Monitoring Javascript-Applications with Kieker

Master’s Thesis

Daniel Schmidt

November 23, 2016

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Wulf
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,
Abstract

The larger the code base of an application becomes the harder it is to anticipate performance bottlenecks. Having a sufficient performance is vital to most applications, so developers need to focus on finding the most significant bottlenecks. To locate the bottlenecks, developers need to make use of static and dynamic software analysis. The Kieker framework provides a way to monitor applications and discovers the bottlenecks of an application. It brings support for various languages, such as Java, C, Delphi, and Perl.

In this thesis, we add the support for the ECMAScript programming language, commonly known as Javascript, to Kieker. Javascript started as a language for the web and had been designed to add interactivity to web pages. As the Web 2.0 movement began, the typical size of a Javascript application grew and therefore also its complexity. With a growing complexity comes the risk of introducing performance bottlenecks, as problems may not be visible from the beginning and as the general resource usage of the application grows. Additionally, developers use Javascript in other contexts as well, for example, server, command line or mobile applications. This heterogeneous purpose of application and the lack of a compilation step for Javascript applications introduce problems with the instrumentation. We build and evaluate an aspect-oriented programming approach for Javascript applications which fit our need for the instrumentation. We designed a monitoring component which can track function and method calls in Javascript and evaluate it towards its feasibility and performance. We built a command line interface for automatically instrumenting Javascript applications in a user-specified manner.
Contents

1 Introduction ................................. 1
  1.1 Motivation .................................. 1
  1.2 Goals ...................................... 1
  1.2.1 G1: Evaluation of Existing Aspect Oriented Programming Frameworks and Libraries .................................. 1
  1.2.2 G2: Implementation of a Method to Monitor Javascript Applications ......................................................... 2
  1.2.3 G3: Implementation of an Efficient Way to Adapt Kieker for Javascript Applications ............................................. 2
  1.2.4 G4: Evaluation of Developed Way to Monitor Javascript Applications with Kieker ................................................ 2
  1.3 Document Structure .......................... 2

2 Foundations and Technologies ................. 5
  2.1 The Language Standard ECMAScript ............ 5
  2.1.1 Execution Contexts ......................... 6
  2.2 The Programming Language Javascript ............ 6
  2.2.1 Ecosystem .................................. 6
  2.2.2 Functions .................................. 8
  2.2.3 Proxy ..................................... 9
  2.3 Modules ..................................... 9
  2.4 The Web Worker Application Programming Interface ......................................................... 10
  2.4.1 Dedicated Worker ........................... 11
  2.4.2 Shared Worker ............................... 11
  2.5 Aspect Oriented Programming .................. 12
  2.6 Application Monitoring .......................... 13
  2.7 The Application Monitoring and Analysis Framework Kieker ......................................................... 13
  2.7.1 Adding Language Support to Kieker with the Data Bridge ......................................................... 14

3 Developed Approach For Javascript Instrumentation With Kieker 15
  3.1 Usage Of Kieker-Javascript .................... 15
  3.1.1 Without the Command Line Interface ........... 16
  3.1.2 With the Command Line Interface .............. 18
  3.2 Implementation Details of Kieker-Javascript ....... 22
  3.2.1 Structure of Kieker-Javascript ................. 22
  3.2.2 Data Flow .................................. 23
  3.2.3 Packages .................................. 24
## Contents

4 Evaluation 29  
4.1 Aspect Oriented Programming in Javascript 29  
4.1.1 Criteria 29  
4.1.2 Possible Libraries 37  
4.1.3 Results 39  
4.2 Evaluation of Kieker-Javascript 41  
4.2.1 G1: Evaluation of Feasability 41  
4.2.2 G2: Evaluation of Performance 46  

5 Related Work 65  

6 Conclusions and Future Work 67  
6.1 Future Work 67  
6.1.1 Performance Improvements 68  
6.1.2 Additional Functionality 68  
6.1.3 Structural Changes 69  

Bibliography 71
1.1 Motivation

Everyday internet users are facing websites, many of them are web applications written in or compiled to Javascript. Especially modern web applications rely heavily on the performance of Javascript, and some of them struggle to keep a 60 frames per second rendering performance. Javascript is not only used by the browser but also in other environments like servers, command line interfaces, native mobile devices and desktop application and also in these contexts performance is almost always a key concern.

The monitoring software Kieker, developed at the CAU Kiel, provides valuable information about the performance and runtime behavior of monitored applications. It is available for a variety of programming languages including Java, COBOL, and Perl. To help Javascript developers build faster applications with the aid of Kieker monitoring is the goal of this thesis. The thesis will also evaluate the usability and performance of the developed software so that it only adds little performance overhead to the monitored application. To maintain a good performance is crucial, as a 60 frame per second rendering means that the complete page only has 16ms to render, resulting in about 10ms for the Javascript part of the application.

1.2 Goals

Our goal for this thesis is to monitor Javascript Applications with Kieker. To achieve this goal we need to achieve the following subgoals:

1.2.1 G1: Evaluation of Existing Aspect Oriented Programming Frameworks and Libraries

We try to find a proper Aspect Oriented Programming Framework to monitor a Javascript Application without needing to develop a new way of intercepting every function call.

---

1 A survey amongst StackOverflow issues and Github Projects recently has shown that Javascript is currently the most popular language: https://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/

2 For more information on rendering performance, please see Google’s Guide to Rendering performance
1. Introduction

minimal requirements for such a library are that it is capable of intercepting function and
method calls without breaking the existing behavior of the application. Additionally, the
instrumentation process should be easy to integrate into any given Javascript application.
An optional goal would be an excellent performance regarding execution overhead, so
found Frameworks and Libraries will be evaluated regarding this benchmarking property.

1.2.2 G2: Implementation of a Method to Monitor Javascript Applications

We implement a Javascript library or build tool which uses one of the existing Aspect
Oriented Programming Frameworks or an own approach to monitoring a given Javascript
Application. For this purpose we will instrument multiple demo applications of different
sizes to verify the feasibility. An ideal result would be a log entry for every function call,
including name, execution time, and parameters of it. Another property of an ideal result
would be its capability of functioning in multiple browser environments. An optional goal
will be to add as few performance overhead to the application as possible. For this purpose,
the execution time of the demo applications will be measured with the goal to minimize
the rise after instrumentation.

1.2.3 G3: Implementation of an Efficient Way to Adapt Kieker for Javascript Applications

We integrate the result of G2 into Kieker using its Data Bridge. The integration includes an
efficient way of transmitting the probes on the client side and the processing on the server
side. An optional goal will be to create a way to specify the transmission method of Kieker
(e.g. TCP, Web Sockets, or the local filesystem).

1.2.4 G4: Evaluation of Developed Way to Monitor Javascript Applications with Kieker

Our evaluation of G3 shows the feasibility, performance, and versatility of the developed
software. To achieve a high certainty for the usefulness of the solution, it needs to be evalu-
ated with multiple demo applications against real-world criteria, for example, execution
overhead, memory consumption, and supported environments.

1.3 Document Structure

In Chapter 2 we discuss the concepts necessary to understand this thesis. In Chapter 3 we
explain the developed tool for instrumenting Javascript applications, kieker-javascript.
The evaluation is done in Chapter 4, providing insights on the feasibility and performance
1.3. Document Structure

of Javascript applications. Other papers concerning this topic are evaluated in Chapter 5, and the conclusion is drawn in Chapter 6. The future work is described in Section 6.1.
This chapter describes the technologies and underlying principles used in this thesis.

2.1 The Language Standard ECMAScript

ECMAScript is the language specification, standardized by Ecma International in ECMA-262 [8]. “ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment.” [8, Section 4]

Instead of using classes as an abstraction layer ECMAScript uses Objects, which one may declare in different forms. One is the literal syntax, by just stating `{ a: 3 }` for an object with the attribute `a`, which has the value 3. Another one is declaring a constructor. This is a function which creates objects and executes code to initialize them. It has a property named “prototype”, which is used for inheritance.

“Every object created by a constructor has an implicit reference (called the object’s prototype) to the value of its constructor’s “prototype” property. Furthermore, a prototype may have a non-null implicit reference to its prototype, and so on; this is called the prototype chain.” [8, Chapter 4.2.1] By doing so, the prototypal inheritance system is more flexible than the class-based one, as it allows an object to change its inheritance dynamically at runtime. This improvement comes at the cost of verbosity, as it obfuscates the inheritance structure within an application.

Another remarkable feature of ECMAScript is that objects [8, Chapter 15.3] may have functions as attributes and that functions may use functions as arguments. The latter allows a developer to write functional code, which is one of the reasons ECMAScript may be seen as a very flexible language. These attributes results from functions being seen as `Function` objects, which additionally allows introducing inheritance at a function level.

As ECMAScript was built with the browser environment in mind, at a time where browsers only supported to run computations on one core rather than multiple, ECMAScript itself is designed to be single threaded. One of its most fundamental paradigms is event drivenness, which may be utilized to build multi-threaded developer experiences. Native multi-threading was introduced via the WebWorker specification, discussed in detail in Section 2.4. However, the event system has already enabled a multi-threaded behavior at
2. Foundations and Technologies

that time. It works as follows. Javascript code may execute methods provided by the host environment and register a callback to get hold of the result. In the implementation, an event is added to the global event queue as soon as the host environment returns the result. The callback is called as soon as the current computation has finished.

2.1.1 Execution Contexts

An execution context [8, Chapter 10.3] is a tuple of a lexical environment, a variable environment, and a this binding. Each time when control is transferred to executable code, the runtime is entering an execution context. Each function invocation leads to the creation of a new execution context which is then pushed to a stack of execution contexts. In order to monitor an application without breaking the current behavior, it is necessary to preserve the execution contexts to be as in the original application.

2.2 The Programming Language Javascript

Javascript is an implementation of the ECMAScript language specification developed by Brendan Eich in 1995, who worked at Netscape. [33] It is the most common implementation of the ECMAScript specification besides JScript developed by Microsoft. Additionally, Javascript is the name, typically chosen to refer to ECMAScript in general. For this reason, we will use the name in this thesis.

2.2.1 Ecosystem

RedMonk ranks the language as the top one for the time being [29], and one reason might be that it may be used in many contexts. While developed as a language for the application in a browser, it is also utilized in environments like the server, native mobile and desktop apps, as well as command line interfaces. As applications to be monitored may occur in each of these environments, we discuss each of these environments in the following.

Browser

Despite the ECMAScript language specification, Javascript implementation still differs in scope and performance, while still being more stable than the surrounding browser APIs. One reason for this is the diversity in Javascript engines used by the browsers [24]: Chakra used by Internet Explorer, SpiderMonkey used by Firefox and V8 used by Chrome are the most used engines.

They currently all support the ECMAScript 5 specification, but the newer ECMAScript 6 specification has a broad variety of acceptance; Internet Explorer 11 supports only 15% of the specification while Chrome 51 supports 96% of it [7]. Also, the performance of the
2.2. The Programming Language Javascript

engines themselves differs a lot as Zuzak, Ivankovic, and Budiselic [35] suggest even on desktop computers. Although there is no research to the topic of Javascript performance on mobile browsers to be found at the time being, one can only suggest that the performance is a tougher bottleneck compared to desktop devices due to limited hardware capacity.

In this thesis the term current browser describes the following browsers:

- Internet Explorer 11
- Edge 14
- Firefox 49
- Chrome 53
- Safari 10
- Opera 40
- iOS Safari 10
- Chrome for Android 53

Native Applications

When it comes to building native applications with Javascript, a developer has to choose between many different principles. [3] The most common ones are the “Web-to-native wrapper” approach, which embeds a web application in a native one and thus gives the developer the opportunity to access native APIs and the “Runtime” approach. In latter one, a scripting language, in this case, Javascript, is used to interact with the native APIs directly without using an embedded web browser. Both principles rely heavily on the Javascript application to be performant to have a native look and feel, especially when it comes to animations.

Server and Command Line Interfaces

Due to Node.js® Javascript is also widely used in the context of servers and command line interfaces. Node.js® is a JavaScript runtime built on Chrome’s V8 JavaScript engine. “Node.js’s architecture makes it easy to use a highly expressive, functional language for server programming, without sacrificing performance and stepping out of the programming mainstream.” [30] This is one of the reasons Lei, Ma, and Tan [17] states that Node.js® “performs much better than the traditional technique PHP in high concurrency situation.” [17]
2. Foundations and Technologies

2.2.2 Functions

As one of our goals is to measure the execution time of functions in Javascript applications, it is important to understand how functions work. We want to use the instrumented and uninstrumented version in the same way, so we need to be sure our instrumentation process does no harm to the functionality provided by the application. Therefore we will discuss the different kinds of functions in Javascript, as well as the different ways of invoking a function.

Definition

The syntax for defining a function is split into two different kinds: Function Declaration and Function Expression [8, Chapter 13]. The first kind is hoisted to the top of the code where it is defined; the second kind is not. Therefore it may access local variables from where it is defined. The other difference is that Function Declarations need to be named, whereas Function Expressions may be anonymous functions.

Methods

A method is no more than “a function that is the value of a property” [8, Chapter 4.3.27] and is not treated specially in ECMAScript. The call of a method consists of two parts, the lookup of the object value and the function call itself.

Constructors

The way function constructors work is that by calling a function with the `new` keyword a new “this” is set. That means the internal state of the function is resets, and an instance is created. There are built-in constructors as well, for example, `Function`, `Object`, and `Array` which initialize a given type.

Invocation

The most obvious way to invoke a function is by the usage of parentheses at the end of the function name which includes the function arguments as to be seen in Line 1 and 2 in Listing 2.1. As mentioned in Section 2.2.2, functions are also to be considered as a particular type of objects. They may be invoked by calling certain methods on these objects: `call` Line 3-4 and `apply` Line 5-6. [8, Chapter 15.3] [6, Chapter 9.2]

```javascript
1 myFunction();
2 myFunction('my', 3, ['arguments']);
3 myFunction.call(undefined);
4 myFunction.call(undefined, 'my', 3, ['arguments']);
5 myFunction.apply(undefined);
6 myFunction.apply(undefined, ['my', 3, ['arguments']]));
```

Listing 2.1. functionInvocationExample.js

call and apply are defined via prototyping properties and are available for every function in ECMAScript. The main difference between the invocation via parentheses and
2.3. Modules

the methods is that with the latter ones a value for the this keyword inside the method may be set. What distinguishes apply and the other invocation mechanisms is the way they deal with arguments: apply uses an array as argument for the arguments to invoke the function with, whereas the other mechanisms expect them to be named separately. Apply is especially useful when dealing with (partial) forwarding of function arguments in Proxy-like behavior.

2.2.3 Proxy

In the course of this thesis, we evaluated the Proxy-API to its ability to instrument applications. The Proxy-API takes an object as an argument and a handler object and returns a new version of the target. The new version is identical to the passed one, but the handler gets called on certain actions, such as accessing a property or issuing a function call. The usage may be seen in Listing 2.2, where every property access gets logged to the console.

```javascript
var handler = {
    get: function(target, name){
        console.log('Accessing', name);
        return target[name];
    }
};
var target = {
a: 3
};
var result = new Proxy(target, handler);
console.log(result.a);
// logs 'Accessing a'
// logs 3
```

Listing 2.2. proxyExample.js

2.3 Modules

Javascript is a scripting language, and it does not compile to an executable file. This means joining various parts of the application to a complete version may not be done by a compiler. To have this problem solved in a reliable manner the Javascript community has developed several competitive module systems. The ones relevant to this thesis are CommonJS and the ECMA Script 6 language definition. CommonJS is the standard used in Node.js® applications. The syntax, to be seen in Listing 2.3 and Listing 2.4, relies on module.exports for exporting functionality and require(‘...’) for importing functionality.

```javascript
module.exports = {
    meaningOfLife: 42,
    secret: 23,
};
```
2. Foundations and Technologies

Listing 2.3. constants.js

```javascript
var constants = require('./constants.js');
console.log(constants.meaningOfLife);
// Logs '42'
```

Listing 2.4. main.js

ECMAScript 6 modules are the upcoming standard for both the web and Node.js. We chose not to use them in this thesis, as ECMAScript 6 is currently not available to every browser. Nevertheless, our solution may deal with source code in ECMAScript 6, therefore we explain the syntax here. The syntax, to be seen in Listing 2.5 and Listing 2.6, consists of import and export statements, importing and exporting functionality.

```javascript
export var meaningOfLife = 42;
export var secret = 23;
```

Listing 2.5. constants.js

```javascript
import { meaningOfLife } from './constants';
console.log(meaningOfLife);
// Logs '42'
```

Listing 2.6. main.js

As we aim to develop a solution for browsers and browsers lack the consistent support for either of these technologies we chose to user browserify. Browserify is a command line utility that takes Javascript source code with CommonJS module definitions as input and returns a Javascript application. The resulting Javascript application has every required module inlined and a require function available so that the application works as if the environment had a module system.

2.4 The Web Worker Application Programming Interface

We distinguish two fundamental principles of Web Workers [13]. The first one is the Service Worker API, which is used to interfere with network request and manages the caching layer. We will not discuss this kind of workers in detail in this thesis, as they provide background services rather than concurrency. The second principle is workers providing concurrency, such as dedicated workers and shared workers. First ones are linked to their creator whereas latter ones are shared amongst execution contexts of the same origin. Both types will be evaluated accordingly to their fitness in the setting of this thesis. Their interfaces are discussed in the following parts:
2.4. The Web Worker Application Programming Interface

2.4.1 Dedicated Worker

Dedicated Workers are determined to the Javascript context they are created in. Listing 2.7 shows the API for initializing and calling an action on dedicated workers.

```javascript
1 onmessage = function(e) {
2   console.log('Message received from main script');
3   var workerResult = 'Result: ' + (e.data[0] + e.data[1]);
4   console.log('Posting message back to main script');
5   postMessage(workerResult);
6 }
```

Listing 2.7. worker.js [19]

```javascript
1 var myWorker = new Worker("worker.js");
2 myWorker.postMessage([1,2]); // Sending message as an array to the worker
3 console.log('Message posted to worker');
4
5 myWorker.onmessage = function(e) {
6   result.textContent = e.data;
7   console.log('Message received from worker');
8 }
```

Listing 2.8. client.js [19]

In the current browser window context, a new Worker object is initialized, the worker.js script is downloaded and executed. As to be seen in Listing 2.7, the worker script contains the onmessage (worker.js, Line 1), which is global within its context. By calling myWorker.postMessage (client.js, Line 2) this method is invoked within the worker context and at the end postMessage (worker.js, Line 5) is called, which invokes the myWorker.onmessage (worker.js, Line 5) function on the client side. It is important to note that both the dedicated worker and the shared worker have completely separate execution contexts, so variables to the main scope are not available within the browser and vice versa.

2.4.2 Shared Worker

Shared Workers are in contrast to the Dedicated Workers not bound to a specific window context but are available for every tab of the browser. This way, they only need to be initialized once for a given session and origin and called from multiple contexts at once. The API is similar to the Dedicated Worker, except that a shared worker has a port property. This property must be started in the beginning calling worker.port.start() (client.js, Line 2) to have a connection to a running worker instance available and can, later on, be used similarly to the dedicated worker handle, by calling myWorker.port.postMessage (client.js, Line 3) or setting myWorker.port.onmessage (client.js, Line 6). The worker script has a global onconnect (sharedWorker.js, Line 1) function, which is invoked on connection to the worker. Within this method an Event is given as an argument, which contains a ports array property; The worker may register an event listener on these ports to get messages send by the client side. The API for initializing and calling an action on this type of worker looks like this.
2. Foundations and Technologies

```javascript
onconnect = function(e) {
  var port = e.ports[0];

  port.onmessage = function(e) {
    var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
    port.postMessage(workerResult);
  }
}
```

Listing 2.9. sharedWorker.js [18]

```javascript
var myWorker = new SharedWorker("sharedWorker.js");
myWorker.port.start();
myWorker.port.postMessage([1,2]); // Sending message as an array to the worker
console.log('Message posted to worker');

myWorker.port.onmessage = function(e) {
  console.log('Message received from worker: ', e.data);
}
```

Listing 2.10. client.js [18]

2.5 Aspect Oriented Programming

Aspect Oriented Programming is a programming technique that favors cross-cutting concerns like logging, authentication or monitoring. Such cross-cutting concerns are described as programming problems, which occur in many different parts and even through multiple layers of the application. Without Aspect Oriented Programming a problem like logging may be solved individually in each method by separate statements. This leads to a tight coupling between the logging and the logged part of the application, which is an anti-pattern.

The Aspect Oriented Programming approach would be to define an advice, a portion of code. To specify on which occasions the advice method gets called one would define a pointcut. That way only one part of the code has to be altered to change the logging within the whole application.

Concerning the implementation, there are two kinds, which may also be used together: dynamic and static weaving.

Static weaving means that the pointcuts are resolved at compile time, the result is that the advice execution is directly invoked in the targeted method. In contrast to that dynamic weaving just invokes a generic callback for that pointcut, which then reflects the actual advice being executed, so that one may add or remove advice at runtime.
2.6 Application Monitoring

Monitoring describes the continuous observation of the performance of an application during its runtime. Because monitoring is typically done while the application is executed in a real world (often production), context it may give helpful insight of actual performance bottlenecks or bugs within the application or regarding the hardware. [11]

2.7 The Application Monitoring and Analysis Framework

Kieker

Kieker [15] is an application monitoring and analysis tool, usable on a variety of languages including Java, Perl, and Cobol. It collects and analyzes application monitoring data, e.g. method execution times and system-wide monitoring data, e.g. CPU utilization or memory usage [12]. It provides performance analysis of the application and also allows runtime behavior analysis.

![Kieker Data Bridge](image)

**Figure 2.1. Kieker Data Bridge**
2. Foundations and Technologies

2.7.1 Adding Language Support to Kieker with the Data Bridge

In version 1.8 Kieker introduced the Kieker Data Bridge, which provides an option to receive monitoring records from other languages than Java. Its interface allows connectors to register (Inversion-of-Control paradigm), which may receive monitoring data from various sources. The ones implemented are Transmission Control Protocol (TCP) and Java-Message-Service (JMS); this thesis uses the first one. To correctly parse the incoming data an ID has to be sent together with the monitored data; it is used to map the data to the correct origin language and record type. So to add a new language to the supported ones, one would have to extend such a mapping and parse the monitored data in a form similar to the Java records generated by Kieker.
Chapter 3

Developed Approach For Javascript Instrumentation With Kieker

The goal of this thesis is to instrument Javascript applications with kieker. We use the pre-existing Kieker Data Bridge explained in Section 2.7.1 to receive HTTP Requests containing records generated by our solution. The data for the records is gathered by different advices, explained in the following chapters. The gathered data is then serialized and given to a web worker, which proceeds to invoke the HTTP request to the Kieker Data Bridge. The developed library, from now on called kieker-javascript, comes in two modes of usage: CLI and manual.

Both modes take a Javascript application with a package.json file as input and transform it into an instrumented version of the same application, as to be seen in Figure 3.1. A package.json file is used to define dependencies and runtime scripts for Javascript applications, as well as for deploying them as a Node.js™ module. The CLI mode allows adding monitoring to the entire application with just a few lines of configuration, whereas the manual mode allows granular control over which parts of the application shall be monitored. Section 3.1 explains both modes in detail.

As mentioned in Section 2.2.1, Javascript has many different host environments in which it operates. One of the most commonly used ones is the browser. This lead to another constraint for the architecture of kieker-javascript: A browser has to load every byte of Javascript code with HTTP-Requests to a server and may be situated in a relatively slow mobile network. kieker-javascript is developed with this problem in mind and tries to reduce the overall Javascript size as less as possible. How this is exactly achieved may be read in Section 3.2.

3.1 Usage Of Kieker-Javascript

We designed the kieker-javascript framework to be used for the manual and automated instrumentation of Javascript applications. We explain both usage modes in the following sections.
3. Developed Approach For Javascript Instrumentation With Kieker

![Diagram of Instrumentation Process](image)

**Figure 3.1. Overview of the Instrumentation Process**

### 3.1.1 Without the Command Line Interface

To give developers granular control over the instrumentation, kieker-javascript supplies a multi-part Javascript library. Its implementation will be discussed in Section 3.2, but the way it works from a user’s perspective is that it has a core module, multiple advices and multiple writers. The core module orchestrates the communication between the advices and the writers and provides the configuration to each of the advices. The advices gather information specific to their specified purpose and send it to the core module. The syntax for instrumentation is explained below in dedicated paragraphs, as the API differs significantly from one advice to another.

To give the users, especially those who use the manual instrumentation, the possibility to use the advices they add one or more methods to the kieker module. These methods may be used to provide the instrumentation of the application. The writers may only be used indirectly and are specified in the configuration. They are supposed to get the data provided by the advices and write them to a source, e.g. to the console for debugging or to a server running the kieker data bridge.

```javascript
var kieker = require('kieker-javascript-core').default;
```

---

1 var kieker = require('kieker-javascript-core').default;
3.1. Usage Of Kieker-Javascript

```javascript
var adviceFunction = require('kieker-javascript-advice-function').default;

var adviceMethod = require('kieker-javascript-advice-method').default;

var adviceEnvironment = require('kieker-javascript-advice-environment').default;

kieker.config({
    serverUrl: "http://localhost:8000",
    advices: [
        {type: "function"},
        {type: "method"},
        {type: "environment",
            include: ["os", "version", "name"]}
    ],
    writer: [{
        type: "console",
        logLevel: "info"
    }],
    workerUrl: "http://localhost:8000/worker.js"
});

adviceFunction(kieker);
adviceMethod(kieker);
adviceEnvironment(kieker);

window.kieker = kieker;
```

Listing 3.1. Configuration Code

One example for a configuration may be seen in Listing 3.1. From line one to four the required node modules is loaded using nodes require function, which may be emulated in the browser by browserify, as explained in Section 2.3. It is worth noting that every module exports a default property, which allows the usage of the EcmaScript 6 module imports. A statement like `var kieker = require('kieker-javascript-core').default;` may also be written as `import kieker from 'kieker-javascript-core';`

The next section is the configuration call for kieker. It takes a configuration object with various keys that are described in detail in Section 3.2. In line 27 to 29 the imported functions augment the kieker object with specific methods used to instrument the application code. These methods are explained in the next paragraph in detail. The last line shows that the kieker instance is bound to the window object so that it may be used in the next script tags.

The current scope of the implementation provides three advices:

- Function
- Method
- Environment
3. Developed Approach For Javascript Instrumentation With Kieker

The usage of each of them will be discussed here; the implementation details will be explained in Section 3.2.

**Function**  The function advice adds records of a started and ended function call, which contain information about the precise time, the function name, the parameters and the result. This type of instrumentation only works for function expressions, so every function declaration has to be rewritten. To enhance a given function expression into an instrumented version, it has to be encapsulated in a `kieker.instrumentFunction` call, as to be seen in Listing 3.2 line two.

**Method**  The method advice results in the same kind of instrumentation as the function advice, but it is called on an object to instrument every key of it containing a function. The exported method is called `kieker.instrumentMethod`. This kind of instrumentation also includes dynamically set properties and inherited properties, which makes this advice valuable in a strict Object-Oriented Programming environment. A sample usage may be seen in Listing 3.2 line five.

**Environment**  The environment advice is different from the other advices as it is designed to be called only once. It gathers information about the execution context of the script, namely if it is a browser or node.js environment and which version is used. If a browser is detected, it also determines the operating system.

```javascript
1 // Function advice
2 var namedFunc = window.kieker.instrumentFunction(function namedFunc() {...});
3
4 // Method advice
5 var obj = window.kieker.instrumentMethod({
6     method: function () {...}
7 });
8
9 // Environment advice
10 kieker.sendEnvironment();
```

**Listing 3.2. Instrumentation Code**

### 3.1.2 With the Command Line Interface

We decided that we want to provide a method for automated instrumentation alongside with the manual approach. We decided so, as wanted to keep the time to instrument their applications as low as possible for developers. To integrate our instrumentation approach well into the architecture of Javascript applications we chose to develop a Command Line Interface, `kieker-cli`. This CLI may be used during the build process and transforms Javascript files in their original version to an instrumented one.
3.1. Usage Of Kieker-Javascript

The kieker-javascript-cli package handles the automatic instrumentation of Javascript applications with kieker-javascript and tries to integrate seamlessly into the typical project environment. For this reason, it reads the kieker part of the package.json file. The package.json file is the primary configuration file for Javascript projects relying on node modules. The broad acceptance of this pattern in the community was the reason for this design decision.

An example of such a configuration may be seen in Listing 3.3 and will now be discussed:

**General information** Some fields in the configuration like serverUrl and workerUrl are used to determine the working environment of the instrumentation code. They are necessary to instantiate the core communication concepts, e.g. to start the web worker. If no workerUrl is specified, it is computed from the combination of serverUrl and workerOutput.

**Code Generation and Augmentation** As described in Section 3.2.3 the kieker-javascript-cli package generated an instrumented version of the original file and a worker file to handle the writers. For this reason, one needs to provide the input, output and workerOutput fields specified with valid paths.

**Advices and Writers** The advices, and writers fields specify the advice and writer packages to be used and provide options to specify their behavior of them. The possible options vary from package to package, but in general, advices have an exclude property to be taken into consideration when automatically augmenting the code. This topic will be explained in detail in Section 3.2.

Listing 3.3 shows an example configuration for the kieker-javascript-cli package to read. It configures the CLI to take a file named “example.js” (line 5) as entry file and write the resulting instrumented version to “instrumentedExample.js” (line 6). The correlating worker file gets written to “worker.js” and gets loaded from “http://localhost:8000/worker.js”, thanks to the configuration in line 4 and 7.

The advices array from line 9 to 26 contains information for the advice to use to instrument the input file. An example for this instrumentation is to be seen in Listing 3.4 and is explained below. The writer defined in the writer array (line 27 to 29) is the console writer, which logs the resulting records to the browser console.

```javascript
{
  ...
  "kieker": {
    "serverUrl": "http://localhost:8000",
    "entryFile": "./example.js",
    "output": "./instrumentedExample.js",
    "workerOutput": "./worker.js",
```

19
3. Developed Approach For Javascript Instrumentation With Kieker

```
"advices": [{
  "type": "function",
  "exclude": {
    "name": "private_*",
    "anonymous": true
  }
}, {
  "type": "method",
  "exclude": {
    "name": "private_*" 
  }
}, {
  "type": "environment",
  "include": [
    "os",
    "version",
    "name"
  ]
}, {
  "writer": [
    "type": "console",
    "logLevel": "info"
  ]
}]
```

**Listing 3.3.** Kieker Section in a package.json File

The given configuration results for an example entry file into an instrumented version like it is to be seen in Listing 3.4 and a worker script like in Listing 3.5. It is to be noted that the configuration calls to the kieker module and the kieker worker module contain the same information that is specified in the kieker section of the package.json file.
3.1. Usage Of Kieker-Javascript

In general, the command line interface module reads the `package.json` file as a configuration and transforms input Javascript files to an instrumented version. All input Javascript files get transformed into an instrumented version to be seen in Listing 3.4 line 17 to 28. The way the file is annotate depends on the configuration given in the `package.json` file. The specified entry file also gets a initialisation function included, which is directly invoked, as to be seen in Listing 3.4 line 1 to 17. The generated worker file is to be seen in Listing 3.5. It includes the configuration for the writers as well as the initialization of the worker itself.

```javascript
(function(window) {
  var kieker = require('kieker-javascript-core');
  kieker.config({....});
  var adviceFunction = require('kieker-javascript-advice-function').default;
  adviceFunction(kieker);
  var adviceMethod = require('kieker-javascript-advice-method').default;
  adviceMethod(kieker);
  var adviceEnvironment = require('kieker-javascript-advice-environment').default;
  adviceEnvironment(kieker);
  kieker.sendEnvironment();
  window.kieker = kieker;
})(window);

var namedFunc = window.kieker.instrumentFunction(function namedFunc() {
  console.info('Named function called');
});

function private_AwesomeFunc() {
  console.info('Named function called');
}
```
3. Developed Approach For Javascript Instrumentation With Kieker

```javascript
(function () {
    var scopedFunctionDeclaration = window.kieker.instrumentFunction(fs

function scopedFunctionDeclaration());

var obj = window.kieker.instrumentMethod({
    method: function () {...

});

})
```

**Listing 3.4. Automatically Generated Entry File**

```javascript
'use strict';
var worker = require('kieker-javascript-worker')({

onmessage = worker.onMessage;
```

**Listing 3.5. Automatically Generated Worker File**

### 3.2 Implementation Details of Kieker-Javascript

To understand our approach, it is important to know certain details of the implementation, which we discuss in the following parts. First, we give a general overview of the structure of our project, and afterward we discuss every package in detail.

#### 3.2.1 Structure of Kieker-Javascript

The kieker-javascript project consists of two main folders, examples, and packages. The examples folder contains multiple possible use cases in various sizes for each type of instrumentation technique. Each of those examples has documentation for new developers to start the application and has the required modules directly linked to the packages folder to ensure a working integration in the further development process.

The packages folder consists of the different exported node modules and therefore contains the basis for the implementation. It is to be noted that this project uses Lerna as the foundation of the module export and management. Lerna is an open source tool which aims to make the development process of multiple corresponding node modules easier to manage.

It supports package installation and linking, but also the publishing of packages with semantic versioning. In semantic versioning the current version is described with the scheme `Major.Minor.Patch`, so e.g. 1.2.3 means it is the first major version with the second minor version and the third patch level. To increase the major version, the API of the exported package has to change; to increase the minor version functionality has to be added in a backwards-compatible way;
3.2. Implementation Details of Kieker-Javascript

to increase the patch version one needs to add a backwards-compatible bug.

To get a rough understanding of how the complete solution works, Section 3.2.2 will present the data flow, followed by implementation details of the separate packages in Section 3.2.3.

3.2.2 Data Flow

The goal of *kieker-javascript* is to gather instrumentation data from various sources and send it to defined destinations as a server’s HTTP port or save it to the filesystem. One key constraint was to limit the effect on the execution time of the instrumented application; this is why every suitable processing is situated in a worker thread. This separation is suggested in Figure 3.3 by the rectangle with the caption “worker” on the right side. The term worker refers to one dedicated worker, as explained in Section 2.4. As Section 4.2.2 shows that the throughput of *kieker-javascript* is sufficient with one worker started, we only start that much in order to keep the RAM consumption low. We decided to use dedicated worker instead of shared worker, as the browser support for shared worker is lower than the one for dedicated workers.

The general data flow as seen in Figure 3.3 is as follows: The advices, represented by block 1, gather data specific to their domain. It is worth noting that each advice operates completely separated from other advices of the same or other types. Each of them uses the sendRecord provided by the core module, represented by block 2, to send a representation of its instrumentation result, called the record, to the core module. The core module then uses the worker interface described in Section 2.4 to send the worker an asynchronous notification that a record was generated. Due to the asynchronicity, even if the further processing may be slow the messages would only queue in a message queue handled by the browser. This would therefore not result in a noticeable lag in the main thread of the browser.

The next step is the receiving of the records by the worker package, as represented by block 3. The worker package then proceeds to send the records in a synchronous way to every registered writer, as to be seen in block 4 and 5. This behavior may occur slow, but as mentioned earlier the worker thread operates without any significant effect on the main thread, whereas spawning another worker adds at least 50MB of ram usage to the browser.
3. Developed Approach For Javascript Instrumentation With Kieker

![Diagram of Dataflow]

**Figure 3.3.** Schematic Diagram of the Dataflow

### 3.2.3 Packages

There are several packages included in the kieker-javascript ecosystem, providing different functionalities:

- kieker-javascript-core
- kieker-javascript-worker
- kieker-javascript-cli
- kieker-javascript-advice-function
- kieker-javascript-advice-method
- kieker-javascript-advice-environment
- kieker-javascript-records
3.2. Implementation Details of Kieker-Javascript

The complete monitoring functionality is split up into these packages, so that developers may configure and extend the functionality to their needs.

Core and Worker

The core package and the worker package provide the foundation of Kieker-Javascript. Although both are designed to be extended, the core package by advices and the worker package by writers, they achieve this goal differently: The core package returns an object which may be used to initiate a configuration process and to issue the processing of a new record. The extension works by adding new methods to this object, which then may issue records via kieker. That way the advices may expose an arbitrary amount of methods to the user and can even rely on each other.

In contrast adding a writer to the worker only needs a line in the configuration, as every available writer is already included in the worker package. It is to be noted that the worker package should be included and hooked to the onMessage method in the worker, but it does not start a new worker. This task is performed by the core package, which also polyfills web worker for environments without an implementation for dedicated workers.

Command Line Interface

The main goal of the command line interface package is to provide an easy configuration interface for automated instrumentation of the application code. The configuration for this task, described in Section 3.1.2, is situated in the package.json file most of the modern javascript projects already have in usage.

The process the command line interface performs split into three stages: First, it gathers every needed information by reading the package.json and the input file and provides valid default values. Before the last stage writes the output files, the second stage performs various tasks based on the provided information. That tasks may be split into two parts: the worker and the instrumentation of the application code. The worker part consists of writing a worker file that has the configuration inlined. The instrumentation task is more challenging and illustrated in Figure 3.2.

The content of the file is transformed into an abstract syntax tree by the Babylon Parser. The abstract syntax tree representation allows having advices export a visitor, which may then iterate over the tree and mutate it into an instrumented version. After
3. Developed Approach For Javascript Instrumentation With Kieker

Each configured advice has mutated the syntax tree babel-generator transforms it into the javascript representation that is then written into the output file.

**Advice: Function and Method**

The most fundamental advices are the function and method advices, which gather the start and end of any function or method execution alongside with any relevant information about the function or method being executed. This data contains the name, call arguments, stack and the result of the execution. Both advices are implemented similarly: They expose a function to wrap the function or object around which then has to be assigned to the variable which would have normally be used. The reason for that is that the Proxy API explained in Section 2.2.3, was used to implement the callback behavior. Due to the necessity for the variable assignment function declarations are not supported and need to be transformed to equivalent function expressions.

The way the Proxy API is used is that a handler is assigned, which gets called instead of the native function the “apply” - Method of the function or the “get” - Method of the object is invoked directly or indirectly. This way it is possible to invoke other functions before the original function gets invoked like the one used to send records to kieker. It is noteworthy that this part is extremely performance sensitive as it gets invoked very often.

Listing 3.6 shows a similar implementation to the one used in kieker-javascript-advice-function. As to be seen in line 3 and 5 the start and end time of the function call gets logged to the console. In line 4 the result of the original function is computed and returned in line 6. In line 11 the proxied version of uninstrumentedFunction gets assigned to instrumentedFunction.

```javascript
var handle = {
  apply: function (target, thisArg, argumentsList) {
    console.log('Start Time', currentTime());
    var result = target.apply(thisArg, argumentsList);
    console.log('End Time', currentTime());
    return result;
  }
};

function uninstrumentedFunction() {};

var instrumentedFunction = new Proxy(uninstrumentedFunction, handle);
```

Listing 3.6. Proxy-API example

We use the Proxy API instead of exchanging the values of an object which are functions or overwriting the apply method of a function. Exchanging the values of an object happens at a specified time. This means if the Javascript code adds keys to the object or exchanges the prototype we miss parts of the application in the monitoring. We chose the Proxy API over overwriting the apply method of a function as we wanted to have a concise solution.
3.2. Implementation Details of Kieker-Javascript

As to be seen in Figure 3.4, fundamentally a callback is invoked after a function or method is executed. It issues the generation of a record using the `kieker.sendRecord` function. The record sent contains a type information, which is then utilized in the worker to serialize the record to a format which the kieker instance, here exemplarily displayed as a server running the kieker data bridge, may read.

![Figure 3.4. Schematic Diagram of User Interaction With The Function Advice](image)

**Advice: Environment**

The environment advice delivers relevant information about the execution environment by using “Platform.js“, an open source module. The open source module is included in the advice and exports an object containing the information.

**Records & Writers**

After gathering the information, it needs to be processed so that kieker can read and work with the data. For this reason, the developed library needs to do two things: make the data available for kieker to read and format the data in a way that kieker may read and parse it. The first thing is done by the writers and explained later on; the second one is done by
3. Developed Approach For Javascript Instrumentation With Kieker

kieker-javascript-records.

This is a bridging project filling the gap between Java and Javascript standards. The way this works is that the build script included in the javascript package generates the implementations from the “instrumentation-languages” repository, transfers the javascript implementation to its repository and writes an index file, which has the record types as keys and the required files as values. That way the record generators exposed by the “instrumentation-languages” record language DSL implementation are easily retrievable in the worker package.

The writers are configured through the worker config call. There are currently two implementations, the console writer, used for debugging, logs every given record to the console. The HTTP writer sends HTTP requests to a running instance of the kieker data bridge. The fetch API is used for these HTTP requests, which allows having new mechanisms like promises to be utilized. It is available in modern browsers, and there are polyfills available for older ones.
Chapter 4

Evaluation

In this section, we evaluate different approaches for aspect-oriented programming in Javascript in Section 4.1. We also evaluate our instrumentation approach for Javascript applications, kieker-javascript regarding feasibility and performance in Section 4.2.

4.1 Aspect Oriented Programming in Javascript

As to be seen in Section 2.5, aspect-oriented programming is the best solution to add a cross-cutting concern as application monitoring to an application. Such a framework has various requirements to be considered complete and useful for the usage as part of the application monitoring in Kieker for Javascript. These requirements are along with development specific criteria like for example code quality, and active development of the library are explained in detail in Section 4.1.1. Afterwards, the existing and own possible solutions are briefly explained in Section 4.1.2; a detailed analysis is presented in Section 4.1.3.

4.1.1 Criteria

To choose the best fitting Aspect Oriented Programming solution for this use-case, each possible solution needs to be evaluated against the same set of criteria. Those criteria are to be seen in Table 4.1 and will be explained and justified in this section. Table 4.1 additionally shows each metric of the criteria and with them the possible values for that metric. As some criteria are more important than others, the magnitude of impact they provide to the result will be explained afterward.
### Table 4.1. Criteria for Aspect-Oriented Programming Libraries

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project Activity</strong></td>
<td></td>
</tr>
<tr>
<td>Last Relevant Commit</td>
<td>Date</td>
</tr>
<tr>
<td><strong>Scope of Instrumentation</strong></td>
<td></td>
</tr>
<tr>
<td>Marks below refer to Listing 4.1</td>
<td></td>
</tr>
<tr>
<td>Global Function</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Global Anonymous Function</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Statical Method C</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Statically Added Method D</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Inherited Method</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Dynamic Method</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Dynamic Inherited Method</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Invocation by Call Method</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Invocation by Apply Method</td>
<td>Yes or No</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td></td>
</tr>
<tr>
<td>Runtime Overhead of Function Calls(^a)</td>
<td>Time in ms</td>
</tr>
<tr>
<td>Runtime Overhead of Method Calls</td>
<td>Time in ms</td>
</tr>
<tr>
<td><strong>Weaving Technique</strong></td>
<td></td>
</tr>
<tr>
<td>Need to change code</td>
<td>Yes or No</td>
</tr>
<tr>
<td><strong>Application Programming Interface</strong></td>
<td></td>
</tr>
<tr>
<td>Tracks beginning of function call</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Tracks end of function call</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Event Handling</td>
<td>Callbacks, Callbacks With Handle Object, Event Stream</td>
</tr>
<tr>
<td><strong>Code Quality</strong></td>
<td></td>
</tr>
<tr>
<td>Static Type Checking</td>
<td>Yes or No</td>
</tr>
<tr>
<td>Number of Automated Tests</td>
<td>Non-negative Number</td>
</tr>
<tr>
<td>Unit Tests</td>
<td>Yes or No</td>
</tr>
<tr>
<td>E2E or Integration Tests</td>
<td>Yes or No</td>
</tr>
<tr>
<td><strong>Filesize</strong></td>
<td></td>
</tr>
<tr>
<td>Filesize of the library</td>
<td>Amount of kb</td>
</tr>
<tr>
<td><strong>Documentation</strong></td>
<td></td>
</tr>
<tr>
<td>Quality</td>
<td>One of “non-existant” / “little detail” / “moderate detail” / “extensive detail”</td>
</tr>
<tr>
<td><strong>License</strong></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Name of the license</td>
</tr>
</tbody>
</table>

\(^a\)The values are measured on a MacBook Pro Mid 2014, 2.8GHz, 16 GB with the Chrome Browser in version 54

\(^b\)
4.1. Aspect Oriented Programming in Javascript

**Project Activity**

Javascript is a programming language, which is commonly used in the browser environment, as discussed in Section 2.2.1. This is an environment, which underlies constant change, due to improvements made by browser vendors, not only regarding the language itself but also regarding the provided APIs. For that reason, the whole Javascript ecosystem is a fast-changing environment, which tends to lead to faster software aging.

Therefore, for us, the most important criteria for choosing a library is the maintenance status. As software is either marked as deprecated or has silently been abandoned, the chance of getting feedbacks to bugs or another relevant information is not as high as on an actively maintained project. Additionally, as the languages environment changes the best practices may also change, which may lead to interoperability problems with more current code.

The metric to be used to determine if the maintenance is given is the time until the last relevant commit was done. The last relevant commit is a commit changing actual code of the library and does not include commits that e.g.

- change a dependencies version
- alter the readme
- change the license
- add / remove an example

This metric reflects the intention of this criteria, as an actively maintained and developed library should have constant updates to the source code. Another relevant metric may be the last time since an issue has been resolved. As Javascript libraries often only have a single purpose and less than 20 lines of code, not every library may have issues in their repository. Therefore, the first metric is preferred. The value range for this metric is the date of the last relevant commit; the newer this date is, the better.

**Scope of Instrumentation**

As the main goal of this thesis is to show a way to monitor javascript applications, measurement is an important part. Therefore, the scope of measurement is an important criterion for the selection of the part of the solution, which tracks the method calls. By scope of measurement, it is referred to the detail of measurement, or in other words the lack of missing tracked method calls in a Javascript program. The following program will be used for the evaluation and includes multiple difficulties for the AOP library to master. Those difficulties may be separated in the function declaration, discussed in Section 2.2.2 and the function invocation, mentioned in Section 2.2.2.
4. Evaluation

Listing 4.1. measureFunctionCalls.js

The shown code snippet shows the most common ways for Javascript functions to be defined and invoked. More exotic patterns like the Proxy API has not been included, as they are not part of the ECMAScript 5, but the ECMAScript 6 syntax. The metric used to determine the scope of instrumentation is for each of the function calls noted with “Result A-I” in Listing 4.1 if it may be monitored.
4.1. Aspect Oriented Programming in Javascript

Performance

As Javascript is a single threaded language (see Section 2.1), the monitoring code execution blocks the execution of the actual application code. Additionally, in the browser, events such as animations or DOM changes invoked by the user or by the Javascript code may occur and cause rerenderings and relayoutings. In such a situation, it is estimated that a browser only has 10ms for the actual application Javascript code to run. Resulting from that, the monitoring code underlies severe performance constraints, which have to be considered when choosing an AOP framework.

The metric used to determine the fitness of a framework will be the execution overhead it causes. Given Listing 4.2 the value $(D - C) - (B - A) = \text{instrumentedExecutionTime} - \text{originalExecutionTime}$ is calculated with a significant amount of repetitions. The lower this value is, the better the solution is suited for the usage in this thesis.

```javascript
1 // global functions
2 var obj = {
3   instrumentedFunction: function() {},
4   originalFunction: function() {},
5 };
6
7 // Take time A
8 obj.originalFunction();
9 // Take time B
10
11 // Take time C
12 obj.instrumentedFunction();
13 // Take time D
```

Listing 4.2. Measurement of Execution Times

Weaving Technique

As explained in Section 2.5 there are different weaving techniques, which may be used to instrument a program. The difference, which is important in our context, is the explicitness a developer has to face when using the AOP library. In general, for our use case, the less explicit is the best approach, as cluttering the observed codes with monitoring statements obfuscates its original intends. The question used to determine if the underlying technique may be beneficial for this approach is very simple:

Does the AOP library require any changes to existing source code?

Please note that source code is in this context defined as the code a developer writes. Many developers use transpilers in modern Javascript applications to transform the ECMAScript 5 to ECMAScript 6 code. Due to the definition, the resulting code may be automatically transformed by a library; that would still lead to a positive test result.
4. Evaluation

Application Programmer Interface

To use an aspect-oriented programming library it has to expose an interface to track events in the programming language, such as function invocations. Therefore, such an API has to be evaluated according to their fitness for the monitoring component of this solution. In this scenario, a good API would provide a way to track the beginning and end of a function call in an easy way. The first metric is yes or no if an application programmer interface is available.

Another metric is how modern the API conception is; the possibilities will be presented and rated in the following:

Callbacks In the earlier days of Javascript, such behavior would be exposed by callbacks. A possible usage of such an API may look like this:

```javascript
function toBeInstrumented() {}

// instrumentation
library.instrument(toBeInstrumented, function() {
  // called as method is invoked
}, function() {
  // called as method has finished
});
```

Listing 4.3. Example for Instrumentation with Callback API

This would be the worst kind of API to deal with and will, therefore, be rated one on the scale.

Callbacks With Handle Object This pattern was also used in the earlier days of Javascript, but it is easier to use than the former one, as you may simply pass the handle around to other functions. This allows a higher grade of decoupling for the software components and is therefore preferred. The usage of such an API may look like this:

```javascript
function toBeInstrumented() {}

// instrumentation
var handle = library.instrument(toBeInstrumented);

handle.methodInvoked(function() {
  // called as method is invoked
});

handle.methodFinished(function() {
  // called as method has finished
});
```

Listing 4.4. Example for Instrumentation with Handle API

As explained above this pattern is slightly better. Therefore it will be rated two on the scale.
4.1. Aspect Oriented Programming in Javascript

**Event Stream**  The best pattern for our approach is the event stream pattern. It is typically used in reactive programming, an approach that aims to make interaction with programs faster and more natural. In this pattern, an event is emitted as a certain action is triggered. Other software components, which are aware of this event stream may register on one or multiple events and get functions invoked as the event occurs. Such an API may look like this:

```javascript
function toBeInstrumented() {}

// instrumentation
var stream = library.instrument(toBeInstrumented);

stream.on('methodInvoked', function() {
    // called as method is invoked
});

stream.on('methodFinished', function() {
    // called as method has finished
});
```

Listing 4.5. Example for Instrumentation with Stream API

Please note that line one to four do not necessarily have to be situated in the same file to work. Additionally, the only little effort is needed to transform a stream into a pipeline, which processes the raw data and it is possible to transmit the events through the network to another process, which may do actual work on this data. We rate this approach with the best rating, a three.

**Code Quality**

As the aspect-oriented programming library builds the root of this project, the success of the Javascript monitoring component dependents highly on the quality of it. To rate this several small metrics are used, which are explained below:

**Static Type Checking**  Javascript has a dynamical type system. In contrast to statically typed programming languages, which throw an error at compile time for possible type incoherences, Javascript throws them at runtime. This means the unexpected behavior is only experienced at runtime, which increases the uncertainty of the library working correctly.

In Javascript, there are many ways to achieve static type checking:

- static type checkers, for example, Flow
- languages with static type system that compile to Javascript, for example, Typescript

Both approaches equal each other in the grade of certainty they provide. Therefore, they both will be treated equally in this evaluation.
4. Evaluation

Tests  Another indicator of a high code quality is the presence of automated tests and the type of these. In general, there are three different types of tests:

- Unit level tests describe the behavior of a single software component
- Integration level tests describe the behavior of two or more software components working together
- End to End level tests describe the behavior of a software with mimiced natural interactions

To determine the code quality improvements through tests in a moderate amount of time, the number of test cases and their level is measured. It is assumed that a project with more tests on one level than another provides more certainty about the expected behavior. Additionally, many tests increase the level of documentation, as test cases describe the expected behavior in a detailed way. An additional metric will be the presence of unit tests and the presence of integration or end to end level tests. Later ones are not separated, as in small software components there may be no need to have multiple components to test the integration of.

Filesize

Javascript code executed in the browser has first to be loaded through network requests. This means an HTTP request is invoked, and the data is requested from a server, adding additional round trip time to the page load time. On mobile devices with high latency network access this time is significantly high. Consequently, it has to be taken into consideration.

To measure this, the size of the files to be included in the project is measured in kb. The code a library may need to add to source files already taken into consideration at Section 4.1.1.

Documentation

To use third party software efficiently, a documentation is needed, otherwise, one would have to understand the source code before using it. As this may take a significant amount of time, a good documentation should be taken into consideration when picking an aspect-oriented programming library.

To determine the quality of the library, a rating from “non-existent” through “little detail”, and “moderate detail” to “extensive detail”. Whereas “non-existent” is self-explanatory, we specify “little detail” as a documentation which only provides installation instructions. We define the term “moderate detail” as a documentation containing installation instructions and basic usage examples. A documentation of “extensive detail” also provides multiple examples for each possible use case and a detailed explanation of the underlying technology.
4.1. Aspect Oriented Programming in Javascript

License

This thesis deals with the extension of the Kieker monitoring framework, which is an open source project licensed under the Apache License, Version 2.0.[1] Therefore, when choosing an aspect-oriented programming library, only the solutions with a permissive license are relevant. We specified the term “permissive license” as a license, which allows the usage of the software and the redistribution under the same license. Otherwise, users may be needed to buy their own license for this component of our software system, which is not the goal of this project.

The metric used for this criterion is the name of the license itself.

Weighting of Criteria

The most important criterion is chosen to be maintenance, as an unmaintained solution would be difficult to use in this and following projects. This leads to the decision that completely unmaintained or abandoned projects may be excluded as possible solutions. This includes that such solutions will not be further evaluated regarding the scope of measurement and performance overhead.

The next important criterion in this evaluation is the scope of measurement, as only partly instrumented applications may not give enough insight of the performance and behavior. Finally the performance is taken into consideration. Although it is important for the final solution this topic is the one that may be solved within or in work resulting from this thesis the easiest.

4.1.2 Possible Libraries

meld

Meld [20] is an aspect-oriented programming library for Javascript. It claims that “it allows you to change the behavior of, or add behavior to methods and functions (including constructors) non-invasively.”[20] An example of the usage, as given in the projects README-File would be:

```javascript
var myObject = {
  doSomething: function(a, b) {
    return a + b;
  }
};

// Call a function after myObject.doSomething returns
var remover = meld.after(myObject, 'doSomething', function(result) {
  console.log('myObject.doSomething returned: ' + result);
});
myObject.doSomething(1, 2); // Logs: "myObject.doSomething returned: 3"
```
4. Evaluation

```javascript
remover.remove();
myObject.doSomething(1, 2); // Nothing logged
```

**Listing 4.6. measureExecutionTime.js**

**aspect.js**

Aspect.js [2] is an aspect-oriented library, which uses the Javascript Decorators[14], a language feature proposed for the ECMAScript 7. Additionally to achieve that goal it depends on the ECMAScript 6 class syntax, which excludes older browsers from using the Javascript code as it is. The example below is taken from the project’s README-File and shows how one may instrument a Javascript class with aspect.js. It is to be noted that this example is using Typescript[10], a language that compiles to Javascript.

```javascript
import {beforeMethod, Wove, Metadata} from 'aspect.js';

class LoggerAspect {
    @beforeMethod({
        classNamePattern: /^Article/,
        methodNamePattern: /^(get|set)/
    })
    invokeBeforeMethod(meta: Metadata) {
        console.log('Inside of the logger. Called ${meta.className}.${meta.method.name} with args: ${meta.method.args.join(', ')}.');
    }
}

class Article {
    id: number;
    title: string;
    content: string;
}

class ArticleCollection {
    articles: Article[] = [];
    getArticle(id: number) {
        console.log('Setting article with id: ${id}.');
        return this.articles.filter(a =>
            return a.id === id;
        ).pop();
    }
    setArticle(article: Article) {
        console.log('Setting article with id: ${article.id}.');
        this.articles.push(article);
    }
}

new ArticleCollection().getArticle(1);
```

38
4.1. Aspect Oriented Programming in Javascript

35  // Result:
36  // Inside of the logger. Called ArticleCollection.getArticle with args: 1.
37  // Setting article with id: 1.

Listing 4.7. measureExecutionTime.js

The evaluation of this library was unfortunately not possible, as the provided interface specifically targets ECMAScript 6 classes. Those were not part of the evaluation, and a function or method may not be instrumented separately.

Photon.js

Photon.js[22] is a source-to-source compiler that enables dynamic instrumentation of JavaScript programs. In contrast to the previous approaches it is not a library to be included in the project, but a source code transpiler. This means it is a program which transforms Javascript source code to an equivalent monitored Javascript representation. After installing Photon.js, this is done by running photon test.js on the command line, where test.js is a Javascript source file.

In the tests conducted in this thesis, the compiled photon.js code was not able to run with a simple example, so this library will not be further evaluated.

traceGL

traceGL[31] takes a similar approach as Photon.js, as it compiles the Javascript source code to an instrumented version. The fundamental difference is that the instrumented code produces a log, which may be inspected using the included web UI.

In the tests conducted in this thesis the compiled traceGL code was not able to run with a simple example, so this library will not be further evaluated.

4.1.3 Results

The evaluation of the aspect-oriented programming solutions in Javascript had the following result:
4. Evaluation

Table 4.2. Criteria for Aspect-Oriented Programming Libraries

<table>
<thead>
<tr>
<th>Criterion</th>
<th>meld.js§</th>
<th>Proxy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROJECT ACTIVITY</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Relevant Commit</td>
<td>18.04.2014 (6df48e)</td>
<td>22.05.2016</td>
</tr>
<tr>
<td><strong>SCOPE OF IMPLEMENTATION</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Marks below refer to Listing 4.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result A</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Result B</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Result C</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Result D</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Result E</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Result F</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Result G</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Result H</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Result I</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>PERFORMANCE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Overhead of Function Calls</td>
<td>n/A</td>
<td>0.00579688 ms</td>
</tr>
<tr>
<td>Execution Overhead of Method Calls</td>
<td>n/A</td>
<td>0.01552847 ms</td>
</tr>
<tr>
<td><strong>WEAVING TECHNIQUE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Need to change code</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>APPLICATION PROGRAMMER INTERFACE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Track beginning of function call</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Track end of function call</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Modernity</td>
<td>Callbacks With Handle Object</td>
<td>Stream</td>
</tr>
<tr>
<td><strong>CODE QUALITY</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Type Checking</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Number of automated Tests</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Unit tests</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>E2E or Integration tests</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>FILESIZE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Included Files</td>
<td>4.69 kB</td>
<td>292 B</td>
</tr>
<tr>
<td><strong>DOCUMENTATION</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quality</td>
<td>moderate detail</td>
<td>extensive detail</td>
</tr>
<tr>
<td><strong>LICENSE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Source</td>
<td>MIT</td>
<td>MIT</td>
</tr>
</tbody>
</table>

§It is to be noted that the results given in “Scope of Instrumentation” rely on the documentation, as the library did not call a callback on function call at the time it was tested.
4.2 Evaluation of Kieker-Javascript

In this thesis, we have built a solution to instrument Javascript applications with Kieker. We use the GQM approach evaluating this solution in this chapter. For this we first need to specify the relevant goals for this project, in our case these are the following:

The first one is feasibility, which we discuss in Section 4.2.1. The second one is performance, which we discuss in Section 4.2.2.

We define multiple research questions for each of the goals. Each research questions aims to get insight to which degree the goal has been reached.

4.2.1 G1: Evaluation of Feasability

To show that our solution is feasible we need to answer the following research questions:

- RQ1: Is it feasible?
- RQ2: How many different kinds of records can be tracked?
- RQ3: On which platforms does the solution work?

To answer these questions, we determine the following metric:

- M1: Number of Kieker Records written

The basic setup used to meet the research questions consists of two parts, the Kieker Data Bridge instance, and a Javascript application. The Kieker Data Bridge instance listens to POST-Requests on http://localhost:8080/kieker and is active during the experiments. It is configured to write the monitoring records directly to the file system.

The Javascript application consists of the HTML file shown in Listing 4.8, the package.json file to be seen in Listing 4.9 and an individual Javascript file. To answer different questions, the content of the Javascript file varies with the investigated research question. These changes and variations made to parts of the package.json are declared in the chapters for the individual research questions.

```
<html>
<body>
<script src="js/instrumentedBundle.js"></script>
</body>
</html>
```

Listing 4.8. index.html
4. Evaluation

The HTML file seen in Listing 4.8 includes the instrumented and bundled Javascript
code in line 3. It is necessary that it loads the bundled version because kieker-javascript
relies on a module system which is not available to the browser. Therefore, as to be seen in
Listing 4.9, a part of the build process runs browserify. Browserify is a command line
utility that transforms Javascript code with module definitions and imports to a bundled
version that includes the required modules.

To start this example, one has to run the script defined in Listing 4.9 by running

code, it is necessary that it loads the bundled version because kieker-javascript
relies on a module system which is not available to the browser. Therefore, as to be seen in
Listing 4.9 line 8, a part of the build process runs browserify. Browserify is a command line
utility that transforms Javascript code with module definitions and imports to a bundled
version that includes the required modules.

To start this example, one has to run the script defined in Listing 4.9 by running

```javascript
1 { 
2   "name": "feasibility",
3   "version": "1.0.0",
4   "description": "",
5   "main": "main.js",
6   "scripts": {
7     "instrument": ".node_modules/.bin/kieker",
8     "inlineDependencies": "browserify --external babel-traverse --external babel-types js/
9           instrumentedMain.js -o js/instrumentedBundle.js && cp worker.js instrumentedWorker.js &&
10        browserify instrumentedWorker.js -o worker.js && trash instrumentedWorker.js",
11     "build": "npm run instrument && npm run inlineDependencies",
12     "start": "trash node_modules/ && npm install && npm run build && python -m SimpleHTTPServer"
13   },
14   "kieker": {
15     "serverUrl": "http://localhost:8000",
16     "entryFile": ".js/main.js",
17     "output": ".js/instrumentedMain.js",
18     "workerOutput": ".js/worker.js",
19     "files": [],
20     "advices": [{
21       "type": "function"
22     }],
23     "writer": {
24       "type": "http",
25       "url": "http://localhost:8080/kieker"
26     }],
27   },
28   "dependencies": {
29     "kieker-javascript-advice-environment": "latest",
30     "kieker-javascript-advice-function": "latest",
31     "kieker-javascript-advice-method": "latest",
32     "kieker-javascript-cli": "latest",
33   }
34 }
```
4.2. Evaluation of Kieker-Javascript

```
"kieker-javascript-core": "latest",
"kieker-javascript-worker": "latest",
"kieker-javascript-writer-http": "latest"
},
"devDependencies": {
  "browserify": "^13.0.1"
},
"author": "Daniel Schmidt",
"license": "ISC"
}
```

Listing 4.9. package.json

RQ1: Is it feasible?

To answer this question, we decided to show that we can track a function call in Javascript. This means that due to the invocation of a instrumented function an HTTP Request has to be sent to the Kieker Data Bridge, which then writes a Kieker Monitoring Record to the file system. To measure this, the Javascript of the application is set to the self-invoking function to be seen in Listing 4.10. Through the build process discussed in Section 4.2.1 this Javascript file gets built to an instrumented version, and afterward, a bundled version is created. For the sake of space, only the instrumented, but not bundled versions will be discussed here. It is to be seen in Listing 4.11. The code in line 1 to 9 is used to initialize the instrumentation solution as discussed in Section 3.1.2. In line 11 the instrumented version of an empty function gets immediately called.

Executing this Javascript file by opening it in the browser as previously described results in two files being written by the Kieker Data Bridge. The first one is the .dat file containing the actual record, being shown in Listing 4.12. The second one is the kieker.dat file containing every used record type, being shown in Listing 4.13. This answers the research question with yes, it is feasible.

```
(function () {})();
```

Listing 4.10. main.js

```
(function (window) {
  var kieker = require('kieker-javascript-core');
  kieker.config({
    "serverUrl": "http://localhost:8000",
    "entryFile": "/js/main.js",
    "output": "/js/instrumentedMain.js",
    "workerOutput": "/worker.js",
    "files": [],
    "advices": ["type": "function",
                 "implementation": {}],
    "writer": ["type": "http",
               "url": "http://localhost:8080/kieker"],
    "workerUrl": "http://localhost:8000/worker.js"
  });

  var adviceFunction = require('kieker-javascript-advice-function').default;
  adviceFunction(kieker);

  window.kieker = kieker;
})(window);
```

Listing 4.10. main.js
4. Evaluation

```javascript
window.kieker.instrumentFunction(function () {});
```

Listing 4.11. instrumentedMain.js

```plaintext
$0;1476710142705686433;1.13-SNAPSHOT;KIEKER;Daniels-MBP.wlan.uni-kiel.de;1;false;0;NANOSECONDS;1

Listing 4.12. kieker-20161017-131542706-UTC-000-Thread-1.dat

1 $0=kieker.common.record.misc.KiekerMetadataRecord
2 $1=kieker.common.record.controlflow.OperationExecutionRecord
3 $2=kieker.common.record.misc.KiekerBrowserEnvironmentRecord

Listing 4.13. kieker.map

RQ2: How many different kinds of records can be tracked?

This research question requires that the Javascript program is instrumented with every advice provided by the kieker-javascript package. Therefore the package.json's kieker advice field has to include the configuration to be seen in Listing 4.14 from line 5 to 14. Besides, the Javascript code needs to have each advice target included once, so the Javascript file in Listing 4.15 was used as instrumentation basis.

This basis is transformed to the instrumented version to be seen in Listing 4.16. The instrumentation call for kieker-javascript-advice-environment is to be seen in line 13. The invocations of the instrumented function and method are to be seen in line 18 and 22. Each of those calls should lead to a separate POST-Request to the Kieker Data Bridge and result in an entry in the Kieker Record.

As to be seen in Listing 4.17 and Listing 4.18, the record includes three different records with two different types. Therefore we conclude that two record types may be tracked.

Regarding the total of 20 records, this means 10% of Kieker may be used in kieker-javascript.

```json
{ ...
"kieker": {
...
  "advices": [{
    "type": "function",
    "exclude": {
      "anonymous": true
    }
  }, {
    "type": "method"
  }, {
    "type": "environment"
  }],
```

44
4.2. Evaluation of Kieker-Javascript

Listing 4.14. package.json

```javascript
(function named() {}());
var obj = {
  foo: function () {
  
  }
};
obj.foo();
```

Listing 4.15. main.js

```javascript
(function(window) {
  var kieker = require('kieker-javascript-core');
  kieker.config({

  var adviceFunction = require('kieker-javascript-advice-function').default;
adviceFunction(kieker);

  var adviceMethod = require('kieker-javascript-advice-method').default;
adviceMethod(kieker);

  var adviceEnvironment = require('kieker-javascript-advice-environment').default;
adviceEnvironment(kieker);
  kieker.sendEnvironment();

  window.kieker = kieker;
})(window);
```

Listing 4.16. instrumentedMain.js

```javascript
$(0;1473424179628229296;1.13-SNAPSHOT;KIEKER;Daniels-MBP.wlan.uni-kiel.de;1;false;0;NANOSECONDS;1
$2;1476710249321832713;Chrome;53.0.2785.143;OS X;10.11.6
$3;1476710249321832713;Chrome;53.0.2785.143;OS X;10.11.6
$4;1476710249321832713;Chrome;53.0.2785.143;OS X;10.11.6
```

Listing 4.17. kieker-20161017-131548606-UTC-000-Thread-1.dat
4. Evaluation

As RQ1 relied on the prerequisite that it works in browsers in general. The current implementation relies heavily on the use of web workers, the window object, and the fetch API. As these techniques are not available on Node.js® environments, it is clear that kieker-javascript does not work on this platform. Therefore the evaluation in this section is restricted to the different browsers.

The test setup equals the one described in RQ1, but the URL is visited by different browsers. The results are displayed in Table 4.3. The “Worked” column refers to whether the browser sends a POST-Request to the Kieker Data Bridge.

<table>
<thead>
<tr>
<th>Browser</th>
<th>Worked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android Browser 52+</td>
<td>Yes</td>
</tr>
<tr>
<td>Chrome 49+ 14</td>
<td>Yes</td>
</tr>
<tr>
<td>Chrome for Android 53+</td>
<td>Yes</td>
</tr>
<tr>
<td>Edge 13</td>
<td>No</td>
</tr>
<tr>
<td>Edge 14</td>
<td>Yes</td>
</tr>
<tr>
<td>Firefox 47+ 14</td>
<td>Yes</td>
</tr>
<tr>
<td>IE</td>
<td>No</td>
</tr>
<tr>
<td>iOS Safari</td>
<td>No</td>
</tr>
<tr>
<td>Opera 40+</td>
<td>Yes</td>
</tr>
<tr>
<td>Opera Mini</td>
<td>No</td>
</tr>
<tr>
<td>Safari</td>
<td>No</td>
</tr>
</tbody>
</table>

4.2.2 G2: Evaluation of Performance

As mentioned in Section 1.2, an important focus of this work is to show that instrumenting Javascript applications in a performant manner is feasible. Therefore the performance was chosen as second evaluation goal. As mentioned in Section 4.2.1, the platforms for this evaluation are limited to browsers.

If not otherwise specified Google Chrome in version 53.0 is used on a MacBook Pro Mid 2014 with a 2,8 GHz Intel Core i5 processor and 16 GB 1600 MHz DDR3 RAM. We chose it, as Google Chrome has an extensive set of tools to measure application performance.
4.2. Evaluation of Kieker-Javascript

To answer most of the following research questions it was assumed that the behavior of Chrome is characteristic for most web browsers. To show that our solution is performant we need to answer the following research questions:

- RQ4: How much execution overhead does the instrumentation add to the application?
- RQ5: How many records may be tracked in a certain period of time?
- RQ6: How much RAM usage is added by the instrumentation?
- RQ7: Does the RAM usage change over time?
- RQ8: How precise is the measurement of compared to other sources?

Additionally, we used the following metrics to determine the answers to the research questions.

- M1: Number of Kieker Records written
- M2: Overall Javascript execution time
- M3: Time between Javascript call and file write
- M4: RAM usage
- M5: Execution time of one Javascript function

As the performance of kieker-javascript may vary with different applications as a basis we conducted the tests in various scenarios. The scenarios consist of different applications being automatically monitored by the kieker-javascript-cli.

- S1: TODO MVC

After describing these scenarios in Scenarios, we will discuss the relevance and measurement processes in RQ4 to RQ8. Finally, the results of the research questions are presented in Results.

Scenarios

S1: TODO MVC We chose the Vanilla Javascript example from the TodoMVC project as an example. The TodoMVC project was founded by Addy Osmani to help web developers decide for one of the many frontend frameworks to use. The project currently provides examples for over 50 frameworks or programming languages compiling to javascript.

Each example has the same scope: Providing a to-do list with the same well-defined feature set. The Vanilla Javascript example refers to a pure javascript implementation with
4. Evaluation

no frameworks used. We chose it, as adding even a reasonably complex frontend framework would add a tremendous amount of complexity, resulting in an example that may not be called trivial.

Figure 4.1. TODO MVC - Vanilla Javascript

The application used may be seen in Figure 4.1, and the logging entries generated during the usage of the instrumented version in Figure 4.2. The logging entries were generated with the kieker-javascript library, using the console writer for demonstration purposes. The instrumentation was done automatically using the kieker-javascript-cli package with settings indicating that every function should be instrumented.
4.2. Evaluation of Kieker-Javascript

The configuration to be seen in Listing 4.19 was used for the automatic instrumentation.

```json
1 {  
2   "kieker": {  
3     "serverUrl": "http://localhost:8000",  
4     "entryFile": ".js/helpers.js",  
5     "files": [  
6       {  
7         "cwd": "js",  
8         "dest": "build",  
9         "src": "**/*.js"  
10       }  
11     ],  
12     "output": ".build/helpers.js",  
13     "workerOutput": ".build/worker.js",  
14     "advices": [  
15       {  
16         "type": "function"  
17       },  
18       {  
19         "type": "environment",  
20         "include": [  
21           "os",  
22           "version",  
23           "name"  
24         ]  
25       }  
26     ],  
27     "writer": [  
28       {
```
4. Evaluation

To gather test data in a reproducible manner, we chose to include a small helper file into the project. It is not included in the instrumentation and adds therefore only a small and consistent overhead to the Javascript execution. This helper, as to be seen in Listing 4.20, exposes a buildTodo function, which adds a Todo item to the list and deleteTodo, which does the opposite.

```javascript
var elem = document.querySelector('.new-todo');
var click = new MouseEvent('click', {
  'view': window,
  'bubbles': true,
  'cancelable': true
});

function buildTodo(value) {
  elem.value = value;
  elem.dispatchEvent(new CustomEvent('change'));
}

function deleteTodo() {
  document.querySelector('ul li:first-child .destroy').dispatchEvent(click);
}
```

Listing 4.20. helper.js

S2: Flexbox Froggy  We chose Flexbox Froggy as an example, as its Javascript code heavily relies on the usage of methods in contrast to the Javascript code of S1. Flexbox Froggy is an open source project by Thomas Park, which aims to teach the CSS Flexbox specification by playing a game. It contains 24 challenges showing every aspect of the specification with a text field to fill the answer in, a CSS construct and several frogs and leaves as game elements. The first challenge is to be seen in Figure 4.3 and shows the usage of justify-content.
4.2. Evaluation of Kieker-Javascript

As the next button is clicked, the text and CSS get exchanged by the Javascript, which is the functionality we want to base our test on. As in S1, we want our test to be easily reproducible and therefore include the Javascript helper shown in Listing 4.21.

```
var nextEl = document.querySelector('#level-counter .arrow.right');
var prevEl = document.querySelector('#level-counter .arrow.left');
var click = new MouseEvent('click', {
  'view': window,
  'bubbles': true,
  'cancelable': true
});

function next() {
  nextEl.dispatchEvent(click);
}

function prev() {
  prevEl.dispatchEvent(click);
}
```

Listing 4.21. helper.js

It exposes a `prev` and `next` function, navigating through the examples.

As the implementation of Flexbox Froggy relies on methods, the instrumentation configuration to be seen in Listing 4.22 uses the `kieker-javascript-advice-method` package for instrumentation.
4. Evaluation

```
... 
"kicker": {
  "serverUrl": "http://localhost:8000",
  "entryFile": "./js/levels.js",
  "files": [
    {
      "cwd": "js",
      "dest": "build",
      "src": "**/*.js"
    }
  ],
  "output": "./build/levels.js",
  "workerOutput": "./worker.js",
  "advices": [
    {
      "type": "method"
    },
    {
      "type": "environment",
      "include": [
        "os",
        "version",
        "name"
      ]
    }
  ],
  "writer": [
    {
      "type": "console",
      "logLevel": "info"
    }
  ]
}
```

Listing 4.22. package.json

**S3: PM - Code Galaxy Visualization**   The PM project by Andrei Kashcha aims to visualize dependencies for packages in different package managers like npm, bower or brew. For this, it displays packages as nodes and dependencies as edges, creating a colorful code galaxy visualization. The galaxy shown in Figure 4.4 shows the core of the Angular project, a framework for Javascript applications, with its dependencies.
4.2. Evaluation of Kieker-Javascript

We chose this project as it has a relatively high complexity and a good mixture of functional and object-oriented programming approaches combined. We also chose to instrument only a certain part of the application, to show that this is feasible. As to be seen in Listing 4.23 line 9, we chose to instrument the `galaxy/native` directory of the application. It encapsulates the low-level logic, like rendering and reacting to user input, meaning that it is potentially heavily used code.

```
1 { 
2     ... 
3     "kieker": { 
4         "serverUrl": "http://localhost:8081", 
5         "entryFile": "./_src/galaxy/service/appEvents.js", 
6         "es6": true, 
7         "files": [ 
8             { 
9                 "cwd": "_src/galaxy/native", 
10                 "dest": "src/galaxy/native", 
11                 "src": "*.js" 
12             } 
13         ], 
14         "output": "./src/galaxy/service/appEvents.js", 
15         "workerOutput": "./worker.js", 
16         "advices": [ 
17             { 
18                 "type": "function", 
19                 "exclude": { 
```
4. Evaluation

```javascript
"name": "appConfig"
}
}, {
"type": "method"
}
},
"writer": [
{
"type": "console",
"logLevel": "info"
}
]
],
... ...
```

**Listing 4.23. package.json**

It is also to be noted that the application uses ECMAScript 6 code. Therefore the `es6` flag is set in the configuration.

In order to instrument the specified part of the application, we had to refactor the files targeted by the instrumentation. This was necessary, as the application relies heavily on nested functions to be available at the top of the scope. At the current state of development, kieker-javascript is not able to transform these files, as the visitor does not mind dependencies on other functions or data upon instrumentation. This would be necessary, as the visitor of kieker-javascript-advice-function transforms function declarations to function expressions in order to wrap an instrumentation call around them. Figure 4.5 shows a small example of such a refactoring; both versions operate the same way.

```
1 a();
2 function a() {
3   c();
4   b();
5 }
6
7 function b() {
8   c();
9 }
10
11 function c() {}
12
```

(a) Before

```
1 var c = function c() {};
2 var b = function b() {
3   c();
4 }
5
6 var a = function a() {
7   c();
8   b();
9 }
10
11 a();
```

(b) After

**Figure 4.5. Refactoring to Function Expressions**
RQ4: How much execution overhead does the instrumentation add to the application?

As the instrumentation adds Javascript code to the application that keeps track of events in the application the execution time of every Javascript controlled action increases. In case of kieker-javascript, the relevant events for this research question are function and method calls, as the execution environment record is only fired once in the beginning. By using additional computation time on every function call, these events have the power to heavily influence the performance of an application.

To measure both kinds, we have chosen to measure this time with both previously described scenarios. Combined they provide methods and functions to monitor for both performance critical advices, kieker-javascript-advice-function and kieker-javascript-advice-method.

To measure the influence of the instrumentation to the runtime behavior of applications we assumed that we get the most relevant measurement by measuring the total javascript execution time. Measuring only one functions execution time would not give significant results, as the execution overhead may vary on parameters like function execution time or shape of function arguments. Also measuring one function alone could guide the browser to a heavier usage of optimizations which may distort the result. Therefore comparing the overall Javascript execution time of an instrumented and uninstrumented version of the same non-trivial Javascript application gives us the most significant results.

To gain more trust in the results, we chose to test this with multiple applications and for one application on multiple browsers. Testing only one application could have distorted the results as it would expose kieker-javascript with only one mixture of advices needed and one set of arguments given. Therefore each scenario is chosen to have a different set of functions and function types to ensure a good level of certainty that the results are valid.

The first scenario is S1: TODO MVC. To measure the same work load on every run, we used the helper functions mentioned in Listing 4.20 in the code to be seen in Listing 4.24. It creates 100 TODO items and deletes them afterward.

```javascript
for (var i = 0; i < 100; i++) {
    buildTodo('Todo Item ' + i);
}

for (var i = 0; i < 100; i++) {
    deleteTodo();
}
```

Listing 4.24. test.js
4. Evaluation

The result of this test on the main browsers is to be seen in Figure 4.6. The browser versions used in this test are Google Chrome 53.0.2785.143 (64-bit), Safari 10.0 and Firefox 39.0. The increase of execution time ranges from 58% on Firefox to 268% on Chrome.
4.2. Evaluation of Kieker-Javascript

The second scenario tested is **S2: Flexbox Froggy**. The experiment was to navigate forward and backward through all 24 examples. To perform this task, the Javascript snippet shown in Listing 4.25 was used.

```javascript
for (var i = 0; i < 23; i++) {
    next();
}
for (var i = 0; i < 23; i++) {
    prev();
}
```

**Listing 4.25. test.js**

![Figure 4.7. Flexbox Froggy: Javascript Execution Time](image)

The measured increase in execution time is 53% in Google Chrome, 83% in Safari and 74% in Firefox as shown in Figure 4.7.

The third scenario tested is **S3: PM - Code Galaxy Visualization**. The experiment was to open the npm galaxy and wait until it has completely rendered. There was no snippet for this task, as clicking a link could be done without the risk of accidentally triggering computations.

The execution times in Figure 4.8 show an increase of 17% in Google Chrome, 20% in Safari and 39% in Firefox. This shows that the slow down depends on the detail level of instrumentation.
4. Evaluation

The measurement results show that the performance impact of using the instrumented Javascript version of an application is severe on extensive instrumentation. It also shows that reducing the amount of instrumented code reduces the performance impact. This gives users of kieker-javascript the possibility to adjust the detail level of the instrumentation to a degree where the user behavior is not compromised.

**RQ5: How many records may be tracked in a certain period of time?**

As kieker-javascript is used in environments that may terminate spontaneously, for example, a browser window getting closed by the user. Therefore it is important that kieker-javascript is able to send as many records as possible in a given amount of time. Additionally, the Kieker Data Bridge needs to be capable of accepting the necessary amount of HTTP-Requests.

To test the solutions ability to send and collect a high amount of records we conducted the following test. We use the sendRecord function exposed by kieker-javascript-core to send a record as to be seen in Listing 4.26.

```javascript
// Require kieker & initialize it
var record = {...};

while (true) {
    kieker.sendRecord(record);
}
```

*Listing 4.26. test.js*
4.2. Evaluation of Kieker-Javascript

We use an endless while loop to send as many records as possible. After these requests get send, we open a data bridge and close it after a certain amount of time. The number of received records gives clarity about the throughput performance of kieker-javascript.

![Graph showing record throughput](image)

Figure 4.9. Record Throughput

Figure 4.9 shows that the recording of records is almost in the tested range. The amount of records being issued is extremely high due to the test setup. Having a record throughput like the one tested in a long period of time would result in an unresponsive browser, finally resulting in a crash. Therefore it can be safely assumed for most applications that the tested high-frequency timeframe of ten seconds is sufficient. This leads to the conclusion that the limitations of kieker-javascript regarding throughput are high enough even for modern Javascript heavy applications.
4. Evaluation

RQ6: How much RAM usage is added by the instrumentation?

As \texttt{kieker-javascript} generates and sends records towards the Kieker Data Bridge, it generates intermediate data to be stored. This data, mostly consisting of records, should be visible in the RAM usage of an application. Also, these records, after they are serialized for the HTTP-Request should be able to be collected by the garbage collection. This means the best possible result would be to see the RAM usage of an instrumented version of an application slightly above the one of the uninstrumented version.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{TODO_MVC_RAM_usage.png}
\caption{RAM usage of TODO MVC}
\end{figure}

60
4.2. Evaluation of Kieker-Javascript

Each scenario was tested the same way as in **RQ4**. To gather the RAM usage data Google Chrome in version 53 was used with a timeline recording, which included RAM usage. We chose Chrome as it has a detailed UI showing the RAM usage for the whole recording. The blue line shows the size of the heap at a given point; the green one shows the amount of stored DOM nodes. This includes the nodes attached to the DOM and the ones stored in memory.

To measure the RAM usage is important as finding memory leaks that increase RAM consumption over time may lead to crashing applications if they are used for a significant amount of time. This is more important for mobile devices, but could also apply to desktop computers if the user chooses to use the application for a sufficient amount of time.

The first scenario is **S1: TODO MVC**. The test setup is the same as in **RQ4**. As to be seen in Figure 4.10 the general RAM consumption is significantly higher in the instrumented version. Also the consumption drops in the end, which is a sign that no memory leaks are introduced in the `kieker-javascript-advice-function` package.

The second scenario tested is **S2: Flexbox Froggy**. The test setup is the same as in **RQ4**. As to be seen in Figure 4.11, the RAM consumption of the instrumented version is significantly higher compared to the uninstrumented version. It is also to be seen that the consumption drops after the garbage collection to a slightly increased level compared to the uninstrumented version. This is a sign that there are no memory leaks in `kieker-javascript-advice-method`.

The third scenario tested is **S3: PM - Code Galaxy Visualization**. The test setup is the same as in **RQ4**. Figure 4.12 shows the same result as the previous scenarios.

Taking each scenario into account, the test conducts that there is no memory leak in the current implementation of `kieker-javascript`. It also conducts that the amount of RAM overhead introduced is moderate. One possible cause of this increase is the introduction of a dedicated web worker to the application.
4. Evaluation

Figure 4.11. RAM usage of Flexbox Froggy
4.2. Evaluation of Kieker-Javascript

RQ7: How much does the RAM usage change over time?

As explained in RQ6 the RAM usage is important for the behavior of an application, especially for long running ones. In order to use kieker-javascript in a performant way, it has to be clear that the RAM consumption does not increase over time. To test this we conducted the same tests as in RQ6, but let the measurement perform without any action afterward. We tested all scenarios to a length of ten minutes, but no increase of the RAM usage could be measured.
4. Evaluation

RQ8: How precise is the measurement of compared to other sources?

To measure the execution time of short running functions, the precision of kieker-javascript needs to be high enough to gather meaningful data. We checked the detail level of different sources and compared them in the chart to be seen at Table 4.4. The sources include programmatically available sources, such as different browser APIs and manual ones, like the Chrome Developer Tools.

<table>
<thead>
<tr>
<th>Source</th>
<th>Precision</th>
<th>Programmatically available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date.now</td>
<td>Milisecond</td>
<td>Yes</td>
</tr>
<tr>
<td>Performance.now</td>
<td>One thousandth of milisecond</td>
<td>Yes</td>
</tr>
<tr>
<td>Chrome Developer Tools</td>
<td>One hundredth of milisecond</td>
<td>No</td>
</tr>
<tr>
<td>Firefox Developer Tools</td>
<td>One hundredth of milisecond</td>
<td>No</td>
</tr>
<tr>
<td>Safari Developer Tools</td>
<td>One hundredth of milisecond</td>
<td>No</td>
</tr>
</tbody>
</table>

We found that kieker-javascript would be the most precise with Performance.now. We also found that the Performance-API is only available in modern browsers and therefore settled for Date.now.
In this chapter, we discuss related work on application performance and user behavior profiling for Javascript applications.

**Jalangi: a selective record-replay and dynamic analysis framework for JavaScript** In this work Koushik Sen and Swaroop Kalasapur developed a “dynamic analysis framework”[28] for Javascript. It has features like selective record-replaying and shadow values and allows dynamic analyses like concolic testing, tracking origins of null and undefined, detecting likely typed inconsistencies, dynamic taint analysis, and a simple object allocation profiler. Each of these features and analysis methods will be discussed briefly, and the solution will be compared to this one afterward.

Selective record-replaying means that a specific part of the program may rerun with a previously recorded user session, enabling the developer to understand errors in greater detail. Shadow values enhance normal program values with meta information, allowing shadow execution. With shadow execution, an analysis can update the shadow values with the actual execution.

Concolic testing allows the generation of new test values by inspecting the concrete execution path of a function and proposing similar input values, allowing to possibly test other execution branches. The developed solution also allows tracking null or undefined values to their origin, which are a constant source for runtime exceptions in Javascript. Detecting likely typed inconsistencies serves the same purpose. The dynamic taint analysis allows to check the program for possible data flows between two sources, and a static profiler observes the number of objects in a given allocation site, monitoring for eventual memory insufficiency.

This approach is a much more heavy-weight approach than the developed one in this thesis. It slows down the execution by 26 times, which is huge compared to the 0,00579688 ms to 0,01552847 ms execution overhead in our solution. This performance comes at the price of not having this many features and analyses. The only analyses which could be built upon our solution would be a function level detection of likely typed inconsistencies and concolic testing.

**An analysis of the dynamic behavior of JavaScript programs** [26]
5. Related Work

This thesis by Gregor Richards contains a study about the dynamic behavior of typical Javascript applications. It took 18 different websites into account and researched among other things if those websites rely on dynamically setting the prototype and adding and deleting of properties on objects. Their result was that these language features are significantly used and may not be omitted easily. This supports the design decision made for the kieker-javascript-advice-method package to favor the Proxy API over statically replacing the keys.


In this poster by Laurent Christophe a way to monitor Javascript applications with the Proxy API is described on a theoretical level. It aims to monitor the application in greater detail by wrapping every Javascript expression, including literals, objects, arrays, unary & binary expressions, functions, and the global window object into functions which track the execution and provide the same interface. Compared to our approach this allows tracking the application behavior in greater detail, giving the user more insights on the application. As every setting of values is handled through Proxies, one may even track the state of each scope in the application at a given point of time.

It is to be added that the kieker-javascript aims to be modular by design, allowing the same kind of instrumentation in further iterations on the project. With kieker-javascript-cli and the visitor exports, it would even allow instrumenting applications in this detail automatically. Also, the user needs to bare in mind that this detailed level of monitoring comes at the cost of execution performance overhead. The poster did not include any performance studies, but calling a function that executes an expression and more is more expensive than executing the same expression.
In this thesis, we built and evaluated a solution for instrumenting Javascript applications with the Kieker Framework. The developed solution is named kieker-javascript. To build it on a performant and solid basis we also evaluated different approaches to Aspect-Oriented Programming in Javascript.

After we had presented the foundations and technologies in Chapter 2, we discussed our implementation of kieker-javascript in Chapter 3. We built a monitoring component with the Proxy-API, used the WebWorker-API for performance reasons and implemented the measurement of functions and methods. We also implemented a Record type for browser environments and built a command line interface, kieker-javascript-cli. Kieker-javascript-cli may be used to automatically instrument a Javascript application without any further knowledge of kieker-javascript. It relies on the Babel project, which provides an abstract syntax tree parser, traverser and code generator.

In Chapter 4 we designed and conducted experiments to test both the feasibility and the performance of our approach. Regarding feasibility the integration of kieker-javascript into the Kieker framework was tested. The performance was examined regarding execution overhead, RAM usage, throughput, and accuracy.

The results show that kieker-javascript integrates well into the Kieker Framework. The performance analysis conducted that the level of detail in the instrumentation heavily influences the performance of the instrumentations. This leads to the conclusion that user of kieker-javascript need to determine their instrumentation focus before instrumenting the application. The results also conduct that further work is necessary to improve the performance of kieker-javascript.

### 6.1 Future Work

Generally speaking, this project is a prove of concept, which may be used in certain projects but comes with several shortcomings due to different constraints. Those constraints include the time it was developed, as Node 6 with Proxy support came out at the end of the project runtime. Also to improve the performance or the usability, even more, the project
6. Conclusions and Future Work

duration was too short, this is why this chapter focusses on possible future tasks to improve kieker-javascript.

6.1.1 Performance Improvements

As pointed out in the previous sections, the essential quality of service for the instrumentation software is performance. Particularly important is to keep the execution overhead to the monitored function as low as possible, so that applications run as fluid and fast. This constraint is even more important today, as smartphones with lower computation power are used to browse the web. There are multiple ways to improve the performance, which may also be used in parallel to ensure an optimal improvement.

The most crucial part of optimizing the explained kind of performance is to reduce the time spent on the same execution thread as the application code. kieker-javascript-advice-function, kieker-javascript-advice-method, kieker-javascript-core are the parts of kieker-javascript being executed in this thread. There are two ways of optimizing the runtime behavior: reducing the number of times the code gets executed or reducing the time each execution takes.

There are several spots where the first optimization may be used to increase efficiency: The easiest optimization may be made in the kieker-javascript-advice-method package. The current implementation wraps the given object into a proxy. In fact, on every access to a property key, the value, if it is a function, gets replaced with a new instance of the instrumented version of the value. This behavior serves its purpose as it allows to instrument methods added dynamically by extending the object or setting its prototype.

Essentially, there are two ways to achieve a better performance here; one would be to replace keys on the object, which are functions, as soon as the program loads. This technique reduces the execution overhead added by the advice itself to zero, only leaving the execution overhead of the kieker-javascript-core function sendRecord to be taken into account. This performance improvement comes at the cost of missing methods, which may be added to the object on runtime. As Gregor Richards states in “An analysis of the dynamic behavior of JavaScript programs” [26], this behavior is uncommon with application code, but common with framework code. With this statistic in mind, it might be useful to separate this approach of monitoring methods into another advice, so that users of this software may choose which approach works best for them.

6.1.2 Additional Functionality

The current implementation of kieker-javascript is focussed on the browser as the execution environment. We chose to do so as Node.js® had no support for Proxies at the time we started the development. In version 6 the Node.js® project added support for Proxies, so
there is no reason to not extend kieker-javascript to support Node.js®. To build support for Node.js® and other non-browser environments we would have to exchange every window reference with something more suitable. This would primarily include changing the instrumentation calls and the preparation step added by kieker-javascript-cli.

6.1.3 Structural Changes

The current approach for automatic instrumentation relies on the kieker-javascript-cli. It uses Babylon and Babel-generate internally, which are both part of the Babel transpiler project This project also includes a plugin structure, which would allow us to export kieker-javascript as a Babel plugin.

As many projects use babel to transpile their ECMAScript 6 code to ECMAScript 5 code shifting from a separate command line utility to a Babel plugin would reduce the mental overhead for developers. Additionally, there would be no need to adjust the build process, leading to an easier usage of kieker-javascript.
Bibliography


Bibliography


[31] Tracegl | tracegl transforms your javascript, injecting monitoring code that produces a log of everything that happens. URL: https://github.com/traceglMPL/tracegl (cited on page 39).


