for HTML and XHTML, W3C Recommendation 28 October 2014*. URL: https://w3.org/TR/html5/ (visited on 11/18/2016).

Hohpe, Gregor and Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional. ISBN: 978-0-321-20068-6.

Hors, Arnaud Le et al. (2000). *W3C Document Object Model Core*. URL: https://w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html#ID-2141741547 (visited on 11/18/2016).

Intel (2016). *7th Generation Intel® Processors i7 Datasheet Volume 1 of 2*. URL: http://intel.com/content/www/us/en/processors/core/7th-gen-core-family-mobile-u-y-processor-lines-datasheet-vol-1.html (visited on 11/18/2016).

Internet Explorer Team Blog (2010). *The Architecture of Full Hardware Acceleration of All Web Page Content*. URL: https://blogs.msdn.microsoft.com/ie/2010/09/10/the-architecture-of-full-hardware-acceleration-of-all-web-page-content/ (visited on 11/18/2016).

Jansen, Remo H. (2016). *The current state of dependency inversion in JavaScript*. URL: http://blog.wolksoftware.com/the-current-state-of-dependency-inversion-in-javascript (visited on 11/18/2016).

Jbanov, Yegor and Tobias Bosch (2016). *Design Document: Angular 2 Rendering Architecture*. URL: https://docs.google.com/document/d/1M9FmT05Q6qpsjgvH1XvCm840yn2eWEg0PMskSQz7k4E/edit (visited on 11/18/2016).

Kambona, Kennedy, Elisa Gonzalez Boix, and Wolfgang De Meuter (2013). "An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications." In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. DYLA '13. ACM, 3:1–3:9. ISBN: 978-1-4503-2041-2. DOI: 10.1145/2489798.2489802. URL: http://doi.acm.org/10.1145/2489798.2489802.

Kelonye, Mitchel (2014). *Mastering Ember.js*. Packt Publishing. ISBN: 978-1-78398-198-4.

Knol, Alex (2013). *Dependency Injection with AngularJS*. Packt Publishing. ISBN: 978-1-78216-656-6.

Krasner, Glenn E and Stephen T Pope (1988). "A description of the model-view-controller user interface paradigm in the smalltalk-80 system." In: *Journal of object oriented programming* 1.3, pp. 26–49.

Kuuskeri, Janne (2011). "Experiences on a Design Approach for Interactive Web Applications." In: *WebApps*. Ed. by Armando Fox. USENIX Association.

Looper, Jen (2015). *What is a WebView?* URL: http://developer.telerik.com/featured/what-is-a-webview/ (visited on 11/18/2016).

Marr, Deborah T. et al. (2002). "Hyper-Threading Technology Architecture and Microarchitecture." In: *Intel Technology Journal* 6.1, pp. 4–15. ISSN: 1535-766X. URL: http://developer.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf.

Martin, Robert C. (1997). "The Dependency Inversion Principle." In: *Object Mentor*, pp. 1–12.

— (2000). "Design principles and design patterns." In: *Object Mentor*, pp. 1–34.

— (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Prentice Hall PTR. ISBN: 978-0-13-235088-4.

Microsoft (2016a). *Default Document Documentation for IIS7*. URL: https://iis.net/configreference/system.webserver/defaultdocument (visited on 11/18/2016).

— (2016b). *TypeScript Language*. URL: https://typescriptlang.org/ (visited on 11/18/2016).

— (2016c). *Windows Performance Monitor Webpage*. URL: https://technet.microsoft.com/en-us/library/cc749249(v=ws.11).aspx (visited on 11/18/2016).

Microsoft Patterns and Practices Team (2009). *Microsoft® Application Architecture Guide (Patterns & Practices)*. Vol. 2nd Edition. URL: https://msdn.microsoft.com/en-us/library/ff650706.aspx.

Microsoft Software Developer Network (2012). *Internet Feature Controls - Browser Emulation*. URL: https://msdn.microsoft.com/library/ee330730(v=vs.85).aspx#browser_emulation (visited on 11/18/2016).

— (2014). *Internet Feature Controls - GPU Rendering*. URL: https://msdn.microsoft.com/library/ee330730(v=vs.85).aspx#browser_emulation (visited on 11/18/2016).

— (2016a). *Component Object Model Documentation*. URL: https://msdn.microsoft.com/de-de/library/aa286559.aspx (visited on 11/18/2016).

— (2016b). *WebBrowser Control Reference*. URL: https://msdn.microsoft.com/en-us/library/aa752040(v=vs.85).aspx (visited on 11/18/2016).

Microsoft TechNet (2016). *Windows Performance Monitoring Tool - Processor Counters*. URL: https://technet.microsoft.com/en-us/library/cc938593.aspx (visited on 11/18/2016).

Mikowski, Michael and Josh Powell (2013). *Single Page Web Applications: JavaScript end-to-end*. 1st ed. Manning Publications. ISBN: 978-1-61729-075-6.

Morales-Chaparro, R. et al. (2007). "MVC web design patterns and rich internet applications." In: *Proceedings of the Jornadas de Ingenier ia del Software y Bases de Datos*.

Mozilla Developer Network (2016). *Multiprocess Firefox*. URL: https://developer.mozilla.org/en-US/Firefox/Multiprocess_Firefox (visited on 11/18/2016).

Mustach Team (2016). *Mustache: Logic-less Template Engine*. URL: https://mustache.github.io/ (visited on 11/18/2016).

Oakley, Glen and M. Pulimood (2012). "Optimized Performance of Web Applications." In: *The Elucidator, Vol. 1, No. 1, Article 1*.

Osmani, Addy (2012). *Learning JavaScript Design Patterns*. O'Reilly. 251 pp. ISBN: 1449331815.

Overson, Jarrod and Jason Strimpel (2015). *Developing Web Components*. O'Reilly. 250 pp. ISBN: 978-1-4919-4902-3.

Parviainen, Tero (2015). *Change And Its Detection In JavaScript Frameworks*. URL: https://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html (visited on 11/18/2016).

Paulweber, Michael and Klaus Lebert (2014). *Mess- und Prüfstandstechnik*. Springer Vieweg. ISBN: 978-3-658-04452-7. DOI: 10.1007/978-3-658-04453-4.

Pautasso, Cesare, Olaf Zimmermann, and Frank Leymann (2008). "Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision." In: *Proceedings of the 17th International Conference on World Wide Web*. WWW '08. ACM, pp. 805–814. ISBN: 978-1-60558-085-2. DOI: 10.1145/1367497.1367606.

Petitit, François and Mickaël Tricot (2014). *The new Web application architectures and their impacts for enterprises*. URL: http://blog.octo.com/en/new-web-application-architectures-and-impacts-for-enterprises-1/ (visited on 11/18/2016).

Pisman, Kwinten (2016). *How the hell does zone.js really work?* URL: http://blog.kwintenp.com/how-the-hell-do-zones-really-work/ (visited on 11/18/2016).

Precht, Pascal (2016). *Dependency Injection in Angular 2*. URL: http://blog.thoughtram.io/angular/2015/05/18/dependency-injection-in-angular-2.html (visited on 11/18/2016).

Ranganathan, Arun, Jonas Sicking, and Marijn Kruisselbrink (2016). *W3C File API*. URL: https://w3c.github.io/FileAPI (visited on 11/18/2016).

Rauschmayer, Axel (2016). *Exploring ES6 - Upgrade to the next version of JavaScript*. URL: http://exploringjs.com/es6/index.html#toc_ch_modules.

React (2016). *Reconciliation in the React Library*. URL: https://facebook.github.io/react/docs/reconciliation.html (visited on 11/18/2016).

Reis, Charles and Steven D. Gribble (2009). "Isolating Web Programs in Modern Browser Architectures." In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. ACM, pp. 219–232. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519090.

Rossi, Gusatvo et al. (2008). *Web engineering: modelling and implementing web applications*. Vol. 12. Springer-Verlag New York Inc.

Russinovich, Mark E., David A. Solomon, and Alex Ionescu (2012). *Windows Internals*. Microsoft Press,U.S. 672 pp. ISBN: 978-0-7356-6587-3.

Schuchert, Brett L. (2013). *DIP in the Wild*. URL: http://martinfowler.com/articles/dipInTheWild.html (visited on 11/18/2016).

Shklar, Leon and Rich Rosen (2009). *Web Application Architecture*. Wiley John + Sons. 440 pp. ISBN: 978-0-470-51860-1.

StatCounter (2016). *StatCounter Global Stats Web Browser Usage 2015-2016*. URL: http://gs.statcounter.com/#all-browser-ww-yearly-2015-2016-bar (visited on 11/18/2016).

Stefanov, Stoyan (2010). *JavaScript Patterns*. O'Reilly. 232 pp. ISBN: 978-0-596-80675-0.

Templin, Reagan (2007). *Understanding Sites, Applications, and Virtual Directories on IIS 7*. URL: https://iis.net/learn/get-started/planning-your-iis-architecture/understanding-sites-applications-and-virtual-directories-on-iis (visited on 11/18/2016).

The Chromium Projects (2016a). *Blink: The Rendering Engine of Chromium*. URL: https://chromium.org/blink (visited on 11/18/2016).

— (2016b). *Design Document: Chromium Multi-process Architecture*. URL: https://chromium.org/developers/design-documents/multi-process-architecture (visited on 11/18/2016).

— (2016c). *Design Document: Chromium Multi-process Architecture*. URL: https://chromium.org/developers/design-documents/multi-process-architecture (visited on 11/18/2016).

Wiltziusa, Tom, Vangelis Kokkevis, and the Chrome Graphics team (2014). *Design Document: GPU Accelerated Compositing in Chrome*. URL: https://chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome (visited on 11/18/2016).

Zeigler, Andy (2008). *Internet Explorer Team Blog - IE8 and Loosely-Coupled IE (LCIE)*. URL: https://blogs.msdn.microsoft.com/ie/2008/03/11/ie8-and-loosely-coupled-ie-lcie/ (visited on 11/18/2016).