Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MANUEL TREMMEL
MASTER THESIS

SERVER-SIDE SCRIPTING IN THE SWEBLE ENGINE

Submitted on 24 July 2015

Supervisors:
Hannes Dohrn
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 24 July 2015

License

Erlangen, 24 July 2015

Abstract

The Sweble Engine is a wiki software built around the powerful "wiki object model" (WOM), which represents the full state of the wiki. This thesis adds scripting support to Sweble so that scripts embedded in a wiki page, can manipulate the WOM and hence the state of the wiki. The focus of this thesis is on embedding JavaScript as a programming language, but also other scripting languages can be used. Wiki events such as rendering, saving resources and submitting forms trigger functions defined in the script. The result is a rapid prototyping environment based on Wikitext and script languages which helps users to create simple and well-factored Wiki applications. The implemented Sweble scripting module allows for collaboratively developing script libraries inside the Wiki environment that be can be included by end-users with little or no coding.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Wikis are collaborative content management systems. A well-known representative of a wiki system is MediaWiki, which is driving Wikipedia. Wikis are also used in companies and for personal projects. Mediawiki Wikitext is a common markup for Wiki systems, however, due to its evolution, it comes with numerous problems. It has not been formally defined and it has not been designed as a language which is easy to parse. To remedy this situation, the Sweble project (cf. Dohrn and Riehle, 2011) defines a wiki object model (WOM) which can represent all resources within a Wiki system. An example for such a resource is an article resource written in Wikitext. Its wiki markup elements are accessible via a standardized API which is currently available in Java only. The Sweble parser transforms Wikitext into this WOM representation and is able to convert it back without loss. WOM may also be used to transform any other type of Wiki markup into the WOM representation by using a parser for that markup. Using the WOM has several advantages. First, Wikitext has become increasingly complex and thus adding new elements to the Wikitext markup language is not straight-forward. Secondly, working with an abstract representation has not been possible so far because Wikitext is practically defined by the Wikitext parser in Mediawiki. With Sweble and the WOM representation of a Wiki article, an abstract representation exists, which can be read and manipulated by a 3rd party script or application. This is especially interesting to enable end-user programming. End-user programming is not done exclusively by programmers with a formal IT education, but also by users with different background. In fact, for 30% of all new jobs in the USA programming skills would help in solving daily tasks (cf. Ko, Myers, Aung et al., 2004). Therefore, end-user programming is useful in personal and professional environments.

The web and the web browser may be enriched with tools that allow them to collect and annotate information to ease every-day work (cf. Cypher, Dontcheva, Lau and Nichols, 2010). As part of the web, also wiki systems are ideal for end-user programming. To assist users when customizing their wiki system and

adding dynamic functionality, it is the task of the wiki environment to provide adequate tools to intuitively write reusable and maintainable code. Even more importantly, this scripting module allows more experienced developers to collaboratively develop tools that can be embedded by end-users with little or no coding. Therefore, users do not depend on top-down tools developed or added by the administrator. Therefore, decentralized tool development is made possible through the script library approach. The idea of this work is to provide these tools through a Wiki system, which end-users to customize, automate and create mashups, which constitute typical end-user requirements (cf. Cypher et al., 2010). A more detailed discussion of end-user programming will be discussed in detail in section 3.1.

To provide users with facilities to improve their effectivity in their daily use of wiki systems, a scripting module for the Sweble Wiki has been created as part of this thesis. Scripting also increases the number of possible applications of a Wiki. With scripting support, small applications can be written inside the Wiki. For example, it is possible to add or manage information more conveniently by using custom forms provided by the Sweble scripting module.

1.2 Scope

The scope of this work is to provide server-side scripting and not to provide a fully-fledged web application framework, which rivals other script languages found in the web such as PHP. This is also not supported by the Sweble Wiki framework (yet). The goal is to enable users to write simple applications within Wiki resources. The decision was not to embed the scripting functionality as a templating language, which is an approach found in many web programming languages such as PHP. Instead, the scripting language operates with WOM objects, which is a representation of the Wiki and its resources in the form of a tree. This requires all scripts to perform manipulations on the WOM tree in order to change the resource displayed to the user. This change may be temporary or (in case of a form submission), also permanent by committing a writable transaction with the changes. Scripting expressions are an exception to this rule, as their result is displayed when viewing the resource. In other words, to implement simple dynamic functionality, one is not required to use and be aware of the WOM.

Scripting is not intended for more sophisticated applications, where a Sweble Wiki module would be a better fit. The Sweble scripting module is intended only for small applications of little complexity. Using scripting is especially helpful when script applications are created by privileged users. In this case additional functionality can be implemented or get delegated without requiring the administrator to add another module. A detailed discussion when to use the scripting

module or when to create a Sweble module can be found in section 4.13.

1.3 Research question

The main research question that guides this work is how to design concepts for end-user suitable programming languages within the Sweble Wiki. At the same time, the scripting module should be a rapid prototyping environment. As a rapid prototyping environment, concepts have been developed which allow for reusable, maintainable, evolvable and modular code.

While the practical work of this thesis will be the implementation of basic scripting support, the research section will discuss end-user programming and what qualifies scripting languages for end-user programming. Only a few features of end-user programming have been implemented, such as sample scripts for adding rows to tables based on form data with automatic mapping of form elements to he cell below a specific heading. Development of script repositories which can be included with little or no coding has not been the scope of this thesis. However, the foundation for these script repositories has been created and is fully functional. Also, due to time limitations, further improvements of the scripting module such as programming by demonstration as discussed in section 3.1 have not been possible. Nevertheless, the future work section as well as section 3.1 will discuss possible improvements based on the implemented scripting module which can be the base of end-user studies how different approaches will be accepted and appreciated in the context of a Wiki system.

End-user suitability ensured mostly by handling WOM complexities by libraries which might be embedded by the end-user without understanding the underlying concepts of those scripts. As the Java API for WOM manipulation only provides little abstration for low level elements such as round-trip data and text child nodes, direct exposures of end-user programmers to this

In chapter 2, related work is discussed as the basis for the comparative evaluation and to explain other programming languages, frameworks and standards which have inspired features of the scripting language. Chapter 3 introduces the ecosystem of the scripting module, i.e. end-user development, web programming languages and how scripting languages help for end-user programming compared to non-sccripting languages. Also rapid prototyping which is convenient to do with scripting languages and the Java Scripting API which provides the base for scripting support are discussed in this chapter. The concepts developed with the implemented Sweble scripting module are presented in chapter 4, such as the script and form markup and the API that external scripts can use. This chapter also explains how to use the scripting module with a focus on the concepts for

end-users. After the research chapters and the methods chapter, the research part of this thesis is ended with a comparative evaluation of the scripting module with other web programming languages such as PHP.

The next chapters give a detailed overview of how the scripting module was created. The design concepts are discussed in chapter 6, while the implementation details can be found in chapter 7.

2 Related work

The WOM is a novel approach of representing the state of a Wiki. The reason why Node.js is addressed in this context is to show how other server-side implementations provide features such as events which are currently not provided by PHP. This will be the foundation of the comparative evaluation in chapter 5. Also, other approaches of including script programming languages to Java will be discussed here.

This chapter will briefly introduce several technologies. One of them is the Web Scripting Framework which is based on the Java Scripting API and therefore the same technology which is used by the Sweble scripting module. Thus, the Web Scripting Framework is an example for generic scripting support. Also, the framework Wicket is explained briefly which is used by the Sweble Wiki and has a templating approach that might be useful to combine with the Sweble scripting module to be able to implement functionality not exclusively by WOM manipulation. Moreover, client-side HTML its representation via the DOM and DOM manipulation via JavaScript is addressed in this chapter. Another programming language described here is PHP, which is a popular language for the web. Finally, XSLT is an example for a domain-specific language for producing XML or HTML through tree manipulation with the property that the page is always well-formed.

2.1 Node.js

The server-side framework Node.js is designed for developing asynchronous event-driven server applications. Its novel approach has made Node.js popular in the last years. It is based on the V8 engine by Google. Efforts of browser vendors to constantly improve the speed of their JavaScript runtime environments has helped JavaScript and therefore Node.js to benefit from the good performance of the V8 engine. A node server may have several single-threaded processes which handle requests. Due to native event support and callbacks, waiting times for I/O can be used for handling other requests instead of blocking. Its functional programming approach differentiates Node.js from other web programming lan-

guages such as PHP. For example, higher-order functions are used for all heavy I/O operations. While PHP gives procedural programming styles an advantage by better performance, Node.js offers functional programming with callbacks. With a wide range of I/O features such as socket, TLS/SSL or UDP connection support, Node.js offers custom server features which PHP does not offer. Using a single-threaded asynchronous event-based architecture without multi-threading makes scripts more readable, more maintainable and reduces the complexity and pitfalls of multi-threading. The downside of the events and asynchronous I/O is that handling different events in their specific contexts can be challenging for programmers (Tilkov and Vinoski, 2010).

The motivation of Node.js programmers is often to reuse client-side validation code on the server-side and vice versa[1]. To do so, modules like jsdom exist which are able to render HTML text and perform jQuery operations on the DOM (Insua, n.d.).

Node.js and its approach of reusing client-side verification code, and its event handling model has inspired the Sweble scripting module.

JavaScript has become popular on the web and is used for sophisticated applications. Sophisticated modules require modularity which can be achieved by dynamic loading, however modularity is not supported by JavaScript natively. Therefore, Node.js uses a module system which loads modules by calling require(). Require loads all dependencies which are not resolved including possible recursive dependencies. However, as Herman and Tobin-Hochstadt point out, due to lack of language support, modules in Node.js require programmers to use and detect the pattern correctly. Also reducing the number of global variables does not elimate global namespace pollution which still might collide with other modules. Also, as a custom construct, circumventing it is possible (cf. Herman and Tobin-Hochstadt, 2011).

2.2 Web Scripting Framework

The Java Specification Request 233 (JSR 233) intended to provide a generic API for scripting and to provide templating support for servlet containers. For example, the Web Scripting Framework can be used for integrating PHP and Java, where PHP can serve as templating engine for the view for Java applications. Also, due to the popularity of PHP in the web application environment, offering to reuse and integrate existing PHP code with Java helps to attract more programmers. It also removes barriers of porting PHP-based applications to Java. The freedom

[1] also of the Sweble scripting module which uses JavaScript as main scripting language for reusing client-side code.

to combine different software modules and components (e.g. CMS) to create a new web application gives more options for Java developers. The goal of the Web Scripting Framework is to help migrate existing PHP applications into Java with less effort (cf. p. 447ff., Bosanac, 2007).

The Web Scripting Framework is another project which uses the Java Scripting API which is used by the Sweble scripting module.

2.3 Wicket

Wicket is a component oriented web framework with an active community support, started by Jonathan Locke in 2004. The idea of Wicket was to overcome shortcomings in the designs of several mainstream Java standards for web development. Wicket allows both classical web application and Rich Internet Applications (RIA) similar to desktop applications. For RIA, Wicket handles the necessary technical details to make it as easy as web application development. The component oriented approach uses POJO objects and is thus light-weight as Swing (cf. p. 7-10, Förther, Menzel and Siefart, 2010).

The Wicket framework takes care of state management and session handling. For example, entered information is stored in the session when the user navigates to another page. In contrast to Model 2 frameworks, Wicket is handling low-level HTTP protocol specifics for the developer. Also, Wicket templates are free of UI business logic. Wicket templates are also valid and renderable (X)HTML which allows web designers and developers to work in parallel. As development is done mostly in Java and templates (panels) can be reused or extended, developers can make use of all features which IDE provides for refactoring. The downside of Wicket is that easier development comes at the price of harder scalability as only the server node that has the session content stored is capable of handling client requests. However, the advantages of reuse and better maintainability compensate the lack of Representational State Transfer (REST) as suggested by Roy T. Fielding (cf. p. 4-14, Dashorst and Hillenius, 2008).

Generally there is just a Java and a XHTML template, non-Java configuration is minimal in Wicket. Also, Wicket displays readable errors in the development mode (cf. p. 7-10, Förther et al., 2010).

Wicket has been used for implementing the UI for Sweble wiki (e.g. logger, interactive scripting) and its wiring of Java code with HTML templates is a good example for a templating mechanism which might be useful in subsequent versions of the Sweble scripting module.

2.4 HTML, DOM and JavaScript

HTML on the client-side also uses inline scripts (e.g. with the onclick or onload attribute), external scripts and script references (cf. Le Hors, Raggett and Jacobs, 1999). The referencing of external script resources is a defacto standard and widely used on the web. In order not to invent yet another standard, the way how external scripts are referenced in JavaScript will serve as a standard.

Browsers transform the HTML sent by the server to a tree-based DOM representation. The DOM can be manipulated by adding, removing child nodes or editing text content. Also, setting innerHTML of a node adds a set of elements to the DOM. This is similar to the WOM, even though a concept similar to innerHTML does not exist in the WOM (cf. Wood et al., 1998).

2.5 XSLT

XSLT is a language to transform XML documents to other documents by dispatching nodes and handling nodes by templates which match these nodes. (cf. Clark, 1999). XSLT is too complex for end-users, but as domain-specific language for XML generation, it ensures valid HTML which is not guaranteed with string concattenation. Also, its approach creates target documents by traversing and dispatching nodes, which is an approach comparable to programmatic WOM manipulation.

2.6 PHP

PHP is one of the most common web script languages. According to Tatroe et al., approx. 78% of the top 1 million website use PHP (cf. Tatroe, MacIntyre and Lerdorf, 2013). It uses placeholders inside a HTML file which are replaced by the content "printed" by statements inside PHP (cf. The PHP Group, n.d.).

PHP is a template language which allows inserting dynamically generated text instead of `<?php ?>` placeholders, which get evaluated by the PHP interpreter. PHP is used by classical content management systems (CMS) such as WordPress. PHP suffers of shortcomings in the initial language design, and many features such as object orientation and namespaces have been added in more recent versions of the language only. Also, global variables which were automatically created from request parameters were a major security issue in early PHP versions until they got removed (Cholakov, 2008).

PHP and other templating languages render pages using string concatenation (cf. The PHP Group, n.d.). However, the rendered result is interpreted as a DOM when rendered by the web browser on the client side. Serializing the HTML based on strings with placeholder may be convenient in some conditions, but comes with disadvantages (cf. Cholakov, 2008). For example, escaping has to be taken care of by the template developer and mistakes or improper encoding of input lead to problems such as Cross-Site-Scripting vulnerabilities (XSS). Also, refactoring templates which are composed by placeholders (and thus utilize string concatenation) is not possible, as the placeholders may included arbitrary text e.g.

```
<?php echo "</td></tr></tab";
echo "le><table><tr><td>"; ?>
```

The PHP sample code is kept unreadable on purpose to show that refactoring is not possible as elements are created by print statement. It is therefore not possible to replace all table nodes by other markup, for example.

3 Background research

3.1 End-user development (EUD)

Nowadays, computer skills are essential for many daily private and work tasks. Most computer users can work with the computer due to the possibilities of understandable or intuitive graphical user interfaces, but have no programming skills. The understanding of how to use a computer for specific tasks is known as computer literacy. Computer literacy is getting more important together with of information and information technology, which creates the need to develop and improve computer skills of people (cf. Konan, 2010). Computer literacy, however, is often not enough to automate repetitive tasks, which is why end-user development (EUD) is helpful (cf. Cypher et al., 2010). End-user development or end-user programming aims to provide the tools and mechanism to enable end-users to easily develop customizations or automations. Parametrization or customization (e.g. using a different view for the information) are not considered end-user programming, as they offer only limited options to choose from (cf. Lieberman, Paternò, Klann and Wulf, 2006).

For many years, IT systems try to be easy to use. IT systems are used in dynamic environments where requirements change quickly or are not known precisely enough before-hand by end-users or customers to enable developers to provide a one-time and long-lasting solution. Also, different users might need different tools to optimize their personal routines within the IT system, with routines which might change on a monthly or even daily basis. Mostly, adaptations of the IT systems by professional developers are costly, time-consuming and not available as quickly as required. Also, due to lack of domain knowledge, programmers cannot always directly provide a solution that matches the requirements of the end-user. Therefore, end-users need to be able to customize their system continuously themselves to achieve a short or medium term solution. There are even IT projects where end-users actively participate in the development process as end-user developers using programming tools which require no programming skills. This end-user software engineering approach has avoided failures of software projects in some scientific experiments and reduces development

costs (cf. M. M. Burnett and Scaffidi, 2013).

End-users use end-user development tools in their area of expertise to support their goals. Due to potential lack of knowledge compared to IT professionals, those programs, spreadsheet formulas or web pages might be of poor quality or even have security flaws. In a professional context, errors due to poor end-user programming can result in a economic loss. Therefore, the interest of researchers is to provide tools that avoid common pitfalls (cf. p. 2ff., Ko et al., 2011). For example, for web pages, a mechanism to prevent cross-site scripting (XSS) or SQL injections or writing access to files or scripts for web page visitors are mechanisms of restricting possibilities for the end-user. For example, the WOM manipulation which is allowed by the Sweble scripting module is not vulnerable to XSS and SQL injections and therefore the Java WOM API is a domain-specific language for creating valid markup.

There are several motivations for end-user programming, which will be introduced in the following by giving use cases where end-user development is desirable. One example is automating repetitive tasks from multiple or dozens clicks to one or two clicks. This is especially useful when the user interface offers to many options or if the user interface cannot know personal information. Also, automated notifications via SMS or email (e.g. a warning message) is a common need which end-users may handle themselves. Mashups are a combination of several sources of information to create or annotate information in a way to make it more useful for the end-user (cf. p. 3ff., Cypher et al., 2010). Spreadsheet programming and creation is an example for a very popular form of end-user programming with more than 50 million estimated users (cf. p. 11, Cypher et al., 2010). The problem of conventional programming languages is that they are "obscure, abstract and indirect" (p. 12, Cypher et al., 2010). That means that the code required for a specific action does not permit syntax errors and the code uses more complex programming constructs than the end-user might expect (e.g. invoking invent handlers to trigger a mouse click on a button instead of "click button"). Without understanding of programming concepts such as event handling, end-user suitable languages need to provided functions and names that the end-user can understand. An end-user can be expected to understand terminology from his daily routines of working with the computer to avoid learning barriers and misinterpretation of functionality.

Strategies to reduce the difficulty for end-users to write script languages are structure editors, where a wizard allows the user to select appropriate commands with a GUI and the wizard creates correct syntax (cf. Cypher et al., 2010). Also, natural development using natural languages or gestures are emerging disciplines which might play an increasingly important role for EUD.

Programming by demonstration (PbD) is a technique where the user performs a task and then the system creates a program to repeat the task for other instances

of “data”. In cases where the action of the PbD system is not unique, the user might be asked to choose the correct action (cf. Cypher et al., 2010). With Programming by specification, the end user describes the desired target system as text or with a visual tool which then generates the target system, provided that the input is understandable (cf. M. M. Burnett and Scaffidi, 2013).

End-user development (EUD) should have a “low threshold” to enable beginners to adapt EUD quickly. At the same time, it should have a “high ceiling” to enable experienced programmers to use powerful tools using widely accepted design principles. This is done by providing metaphors a user can relate with instead of using high-level concepts which can be barriers to novice users. Collaboration in code generation and mutual development where end-users help to design a system are key features of many EUD approaches (cf. p. 3ff, Paternò, 2013).

As briefly introduced before in this section, end-user programming goes so far to have the end-users develop services (e.g. e-government services) themselves with only little interaction of professional developers, who design the underlying model for the application. This is called end-user software engineering. By having end-user developers, having a complete and unambiguous specification of the requirements beforehand is not necessary. The result of an experimental phase showed that end-user developers enjoyed the experience of creating programs with the tools at hand (cf. Fogli and Provenza, 2011). This experiment shows that in the future, applications in the web or for companies might actually be developed partly by the end-user developers - reducing the importance of requirements elicitation.

End-user development approaches have are not limited to web and desktop applications (e.g. browser automation tools and spreadsheet programming). End-user programming has already been applied to mobile devices, for example with the TouchDevelop project by Microsoft. TouchDevelop provides a simple programming language which gives users the chance to program the mobile device without using a computer. A field study of TouchDevelop has shown that a common characteristic is code reuse, especially of own code. ALso, the survey shows that beginners are the largest user group (cf. S. Li, Xie and Tillmann, 2013). Especially the popularity of libraries is a result of that survey which encourages the use of script repositories for the Sweble Wiki

While end-user programming is getting increasingly important, lowering learning barriers of programming languages and programming environments which includes, for example, the editing and debugging tools. Fig. 3.1 shows which kind of barriers occur to end-user programmers when they learn Visual Basic for Excel. As programming skills are missing, end-users work with assumptions which may lead them to barriers where end-users get stuck or which lead to other barriers. There are only edges when the problem is not insurmountable and when the edge percentage is greater than 10%. The result is that while some barriers

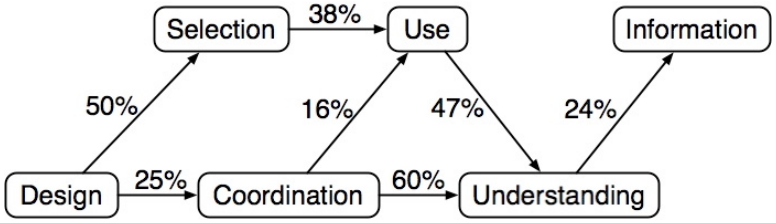

Figure 3.1: End-user programmers try to overcome lack of programming skills (e.g. programming concepts and understanding) with possibly wrong assumptions. "For surmountable barriers, the percent of each type overcome with invalid assumptions, and the type of barrier to which the assumptions led." (cf. Ko, Myers, Aung et al., 2004).

may be passed with invalid assumptions, there is another barrier where the user-ends up with. Based on the evaluation of Visual Basic for Excel, the barriers can be classified universally as (cf. Ko et al., 2004):

- Finding an abstract solution and planning it in the design phase in such a way to have the computer do the intended task (design barriers).
- Also, finding the ideal tools to use to solve a specific problem requires experience which an end-user might not have (selection barriers).
- There are problems of combining behaviors and understanding the rules of composition of algorithms, modules etc. (coordination barriers).
- Unclear or not intuitive programming interfaces mislead the user how to use that interface (use barriers).
- While users sometimes manage to formulate code the way they expect it to work, some external error (understanding barrier, e.g. compiler error) or an internal error (information barrier) occurss when end-users find no means to test their hypothesis (e.g. what is the value of a variable at a specific point in the program?)

Chickenfoot

An example for an end-user programming language is Chickenfoot. Chickenfoot allows end-users manipulate the DOM and therefore a web page on the client side. It can be embedded into Firefox as an extension. Chickenfoot uses a syntax that is very close to the end-user. With intuitive functions like

- find (HTML element or form element),
- click (on link or button),

- check or uncheck (a checkbox),
- enter (information to a text field) or
- pick (from a drop-down list or selection),

common tasks on websites can be automated without using "obscur" JavaScript syntax. Web page components are addressed by keyword matching. As the results are visible directly, the user has direct feedback of what the code does (cf. p. 39ff, Cypher et al., 2010 , p. 5ff, Paternò, 2013).

For example,

```
enter("Sweble Wiki");
click("Search");
```

is sufficient to trigger a search on a search engine website (cf. p. 39ff, Cypher et al., 2010). This is a synax which can be offered by libraries using the Sweble scripting module. Therefore, using a Chickenfoot-like syntax can be part of a simplification approach help doing simple tasks with little effort. Especially the keyword matching approach of Chickenfoot should be favored over selection mechanisms such as XPath, as XPath is more complex to learn.

CoScripter

CoScripter helps to collaboratively record and automate tasks on the web with its human-readable scripting language ClearScript. It consists of an online repository of scripts which allows loading the script to the browser extension. There, the script can be inspected step-by-step, with the code and the affected web page element highlighted. CoScripter is inspired by Chickenfoot but due to its human-readable language it is even better suited for end-users without programming experience (cf. p. 86ff, Cypher et al., 2010 ; p. 5, Paternò, 2013).

Sloppy programming

With sloppy programming, interpreters/compilers allow "pseudonatural language instructions" and do not return a syntax error, but try to find the closest code that would be valid. This is similar to search engine suggestions when the user mistypes (cf. p. 290, Ko et al., 2004 ; p. 289ff, Cypher et al., 2010 ; (cf. p. 3ff, Paternò, 2013)). Sloppy programming can be used in the Sweble scripting module by choosing a programming language which allows for sloppy programming.

3.2 Web programming languages and their problems

Web programming languages such as PHP have evolved over time and have been rarely engineered using scientific principles. The goal of backwards-compatibility comes with the disadvantage that improvements in language design can only be of minor nature and cannot break with previous design decision, even though they have been proven problematic. At the same time, web applications have become more and more pervasive. In the following, common problems with web programming languages will be introduced, which will serve as a foundation for the comparative evaluation in chapter 5.

3.2.1 Weaknesses of templating languages

On the web, script programming languages which allow "quick and dirty" rapid prototyping programming enjoy great popularity. An example for such a programming language is PHP, which has been briefly introduced in section 2.6

From a programming point of view, PHP is normally not using the expressiveness of tree-based data structures such as XML, as it renders pages (mostly to HTML) by string concatenation. String concatenation means that the output of a script code may call a print function/procedure which will append its argument to the HTTP response (cf. The PHP Group, n.d.). The use of string concatenation in PHP also increases the risk of producing invalid HTML code. Examining a PHP script if it may produce invalid HTML code is not straight-forward, as invalid HTML may be produced in rare or exceptional cases due to conditional statements.

```
<html>
<php
$attr=' class="color:red;"';
echo "<div$attr>".$_REQUEST['param'].'</div>';
?>
</html>
```

This code sample shows how XSS is possible, which might execute malicous code on the client-side and also destroys the validity of the generated XML, e.g. with

```
$_REQUEST['param']='<script>';
```

Also, if the first or last line of code was missing (i.e. `<html>`) the PHP would never generate valid HTML code even if input validation was used for the parameter.

This shows that PHP does not guarantee to produce valid HTML, while other languages such as XSLT guarantee valid HTML/XML.

The design of the PHP programming language mislead especially inexperienced programmers to adapt problematic programming practices. Also, security vulnerabilities come also from poor language design. Using object oriented programming in PHP reduces the execution speed, as instantiating PHP objects takes longer than Java objects. Therefore, procedural scripts are favored over well-factored object-oriented applications in terms of performance. In his paper, Cholakov (Cholakov, 2008) also mentions the lack of strict typing and the lack of pre-compilation optimization. However, as Bosanac points out, this is a characteristic of script languages and therefore is not a PHP-problem, but a question of scripting versus "system programming languages" (Bosanac, 2007).

Another problems identified by Cholakov (Cholakov, 2008) is that configuration can influence the behavior such as short open tags (<? instead of <?php). Future scripting language implementations such as the scripting module of the Sweble Wiki should avoid configuration options which reduce portability of code. Also, events are not supported natively. Current PHP versions do not allow multithreading even though multiple core CPUs are common for web servers and get increasingly important for modern web programming frameworks.

The weaknesses discussed in this section are the base of some arguments in the comparative evaluation in chapter 5.

3.2.2 Security problems

The pervasive use of web programming languages comes also with security problems. HTML5, for example, now allows graphical browser games and offline data storage, reducing the gap between desktop applications and web applications further (cf. WHATWG, n.d.). However, new features come with security threats. For example, 3D browser games require code originating from a website to be executed on the graphics card, potentially exploiting some hardware bugs. At the time of development, the graphic cards were not expected to run computations coming from potentially untrustworthy sources such as the web. While malicious client code may come from an untrustworthy website, XSS can even bring untrustworthy code to generally trusted websites.

Information security, which ensures integrity and confidentiality, requires information flow policies to be enforced. While many web scripting languages provide ad hoc mechanisms to ensure information flow security, they do not enforce those policies (P. Li and Zdancewic, 2005). Information flow may be controlled by scripts which validate user input. The Sweble module avoids information leakage by giving scripts only access to data which is permitted in the current transaction.

In the following, typical web security problems such as Cross-Site-Scripting and SQL injections will be introduced to demonstrate what scripting frameworks need to do to avoid vulnerabilities.

XSS (Cross-Site-Scripting) exploits vulnerabilities that normally come from a lack of input validation. A source controlled by the attacker (e.g. an "infected" URL inside a URL) may direct the user to the correct website, but still manipulate the DOM through parameters so that external scripts are loaded which manipulate the DOM further so the data entered by the user ends up at a sink controlled by the attacker and thus can get abused (cf. Fogie, Grossman, Hansen, Rager and Petkov, 2011). For example, XSS is the base of many credit card frauds. The problem is that input validation has to be done inside the script explicitly as most web programming languages are not domain specific language (DSL) which handle security transparently. A DSL in the context of HTML template or web page generation would take care of escaping and breaking out of tags itself. Instead, the web developer has to take care of proper escaping himself and when doing a mistake, a vulnerability can result. There are frameworks which help to ensure proper escaping of data that comes from user input, but full or partial XSS-safety is not a feature of most web programming language itself.

SQL injections store malicious code in the database (persistent SQL injections) or use poor SQL query formulation to leak information inside the database to sinks which are controllable by the outside(reflected SQL injections). This can be used to disclose confidential data such as passwords to a hacker. Poorly designed programming language features favor SQL injections, such as the mysql extension by PHP. PHP's mysql extension should not be used, but replaced by mysqli or alternatives (cf. Tiwari, 2014).

3.3 Scripting languages

This section will describe characteristics of scripting languages and what differentiates them for non-scripting languages.

The idea of scripting languages is to create more sophisticated applications with less effort (cf. p. 5, Bosanac, 2007). Scripting languages are high level programming languages, which are also referred to as third-generation programming languages.

"Scripting language[s] [are] to write user-readable and modifiable programs that perform simple operations and control the execution of other programs" (cf. p. 12, 25-28, Bosanac, 2007). The availability of source code is common even though not necessarily the case when code is secret (cf. p. 13, Bosanac, 2007).

Scripting languages often use dynamic typing, which means that no type information is provided when declaring a variable. They distinguish themselves from non-scripting languages in terms of run-time performance and debugging complexity. Scripting languages are less performant due to dynamic typing and the interpreter overhead. Debugging might be more difficult due to dynamic typing (cf. p. 678, R. Liguori and Liguori, 2014).

Code evaluation with functions like eval is supported by several scripting languages. For example, Python and JavaScript provide an `eval()` function which executes code and therefore allows treating code as "data". This is possible, as the interpreter is present whenever code is executed. Scripting languages allow closures which is passing code as function argument, which is a concept similar to list comprehension in declarative languages. Also, scripting languages often allow to pass functions as arguments (cf. p. 19ff, Bosanac, 2007).

Moreover, script languages also differentiate themselves, as they

- may include domain-specific features (cf. p. 677, R. Liguori and Liguori, 2014),
- are embeddable,
- are extensible and
- are easy to learn and use

(cf. p. 71, Bosanac, 2007)

Typical applications of scripting are

- Unix shell languages (cf. p. 41, Bosanac, 2007)
- Prototyping e.g. with Python (cf. p. 47, Bosanac, 2007)
- End-user customization with so-called macro languages, e.g. VBA for Microsoft Excel or macros in LibreOffice (cf. p. 49, Bosanac, 2007)

3.4 Rapid prototyping

Another goal of the Sweble scripting module is to provide a rapid prototyping environment.

Scripting languages lack overhead of declaring variables and therefore are ideal for rapid prototyping. In the prototyping development model, first requirements are analyzed and a prototype is developed to present a solution for the requirements. This prototype is presented to the users who evaluate the prototype or to clarify

misunderstandings between client and developers and detect missing features (cf. p. 44-45, Bosanac, 2007).

Scripting languages have more powerful statements and data types than system programming languages. This shortens the code and leads to higher level code. Also, due to dynamic typing, less code needs to be written. Moreover, as no compilation and linking is required, testing changed code requires less time. However, interpreted languages consume hundreds or thousands of machine cycles, while the average system programming requires only five machine cycles, as Bosanac points out. However, this performance loss is a minor problem as hardware is getting more and more powerful and allows programming languages to get more human-oriented. Therefore, scripting languages help to improve development speed and are ideal for rapid prototyping and increase productivity (cf. p. 28, Bosanac, 2007).

3.5 Scripting in Java

Several approaches have been designed to allow for scripting in Java, such as the Bean Scripting Framework and JSR-233.

The Bean Scripting Framework (BSF) is a Java scripting framework developed by IBM which has the been moved to Apache. BSF supports only scripting languages with an interpreter implemented in Java or native engines implementing the Java Native Interface, or in short JNI (cf. p. 246, Bosanac, 2007).

While BSF is a stable project and matches the requirements for scripting, there was a need for Java itself to allow for scripting natively with simpler inclusion of upcoming scripting languages and more features. The Java Scripting API was introduced in Java SE 6. Examples of features of the Java Scripting API which are not supported by BSF are `eval()`, compiling code to bytecode and a dynamic script engine discovery mechanism and defining binding scopes. Also, the Java Scripting API has a generally a "cleaner API" compared to BSF(cf. p. 408, 445, Bosanac, 2007 , p. 175, R. Liguori and Liguori, 2014).

Java Specification Request 233 (or JSR-233) is a Java abstraction layer which allows using several scripting languages within Java. JSR-233 or "Scripting for the Java Platform" is a specification with originally two parts: General Scripting API and Web Scripting API. The General Scripting API deals with script integration in Java applications in general, while the Web Scripting API has the goal of embedding scripts in servlet containers or integrate existing Java applications and dominant web scripting languages such as PHP(cf. p. 389, Bosanac, 2007). The Web Scripting Framework has been discussed in section 2.2.

The bindings mechanism is provided by Java to the script engine. It is a simplified and standardized way of passing data between the host language (here Java) and the script written in an arbitrary scripting language. Bindings implement the interface Java.util.Map interface (cf. p. 25ff, Sharan, 2014) and therefore a "simple" key-value pairs.

The JSR-233 is designed to ease the integration of all kinds of script languages with the goal of portability (cf. p. 391, Bosanac, 2007).

Available JSR-233 scripting languages

The script languages which are supported by the JSR-233 are, for example:

- BeanShell
- Clojure
- FreeMarker
- Groovy
- ACL with Jacl
- JavaScript (with Nashorn or Rhino)
- Jawk
- Jelly
- JEP
- Ruby (with JRuby)
- Python (with Jython or JPython)
- Scala
- Sleep
- TCL/Java
- Visage
- JudoScript
- ObjectScript
- Velocity

(cf. Appendix B, R. Liguori and Liguori, 2014 and p. 122, Bosanac, 2007).

4 Methods: Usage and API of the scripting module

4.1 Events of script invocations

There are several events where triggering script evaluation can be useful. In the case of a Wiki, script evaluation is possible

- when rendering a page with external scripts or script expressions (onRender),
- when saving a resource after editing it with the resource editor (onSubmit) and
- when submitting a form on a resource (onSave).

Therefore, embedding script snippets is possible in three variants:

- inline script expressions (just in case of onRender),
- external scripts, by referencing one or more external scripts from within an article resource which will be evaluated,
- external scripts, by referencing one or more script resources which will be evaluated and
- interactive scripting (entering code and evaluating it when a button is clicked).

Script types (i.e. the script language) can be provided to script expressions, external scripts and script resources. Script references, however, may not have a script type. This is due to the fact that the referenced script is supposed to provide the script type which may also change over time transparently.

The preferred way of adding server-side logic to a resource is by referencing an external script. An external script can be a (JavaScript) script resource. Another way is to use the tag extension external-script. Surrounded by <external-script> and </external-script> in the familiar XML syntax, a script can be placed within

an article. When several scripts exist on an article, addressing a specific script is not unique. Then an external script needs an id attribute, which allows other articles to address the script using

```
<script src="/path/to/article#id"></script>
```

Referenced scripts are not evaluated on the resource where they are placed. Instead, they are displayed only with syntax highlighting. The intention is that one can have several scripts in different script languages with documentation Wikitext around these scripts.

Scripts running in the Sweble engine can make use of the WOM Java API (cf. Dohrn and Riehle, 2011) by accessing the documentbinding or using the arguments passed to the onRender, onSubmit and onSave.

4.2 Script expressions

Script expressions are evaluated when rendering the page. Possibly, scripts evaluated on rendering use values provided by scripts triggered after submitting a form. The approach of using script expressions is comparable to templating as the script expression is replaced by the evaluated code. However, more-liners are not supposed to be implemented using script expressions. To avoid using script expressions for sophisticated tasks (resulting in a unclear notion for the end-user of what will be the rendered result when more than one statements are presents), the script has a limited context (cf section 4.8.1.

Script expressions are containers for simple inline expressions written in a specific script language. For example, a mathematical computation written in a script language such as `9*9;` is an expression which yields a result, in this case `81`. A return statement is not required for script expressions. Script expressions have several attributes such as the script type which defines the script language media type of the script. Also, script expression have the evaluationTime attribute with the default of intermediate (side effect free) and the alternative options before and after.

Script expressions may use variables defined by previous script expressions or previous referenced external scripts or referenced script resources.

Examples:

```
<script eval="server" type="application/javascript">9*9;</script>
```

```
<script type="application/python" evaluationTime="intermediate">
print "hello world"
</script>
```

While the first sample script is in JavaScript, the second script is written in Python. The requirement for executing Python scripts is that the Python .jar file is present in the classpath. The eval attribute is currently optional as the wiki does not yet support client-side scripting.

Allowing script expressions in Wikitext eases the task of including dynamically generated text into the WOM. Without script expressions, the programmer needs to be able to address the position first by creating placeholders and then has to write an external script to manipulate that WOM element. To make this special case more convenient, expressions are allowed inline to allow for quick templating.

Inline expressions are intended for very simple scripts, ideally one-liners. For example, returning the current date or counting list elements on a page (e.g. by using an external library). Script expressions have a limited document.getContext() binding to avoid "abusing" script expressions for more sophisticated applications, as this comes with several problems and may cause confusion (e.g. with more than one statement, which one will be printed?)

4.3 Referencing external scripts and script resources

External scripts are scripts which themselves are not executed when they appear on a page. Instead, they need to be referenced to be evaluated. External scripts are intended for any more sophisticated script than script expressions.

As all external scripts run in the same context and therefore have access to the variables and functions defined by previous scripts, external scripts are ideal for use libraries. At the same time, scripts evaluated for rendering pages for different user are invisible to each other. Libraries consisting of one or more external scripts referenced by the script using the libraries can handle complex aspects of the WOM manipulation. Using libraries, end-users may use simple concepts for specific applications. Inside the Wiki ecosystem, collaboratively developing scripts and documening them

4.3.1 Script references

Script references are required to trigger script evaluation when no script expression is used. Script references hold a source WRI of an external script or of a scripting resource. In case of a referenced external script node (i.e. a script embedded in an article resource), an external script ID can be used to address a specific external script on an article. If no script ID is provided by using a URL

fragment (i.e. after hash symbol #), then all scripts defined by external script nodes on the article resource are evaluated.

A resource may have multiple script references. All scripts referenced by script references run on the same context. This means that a script running after another script (e.g. the 2nd referenced script) may read variables created by the previous script and can see all changes done by the previous scripts. This is similar to the way how the script tag in HTML works with external scripts. The resource or article which contains the script reference evaluates the referenced script resource or external script. The resource with the reference is also passed to the external script as argument and is available as context resource through `document.getContext().getContextResource()` where applicable.

When several script references are placed in one article or resource, then the scripts are defined in the order of the script reference tags. The referenced scripts can be in any scripting language supported by the Scripting API and enabled by the Sweble wiki administrator. Mixing differnet script languages and accessing the same scripting context is a feature provided by the Scripting API ScriptContext. Therefore, variables defined in one referenced script can be accessed by subsequent scripts. Due to the generic nature of the Scripting API, any script language can therefore be used.

4.3.2 External script nodes

Exernal scripts nodes are XML nodes inside an article resource which contain code. When a page has an external script node, it is displayed with syntax-highlighting. However, external script nodes on ab article are not evaluated. External script nodes are only evaluated on the resource on which a script reference for that external script node exists. External script nodes may be in any scripting language. The script language is defined by the type attribute of the external-script XML element in the Wikitext. Also, using different scripting languages for different external script nodes in one article resource is possible.

External script nodes are well-suited for libraries as code and Wikitext documentation can be on the same article resource. Therefore, the surrounding article can explain the presented algorithm and how to use it for own projects. The documentation is not inside the script in the form of programming language comments, but readable and can use Wikitext features such as lists, text formatting and tables for better readability.

External script nodes may have the attribute id to allow script references to reference one of many external script nodes in an article resource. Without the id attribute, all external scripts nodes inside an article resource will be evaluated.

evaluation time	before[1]	before	intermediate	after
may modify	no	yes	no	yes
may return	no	yes	yes	yes
access to original WOM	yes	yes	yes	yes
access to modified WOM	no	no	no	yes

Table 4.1: Evaluation time and possibilities of scripts regarding WOM manipulation and/or returning values

Example:

```
<external-script id="hello-world" type="application/javascript">
function onRender() {
return "hello world";
}
</external-script>
<script src="ArticleName#hello-world"></script>
```

4.3.3 Script resources

Script resources are resources inside the Sweble wiki. This ressources contains the script code to be evaluated. When creating a new script resource, apart from the code, also the evaluationTime and script language needs to be defined using the media type of the script language. The script language media type must start with application/, but does not need to be defined as only a subset of all available scripting language will be made available by the administrator. In case a script language is not known, referencing it will display an error on the referencing resource. This can be fixed by the administrator by adding the script language to the Sweble Wiki server.

A script resource can only have code in one specific script language. A script resources may only contain one script unlike external script nodes which can appear inside an article several times. Details about the script resource can be found in the section 7.5.

The disadvantage of script resources compared to external script nodes is that documentation is inside script code and therefore less readable for an unexperienced end-user.

4.4 Evaluation time

Each script may have an attribute evaluationTime. Possible values are before, intermediate and after. The default is intermediate, i.e. evaluating scripts as during the visitation. What evaluation time is used is determined by the script reference. The referenced script (i.e. script resource or external script node) may have also an evaluationTime attribute. If present, the script is only executed when the evaluationTime attribute of the script reference and referenced script match. If not present, the evaluationTime of the script reference will be used. It is possible to declare more than one allowed value for evaluationTime of a external script by separating all allowed values by space, e.g. `evaluationTime="before after"`

- before: Before any pre-rendering is done, the required WOM tree is cloned and can thus be modified. The script has access to the original WOM tree. The returned tree is processed further by the pre-renderer. When accessing or modifing elements on the page, this is done on the elements when they are not expanded (in case of tag extensions) and the returned nodes may still be transformed into something else. For example, when a Transclusion is returned, then this transclusion will be expaneded before rendering the page to the user.
- intermediate (default): The default case occurs when attribute evaluationTime is not provided or when the value of the attribute evaluationTime is intermediate. Then, the script is evaluated during stacked visitor processing. Therefore, the script must be side effect free to avoid complications between visitors. It may return a value which is inserted instead of the original script node in the result tree. The node might be redispatched and transformed into something else, e.g. when returning a external-script node, it will be transformed to a syntax-highlighted preformatted text. It has only reading access to the original tree and not to the semi-modified result tree because it is nondeterministic, i.e. some nodes might be processed while others for no obvious reason are unprocessed. Therefore, scripts evaluated at this evaluationTime have to be side-effect free.
- after: Scripts at evaluationTime after have access to the original and result tree. The returned node is not processed by the pre-renderer, therefore a script node should return a value or a sub tree which does not require further processing.

[1] without cloning

4.5 Execution model

Scripts are run in a multi-user environment. Scripts running to render pages for different users run seperately and have no side-effects on each other.

Scripts are not only run, but also created in a mult-user wiki environment. This means that privileged users might create or modify scripts while some attempts of unauthorized users to create scripts need to be prevented. At the same time, editing a page without modifying the script itself needs to be allowed - provided that the user has the privilege to edit pages. Therefore, anybody who saves a script and or creates changes code is stored internally. When this user is authorized to sign scripts, those scripts will be evaluated by all users accessing that resource. If a user is not authorized, an error is displayed at the location of the script tag and the script is not executed. To enable execution for that script, an authorized user needs to save the script again. If a unauthorized user changes an article resource without changing the code, the signature of the last editing user remains which means that the script can still be evaluated if it was signed before the save operation.

If no permission of the stored last user for script execution exists, an error is displayed at the location of the script tag and the script is not executed. To enable execution for that script, an authorized user needs to save the script again.

Also, scripts may contain malicious code and therefore mechanisms is required to protect the Wiki system from loosing its integrity.

4.6 Forms and form elements for Sweble resources

Server-side scripting alone does not yet give enough flexibility for more sophisticated applications. Forms are necessary to provide data for script logic and to trigger script evaluation, such as onSubmit, i.e. when a form is submitted.

Form and form element generation is triggered by placing default HTML 4.x form tags inside the WikiText. A form as well as form elements are identified by the attribute name. This attribute name decides how to access the content of the form in the script. A specific form element can be addressed by its form name in combination with its form element name.

Form elements can be populated with default data on first encounter within a user session. The default values can be passed using standard HTML syntax. For example, to pre-populate a text field, the attribute value can be set to a specific value. To pre-populate a text area, the text content of the text area

can be filled. Similarly, to do a default selection for radio buttons, checkboxes is `checked="checked"` as attribute. For <`select`> dropdown or multiple choice options, the attribute `selected="selected"` of `<option>` elements means that the respective option is selected by default.

The following form elements have been implemented as a selection of all available Wicket form elements and can be used with the following markup

- text input field, i.e.
  `<input type="text"/>`
- text areas, i.e.
  `<textarea/>`
- dropdown lists, i.e.
  `<select><option/></select>`
- multiple selection lists, i.e.
  `<select multiple="multiple"><option/></select>`
- checkboxes, i.e.
  `<input type="checkbox"/>`
- radio choices, i.e.
  `<input type="radio"/>`
- buttons, i.e.
  `<input type="button"/>` or <`button/`>

An example for using forms is shown here:

```
<form>
<input type="text" name="item"/>
<input type="submit" value="Add item"/>
</form>
<script src="ScriptResourceName"></script>
```

Also, there is a script resource with the WRI "ScriptResourceName" (script type: application/javascript) with the following code:

```
function onSubmit(formData, resource) {
resource.getBody().appendChild(document.createStrong(formData.item
   ));
}
```

When submitting the form, the resource is appended temporarily with a bold text with the value provided in the text field.

It is possible to define default values for form elements. The value attribute of a text field, the text content inside an text area and the selected or checked attribute of drop down or multiple choice lists resp. radio buttons or check boxes are the values that are displayed as default when the user accesses the form for the first time. The user may then reuse or overwrite those values.

4.7 API

Several events such as onRender, onSubmit and onSave can be handled by external scripts.

For example in case of onRender define events, a variable called onRender or a function called onRender needs to be defined globally. However, this allows for defining just one function for each event. To define more functions for one event, the function might dispatch to other functions

4.7.1 onRender

In case onRender is defined, the article resource containing the script reference will run the script before rendering the page.

onRender is called with one argument which is the root node of a resource.

The onRender function of a script is evaluated each time the page is accessed, even though future versions of the scripting module might cache the result of script evaluation and only re-render when a resource has been changed. If caching is used, dynamic scripts such as printing the current time will change their semantics.

If the script has written to System.out (e.g. with `print('');`) then this is used instead of the returned value. The reason for favoring the System.out is that the result is more reliable than the returned value, as the last value is returned even if it is not apparent for the user. In other words, the print statement has precedence over the return statement.

For security reasons, persistent modifications during onRender will throw an error to avoid an excessive number of commits.

4.7.2 onSubmit

onSubmit is called with two arguments, where the first argument is the root node of the resource (e.g. article resource) and the second argument is a key-value-map

of the submitted for elements.

When several forms exist on a page, the onSubmit function may do get the form name and even the button name to dispatch to an appropriate submit handling function.

4.7.3 onSave

If the onSave function is defined inside the script, then script evaluation of this function is triggered when the article with the script reference is saved. The article with the script referencce is the context resource on which the script may perform reading and manipulation operation.

onSave is triggered whenever changes to the observed pages occur in the form of a save operation on an article. The argument passed to onSave is the context resource node.

4.7.4 Event listeners

While the functions (or variables) onRender, onSubmit or onSave are well-suited for end-users, experienced programmers might prefer to use event listeners. Explaining event listener concepts to end-user programmers is too complex, and therefore event listeners have been implemented only for helping experienced programmers to develop libraries more efficiently. For example, only with event listeners it is possible to register more than one function to be called on specific events. Developers might work around this shortcoming by developing an own dispatching mechanism inside those functions (e.g. onRender), but it is easier when the scripting module provides that functionality.

Therefore, the scripting module provides the function document.addEventListener, which accepts an event name as first and a function as second argument. The event name can be render, submit and save. However, it is also possible to use custom event names for other purposes. For example, a user might register events named onInvalidFormSubmission and invoke an event and thus all functions registered for that event. Generic event support has been implemened as the event based model (e.g. Node.js cf. section 2.1) is gaining popularity

All scripts may access the default binding document. The reason why document has been chosen instead of a WOM-specific name is that existing scripts need `document.createElement` and other functions based on the document binding. To reduce the overhead of rewriting existing libraries, this convention has been used. Also, the binding name document is generic. The document objects provides

Script type	onRender	onSubmit	onSave	Wikitext[2]
Script expression	yes	no	no	no
Script reference	yes	yes	yes	N/A
Exernal script nodes	yes	yes	yes	no
Script resource	yes	yes	yes	no
Form elements	no	yes	no	yes

Table 4.2: Characteristics of elements introduced by the scripting module

a `getContext()` method which will return a context object depending on where the

4.8 Bindings and context available to scripts

The objects and methods available to different scripts are listed in the Appendix. Those bindings provide the user with methods to manipulate the WOM, to write log messages, to get read-only and writable transactions etc.

4.8.1 Context of script expressions

The object returned via `document.getContext()` has less methods (called functions in JavaScript) than external scripts or interactive scripting. This is due to the concept that script expressions should not contain complex code. Instead, what is expected is one expression statement only. A resource is present for the context.

4.8.2 Context of external scripts

The context of external scripts may get the context resourcce, perform operations on the document (e.g. createElement). And may do more sophisticated operations such as getting the current transaction, getting the a list of tables and others.

Scripts in onRender cannot get a writable transaction, as otherwise each page load might create a new commit.

When a form is submitted, the onRender method has access to getting the form name, all form values and the button which has been submitted. This is to allow more than one form per resource and also more than one button per form.

[2]may contain Wiki markup?

4.8.3 Context for interactive scripting ("CLI")

The interactive scripting context differs from the script expression and external script context in the absence of a context resource. Apart from that, it has almost the same potential as the external script context.

4.9 Simple syntax

End-user programming (cf. 3.1) suggests a very simple syntax

(e.g. click("buttonname");).

The Sweble scripting module in this work encourages the use of libraries, which are developed in any script langugages supported by the Sweble scripting module. For example, it is possible to use a Phyton library for a JavaScript snippet. Libraries defined inside the Sweble scripting module are recommended to be well documented (e.g. source code and documentation embedded inside a Wiki article resource). Due to the nature as Wiki article resource with syntax-highlighted script snippets inside, the documentation is available for all end-users and always up to date. Doubts in the documentation may be corrected instantaneously. The idea is to use a "subsection" of the Wiki (e.g. namespace or "folder") as script repository with script libraries which might be useful for the specific Wiki domain. Therefore, a Wiki can use a set of libraries which define a domain-specific language for the most common cases of that Wiki. Alternatively, it would be possible to implement all possible end-user functions inside the API of the ScriptingContext available to all scripts. However, the simple syntax would go hand in hand with a lot of of globally defined functions for specific domain tasks (e.g. adding a table row from a submitted form), which "pollutes" the global namespace (cf. Herman and Tobin-Hochstadt, 2011). This is an disadvantage for library developers who might need to prefix their function names, which makes those function names less usable for end-users. Therefore, loading the required script libraries which then defined end-user suited functions increases the flexibility and the usability.

JQuery-like syntax

JQuery is a framework or library to reduce verbosity of JavaScript and to make coding more concise. JQuery works heavily with CSS-inspired selectors to operate on sets of elements. In JavaScript, especially in early versions of JavaScriptaccessing and manipulating sets of elements requires a loop and additional variable assignments, which is cumbersome (cf. Freeman, 2013). While jQuery is not directly useable, a jQuery-like syntax can be used when using libraries.

Example for an external script:

```
var ctx = document.getContext();
$(ctx.getDivs(element)).forEach(function(node) {
  document.log(document.getTextContent(node));
});
```

In this example, the function $ converts the Java list to a JavaScript array which is a functionality provided by a referenced external script library. A sample resource with this code can be found when starting the Sweble Wiki with the scripting module in debug mode. forEach is a default construct of JavaScript(cf. p. 132, Resig and Bibeault, 2013).

4.10 Script repositories

The target group of scripting is not necessarily the end-user due to complexities of the Java WOM API which needs transparent handling of round-trip data and text nodes to improve user acceptance. However, the end-user can greatly benefit from collaboratively created script repositories by more experienced developers which are tailored for specific problems. Those libraries can be embedded due to the external script concept used with a single script reference without code. In fact, adding this script reference can be handled by a wizard developed using the form elements and script nodes of the Sweble scripting module. This wizard would let the user decide on which resource the reference is to be added and if needed might add code to address only one out of many elements on the page (e.g. ordered or unordered lists, tables etc.). This is the principle of no coding discussed in section 3.1. Adding script references to a page by scripts is possible due to the expressiveness of the WOM. Alternatively, those libraries can be added by a programming by demonstration tool which observe the actions of a user and use libraries to keep the generated code readable.

In contrast to copy & pasting from other websites, using libraries inside a Wiki are more trustworthy due to collaborative checking and can be expected to do the desired work which is not the case with script snippets originating from blogs, forums and other web pages. Also due to domain knowledge, those WOM manipulations can be more spezialized and contain documentation that is more helpful for the end-user.

Require modules

While end-user development focusses on simplicity for the end-user, experienced developers have other goals, such as reusability and maintainability. This can be achieved by a modular software design and dynamic loading of dependencies. A common approach is the require() mechanism(cf. Herman and Tobin-Hochstadt, 2011), which is not supported by the scripting module natively as it can be embedded with little effort

A require machanism is not supported by the scripting module natively as the implementation would be dependent on specific script languages. However, such a module may be implemented using a script library. The library can call the getCode() method of the required script resource to get the code as String. This code can be passed to eval(). To avoid duplicate code loading, this script needs to keep a list of all referenced scripts

4.11 Permissions

Executing untrusted code and giving users access to the script logging or the interactive scripting is problematic. Users might get access to details which might be used to compromise the system. Therefore, the Sweble scripting module defines the following permissions:

- P_SCRIPT_SIGNER: The user is allowed to sign scripts. This means that scripts created or modified by the respective user will be evaluated by all users accessing the resource.
- P_BLOCK_SCRIPTS: The user is allowed to block scripts.
- P_UNBLOCK_SCRIPTS: The user is allowed to unblock scripts.
- P_SHOW_SCRIPT_LOGGER: The user is allowed to view and use the scripting logger page.
- P_INTERACTIVE_SCRIPTING: The user is allowed to view and use the interactive scripting feature.

4.12 End-user tools

The scripting and form mechanism is a feature for the Sweble wiki, but it is not directly visible on pages itself unless its features get used on resources, e.g. in forms. However, two Wicket pages have been created where users can evaluate

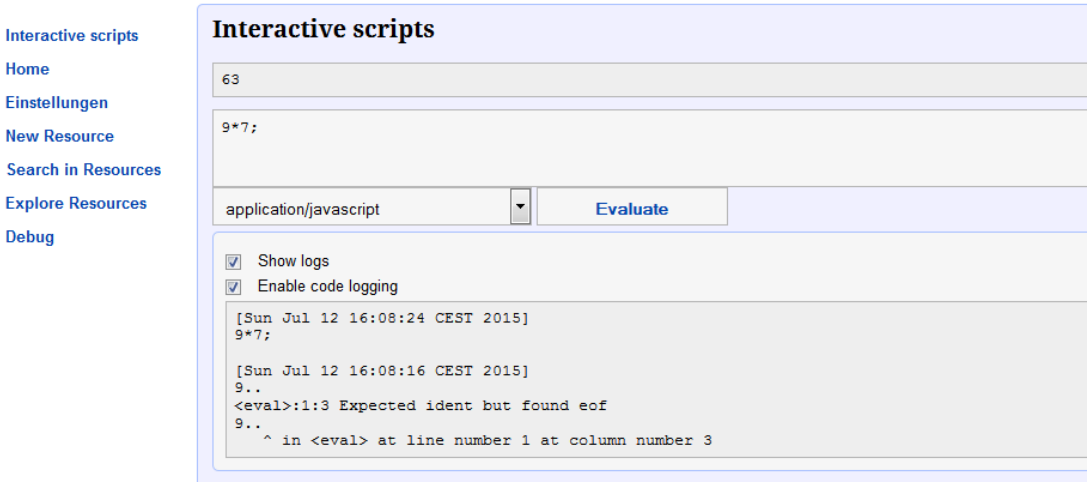

Figure 4.1: Screenshot of the interactive scripting page in the Sweble wiki provided by the Sweble scripting module.

own script (interactive scripting) and may debug pages with errors. These are the two points of contacts with the end user. Moreover, script resources also have their own user interface which is similar the UI of the article resource with the text fields evaluation time and script language (type).

4.12.1 Interactive scripting

Interactive scripting allows entering script snippets and evaluate them "on-the-fly" by pressing the "Evaluate" button. The result is immediately returned and potential error messages are returned. The page allows choosing the script language from all available script languages. Also, it is possible to display a history of all evaluated script snippets (e.g. to compare different codes or algorithms).

Such an interactive console is known by many programming languages. Also, modern browsers mostly have a web console, where JavaScript code can be executed and thus read or manipulate the state of the DOM. The same is possible with the interactive scripting console, which has a menu item in the Sweble wiki when activated. The result of the interactive script input is directly displayed if applicable. Potential exceptions are displayed and enqueued in the history of the interactive scripting console to compare new evaluation results with previous results. This functionality is common for interactive consoles in browsers (e.g. Firefox 38[3] or Chrome 33[4]) and for many programming languages. A user privilege is required for using the interactive scripting feature, as persistent manipu-

[3]cf. https://www.mozilla.org/en-US/firefox/new/ - last visited on 7 July, 2015
[4]cf. www.google.com/chrome/ - last visited on 7 July, 2015

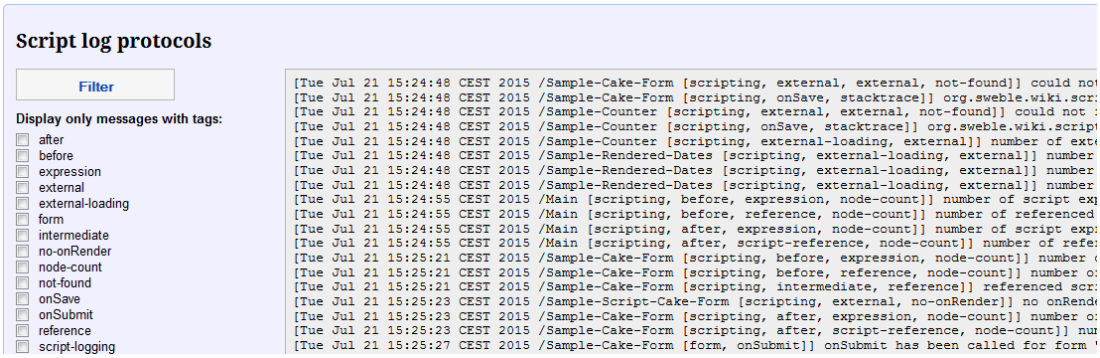

Figure 4.2: Screenshot of the script logging protocol page in the Sweble wiki provided by the Sweble scripting module.

lations might modify the Wiki's state in an undesired way if no authentication is performed.

4.12.2 Script logging

The log messages created by individual scripts can be displayed to the end user with the adequate permission to view the script log. All debug, warning and error messages of the scripting module are annotated with the path (i.e. WRI of the accessed resource), with the date and time and also with tags. Those tags help to filter only relevant messages. This is necessary, as this specific log is not supposed for the administrator, but for the end user. While the administrator log is usally in text form and can be filtered with commands such as grep on Linux system, the end user has not the tools at hand (i.e. command line filtering) nor the knowledge to filter himself. Therefore, a filtering mechanism has been implemented to select which tags should be included in or excluded from the result set. Also, filtering by path (i.e. WRI) and date/time is possible.

A selection of tags used by the scripting module and displayed by the script logger are described in the following:

- expression: logging is about script expressions (e.g. an error in a script expression)
- external: logging is about an external script
- before: logging occured during evaluationTime before
- after: logging occured during evaluationTime after
- no-onRender: no onRender function is defined in the script
- not-found: the referenced resource was not found

- stacktrace: a stacktrace of an error (displayed without new lines to have one log message per line)

The script logging can be opened via the link "Script protocols" under settings and opens a Wicket page. Apart from filtering, also emptying the log messages is a supported operation for the user.

4.13 Sweble module vs. scripting

When new functionality is to be implemented, it can be done as a module for the Sweble wiki in Java. Alternatively, it may be implemented in a scripting language of the choice of the user - provided that it has been enabled by the administrator of the Sweble wiki.

Sweble module

For administrator-driven or developer-driven features implementation as a Sweble module in Java is favorable. Java modules can be easily deployed and can harness the full expressiveness of the Sweble wiki. A powerful IDE such as eclipse can be used for creating Sweble modules. Git or SVN can be used directly for version control of Java Sweble modules while WOM data stored in databases can only be subject to version control when this feature has been implemented in Sweble wiki. Also, Maven can be used for development in Java and, for instance, resolve dependencies to other projects. All these features of using an IDE are not directly usable when using a scripting language, unless these features are included into the Sweble wiki.

Scripting

Simple applications based on forms to edit specific information represented in resources such as articles are more convenient to implement using a script language, as any user with sufficient privileges can contribute to the creation and improvement of these scripts. Based on own observations current wikis such as Wikipedia [5] share mostly knowledge and to some extent images with Wikimedia Commons[6]. Enriching the concept of Wikipedia, the script markup and the script resource can be used to share code and collaboratively create small and well-defined script applications and helper tools, which will be explained in the

[5]cf. https://www.wikipedia.org/ (last visited: 09/07/2015)
[6]cf. https://commons.wikimedia.org/ (last visited: 09/07/2015)

next paragraph. Especially the fact that code can be evaluated and be displayed (external-script) on the same page makes it easier to demonstrate code in-place and thus is a foundation for a collaborative repository for simple scripts (e.g. simple sort algorithms).

Helper tools in this context are Wiki resources which contain instructions and a UI optimized by the end-user for the end user. These pages and the forms on these resources can be the starting point to trigger simple WOM modifications or using the refactoring mechanism provided by Sweble. Therefore, the scripting module may serve as a tool for wiring features of the Sweble wiki which are not directly accessible to the end-user. This is similar to wiring as done in Unix shell scripts (cf. 3.3)

The general advantage of scripting is speed of development which allows rapid prototyping. Using many prototyping cycles with the Sweble wiki is time-consuming on slower computers, as the time required for re-compiling all Java code and executing it, loading the Guice modules and starting the web server takes more time than entering code in the interactive scripting page and pressing "Evaluate". Also, the history of the last evaluated codes can help to try out and evaluate several potential solutions in a trial and error approach.

However, as anybody might edit scripts, security checks are in place which reduce the possibilities of what can be done with a script to avoid malicious use.

5 Comparative evaluation of the Sweble Scripting module with PHP

In the following, the Sweble scripting module will be compared with the well-known web scripting language PHP along several dimensions, including readability, ease-of-use and evolvability.

5.1 Readability

First of all, writing readable scripts is possible with both PHP and Sweble's JavaScript module. It is up to the developer to separate different concerns and to document code readably. However, PHP makes it easy and efficient to program "quick and dirty". Having the controller, model and view (i.e. the HTML page with PHP snippets in one page) is possible and very convenient at the first glance. Only if the programmer uses a web framework or uses one's own convention of separation of concerns readable and manageable code is possible. That means, per se PHP is not designed to be give readable scripts. In combination with some inconsistent conventions in PHP (e.g. order of parameters), sophisticated scripts are hard to read. Also, reusable code requires the use of several files which need to be loaded and parsed for each page request (if no PHP opcode cache is used). Therefore, for optimization purposes, having logic and the view in one file gives a performance benefit, which leads to situations where well-designed scripts require custom caching mechanisms which are not provided by PHP itself to run.

Within the Sweble environment, the way to add scripts supports to use widely accepted design patterns. While using script expressions are possible to avoid complicated and not easily readable code snippets to set the content of a placeholder, the primary approach for adding scripting is by referencing external scripts.

5.2 Ease-of-use

Simple PHP snippets and Sweble's scripting expressions are easy to use. Using a reference or example code, it is a matter of copy and paste to implement simple scripts.

Manipulating the WOM tree, however, requires a basic understanding of the underlying tree data structures to understand the effects well. However, due to automatic escaping inside Sweble, assessing the ease-of-use to create secure resources that are not vulnerable to XSS (cross site scripting) attacks, this is certainly easier with Sweble. With Sweble's scripting module, properly escaped input and output comes out of the box and requires no custom coding. On the other hand, to achieve the same result with PHP, a notion of where the printed text is placed is obligatory. Therefore, with PHP it is necessary to understand escaping rules to figure out why a specific text is not displayed when no escaping is performed (e.g. <). For example, an HTML/XML attribute value needs different escaping than the text content within an HTML/XML element. Therefore, while adding some understandable complexity on the side of the structural representation of the WOM, it comes with the benefit of security and correct display out of the box.

The naive scripting language approach within the Sweble Wiki cannot compete with other templating languages such as PHP. Writing simple functionality is way more cumbersome compared to the templating approach used in PHP and many other web frameworks. For the end user developer, a wide knowledge and understanding of the WOM is required to be able to write simple applications. Simple placeholders such as mathematical computations, however, are as simple to express as in templating languages.

5.3 Reusability

The advantage of a Wiki environment is that code inside the Wiki by trusted users may be reused. While copy and pasting scripts from the web for programming in PHP and similar languages comes with the risk that there are security problems, especially wikis with many users and many code reviewer can provided and maintain code libraries which are reused on many pages. Fixing security problems centrally or improving functionality can therefore be done by updating the library as code can be embedded by just referencing an external script, which works with a generic set of pages. An example for such a generic mechanism is the script "cake planner" (see Appendix) which handles form submissions by automatically detecting in which table cell a specific form field is to be inserted.

Generic nature

Manipulating the WOM without a templating language is significantly more difficult for end-user programmers due to the need to understand the underlying concepts, but it allows more generic manipulations.

Templating languages tend to be more explicitly programmed than scripts which manipulate the WOM. Based on the nodes encountered, specific events are triggered inside the code, manipulating the state accordingly. Manipulating the WOM/-DOM is able to yield more loosely coupled code that can be packed in libraries which can be reused. This is advantageous because the same experienced programmers can write libraries which can be simply reused also by inexperienced programmers, thus reducing the scripting effort solely to configuration. This helps for end-user programming as users can simply combine existing scripts from repositories to achieve the desired effect. However, the readability and understandability of having templates (similar to Wicket templates) would reduce the effort of coding significantly. Coding libraries therefore is relatively complex for end-users while using libraries tailored specifically for a certain domain problem can be done by embedding a script reference. To further ease the task of embedding tools or libraries (e.g. cake planner, see Appendix), an alternative approach of deployment is to create a deployment article resource which lets the user enter or select the WRI an article and then can choose for which element on the page the tool should be used. Then, the inclusion of script references may be handled by the deployment article resource, as it can fetch the resource, modify it accordingly and therefore gives the end-user the possibility to add functionality without any coding. This approach without coding is, as discussed in section 3.1 well-suited for end-user interaction and therefore a clear advantage over web programming languages which do not support that (e.g. PHP).

5.4 Performance

Performance considerations and evolvability are strongly interconnected. When maintainability, readability and evolvability comes with poor performance, many developers will favor the performant approach and ignore good software design standards.

Sweble's scripting module also allows for functions/methods to be called upon initialization. This initialization may be relatively complex as it is done only once and there.

Also, due to script caching as bytecode, there is no need for dirty and efficient implementations. Instead, frequently used scripts will reside mostly inside the

script cache and thus cause little or no overhead for execution. Also, some script methods are called only on specific events, such as onSubmit and onSave. Invoking functions on specific events with raw PHP is not natively by PHP. Therefore, due to domain knowledge of possible operations on articles or other resources, script evaluation is only triggered when really necessary.

5.5 Caching

With well-known script expressions and domain knowledge of when scripts are to be called, it is possible to use fine-granular caching with expiry dates of cached items. For example, an expression yielding the year can be cached until the end of the year. This allows performing some potentially heavy computations beforehand and to use a hierarchy of caches based on the frequency of when cached items change. A downside of storing a WOM/DOM on the server-side is that the performance without caching is poor, as the resource rendering process needs to be performed on each page access, transforming the abstract representation into the target representation.

5.6 Evolvability

The performance benefits discussed in the previous section encourage tailoring scripts to specific actions and to use an event-based approach. Event-based programming with listeners is good programming practice and makes software more evolvable. Other scripts may simply define one more listener which will be called as well. This kind of "hooking" to custom actions is not part of PHP (even though this functionality itself may be implemented by a framework) and thus not natively supported.

However, occasionally, refactoring HTML templates is necessary, e.g. to adapt elements to new HTML standards, such as the abandoned XHTML 2.0 standard. XHTML 2.0 was significantly different from the previous HTML versions, which implies that to conform with the standard, all or most pages need to refactor there HTML code base. In this case, refactoring the templates can only be done manually which is very error-prone task. If elements were addressable by tag name and could be manipulated with a simple generic script, such a shift of HTML would require only little extra work to get compliant. However, almost all templating languages which are widely used (except XSLT) do not use the tree structure at hand and thus the of sophisticated refactoring is hard or impossible. Also, due to the arbitrary text placeholders, telling all the classnames or IDs that may be produced out of a template is not possible. This might be handy

for example, to minify the CSS in combination with the HTML classnames. The only workaround is to tidy the HTML after rendering (as Wikimedia does) and apply transformations on rendered pages before they are sent back to the client. However, this is an unnecessary indirection.

Apache Wicket uses also HTML so that no string concatenation is taking place. Instead, the HTML data structure can be parsed and is modified by Wicket itself to add attributes and content to components based on the models provided to Wicket. This approach of using machine-processable data structures is favorable, as it allows for refactoring of code e.g. to adapt a new HTML standard as described in the previous paragraph.

Refactoring of PHP is not possible, unless conventions are used. This is because inside PHP snippets, the surrounding HTML element tags may be closed and opened arbitrarily. A convention that allows for refactoring is to never print HTML tags themselves with PHP's print or echo statement and instead using HTML outside of PHP tags.

AngularJS uses a DOM-based templating system similar to Apache Wicket, where it extends HTML by attributes for server-side processing (cf. Jain, Mangal and Mehta, 2015). Therefore, the problem of potentially invalid HTML/XML and XSS attacks is catered for by the framework. Instead of "string concatenation" like in PHP, the HTML and its DOM representation serve as a domain-specific language (DSL) which ensures the document is valid and well-formed, to speak in XML terminology. AngularJS is JavaScript-based.

The JVM language Scala also does not need to resort to string concatenation to produce valid XML output. With native XML support provided by Scala, it can operate with XML objects as "first class citizen" (cf. Narmontas and Fancellu, 2014). Therefore, one can create and read from XML data structures without coding. This guarantees valid XML if that XML library used correctly. Scala will also take care about correct XML escaping, thus avoiding classical security threats such as XSS attacks. Scala is a JVM language and therefore available as scripting language in the Java Scripting API. Due to its XML support and the fact that WOM is also represented as XML, Scala is a very suitable language for scripting.

With the library approach that has been suggested under Methods in section 4, improvements and security flaws can be fixed centrally. As no copy and paste from web resources or forums needs to be done for the scripting module, but only generic external scripts need to be referenced, improvements in algorithms can be centrally reflected. Also, code reuse can be on a fine-granular level as due to bytecode caching, performance does not suffer when using huge libraries, which gives one more incentive for code reuse which helps evolvability. That means that due to the possibilty of having libraries, those libraries can be improved and

developed further without having to go to individual resources referencing them.

As discussed in section 4.10, also using mechanisms like require is possible when defining appropriate libraries, which allow maintainable and modular scripts through dynamic loading (cf. Herman and Tobin-Hochstadt, 2011).

5.7 Debugging tools

PHP scripts can be debugged reliably only using the error log, because some fatal errors. Even though there is a way to display error messages to the screen when opening the script via its URL, this is not reliable as some errors simply stop script execution (cf. Cholakov, 2008). In contrast, the scripting module provides logging with a script log protocol with features such as filtering by WRI, type of error or message, date etc. The script log protocol is directly accessible via the Sweble Wiki navigation and can use. Therefore, debugging script problems with the Sweble scripting module has been designed more end-user friendly than debugging PHP scripts.

5.8 Security

Despite the poor performance in terms of usability, operations on the WOM have guaranteed properties. One of these properties is that the generated HTML or XML is always valid, which is not the case in PHP. Requirement for that property is that the implementation of the Sweble Wiki is correct, which shall be assumed in this context. Therefore, "breaking out" of the HTML/XML element tag is not possible. This is comparable to a DSL, which also such as XSLT (cf. section 2.5) has a limited expressiveness, but in turn guaranteed properties such as a valid XML. Most template languages which provide no mechanism to automatically escape user content may have vulnerabilities which can be exploited. The lack in input validation that allows XSS (cf. section 7.8) is also the cause of other vulnerabilities such as SQL injection. In the Sweble Wiki, direct database access is not possible for scripts. Access to resources is possible only via the abstraction layers provided by the Sweble Wiki to the end user scripts. Therefore, the current implementation of the Sweble Wiki is not vulnerable to the XSS and SQL injection exploits.

The Sweble module protects the integrity and confidentiality of information by giving scripts only access to data which is a user has access to in the current transaction. Therefore, handling access policies is catered for by the Sweble engine.

5.9 Versatility

Still, PHP is currently much more versatile than the Sweble scripting module JavaScript of this implementation. This is due to the limitation, that no database direct or low level support, image rendering and other features are supported. On the other hand, when abstract database or image rendering support (e.g. vector graphics to PNG/JPG) is added to Sweble, then the programmer does not need to worry about the specific database which is used as this is taken care of by the Sweble framework.

5.10 Direct comparison with PHP

Several concrete problems of PHP have been discussed in section 3.2.1. Those problems where used as a guideline for creating the Sweble Wiki engine.

Especially the problem that clean code (e.g. object orientation) is at a disadvantage in terms of performance is a problem which is addressed by the Sweble scripting module in this work by providing caching of compiled scripts.

The problem of multiple functions and aliases for very similar purposes has been addressed by providing an unform document binding. Also, the naming convention in the Sweble scripting uses the Java naming convention and is therefore more intuitive than PHP with its mix of C-flavored function names, the inconsequent use of underscores in function names and others. Also, the Sweble Wiki currently does not provide low-level access to database functions. Instead, data can be retrieved from Wiki resources. Future implementations might provide data access without the risk of SQL injections.

Also, while PHP lacks native support for events, the Sweble Scripting module is based on events. onRender, onSubmit and onSave are high-level events which do not exist in PHP in a comparable way.

6 Design

6.1 Scripting module

The scripting module has been defined as a logically separate module that can be included or removed in the Sweble Wiki via configuration.

There are several configuration options such as disabling scripting features which are enabled by default. The administrator might do so for debugging or to speed up script evaluation as computational steps can be skipped. For example, script expressions, the before/after/intermediate evaluation time and the onSave event can be skipped. Also, several debugging messages can be enabled if an administrator and developer wants to do debug the scripting module. The configuration also handles cases whether to execute scripts of specific users.

The `<script eval="server"` attribute is optional, but to avoid confusion with client-side it is advisable to use it whenever possible.

6.2 Events of script invocations

As briefly introduced in section 4.1, these events might trigger script evaluation in the context of a Wiki system

- onRender (i.e. external scripts or script expressions while rendering a resource)
- onSave (i.e. external scripts evaluated when saving a resource with the resource editor)
- onSubmit (i.e. external scripts evaluated when submitting a form on a resource)
- and potentially some currently unimplemented events such as onTimerEvent for regular tasks

Whenever a event occurs (e.g. onRender when rendering a resource, onSubmit when submitting a form on a resource), the resource is traversed by a visitor. This resource will also be used as context resource for the script. The collection of script references with matching evaluationTimes is done for all three evaluationTimes (except for the onSave event). First, all the scripts are evaluated and next the respective function e.g. onRender is called. In case of onRender, also the script expressions will be evaluated and the result with replace the script expression.

6.3 "String concatenation" vs. DOM/WOM manipulation

In the ecosystem of WOM, a tree-based (or XML-representable) data structure already exists. Thus, instead of adding scripting functionality to allow for yet another templating language such as PHP, the decision was to use a scripting language to manipulate the WOM. The WOM can be manipulated persistently or per session. This has been implemented by using an event-based DOM modification. Several events are listend to: onRender, onSubmit and potentially in future implementations onTimerEvent.

Advantages

There are numerous advantages of using DOM/WOM manipulation over templating with languages like PHP where the represented data is not extractable. The advantages are based on the semantic representation which is the achievement of the Sweble engine and enables several applications of the Scripting module. Those potentials will be described in the following:

The semantic web is an attempt to represent information on the web also in a machine-readable and in a machine-processable representation. Many traditional website are only intended for human use and not for machine use. Sometimes it is not even desirable to be usable by machines to control the data within the website and avoid 3rd party applications to make use of the information on a specific website. One requirement of semantically annotated websites and thus for the semantic web is to have structured documents and to know what information is represented by the data on the website. The WOM is a structured representation of the information in a wiki which can be the basis for a semantic web application which creates additional value with the information present in the wiki.

A derivative of semantic web technologies in the wiki ecosystem are semantic wi-

kis. Semantic wikis make excessive use of annotation to make information more searchable and retrievable by intelligent agents which answer queries by humans or other machines against the information in a wiki. A WOM/DOM representation makes it easy to utilize the advantages of semantic data as information extraction can be done directly without a previous parsing step. Therefore, even simple applications implemented with the Scripting module of this work can run semantic queries against the wiki.

However, it should be noted that the current wiki implementation does not provide tools to semantically annotate data (e.g. date, price, number, country name) as semantic wikis do. The purpose of this section is only to discuss the advantages of the WOM manipulation for potential future semantic features of the Sweble wiki.

Disadvantages

The disadvantage of having a WOM/DOM representation as the internal data representation is that performance suffers considerably in naive implementations. For each page rendering, the WOM representation has to be transformed to an HTML page. However, one may use the context knowledge to implement an efficient caching mechanism or a caching hierarchy to improve performance. This caching mechanism can outperform traditional templating approaches, as they mostly have no clear knowledge of when the value rendered into a template will expire and thus all script snippets inside a template are evaluated equally on each page load. Therefore, the WOM/DOM implementation in combination with annotation for caching can be improved in performance significantly.

Also, manipulating pages via the WOM requires considerably more code than a templating approach.

Conclusion

The advantage of templating with content-aware templating systems as the one used by Wicket greatly reduce the complexity of code required for performing WOM manipulation options. Therefore, to use both the advantages of the WOM and its semantic data and templating should be provided. To do so, a templating language for the WOM can be created inspired by the templating done in Wicket.

6.4 JavaScript as main scripting language for Sweble

I decided to use JavaScript as primary scripting language for several reasons. First of all, JavaScript may be considered as the lingua franca of the client-side scripting. Therefore, JavaScript is known by many people who have contact with simple web development tasks. Using a language that the end-user is probably aware of or if not, is likely to use again in another context such as private blogs, can increase acceptance of the scripting language. Secondly, JavaScript is considered the default scripting language of the Java Scripting API, which demonstrates its importance as scripting language.

JavaScript offers object orientation and inheritance by prototypes and prototype inheritance which gives developers the tools to develop more sophisticated applications with it (cf. p. 150, Resig and Bibeault, 2013). With it functional programming approach, JavaScript also offers closures (i.e. the scope created when declaring a function), partially applying functions and extending the language which features such as map (cf. p. 92, Resig and Bibeault, 2013). In case of the scripting module, the language can be extended by referencing external script libraries. Due to the success of JavaScript success on the web, it has been continuously improved while missing functionality for older browsers can be added by libraries. The advantage of server-side JavaScript is that cross-browser support issues do not exist as code is only executed on the server. For example, the function (method) forEach can be called on an array, passing a function as argument which is called for each element of the array (cf. p. 132, Resig and Bibeault, 2013).

Furthermore, JavaScript on the server-side is frequently used to validate input before entering it into the database. In our context, the database is the WOM itself, as it has tables, lists and other elements to store structured data. The trend goes to adding the validation on the client-side as well to increase responsiveness. For example, when the user enters a wrong value, the input field itself may change its color to red and a warning message might appear. In a conventional environment, the user might learn about this error only once the form has been submitted. In productive environments, a CAPTCHA may be embedded as well to avoid spam and fake submission and such a CAPTCHA challenge needs to be solved again on every attempt to submit that form. After several attempts, entering a CAPTCHA again and again might lead to frustration and cause the user to give up. Therefore, most web forms nowadays try to warn while entering data, which needs duplication of validation code on the client and on the server side. This is extra work and error-prone, as what is said to be valid on the client side should not return an error after submitting the form to the server and vice

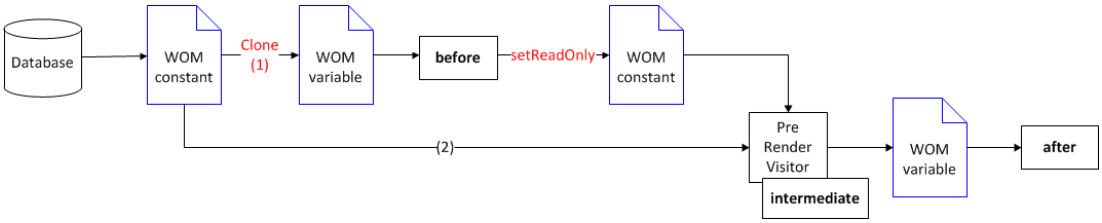

Figure 6.1: The script processing pipeline with the evaluation times "before", "intermediate" and "after".

versa. JavaScript gives the opportunity to reuse validation code on the server side as well as on the client side and thus saves work and avoids unexpected behavior.

6.5 Evaluation time

As figure 6.1 demonstrates, the rendering pipeline works as described here: The database holds the WOM representation of a resource which might be stored in memory for caching purposes. As several page accesses might use this WOM representation simultaneously, the WOM document is read-only. Some scripts might have the need to perform operations at this stage (called evaluation time before in this Scripting module). If a script for evaluation time before exists, then the constant WOM is cloned and the copy may be modified by the script. After this step, the WOM is made read-only.

The next stage in the pre-render visitation pipeline is the PreRenderVisitor . The PreRenderVisitor always receives a constant WOM resource either from the database or the cache or from the evaluation time before scripts. Due to the stacked visitor concept, modifying the nodes by scripts as soon as they are processed leads to a behavior which is hard to predict as some nodes are already processed while others are not. Therefore, scripts running during the PreRenderVisitor stacked visitor processing are required to be side effect free and therefore may not modify the WOM. However, they may return a value (e.g. a String, a number or any Wom3 (i.e. WOM version 3) element potentially with children. If the value is a Wom3 element, it replaces the script reference node and this sub tree gets processed again by the stacked visitor.

The final stage is the processing of scripts at evaluation time after. Those scripts operate with the result tree which is not shared and not cached and therefore may be modified freely. As the visitation is already finished at this stage, nodes are not redispatched or processed further. Returning a value will replace the script reference or expression node by the returned value. If no value is returned, then the script reference of script expression node will by removed.

Other scripts are evaluated on the same context with the bindings of the previous, in the order they are placed on a page. Different Wiki resources use different contexts. Also, script evaluations of different users will always use their own script context. The order of script execution depends on the order of the script references; it does not depend on the order how external-script node are placed.

6.6 Execution model

Several concepts have been introduced to allow for managing the scripts created by Wiki users. Scripts are automatically signed by creating an hash over the string content of the script. This script signature is placed as an attribute of the WOM script, together with the last user who modified the script. To evaluate scripts, the last editing user of the script needs to be authorized to create or modify scripts (cf. section 4.5). A hash code computed over the code of a script or the reference of a script reference and stored as an attribute of the script. When saving a page with scripts, this code hash is compared with the hashed new code which might be potentially edited. If there is a hash mismatch, then the last user attribute of the script is changed to the current user. When the script is not signed by an authorized user, an error is thrown.

An alternative approach is to evaluated the last signed script (i.e. go back to the last commit version of a resource where the script is signed by an authorized user). However, this might result in a situation that a script is evaluated on a resource which has a different structure than the one the script was originally designed for. Therefore, evaluating the last signed script might go along undesirable consequences.

Other concepts that have been implemented is blocking a script by adding an attribute to the script that only a user with unblocking privileges may remove again.

6.7 Script logging

End-user scripts do not use the default logger of Sweble, but a custom logger which may be opened by privileged users. This logging mechanism is using tags to help end-users filter log messages.

The filtering is implemented in the classes implementing the FilterConstraints and FilterConstraint interface. A FilterConstraint is a Object value combined with a type (e.g. WRI, date, tag). The filter shown on the Script protocol page of the Sweble Wiki can be set by the user and is then transformed to a collection of

FilterConstraint, i.e. FilterConstraints. When rendering the log messages, a check is performed if the filter criterion is matched and if not, the item is not displayed.

7 Implementation

7.1 Sweble module

The scripting module can define its own

- WOM nodes,
- create menu items,
- custom media types and media type definitions,
- custom permissions and permission groups,
- PreRenderVisitors,
- PreSaveVisitors,
- PreRenderHtmlVisitors,
- custom tag extensions,
- resource transformers and
- it can mount custom pages for the navigation bar of the Sweble wiki.

7.2 Sweble Wiki

The Sweble Wiki stores WOM represents the state of the Wiki and its resource. This approach is not common in the CMS and Wiki environment and will be explained in this section.

7.2.1 Resources

Originally, the WOM represented only article resources. Recent changes in the Sweble Wiki implementation allowed resources. Examples for resources are art-

icle resources, raster or vector graphic image resources, type resources or script resources. Dohrn and Riehle, 2013 demonstrate how XSLT based templates can perform refactorings on wiki articles. Since the wider notion of resources exists, any type of resources may be subject to content refactorings. While transformation with XSLT are ideal for skilled developers due to the complexity of XSLT, programmatic approaches might be more reasonable for simple transformations which can be permanent or temporary. The Sweble scripting module gives the tools for programmatic manipulation of the WOM. All resource types can or might be manipulated by scripts when tools for manipulation are provided by the Sweble engine or its modules.

7.2.2 Transformation and presentation of resources

Each resource is represented by a WOM3 data structure. Depending on the desired presentation type, there are different transformations. For example, an article resource may be transformed to a LaTex file to be transformed to a PDF file in the next step. An article resource may also be represented as a static HTML file. For both LaTex/PDF rendering and static HTML file rendering, Wicket does not appear at all in the rendering process.

The idea of the transformation approach of the Sweble Wiki is that there is just one central representation for any type of data (raster graphic, vector graphic, scripting resource, article resource), which is all internally represented or representable as the WOM XML data structure. The transformation works by providing a source with its media type and a target media type. Then, if such a transformation is implemented, the source will be transformed to the target type. The media types to which one might want to export are XML Wikitext, to HTML, to PDF to image etc.

7.2.3 Transformation of Wikitext to internal representations

As mentioned before, the WOM represents all elements. To do so, the Wikitext is transformed into internal representations.

Tag extension

HTML/XML are supported by Sweble wiki as well as tag extensions. The difference of Sweble's XML elements (Wom3Element) to tag extensions is that markup inside tag extensions is ignored and remains there as string, while content inside

Wom3Elements is parsed and markup is fully interpreted. Therefore, whenever the text content of a node is a different media type such as code, a tag extension is used.

Tag extensions have been used `<external-script>` or for script expressions.

Tag extensions are a Wikitext specific phenomenon. However, also other wiki markup languages can be supported in future versions of Sweble Wiki; then these concepts may have a different name. The scripting module can be integrated to these markup languages easily as well. The only thing that the markup needs to provide are pedants of the Wom3Element and unparsed/unprocessed content similar to tag extensions.

XML nodes

XML/HTML markup is used to represent forms and form elements and script references.

This markup is transformed into a Sweble Wiki representation e.g. S2weButton in the case of a button. This representation in turn will be used by the RenderHtmlFormVisitor, a subclass of the RenderHtmlVisitorBase. It adds visit methods for the form elements defined by the scripting module.

7.3 Markup generation

RenderHtmlFormVisitor extends the RenderHtmlVisitorBase which prints HTML code whenever any form element (i.e. implementing S2weFormElement) handled by the RenderHtmlFormVisitor is encountered. For annotating the markup for further processing by Wicket, the generated HTML uses Wicket attributes with the Wicket namespace. The generated HTML is not passed directly by the server to the client, but parsed by Wicket and the Wicket components are re-rendered by Wicket and enriched with several attributes.

7.4 Forms and form elements

The onSubmit functionality is encapsulated in the custom Wicket form WicketScriptingForm. As Wicket forms may be serialized, the state hold by this form needs to be minimal. Therefore, inside this WicketScriptingForm no references to the transaction system or the scripting module for some configuration parameters

is made. Instead, those variables are populated on runtime with the objects by the injector.

Attribute value verification

To ensure integrity of attributes, the Sweble engine provides an attribute verification mechanism which is used also by the scripting module.

VerifyScriptTypeFn is a mechanism that ensures that the script type always starts with application/*, where the asterix is a placeholder for the respective script language name.

Another mechanism to verify the attribute value for its integrity is WRIStringVerifyFn. It ensures that an entered WRI string is syntactically correct.

The application/javscript type is inspired by Thunderbird and Firefox apps, which use this syntax to mark scripts that have the privilege to access its powerful API. Moreover, it is the default media type defined by the W3C (cf. McCarron, 2009).

7.5 ScriptResource

The Sweble Wiki can have custom resources. Article resources are the default resource type, which was originally the only available resource in Sweble. For the scripting module, a custom script resource has been added. It allows for the operations view, edit and export within the Wicket UI and can be referenced by script tags. That means that instead of an external-script node, a script resource can be referenced.

The text content of the script resource is the script code to be evaluated. A script resource can only have code in one specific script language. A script resources may only contain one script unlike external script nodes which can appear inside an article several times.

Script language

The script type (i.e. the script language) should be allowed to be any string starting with application/*. There exist many script languages and for all of them it is theoretically possible to integrate them by adding the script language to the Java Scripting API and therefore to the Sweble scripting module. If the script language was restricted to script languages existing in the current Sweble Wiki,

importing dumps from other Wikis with more supported script languages might throw an error and skip importing. This might leave out some resources only due to missing scripting languages. One needs to consider that each running instance of a Sweble wiki might support a set of script languages that do not necessarily overlap.

Instead, the favoured approach is to allow any script type tag (i.e. allow existing and potentially new or upcoming scripting languages). If the script type is not understood by the Sweble engine, an error is thrown in case the script is to be evaluated.

Sweble Wiki modules can register TransformerFactories. The required TransformerFactories are script to WOM and WOM to script. Each TransformerFactory defines an accept method for a WOM tansformer source or target which returns if the TransformerFactory can process the source and target or not. The script to WOM and vice versa transformations are identity transformations. Other transformation types are pre save, pre render transformation and resource summarization.

Transformer

The Sweble wiki uses a WOM representation, which is internally represented as XML. Via this intermediate step a source can be transformed into (almost) any target representation. For example, wiki markup can be transformed to a PDF, HTML file or an image. Tranformations can be information-preserving or not. If they are information-preserving, applying the transformation and then inverse transformation will generate the original WOM. A script code to WOM and a WOM to script code transformer have been implemented. Also, a script to resource summary transformer was required.

UI

ScriptResources have the operations view, edit and export within the Wicket UI and can be referenced by script tags. That means that instead of an external-script node, a dedicated script resource can be referenced.

Import

Script resources may be populated from files using the BootstrapFromFlatFiles-Module by the Sweble module. Currently the mapping of these bootstrap files

is done only for .js files. Those files will be mapped to script resources with the script type (media type) application/javascript. As the scripting module provides a generic scripting language support, the use of other script languages is possibly by adding new file names to the BootstrapFromFlatFilesModule.

7.6 Form submissions and markup

A problem of the scripting module and Wicket is that when a resource is rendered and the form is submitted, the markup is not generated again. This is due to the binding of Java objects to the markup. Therefore, the modifications by a script are not reflected directly in the page displayed after submission. To remedy this situation, a reload of the page is enforced.

7.7 Wicket dependencies

Most of the implementation is not Wicket specific. The points of contact are only the forms as well as adding static client-side JavaScript and CSS resources for syntax highlighting. The resource part later on was taken oven by the Sweble project in a modified form. For forms, the HTML generation process was modified to allow for a custom markup input stream that allows Wicket markup. So far, only HTML was allowed. The markup input stream comes with the overhead of parsing the data structure after it has been created from the WOM representation. Alternative implementation approaches that include low-level changes in the Wicket core are not flexible enough for future versions of Wicket.

7.8 Security

For security reasons, JavaScript scripts are evaluated with a class filter. The class filter is a mechanism provided by Nashorn to restrict access to specific classes. This is similar to a custom classloader, with the difference that it is more lightweight and less coding is required. As the functionality of the classloader should remain mostly unchanged and only rules are required to block unwanted classes need to be enforced, the class loader concept reduces duplicate code. Whenever a class is to be loaded, the exposeToScript method is loaded with the class name as string and therefore, low level class accesses can be restricted. In the current implementation, a requirement for loadable class names is that they begin with the Sweble Wiki's package name or are in the java.util package.

Two PreSave visitors are defined by the scripting module: ScriptingPreSaveVisitor and FormVisitor. The ScriptingVisitor processes script tags and external-script tags, which are originally of the type SwcTagExtension and are then transformed to dedicated nodes of their respective types. The FormVisitor transforms XML tags in the Wiki markup (i.e. of the type SwcXmlElement) to dedicated nodes when saving the resource, e.g. S2weForm. When other wiki markup dialects are implemented, the respective equivalents would similarly get transformed by custom visit methods. For evolvability, the goal is to have all further script processing in the pre render and HTML rendering phase independent of the original markup language used.

The scripting module defines the PreRender visitor ScriptingVisitor. The ScriptingVisitor processes script tags and external-script tags. For forms, such a visitor is not necessary as the forms are directly rendered by the RenderHtmlFormVisitor to Wicket HTML.

7.9 Round-trip data (RTD)

Form elements and script elements are transformed into dedicated S2weNodes, e.g. S2weForm before they are saved. They are represented by XML nodes with a custom element name. This representation however by default lacks the round-trip data which is required to reproduce the original code from the WOM representation. The text content of Wom3Text and Wom3Rtd is used for the recursive Wikitext reconstruction.

The ScriptingPreSaveVisitor and the FormVisitor create dedicated extending classes of S2weNode. To allow for reconstruction of the original Wikitext, those nodes have Wom3Rtd nodes which hold the Wikitext in such a way that concattenatting recursively all Wom3Rtd and Wom3Text nodes gives the original Wikitext. As XML or HTML tags are need to be stored as RTD to reconstruct the original Wikitext, those nodes normally have one Wom3Rtd as first child, followed by potentially one Wom3Text and/or S2weNodes children and as last child a Wom3Rtd. The Wom3Text holds the content of the node, such as the label of a button or the content of a text area. To change the text content (e.g. setLabel of a button), the respective Wom3Text node is found its content is updated. For creating RTD, the class RtdToolbox provides several helper methods for creating the RTD nodes recursively and appending them to the S2weNodes.

RTD correction mechanism

Scripts may manipulate the WOM without adjusting the Wom3Rtd nodes. For temporary modifications of the WOM this is not a problem. For example, a form submission may modify the WOM without commiting and therefore display the changes only for the page rendering for the specific user. However, as soon an commit occurs, the changes to the page remain permanent. When viewing the source code or editing the page, the displayed Wiki markup (e.g. Mediawiki Wikitext) does not necessarily represent the stored markup as some RTD information might be missing. Moreover, the end-user cannot be expected to fix the RTD himself, as the concept of RTD is very low level and already complex to handle correctly for experienced developers.

To avoid having an unmatching Wikitext representation which does not represent the WOM exactly, an mechanism to traverse the WOM after a commit or onSave is required. Such a mechanism exists in the Sweble Wiki in the form of a visitor which traverses the document and fixes or removes RTD nodes. Therefore, there was no need to ensure a matching RTD representation by the Sweble scripting module.

7.10 Context of scripts

All scripts can get a custom context object depending on how the script is executed. Within the current implementation, one can distingish between:

- execution as script expression (CtxExpression),
- execution as external script, e.g. script resource or external script during onSave and onSubmit(CtxExternalWritable),
- execution as external script, e.g. script resource or external script during onRender,(CtxExternal) or
- execution with the interactive scripting page (CtxInteractive).

Both external scripts as well and script expressions have a context resource during execution. For example, the page referencing the script onRender is used as context resource. For onSave, it is the resource being saved that is the context resource. In case of onSubmit, it is the article resource containing the form that will server as context resource. The script resource or the article resource containing the executed external script are not used as script resource. Interactive scripts have no context resource, i.e. a resource upon which default WOM operations

are performed. Of course, a interactive script may load a resource and perform operations on it, but this requires to explicitly load that specific resource.

Both external scripts and interactive scripts have access to specific "additional" methods as having more than one statement in those scripts is intended. Those methods include getTx(), as those scripts might commit changes to the WOM.

In case of onRender, committing a change is not allowed for performance reasons. Otherwise, each rendering of a page would result in a new revision of a resource. Therefore, getWritableTx() is not available.

7.11 Embedding JavaScript in Java

One way of enabling end-user scripting would be to use Nashorn, which is a Java-based JavaScript interpreter and a successor of RhinoJS. Nashorn is default scripting engine of the Java Scripting API since version Java 8.

JavaScript, similarly to Wikitext in the current implementation or HTML, allows "syntax errors" and resolves them in a best-effort approach.

Parameters can be passed to a script engine by calling the method put(String paramName, String paramValue) on the script engine. Passing parameters from the script back to Java is done via the method get(String paramName)

7.11.1 Script engine discovery

The ScriptEngineManager is a registry for all scripting engines available to the user (cf. p. 391, Bosanac, 2007).

The Java Scripting API provides its own mechanism to map media types to script engines. To do so, the ScriptEngineManager by the Java Scripting API also allows for getting a script engine by media type name, however in this case script engine mappings cannot be customized. Also, this approach keeps the option open to assign specific script engines to users if this is needed one day.

7.11.2 JSR optimization

If script engines implement the Compilable such as the JavaScript engine Nashorn then the script can be compiled to Java bytecode and thus be executed very efficiently (cf. p. 432, p. 437, Bosanac, 2007). This bytecode can be cached to speed up script execution. This optimized approach is used in the Sweble Scripting implementation. For script caching, an efficient hash is computed over

the source code and the compiled script is store in a LRU (last recently used) cache.

Compilable scripts are cached in the ScriptCache singleton class.

Another interface which can be implemented by scripting engines is Invocable. If implemented, functions defined in scripts can be invoked from the host Java code. The same applies to methods of objects (cf. p. 432, Bosanac, 2007).

A ScriptEngine provides a method `eval()` which accepts a String and a Reader. In the current implementation, the string of the text content of the WOM representation is passed to eval. Passing a reader instance might be favorable in cases where not all the script code should reside in memory. For simple scripts of the size expected for the Sweble engine using a Reader or the String directly should not make a difference (cf. p. 17, Sharan, 2014)

Script contexts define the environment which is used and modified when the script is executed. All script engines have a default script context which is the engine script context (cf. p. 27, Sharan, 2014).

When a script engine's eval() method is called with a script context or binding as 2nd argument, then either the provided script context is used or a new script context is instantiated for the provided binding, leaving the default script context unmodified (cf. p. 40, Sharan, 2014).

eval() returns the return value of the last script expression. This approach is only used for script expressions for the Sweble scripting Module, as (cf. p. 40, Sharan, 2014).

At the moment, the information if a external script (or script resource) contains a onRender method. Therefore, after rendering a script, it should, the script reference should be updated if onRender exists or not. This information should be updated on every change of the involved resources. While for onSave and onSubmit this is also true that the information if such event listeners, it has less impact as these events occur more rarely.

7.11.3 Bindings

The JSR 233 allows defining bindings, which are variables, functions/methods available to the script. This type of binding is known as programmatic binding, as the engine is invoked from the host Java application (cf. p. 442, Bosanac, 2007). For example, the function `document.write('test');` that is used in HTML with JavaScript, might be implemented by binding a document class instance to the script engine which exposes the method write(String message) to the outside.

7.11.4 ScriptContext

The script context is the base on which scripts operate. It is a set of namespaces that a script can access. When the ScriptContext of one script language is provided to another script, then the second script can access the variables and bindings defined by the previous script (cf. p. 442, Bosanac, 2007).

7.11.5 JQuery

The JavaScript library JQuery reduces verbosity of JavaScript and has more powerful statements than the JavaScript itself. The idea was to allow for jQuery-inspired syntax also to manipulate WOM objects. When the manipulation of the WOM is more sophisticated, using the classical WOM interface requires a lot of coding for little functionality. JQuery works heavily with CSS-inspired selectors to operate on sets of elements. In JavaScript, accessing and manipulating sets of elements requires a loop and additional variable assignments, which is cumbersome (cf. Freeman, 2013).

However, jQuery depends on the window and is not directly usable without re-implementing most parts of the jQuery library. Also, the difference of the WOM and the DOM model (e.g. Wom3Text, innerHTML, setTimeout) do not allow for direct usage.

7.12 Unit tests

Many scripting and form tests have been implemented. These unit tests may serve as a good reference and a starting point for writing own scripts as they provide code samples.

Unit tests have to purpose to test functionality and to ensure that after several cycles of code modifications the functionality remains unchanged. On top of that unit tests help to document the code by providing usage examples and demonstrating how edge cases are handled (cf. p. 53-54, Bosanac, 2007)

Regarding forms, all variants e.g. checked and unchecked checkboxes or radio boxes are tested. For text areas, text fields, select choices and multiple select choices and buttons. In all cases, tests exists to ensure that the form elements work as expected and the reconstructed RTD matches the original Wikitext. Also, there are unit tests to check if the pre-rendering leaves the RTD intact.

For scripting, the limitation of testing is that other resources cannot be loaded in the simplified testing ecosystem. Therefore, script references and external scripts

are in the same file. The test cases cover, for example:

- blocked scripts which are supposed to throw an exception when called,
- client-side script expressions which are to be ignored by the server-side scripting module,
- referencing one or more external scripts using the fragment syntax (e.g. `/Article#script1`),
- script expressions and external scripts with different execution times (before, intermediate/default, after),
- handling of incorrect script nodes,
- and failure in case of an invalid script, e.g. with syntax errors.

Testing script mechanisms and forms is not sufficient to test if script execution works as expected. Therefore, additional sample scripts are tested by ScriptCodeTest and ScriptNoJsTest. While the ScriptCodeTest has exclusively JavaScript test cases, ScriptNoJsTest has sample scripts in scripting languages other than JavaScript such as Python or Groovy. Also, mixing different scripting languages is tested there, as variables or bindings defined in a script may get accessed by scripts executed subsequently. Other tests ensure that the order of script execution (i.e. in the order of script references) works as expected. Examples for JavaScript tests are to ensure

- the existence of the Nashorn JavaScript engine,
- the definition of global engine variables,
- that the end-user logging mechanism and the order of log messages works correctly,
- that returned values and using the print() method,
- that separate scripts using the same script engine manager run independently,
- that unsafe operations are not permitted,
- and that returned Wom3Nodes such as Wom3Bold are correctly embedded in the article resource page which references the script.

Unit tests have been especially useful as during my work on my master thesis, some fundamental changes have been done to the Sweble wiki have been performed and required to test the implementation after merging with the scripting development branch with the main development branch.

Due to restructuring of the Sweble Wiki, at the time of implementation, no integration tests have been possible as there was no in-memory database for

tests. Therefore, integration tests have been done manually. Wicket pages such as the interactive scripting page and the logger page might be tested when integration tests are supported using org.apache.wicket.util.tester.WicketTester and org.apache.wicket.util.tester.FormTester. Those test can be done without starting the server process (cf. Gurumurthy, 2006).

7.13 Alternative implementations

Alternatives to the Java Scripting API exists such as the Bean Scripting Framework as covered in the section 3.5.

Instead of defining listeners within a script by defining well-known functions, it is also possible to define several script snippets and define the event when they are called as attribute. The downside of this approach is that no logic can be used in defining these listeners. For example, one might want to use a factory to return the onRender JavaScript function. This can be done easily with the implemented approach. Also, when using libraries and other scripts, it is more convenient not have no context switches between JavaScript and Wikitext several times.

Instead of using Nashorn as script engine for JavaScript, it is also possible use the JavaScript v8 engine by Google. The good performance of the v8 engine has lead to several noteworthy implementation around the v8 engine, one of which is Node.js, an event-driven web server. It is possible to use the v8 engine directly which is known due to its performance. However, as the implementation of v8 is not available in Java communication overhead from Java to the v8 engine. Therefore, v8 engine will only outperform Nashorn for heavy computations.

8 Conclusion

The Sweble scripting module which has been implemented in this thesis is fully functional. It enables end-users include script expressions and references to external scripts into resources. External scripts can be script resources or script snippets inside an article resource, possibly surrounded by Wikitext documentation. External scripts may define functions which are called on specific events such as on rendering a resource, when submitting a form or when saving a resource. Those scripts may perform WOM manipulations which allow for simple web applications inside the Sweble wiki. The script logging mechanism and the interactive scripting command line are intended to aid end-users in writing scripts, even though they are not as powerful as debugging tools of IDEs, but still guide in fixing small problems.

The use of the JSR-233 Scripting API allows integrating almost any scripting language due its generic nature. Therefore, and due to the this work is not limited to JavaScript. However, JavaScript has been the focus of this work and the base of most tests. Support for other scripting languages than JavaScript goes beyond the original scope of this work. Also, with the Java Scripting API, innovative new human readable scripting languages as discussed in Cypher et al., 2010 can therefore be used as well as programming by example approaches.

When coding with the scripting module, functions are provided by the scripting module which allow jQuery-like syntax in combination with libraries (e.g. `$(ctx.getDivs()).forEach(func);` cf. section 4.9). Even though the WOM Java API is not ideal and not intended for end-user programming, this syntax increases understandability for library developers. With script repositories, improvements which affect only a small subset of pages can be done decentralized without requiring a top-down intiative of the Wiki operator or administrator.

The scripting module is specifically tailored for automating and customizing (cf. section 3.1) recurring tasks which can be triggered by scripts and form buttons instead of manual work. While research in end-user programming suggests approaches such as programming by demonstration and a very simple syntax, the approach chosen in this work is allowing for libraries which might be located

inside the Wiki which serves as a script repository. Simple syntax would go hand in hand with a lot of of globally defined functions for specific domain tasks (e.g. adding a table row from a submitted form), which contradicts development guidelines of separating concerns into different namespaces (cf. section). The library or script repository approach enables users to contribute code libraries which can be referenced by less experienced users by including a script reference. In fact, even the task of including the script reference can be automated and handled by a wizard developed using the Sweble scripting module. When script repositories of this sort have been developed based on the Sweble scripting module, this empowers users to automate and customize repetitive tasks without coding.

9 Future work

The Sweble scripting module provides the base for research work in the future.

Most importantly, by collaboratively creating a script repository similar to the script repository of CoScripter (cf. section 3.1), a wider range of sample scripts needs to be implemented in the Wiki. Some samples are already present when starting the Wiki in debug mode. Script libraries need to be grouped, documented and linked between each other to help the user to navigate to the best script which is shipped together with the Wiki. Also, article resources and other resources need to be created to inform users of the possibilities of the scripting module and how it can automate repetitive tasks with little effort or without coding.

In this work, no end user study has been performed, as this was not the scope of the this and it was not possible due to time limitations. Future work includes an in-depth assessment of the strengths and weaknesses of the scripting language approach as it has been implemented within this work. Especially, the usage of libraries which hide some complexities of the WOM and its API might help in the acceptance of the Scripting module and are worth to be investigated further.

Manipulating the WOM works well as long as only the number of elements to read, traverse and manipulate is very low. However, for more sophisticated applications a templating mechanism inspired by Apache Wicket can help considerably to develop applications. Such a template language can be built using markup inheritance or by grouping components similar to as introduced in section 2.3.

The current Sweble Java API for WOM manipulation is complex for end-users and more complex than the DOM API provided for HTML DOM manipulation. For example, setTextContent on a table cell does not set the text content yet, but it is possible to append a paragraph child to which an Wom3Text child is appended with the text content that is to be displayed. For easier scripting, it would be convenient to have an additional method such as setTextValue or similar which can be called on a table cell, list item etc. and adds the required child nodes transparently for the user. The existing setTextContent can and should remain unchanged. This would considerably lower the barriers for end-user programming and help developers to create libraries.

At the moment, applications in the Sweble Wiki are far away from being fully-fleged web applications. For instance, script applications can only store data in resources. Article resources are not always the best fit as a storage location, even though they might serve as a rudimentary "database". Options for databases are key-value stores (e.g. as database resource), conventional SQL database with input validation to avoid SQL injection, noSQL databases or an XML databases (similar to WOM) which can be queried with XPath statements. PHP offers features such as image, audio and file manipulation and processing, creating zip archives (e.g. for download) and authentication. The implementation of these features as Sweble modules which are accessible by scripts would go hand-in-hand with an upgrade in attractiveness of Sweble Wiki as a scripting environment. For example, offering a manipulated image file for download as archived file might be a feature needed on individual resources which might be implemented using scripting.

Also, the fact that scripts operate on well-known data structures can be used for very fine-granular caching. A script, which only prints the current year, for example, can be cached until the end of the year. The performance gains of cache hierarchies might be an interesting topic to investigate further.

Currently, referenced onRender scripts are evaluated on every page load, which enables highly dynamic applications, but comes with poor performance. Possibly, the onRender script results might be cached until a dependent resource changes or another event occurs (e.g. regular events). If such a decision is made, script expression returning e.g. the current date would have a different semantic which needs to be communicated to the end-user in the scripting module documentation.

Some features of the scripting engine have been created conceptually, but have not been implemented due to lack of support by the Sweble Wiki system at the moment. onTimerEvent evaluates a script periodically or at specific times. This is supposed for cases where external data is to be pulled from other wiki resources or other data sources to manipulate the WOM.

End-user programming Cypher et al., 2010 includes also tools for visualization and exploration, which are not yet present in the Sweble Wiki. Also, visual programming might be helpful to create simple programs, even though Cypher et al., 2010 describes that the advantages identified in field studies are limited.

As security mechanism, class filters are currently used for JavaScript scripts (cf. 7.8). However, for other languages the Java Scripting API does not define a way to use a custom class filter. Therefore, for those script languages a custom classloader needs to be implemented - unless the Java Scripting API offers class loaders as part of its generic API in future releases.

To conclude, the Sweble scripting module combined with some improvements provides the foundation for several new fields of application of the Sweble Wiki.

References

Bosanac, D. (2007). *Scripting in Java: Languages, Frameworks, and Patterns*. Pearson Education.

Burnett, M. M. & Scaffidi, C. [Christopher]. (2013). End-user development. *The Encyclopedia of Human-Computer Interaction, 2nd Ed.*

Cholakov, N. (2008). On some drawbacks of the PHP platform. In *Proceedings of the 9th international conference on computer systems and technologies and workshop for phd students in computing* (p. 12). ACM.

Clark, J. (1999). XSL transformations (XSLT) specification. *W3C Recommendation, http://www. w3. org/TR/xslt*.

Cypher, A., Dontcheva, M., Lau, T. & Nichols, J. (2010). *No Code Required: Giving Users Tools to Transform the Web*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Dashorst, M. & Hillenius, E. (2008). *Wicket in action*. Dreamtech Press.

Dohrn, H. & Riehle, D. (2011, July). WOM: An object model for Wikitext.

Dohrn, H. & Riehle, D. (2013). Design and implementation of wiki content transformations and refactorings. In *Proceedings of the 9th international symposium on open collaboration* (p. 2). ACM.

Fogie, S., Grossman, J., Hansen, R., Rager, A. & Petkov, P. D. (2011). *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress.

Fogli, D. & Provenza, L. P. (2011). End-user development of e-government services through meta-modeling. In *End-user development*. Springer.

Förther, R., Menzel, C. & Siefart, O. (2010). *Wicket: komponentenbasierte Webanwendungen in Java*. dpunkt-Verlag.

Freeman, A. (2013). *Pro jQuery 2.0*. Books for professionals by professionals. Apress.

Gurumurthy, K. (2006). *Pro Wicket*. New York: Apress.

Herman, D. & Tobin-Hochstadt, S. (2011). Modules for JavaScript. *Preprint, April*.

Insua, E. (n.d.). jsdom: A JavaScript implementation of the WHATWG DOM and HTML standards, for use with io.js. Retrieved July 5, 2015, from https://github.com/tmpvar/jsdom

Jain, N., Mangal, P. & Mehta, D. (2015). AngularJS: A Modern MVC Framework in JavaScript. *Journal of Global Research in Computer Science*, *5*(12), 17–23.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., ... Myers, B. et al. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, *43*(3), 21.

Ko, A. J., Myers, B., Aung, H. H. et al. (2004). Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium* (pp. 199–206). IEEE.

Konan, N. (2010). Computer literacy levels of teachers. *Procedia-Social and Behavioral Sciences*, *2*(2), 2567–2571.

Le Hors, A., Raggett, D. & Jacobs, I. (1999, December). *HTML 4.01 Specification - 18 Scripts* (tech. rep. No. http://www.w3.org/TR/1999/REC-html401-19991224). W3C.

Li, P. & Zdancewic, S. (2005). Practical information flow control in web-based information systems. In *Computer security foundations, 2005. csfw-18 2005. 18th ieee workshop* (pp. 2–15). IEEE.

Li, S., Xie, T. & Tillmann, N. (2013). A comprehensive field study of end-user programming on mobile devices. In *Visual languages and human-centric computing (vl/hcc), 2013 ieee symposium on* (pp. 43–50). IEEE.

Lieberman, H., Paternò, F., Klann, M. & Wulf, V. (2006). *End-user development: An emerging paradigm*. Springer.

Liguori, R. & Liguori, P. (2014). *Java 8 Pocket Guide*. O'Reilly Media.

McCarron, S. (2009, January). *XHTML Media Types - Second Edition* (tech. rep. No. http://www.w3.org/TR/1999/REC-html401-19991224). W3C.

Narmontas, W. & Fancellu, D. (2014). XML processing in Scala. In C. Foster (Ed.), *Xml london 2014 conference proceedings* (pp. 63–75). XML London.

Paternò, F. (2013). End user development: Survey of an emerging field for empowering people. *ISRN Software Engineering*, *2013*.

Resig, J. & Bibeault, B. (2013). *Secrets of the JavaScript Ninja*. Manning.

Sharan, K. (2014). *Scripting in Java: Integrating with Groovy and JavaScript*. Apress.

Tatroe, K., MacIntyre, P. & Lerdorf, R. (2013). *Programming PHP*. O'Reilly Media.

The PHP Group. (n.d.). PHP: Documentation. Retrieved July 5, 2015, from http://php.net/docs.php

Tilkov, S. & Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, *14*(6), 80–83.

Tiwari, N. (2014, February). Simple Ways to Add Security to Web Development. *Linux J. 2014*(238).

WHATWG. (n.d.). HTML5. Retrieved July 5, 2015, from http://www.w3.org/TR/html5/

Wood, L., Le Hors, A., Apparao, V., Byrne, S., Champion, M., Isaacs, S., ... Sutor, R. et al. (1998). Document object model (dom) level 1 specification. *W3C Recommendation*, *1*.

Appendix A Script document API

Scripts have access to a document object which always offers the same set of methods. Depending on whether the script is executed as script expression, external script (onRender, onSubmit or onSave) or as interactive script, a different context is provided. There is a larget set of methods which are shared between document and the context (Ctx), see Appendix D.

Methods starting with create[Element]() create a new node of the type [Element].

Methods starting with get[Element]s() return an array of all nodes of type [Element]. When a node is provided as argument to that method, only children of the provided node (or resource) are returned.

getParameter() returns global parameters, e.g. the path where script libraries are placed or the name of the Wiki system. Those parameters are defined in the ScriptingModule configuration.

Further documentation of the methods can be found in the JavaDocs or in the Wiki sample pages.

The API provided to scripts is described by the following methods of document:

```
void log(String message);
CtxBase getContext();
Object getParameter(String parameterName);
```

Appendix B External and interactive scripts API

External scripts and interactive scripts have access to the following methods of document.getContext():

```
WikiTx getTx();

WRI wri(String wri);
S2weResource getResourceByWRI(WRI wri);

void addEventListener(String eventName, Function eventFn);
void removeEventListener(String eventName, Function eventFn);
void dispatchEvent(String eventName, Object... args);

String getEventName();

Wom3Node getElementById(String id, S2weResource contextResource);
```

```
Wom3Node getElementById(String id, Wom3Node node);
Wom3Node[] getElementsByTagName(String tagName, Element element);

Wom3Abbr[] getAbbrs(Element element);
Wom3Big[] getBigs(Element element);
Wom3Blockquote[] getBlockquotes(Element element);
Wom3Bold[] getBolds(Element element);
Wom3Break[] getBreaks(Element element);
Wom3Center[] getCenters(Element element);
Wom3Cite[] getCites(Element element);
Wom3Code[] getCodes(Element element);
Wom3Comment[] getComments(Element element);
Wom3DefinitionList[] getDefinitionLists(Element element);
Wom3DefinitionListDef[] getDefinitionListDefs(Element element);
Wom3DefinitionListTerm[] getDefinitionListTerms(Element element);
Wom3Del[] getDels(Element element);
Wom3Dfn[] getDfns(Element element);
Wom3Div[] getDivs(Element element);
Wom3Emphasize[] getEmphasizes(Element element);
Wom3Font[] getFonts(Element element);
Wom3For[] getFors(Element element);
Wom3Heading[] getHeadings(Element element);
Wom3HorizontalRule[] getHorizontalRules(Element element);
Wom3ImageCaption[] getImageCaptions(Element element);
Wom3Ins[] getInss(Element element);
Wom3Italics[] getItalicss(Element element);
Wom3Kbd[] getKbds(Element element);
Wom3OrderedList[] getOrderedLists(Element element);
Wom3UnorderedList[] getUnorderedLists(Element element);
Wom3ListItem[] getListItems(Element element);
Wom3Nowiki[] getNowikis(Element element);
Wom3Paragraph[] getParagraphs(Element element);
Wom3Pre[] getPres(Element element);
Wom3Ref[] getRefs(Element element);
Wom3Repl[] getRepls(Element element);
Wom3Rtd[] getRtds(Element element);
Wom3Samp[] getSamps(Element element);
Wom3Section[] getSections(Element element);
Wom3Signature[] getSignatures(Element element);
Wom3Small[] getSmalls(Element element);
Wom3Span[] getSpans(Element element);
Wom3Strike[] getStrikes(Element element);
```

```
Wom3Strong[] getStrongs(Element element);
Wom3Sub[] getSubs(Element element);
Wom3Subst[] getSubsts(Element element);
Wom3Sup[] getSups(Element element);
Wom3Table[] getTables(Element element);
Wom3TableCaption[] getTableCaptions(Element element);
Wom3TableCell[] getTableCells(Element element);
Wom3TableHeaderCell[] getTableHeaderCells(Element element);
Wom3TableBody[] getTableBodys(Element element);
Wom3TableRow[] getTableRows(Element element);
Wom3Teletype[] getTeletypes(Element element);
Wom3Text[] getTexts(Element element);
Wom3Title[] getTitles(Element element);
Wom3Underline[] getUnderlines(Element element);
Wom3Var[] getVars(Element element);
```

Appendix C External scripts API

External scripts provided the following methods due to the context resource:

```
Wom3Node getElementById(String id);
Wom3Node[] getElementsByTagName(String tagName);

Wom3Abbr[] getAbbrs();
Wom3Big[] getBigs();
Wom3Blockquote[] getBlockquotes();
Wom3Bold[] getBolds();
Wom3Break[] getBreaks();
Wom3Center[] getCenters();
Wom3Cite[] getCites();
Wom3Code[] getCodes();
Wom3Comment[] getComments();
Wom3DefinitionList[] getDefinitionLists();
Wom3DefinitionListDef[] getDefinitionListDefs();
Wom3DefinitionListTerm[] getDefinitionListTerms();
Wom3Del[] getDels();
Wom3Dfn[] getDfns();
Wom3Div[] getDivs();
Wom3Emphasize[] getEmphasizes();
Wom3Font[] getFonts();
Wom3For[] getFors();
```

```
Wom3Heading[] getHeadings();
Wom3HorizontalRule[] getHorizontalRules();
Wom3ImageCaption[] getImageCaptions();
Wom3Ins[] getInss();
Wom3Italics[] getItalicss();
Wom3Kbd[] getKbds();
Wom3OrderedList[] getOrderedLists();
Wom3UnorderedList[] getUnorderedLists();
Wom3ListItem[] getListItems();
Wom3Nowiki[] getNowikis();
Wom3Paragraph[] getParagraphs();
Wom3Pre[] getPres();
Wom3Ref[] getRefs();
Wom3Repl[] getRepls();
Wom3Rtd[] getRtds();
Wom3Samp[] getSamps();
Wom3Section[] getSections();
Wom3Signature[] getSignatures();
Wom3Small[] getSmalls();
Wom3Span[] getSpans();
Wom3Strike[] getStrikes();
Wom3Strong[] getStrongs();
Wom3Sub[] getSubs();
Wom3Subst[] getSubsts();
Wom3Sup[] getSups();
Wom3Table[] getTables();
Wom3TableCaption[] getTableCaptions();
Wom3TableCell[] getTableCells();
Wom3TableHeaderCell[] getTableHeaderCells();
Wom3TableBody[] getTableBodys();
Wom3TableRow[] getTableRows();
Wom3Teletype[] getTeletypes();
Wom3Text[] getTexts();
Wom3Title[] getTitles();
Wom3Underline[] getUnderlines();
Wom3Var[] getVars();
```

For onSave and onSubmit events, also the following methods are provided by the context object:

```
WikiTx getWritableTx();
boolean commitTx(WikiTx tx);
```

For onSubmit events, also the following methods are provided by the context object:

```
String getFormName();
String getSubmitButtonName();
```

Appendix D Shared document/context API

Both document and any context provides always ("minimal shared functionality"):

```
String getScriptEngineName();
Wom3Document getDocument();
Wom3Node createElement(String qualifiedName);
Wom3Node createElementNs(String namespaceUri, String qualifiedName
   );
Wom3ElementNode createElementNode(String qualifiedName);
String getTextContent(Wom3ElementNode node);
Wom3Abbr createAbbr();
Wom3Big createBig();
Wom3Blockquote createBlockquote();
Wom3Bold createBold();
Wom3Break createBreak();
Wom3Center createCenter();
Wom3Cite createCite();
Wom3Code createCode();
Wom3Comment createComment();
Wom3DefinitionList createDefinitionList();
Wom3DefinitionListDef createDefinitionListDef();
Wom3DefinitionListTerm createDefinitionListTerm();
Wom3Del createDel();
Wom3Dfn createDfn();
Wom3Div createDiv();
Wom3Emphasize createEmphasize();
Wom3Font createFont();
Wom3For createFor();
Wom3Heading createHeading();
Wom3HorizontalRule createHorizontalRule();
Wom3ImageCaption createImageCaption();
Wom3Ins createIns();
Wom3Italics createItalics();
Wom3Kbd createKbd();
```

```
Wom3OrderedList createOrderedList();
Wom3UnorderedList createUnorderedList();
Wom3ListItem createListItem();
Wom3Nowiki createNowiki();
Wom3Paragraph createParagraph();
Wom3Paragraph createParagraph(String textContent);
Wom3Pre createPre();
Wom3Ref createRef();
Wom3Repl createRepl();
Wom3Rtd createRtd();
Wom3Samp createSamp();
Wom3Section createSection();
Wom3Signature createSignature();
Wom3Small createSmall();
Wom3Span createSpan();
Wom3Strike createStrike();
Wom3Strong createStrong();
Wom3Sub createSub();
Wom3Subst createSubst();
Wom3Sup createSup();
Wom3Table createTable();
Wom3TableCaption createTableCaption();
Wom3TableCell createTableCell();
Wom3TableHeaderCell createTableHeaderCell();
Wom3TableBody createTableBody();
Wom3TableRow createTableRow();
Wom3Teletype createTeletype();
Wom3Text createText();
Wom3Text createText(String textContent);
Wom3Title createTitle();
Wom3Underline createUnderline();
Wom3Var createVar();
```