Improved Developer Support for the Detection of Cross-Browser Incompatibilities

Alfonso Murolo(✉), Fabian Stutz, Maria Husmann, and Moira C. Norrie

Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland
{amurolo,stutzf,husmannm,norrie}@ethz.ch

Abstract. Various tools are available to help developers detect cross-browser incompatibilities (XBIs) by testing the documents generated by their code. We propose an approach that enables XBIs to be detected earlier in the development cycle by providing support in the IDE as the code is being written. This has the additional advantage of making it clear to the developers where the sources of the problems are and how to fix them. We present wIDE which is an extension to an IDE designed specifically to support web developers. wIDE uses a compatibility knowledge base to scan the source code for XBIs. The knowledge base is extracted automatically from online resources and periodically updated to ensure that the compatibility information is always up-to-date. In addition, developers can query documentation from within the IDE to access descriptions and usage examples of code statements. We report on a qualitative user study where developers provided positive feedback about the approach, but raised some issues to address in future work.

Keywords: Testing · Compatibility · Documentation · Web applications · IDE

1 Introduction

An important part of web development is ensuring that a website will have the desired look and behaviour on different browsers and devices. This is non-trivial given the pace at which web technologies continue to evolve in terms of both their specifications and their implementations in different browsers. Taken together with the diversity of devices and the number of browsers and browser versions, with differences between implementations for specific operating systems, the task of ensuring compatibility across browsers is significant. Responsive design partly addresses the problem by ensuring that adaptation to different viewing contexts is central to the design and development processes [1], however developers still need to keep abreast of variations in browser support for specifications and test their code in a large number of viewing contexts.

Fortunately, there are a number of tools and services that developers can use to test their code and check for cross-browser incompatibilities (XBIs). For example, Browsershots[1] will perform testing of websites on a wide range of browsers

[1] http://browsershots.org.

J. Cabot et al. (Eds.): ICWE 2017, LNCS 10360, pp. 264–281, 2017.
DOI: 10.1007/978-3-319-60131-1_15

in a distributed screenshot factory and, therefore, return feedback in the form of screenshots. However, while these services can detect XBIs, the developer still has to trace the source of any problems and find out how to correct them.

Online resources such as Can I Use[2] help developers by providing compatibility information about which code features are fully supported, partially supported or not supported in different browser versions. Depending on developer experience, these sites may be consulted frequently either during code development, or after testing, to track the source of XBIs. This requires the developer to switch from the IDE to the browser and studies have shown that web developers spend around 19% of their time in a browser [2].

To address these issues and provide improved developer support for the detection of XBIs, we propose an approach that integrates a tool for compatibility analysis into the IDE so that XBIs can be detected in the source code ahead of testing. This is achieved by performing a static analysis of the HTML, CSS and JavaScript source code, using a compatibility knowledge base extracted from online resources such as Can I Use. Developers can specify the set of browser versions that they want to support and they then receive a compatibility report with feedback on the level of support covered by the current implementation, together with links to the parts of the source code which cause compatibility issues. Although our approach could be applied to any IDE, we have implemented it as an extension to IntelliJ IDEA to specifically support web developers. The resulting environment, called wIDE, also provides direct access to documentation extracted from the Mozilla Development Network[3] for code features used so that developers can avoid having to switch between the IDE and the browser.

We discuss the background to this work and related research in Sect. 2 before going on to detail our approach in Sect. 3. The wIDE system that we developed to support our investigations is presented in Sect. 4. We then report on a qualitative user study that we carried out to evaluate the system in Sect. 5. Concluding remarks and a discussion of future work are given in Sect. 6.

2 Background

In the software engineering community, it has been recognised that developers spend a lot of their time browsing the web looking for solutions and documentation. This has led to various proposals for integrating support for browsing activities into the IDE. For example, HyperSource [3] is an IDE augmentation that links source code modifications to browsing histories, making use of a browser extension that logs pages visited, a facility that tracks use activity in the IDE and an interface that allows developers to interact with the browser histories. Since many developers actually prefer to view the full documentation pages in the browser rather than IDE pop-ups, Codetrail [4] instead integrated Eclipse with a browser extension for Firefox which automatically opens documentation pages

[2] http://caniuse.com.
[3] https://developer.mozilla.org.

when the developer browses them in the IDE. As well as creating links between the source code and online documentation based on an analysis of browsing history, Codetrail creates links when a user performs interactions such as copying code from an online resource, thereby guaranteeing traceability in the future.

Often developers browse collections of code examples to assist in the coding task, and the goal of Fishtail [5] is to deliver code examples and documentation related to the current task. The system is integrated with the task management system Mylyn[4] and it tries to determine relevant web resources by analysing the interaction history of the developer with the source code. Suggestions are based on keywords related to program elements such as methods or classes which last changed their *degree of interest*, a value that grows the more frequently the developers interact with them.

XSnippet [6] is another system that suggests relevant code examples to developers. In this case, the system mines Java source code to extract the types used, type hierarchies and method signatures. These are then used to construct search queries that deliver example code snippets from crawled web sources. Muse [7] also has the goal of sharing code snippets, in this case, by extracting them from existing software or developer discussions, pruning irrelevant statements and duplicates.

Rather than coordinating the browser and the IDE, another approach is to integrate online resources into the IDE directly. For example, amAssist [8] introduces a search process inside a Java IDE which makes use of queries to online resources augmented by the usage context as observed by the system. The context is also used to refine the search and rank the results. Similarly, Blueprint [9] is a system targeted at the Adobe Flex community that generates search queries augmented with code context from Adobe Flex builder and creates links between copied code and its source.

Some projects have integrated code examples from online Q&A communities such as StackOverflow into the IDE to help developers find solutions to errors that lead to exception stack traces [10,11]. Seahawk [12,13] integrates StackOverflow code examples directly in the IDE, allowing code snippets to be dragged-and-dropped into the IDE and using annotations for the traceability of snippets with the original crowdsourced solutions. Prompter [14] also goes in this direction, combining the relevance ranking from StackOverflow with an internal ranking based on the context of the code.

Clearly, the problem of switching between the browser and the IDE can also be addressed using the opposite approach of moving the IDE to the browser entirely. Arvue [15] is a browser-based tool that allows developers to create and publish web applications to the cloud. Adinda [16] also investigates the collaborative aspect, integrating web services into the IDE that not only allow traditional development tasks for Java to be performed, but also boost collaboration and communication tasks appropriate for development projects.

In our opinion, introducing support from online sources into existing IDEs (web-based or not) is a preferable strategy as opposed to navigating these resources manually, since modern IDE applications usually provide a baseline

[4] http://www.eclipse.org/mylyn/.

of functionalities which can be easily extended to achieve the desired integration with online resources. Further, it can be argued that such an approach is more likely to achieve wider acceptance since developers are already using these IDEs. We therefore chose to investigate how web developers could be better supported by developing an extension to an IDE that would check for XBIs and provide links to online documentation resources.

A number of tools and services already exist to help developers check for cross-browser compatibility and avoid having to manually test their applications in a wide variety of browser/device viewing contexts. One of the earliest research proposals addressing this problem was by Eaton and Memon [17], who used an inductive model based on HTML tags which end-users and developers were expected to keep up-to-date. Systems such as X-PERT [18], WebDiff [19], and its follow-up CrossCheck [20], rely on detecting XBIs by comparing various cues, such as the visual rendering or the DOM of the web pages. Mesbah and Prasad [21] present a fully automated solution for detecting XBIs under different browser environments based on building and comparing hierarchical screen models. XD-Testing [22] is a recent tool specifically designed for testing cross-device applications on simulated devices where developers write test cases that run in different browser environments, exploiting the paradigm of UI Testing.

A number of commercial tools and services for cross-browser testing exist. These include the previously mentioned BrowserShots as well as BrowserStack[5], BrowserSandbox[6], and Browsera[7]. Such services run the web application on either real or simulated devices, accessing the rendered page in various browsers and reporting layout discrepancies or, in some cases, functionality errors such as console logs.

Our goal was to improve developer support by checking for XBIs earlier in the development process, at the time of writing the code rather than later during testing. This can be done using a static analysis of the source code to anticipate any failures that could arise in services and tools like the ones mentioned above. In contrast to such services, it not only detects an XBI but also identifies the code that is the source of the problem and hence makes it much easier to resolve it. To the best of our knowledge, the only previous work that uses static code analysis to detect XBIs is a recent proposal by Xu and Zeng [23], which aims at finding HTML5 incompatibilities with a manually crafted database of XBIs. In contrast to this, we check for XBIs in CSS and JavaScript as well as HTML, and do so using a compatibility knowledge base generated automatically from online resources.

3 Approach

The static analysis in wIDE can provide documentation and compatibility reports about each specific element in the project using online resources. These elements differ between each supported language, and we refer to them as

[5] https://www.browserstack.com.
[6] https://turbo.net/browsers.
[7] http://www.browsera.com.

Elements of Interest (EOI). EOIs can be identified by each language handler in the system.

3.1 Elements of Interest

EOIs can be of three different types. A *Central EOI* is a crucial type of element in a language which is interesting as a target for documentation and/or compatibility support. A *Satellite EOI* is, instead, an element which is also relevant for compatibility and documentation lookup, but generally depends on a Central EOI. Finally, *Potentially Foreign EOIs* are elements that usually are not relevant for documentation or compatibility lookups. However, in certain cases and conditions, these can still produce results for a documentation lookup or a compatibility analysis.

The languages currently supported in wIDE are the three main client-side languages connected to cross-browser compatibility issues: HTML, CSS and JavaScript. We will now provide details about the different EOIs that we have defined in each language.

HTML. Tags are clearly central EOIs since they are the principal means of structuring and defining the content of an HTML document. Documentation sources will be centred around HTML tags and therefore will be able to provide information about their support in various browsers as well as providing documentation on their usage, together with some examples. These also usually include attributes of the various tags, which we therefore define as satellite EOIs, since they obviously depend on the tag to which they belong. Finally, we define two potentially foreign EOIs in HTML. Attributes may have limitations for their values, implying that, in these cases, only a subset of attribute values can be applied to an attribute, each with a specific effect. This of course means that these effects will be described in documentation sources, and can also be subject to cross-browser compatibility analysis. Plain-text elements are also categorised as potentially foreign EOIs since usually they have no result in documentation lookups. However, sometimes the content of HTML pages can be written in other languages, which need to be analysed by the appropriate language handler.

CSS. Provides four different elements: selectors, pseudo-selectors, properties and property values. While properties are clearly a central EOI, property values necessarily need to be a satellite EOI since the possible values depend on the property. Both are very relevant elements for documentation and compatibility lookup. Selectors and pseudo-selectors, such as: nth-child, are instead generally supported across every browser. Pseudo-selectors are also generally self-explanatory and, therefore, generally not very interesting in terms of documentation lookup. IDE support for automatic suggestions of these pseudo-selectors is, in our opinion, enough to provide support to the developers in terms of usage. For these reasons, we have decided not to include these in our EOIs.

JavaScript is more complicated. Functions are crucial elements in this language, because developers often use a lot of built-in functions that are dependent on the native browser implementation, making them the perfect central EOI. References to built-in objects and types are categorised as satellite EOIs and may be very interesting for documentation purposes. JavaScript also contains many other elements such as language keywords, blocks and other constructs which are common to programming languages in general. While these could return results for a documentation lookup, we argue that these should generally be supported across the various browsers and, more importantly, are not very interesting in terms of documentation lookup. Therefore, we discarded these as candidates for EOIs.

EOIs are the inputs for the two main functionalities offered by wIDE to developers: the documentation lookup and the compatibility scan. The former will be triggered on individual EOIs that appear in the source code, while the latter can be triggered on both single and multiple EOIs at the same time, for example a project-wide compatibility scan. We will now describe both functionalities in detail.

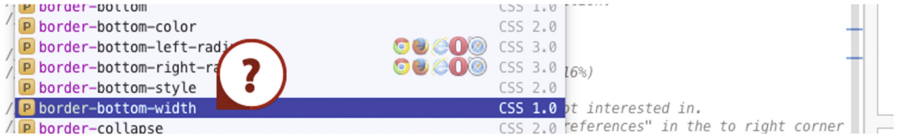

Fig. 1. An IDE suggestion from IntelliJ.

3.2 Documentation Lookup

The documentation lookup process queries online documentation pages, such as the Mozilla Developer Network (MDN). Usually, the content of these resources is rather exhaustive, including syntax of features, code examples, compatibility information and available attributes/parameters and so on. Currently, little support is provided in IDEs for describing web technologies in order to assist web developers in writing code, forcing them to switch to the browser to navigate to the documentation pages online.

However, IDEs usually provide at least suggestions on which features can be used once the developer starts to type, in an auto-completion fashion, as shown in Fig. 1. While the developer navigates the suggestions, for example by hovering on them or through arrow keys, wIDE will request documentation information to be shown in a sidebar panel, dividing it into separated extensible panes (see Fig. 2). As an alternative, a documentation lookup can be triggered from existing source code by highlighting the corresponding EOI, through mouse or keyboard selection, and using a hotkey. The extensible panes offer the advantage of reducing the size needed to display the various sections, still allowing them to be expanded if the developer wants to see more of any of them, thereby reducing the time needed to navigate across the sections and also avoiding information overload.

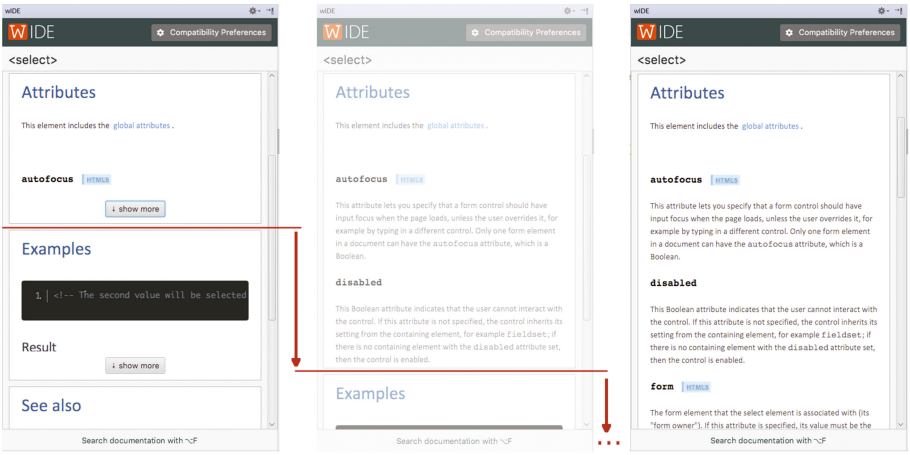

Fig. 2. Extending a documentation pane.

3.3 Compatibility Scan

The second functionality offered by wIDE is the compatibility scan, which can target anything from a single selected EOI to the whole project, scanning for all the EOIs contained in the entire code base. This functionality involves querying an online compatibility knowledge base such as Can I Use. Depending on the type of EOI, sometimes there can also be solutions or work-arounds suggested to improve compatibility for the browsers, for example the use of *polyfills* in the case of JavaScript.

Clearly there needs to be a specified *compatibility set*, which is a set of browsers that the developers want to support, in order to detect any incompatibilities which are relevant for the project. Developers using wIDE can define this set from a preferences panel (see Fig. 3), where they can specify ranges of versions that they want to support. They can also include preview releases, to address cross-browser incompatibilities before these reach a broader set of users, which can be quite useful in the maintenance stage. An additional flag can be selected to keep the project preferences up with new browser releases, as soon as they come out. In this way, when a new browser version is released, it will automatically be included in the compatibility set. By default, all browser versions, from the first release to the latest preview release, are included in the compatibility set.

The compatibility scan can be invoked at three different levels: on a single EOI, on an entire file, and on an entire project. The first level is triggered automatically together with the documentation lookup: an extensible pane about compatibility will be added to the functionality, as shown in Fig. 4. The compatibility scan will show colour-coded results for each browser. For each browser, a vertical bar is presented with the first and the latest versions shown as side labels. The most recent will be at the top, while the oldest will appear at the

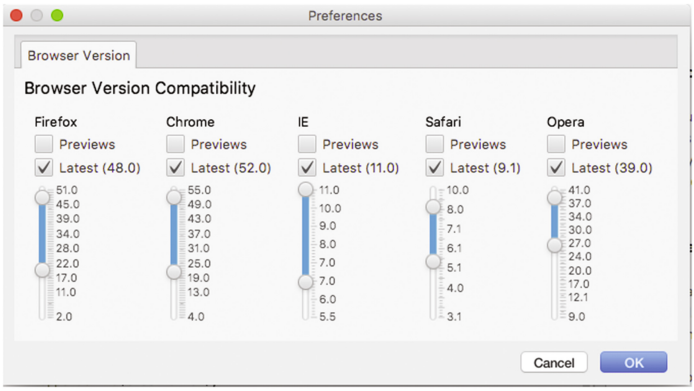

Fig. 3. The preferences panel in wIDE, to set compatibility goals.

bottom. Every transition in the support state, for example a browser starting to support a feature, shows the label of the version implementing the transition on the side. The bar itself is coloured in red if the versions shown do not support the feature, while they are shown in green if they do. Partial support, such as a prefixed, browser-specific implementation, is marked in yellow. Browser versions which are not of interest to the developer, based on the preferences set, are coloured in grey. Finally, the different browser versions are related to their corresponding user base as reported by the knowledge-base, showing the percentage of users that are left out because of the selected EOI. Figure 4 shows this in detail.

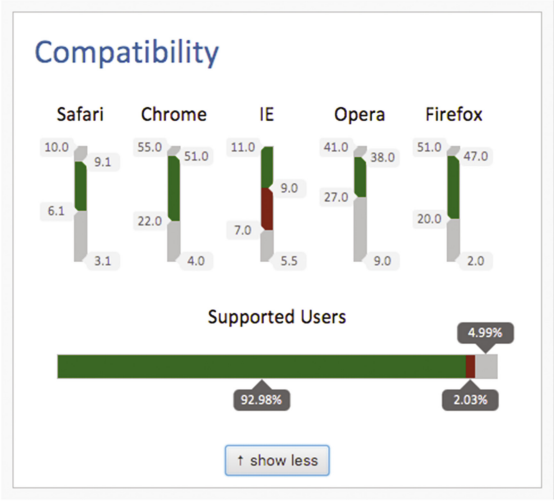

Fig. 4. Compatibility result for an EOI. (Color figure online)

The second and third levels of compatibility scan, require a file traversal to scan for EOIs. Reports for identical EOIs are aggregated, although it is still important for the developer to see how many different occurrences of an EOI there are in the source code, and where they are. For example, if a call to *getElementsByClassName* is used multiple times in the source files, the compatibility report will only perform one query to the knowledge base, while still showing all of the occurrences in the compatibility results. This allows all parts of the code which may need to be altered to be traced, for example, to replace this call with a polyfill implementation.

All the unique EOIs are presented in the compatibility report, each of which can be extended to list the various occurrences in the source code. Clicking on each of these will navigate the source code to the corresponding occurrence. Since the results are aggregated by unique EOIs, these receive a compatibility score which is used to rank the issues by compatibility, in ascending order, so that the most severe issues appear at the top. The compatibility score $C_s \in [0, 1]$ where 0 means that no browser supports the functionality, while 1 stands for full support in every browser version.

Let S_s be the number of browser versions that support the feature, P_s be the number of those that partially support it, Pf_s be the number of those that only

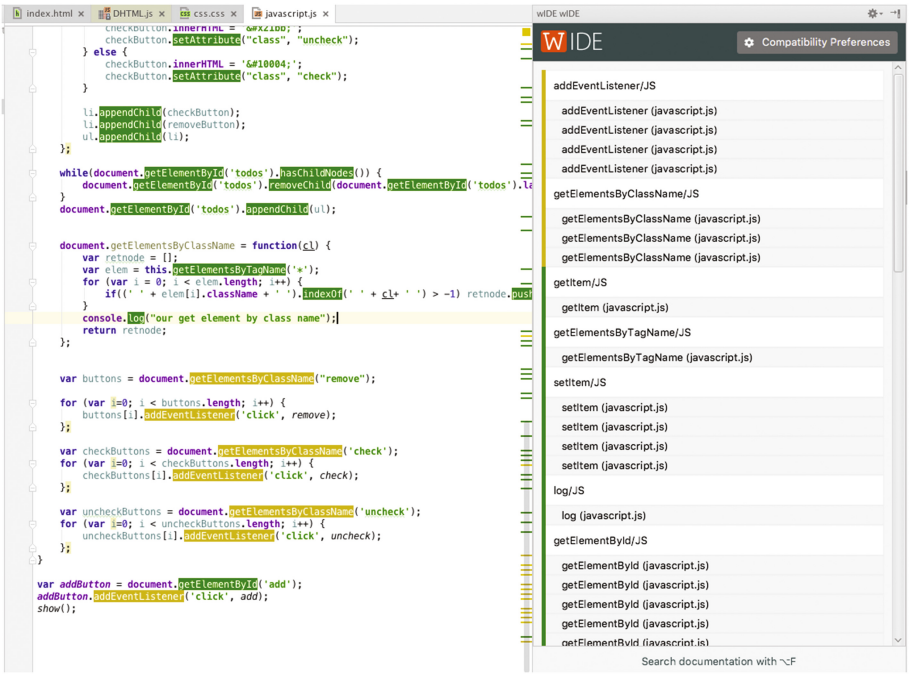

Fig. 5. Compatibility result for a scan of an entire JavaScript file. Issues with higher severity (i.e. lower compatibility score) are ranked at the top, with a colour marking on the side. The same colour is used in the source to locate the issues quickly. (Color figure online)

have the features in a prefixed implementation, N_s be the number of those that do not support the functionality and n be the total number of browser versions in the knowledge base. Then the compatibility score C_s is calculated as follows:

$$C_s = \begin{cases} 0.45 * \frac{S_s + P_s + Pf_s}{n}, & if\ N_s > 0 \\ 0.5 + 0.45 * \frac{S_s}{n}, & else\ if\ P_s + Pf_s > 0 \\ 1, & otherwise \end{cases}$$

An example of the results from a compatibility scan of the second level is shown in Fig. 5. The results are ordered by severity, with a yellow colour tone for the top EOIs, while the ones at the bottom appear green. In addition, the occurrences of the EOIs in the code are also highlighted with the corresponding colour, so the developers can locate the occurrences and the issues in the source code at a glance.

For the last two levels of compatibility scan, the colour coding is more complex than the first level. The compatibility score C_s is mapped to a colour in the range between red and green. The brightest tones of every colour are not used in order to improve readability.

4 Architecture and Implementation

The architecture of wIDE was conceived with the goal of having a shared server that can be deployed for a team or small development organisation. Therefore, it has a client-server infrastructure which is shown in Fig. 6. The client is implemented as a plugin for IntelliJ IDEA, which is responsible for handling the user interaction, parsing the project source code, and communicating with the server. In our implementation of the approach, the server is also responsible for fetching the content from the online sources for the relevant EOIs and parsing it, before caching it in a MySQL database. In this way, the lookup results can be stored in the knowledge base for the entire development team, rather than in each client, thereby minimising the number of queries performed to the sources. In addition, the privacy of the source code in the project is ensured since only the EOI is sent in a request to each online source, and nothing else.

As seen in Fig. 7, each client listens to the various actions that the developer performs in the IDE, such as triggering the lookup of an EOI through a keyboard shortcut. The client contains an appropriate code parser for each language in the language registry, which contains HTML, CSS and JavaScript in our current implementation. Note that other languages such as PHP and Ruby could be added to the registry and we provide more details on how this could be done later in this section.

The identification of EOIs is implemented through code parsing that builds up a Program Structure Interface tree (PSI tree), integrated with IntelliJ. An example of such a tree is shown in Fig. 8. Based on these trees, wIDE automatically detects the best suited EOI for a lookup by searching for pre-defined patterns. For example, in the case of JavaScript functions, the interesting nodes

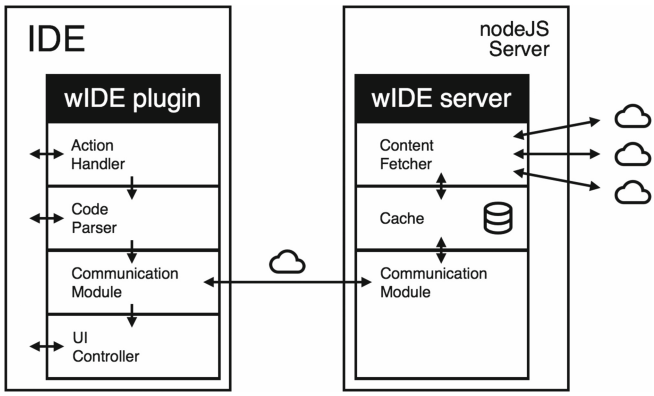

Fig. 6. The architecture of wIDE.

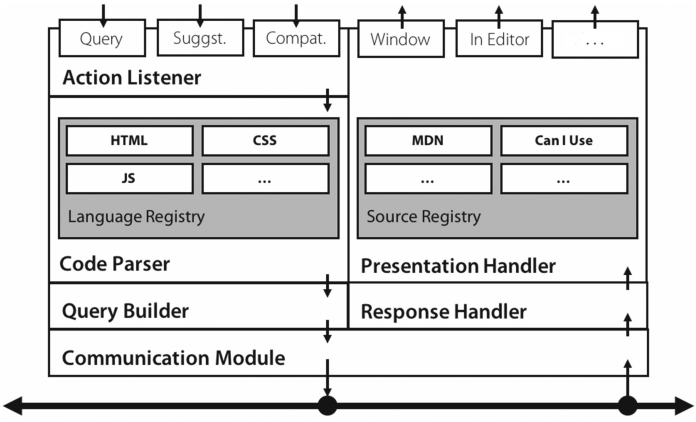

Fig. 7. The components of the wIDE client. The left side processes a query for the knowledge base, while the right side manages the response from the server once the queries have produced a result.

are mostly represented by a REFERENCE node as the first child of a CALL node. Clearly, different patterns need to be applied depending on the language. In addition, different parsing rules have to be considered when wIDE has to handle lookup requests on incomplete code.

Identifying which are the interesting EOIs in a selection of HTML or CSS is quite simple. In HTML, any code selection between the tag delimiters <> will lead to a lookup either of the tag or of the attribute. In the case of CSS, wIDE will only allow lookups on properties, meaning that a selection that contains either the property or its value, will lead in any case to a lookup of the property through the PSI tree.

In the case of JavaScript, wIDE needs to perform an additional step before performing a lookup in the knowledge base because of two main issues. First, the

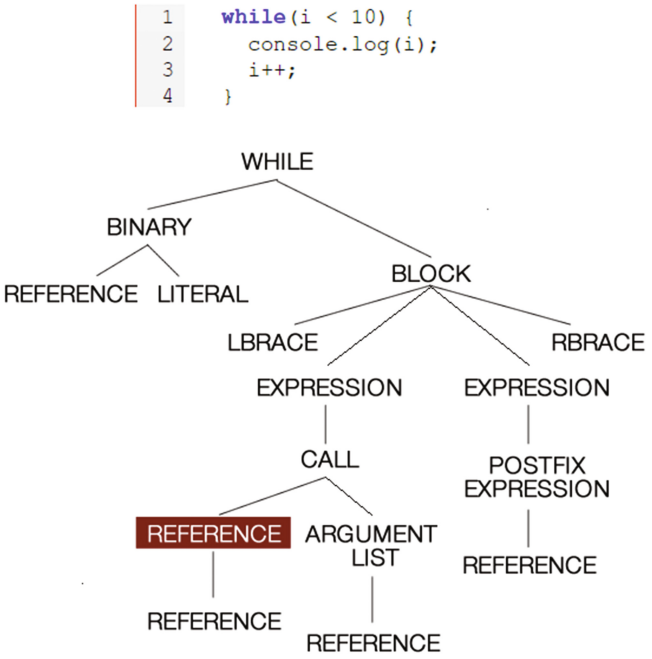

Fig. 8. A simple JavaScript code snippet with the corresponding PSI tree. The red box highlights the EOI which will be found as appropriate for the query, based on pattern matching. (Color figure online)

potential amount of defined functions is unlimited and, second, there could be name conflicts between functions, especially in the case where multiple libraries are included in the project. Therefore, wIDE traces back the function definition to distinguish whether it is natively defined, for example *getElementById*, or was defined in a library that has been included. We found this step quite challenging for two reasons. First of all, JavaScript is loosely typed and the receiver of a function call is not simply distinguishable before runtime. Second, it may be possible that some libraries are available in the project files, while others may be linked from a content delivery network. In our implementation, wIDE is only able to look for JavaScript definitions within the files of the project workspace and the native functions. Once possible definitions have been traced back, wIDE performs a lookup of all potential function definitions, ordered by the probability that each individual function may be the one called at runtime.

Once the EOI is identified through the appropriate language handler, the client communicates with the server by sending a JSON-based request that contains various information items such as the type of EOI and the content, and it even allows for composite requests, as shown in Fig. 9. The composite requests allow for a request to contain additional sub-requests, which can be useful in special cases, such as when a central EOI is sent, and additional EOIs need to be shipped with the parent request, for example satellite EOIs.

```
'WideLookupRequest': {
  'lang': 'HTML',      // required - The language (HTML, CSS, JS)
  'type': 'tag',       // required - The type of the element
  'key': 'form',       // required - The name of the element
  'value': '',         // required for some types of elements
  'children': [        // optional - requests may include subrequests
    {
      'lang': 'HTML',
      'type': 'attribute',
      'key': 'method',
      'value': 'POST'
    }]
}
```

```
'WideLookupResponse': {
  'lang': 'HTML',
  'type': 'tag',
  'key': 'form',
  'value': '',
  'children': [...],      // subresponses, if there were subrequests
  'documentation': [      // one entry for every documentation source
    'mdn': {...},
    ...]
}
```

Fig. 9. The format of JSON requests and responses being exchanged between the wIDE client and the server.

The format of the server response is also shown in Fig. 9. In the server response, the lookup results are attached as a payload of individual objects for each source. Once the response is received, the client can present the received lookup result through the presentation handler of each source, as shown in Sect. 3.

On the server, requests get analysed and, if the response has already been stored from previous queries, it can be returned immediately. If not, the request gets decomposed by a Query handler and a Compatibility handler, which, respectively, will query resources from the corresponding source registry (see Fig. 10). Source handlers will manage the communication with the respective source. Note that while Can I Use offers an API to query for individual EOIs, MDN does not. Once the querying is complete, the result can be sent back to the client and saved in the cache, with an expiration time of 7 days.

As mentioned earlier, wIDE currently only supports JavaScript, HTML and CSS, but could be extended to support other languages commonly used in web applications such as PHP and Ruby, for example to compare support for functionalities across different runtime versions. The architecture is designed so that such extensions can be supported easily. To add support for a language, the developer must implement a language handler for the client and a source handler for both the client and the server. A language handler consists of three main components:

- an *abbreviation* of the language to distinguish on the server side which language is being queried,
- a *language parser*, which identifies the EOIs and builds up the lookup request to the server,

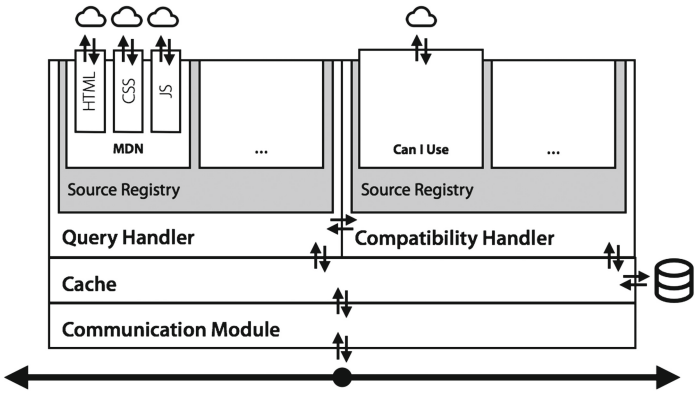

Fig. 10. The decomposition of the wIDE server.

– a *window factory*, that handles the presentation of the lookup result in the wIDE sidebar, allowing for language specific presentation styles of the results.

In addition to the language handler, it may be possible that the existing client source handlers are not able to present any information for the new language being added. Therefore, wIDE provides extensibility of the source handlers for result presentation in the client. Essentially, any implementation of a source handler needs to define three main methods:

– a constructor that takes and parses the server response in JSON format,
– a method that allows the construction and the population of the extensible panes with content from the new source,
– a method to compute compatibility support, if required, for example for different runtime versions in the case of PHP.

Clearly, the source handlers on the server need to implement a method to query and extract the appropriate content depending on each source's public interface. For example, the source handler for MDN will extract data and styling for the appropriate sections from the public website.

5 Evaluation

We evaluated wIDE in a qualitative user study with 9 developers to receive feedback on our approach of integrating documentation and compatibility information into the IDE.

5.1 Tasks and Procedure

We designed two tasks: one that focused on the documentation lookup and one for the compatibility scan. For the *documentation lookup* task, participants were

asked to solve a development task in JavaScript. The task was tailored to include language features that the participants were likely unfamiliar with to encourage the use of the documentation lookup. We asked participants to verify email addresses for a simple newsletter sign up page using regular expressions. This functionality can be achieved with the built-in `RegExp` object. Participants were provided with a skeleton of the application and asked to implement the missing functionality. To solve the task, the participants were allowed to use the wIDE plugin, switch to the browser and ask questions. Browser usage and questions were noted and used in the analysis of the results.

For the *compatibility* task, participants received an existing project and were asked to find compatibility issues. For the identified issues, participants had to analyse if they were crucial to the functionality of the application and resolve them so that the required browser versions were supported. The provided project was a simple to-do application consisting of one HTML, one JavaScript and one CSS file and had the following CSS and JavaScript compatibility issues.

- **CSS border-radius** not supported in IE versions <9. Cosmetic.
- **CSS text-decoration** full support only in Firefox, partially supported by Safari and behind experimental flags in Chrome and Opera. Cosmetic.
- **JavaScript getElementsByClassName** not supported in IE versions <9. Crucial.
- **JavaScript addEventListener** not supported in IE versions <9. Crucial.

Before the first tasks, participants were introduced to the wIDE plugin and guided through an introductory task with the aim of familiarising them with the plugin's features and usage. After all tasks had been completed, participants were asked to fill in a questionnaire. In addition, we logged usage data from the plugin and took notes of the participants' behaviours such as when they struggled, when they used the browser and what they were looking for in that case.

5.2 Participants

We recruited nine participants from our university. Participants were between 22 and 41 years old, with an average age of 27 years. Two participants were female and seven were male. All but one had a background in computer science and the remaining participant was a student of electrical engineering. We required at least basic knowledge of HTML, CSS, and JavaScript to participate in the study and relied on self-assessment for these skills. All participants reported having encountered compatibility issues as developers before our study. When asked about documentation sources that participants were familiar with, the most common answer was *StackOverflow*, which was known to all of them, followed by *w3schools* (6 mentions) and the official *w3c* web standard documentation (5 mentions). Only three users were aware of *Can I Use* and *MDN*.

5.3 Results

For the documentation tasks, participants triggered 11.4 documentation lookups explicitly on average, while 71.5 lookups were trigged automatically as users were

scrolling through the auto-complete suggestions. On average, a user opened 7.7 documentation panes to look at the content. We observed two participants who mainly accessed documentation through the browser and only opened one and two panes, respectively. For the compatibility, we measured 2.5 project scans and 3.2 file scans on average per participant. Analysing the questionnaires, we found that all participants stated that wIDE provides relevant or very relevant information. Some participants would have liked to have even more example-centric content that demonstrates the usage of a feature or provides alternative solutions in the case of compatibility issues. Based on our observations, users might benefit from a natural language access point to the documentation. Our approach based on the auto-complete suggestions requires the user to be aware of the name of an EOI, or at least the first few characters of it. In our study, many participants struggled to find the `RegExp` object and either switched to the browser or asked for help. Only three participants managed to access the documentation for this object using the auto-complete suggestions without help.

The compatibility scans received positive feedback[8]. Suggestions for improvement included automatic background scans for changed code that would highlight problematic elements as the user is writing code. Currently, wIDE will only highlight compatibility issues but developers have to figure out how to address them. Multiple participants requested more assistance for resolving issues. For example, the system could provide alternative solutions or polyfills. One participant suggested to have small tiles that visually display the output of an application loaded into different browsers. Another participant requested an extension for PHP.

6 Conclusion

We have presented wIDE, a system to support web developers obtain compatibility and documentation information directly in the development environment. To provide support for web technologies in the IDE, wIDE builds a knowledge base that centralises information extracted from various online resources and displays it to the user non-intrusively and in-place, thereby reducing the need to switch the context to the browser. The sources are queried through a context-aware parsing of the source code, to provide information about documentation for the various functionalities and their compatibility across various browsers. This helps in locating cross-browser compatibility issues, either before moving to a cross-browser testing stage, or after such stage has been completed and the issues in the source code have to be located and addressed.

Currently, our implementation of wIDE for IntelliJ IDEA supports the languages which are relevant for a cross-browser compatibility analysis, namely HTML, CSS and JavaScript. However, it could be extended to even perform compatibility analysis of server-side languages, with the goal of analysing the compatibility of functionalities across different runtime versions. In addition,

[8] Questionnaire and responses at https://github.com/fabwid/wIDE/blob/master/user_study/userStudyResponses.xlsx - Accessed 21 March 2017.

together with the support for additional languages, wIDE also offers support for introducing new knowledge resources, which might be required if additional languages were added.

We designed and conducted a user study to evaluate the implementation of wIDE in terms of the feedback that it provides to developers. The study consisted of a development task and a compatibility analysis task. The data logs, observations and qualitative feedback provided by the participants have shown that wIDE provides relevant information based on the user's needs. However, they also expressed the desire for additional support for the automatic resolution of compatibility issues, and more assistance in looking for features beyond the current support which relies on the IDE auto-completion.

The problem of automatically resolving compatibility issues could be addressed in future work. For example, in some cases, MDN provides a *polyfill* implementation of missing JavaScript functionalities, which could be included automatically. We also plan to introduce more example-centric sources such as StackOverflow based on compatibility issues in order to improve support for resolving issues once these are located. We have opened the project sources[9] with the hope that this might encourage others to extend the set of languages supported, and even extend the scope of the compatibility analysis to consider server runtimes.

References

1. Mohorovičić, S.: Implementing responsive web design for enhanced web presence. In: 36th International Conference on Information & Communication Technology Electronics & Microelectronics (MIPRO), pp. 1206–1210. IEEE (2013)
2. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R.: Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM (2009)
3. Hartmann, B., Dhillon, M., Chan, M.K.: HyperSource: bridging the gap between source and code-related web sites. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM (2011)
4. Goldman, M., Miller, R.C.: Codetrail: connecting source code and web resources. J. Vis. Lang. Comput. **20**(4), 223–235 (2009)
5. Sawadsky, N., Murphy, G.C.: Fishtail: from task context to source code examples. In: Proceedings of the 1st Workshop on Developing Tools as Plug-ins. ACM (2011)
6. Sahavechaphan, N., Claypool, K.: XSnippet: mining for Sample Code. ACM Sigplan Not. **41**(10), 413–430 (2006)
7. Moreno, L., Bavota, G., Penta, M.D., Oliveto, R., Marcus, A.: How can I use this method? In: Proceedings of 37th IEEE International Conference on Software Engineering, vol. 1. IEEE (2015)
8. Li, H., Zhao, X., Xing, Z., Bao, L., Peng, X., Gao, D., Zhao, W.: amAssist: In-IDE ambient search of online programming resources. In: Proceedings of IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE (2015)

[9] https://github.com/fabwid/wIDE.

9. Brandt, J., Dontcheva, M., Weskamp, M., Klemmer, S.R.: Example-centric programming: integrating web search into the development environment. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM (2010)
10. Cordeiro, J., Antunes, B., Gomes, P.: Context-based recommendation to support problem solving in software development. In: 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE). IEEE (2012)
11. Rahman, M.M., Yeasmin, S., Roy, C.K.: Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In: IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE). IEEE (2014)
12. Ponzanelli, L., Bacchelli, A., Lanza, M.: Seahawk: stack overflow in the IDE. In: Proceedings of the International Conference on Software Engineering. IEEE Press (2013)
13. Ponzanelli, L., Bacchelli, A., Lanza, M.: Leveraging crowd knowledge for software comprehension and development. In: 17th European Conference on Software Maintenance and Reengineering (CSMR). IEEE (2013)
14. Ponzanelli, L., Bavota, G., Penta, M.D., Oliveto, R., Lanza, M.: Mining StackOverflow to turn the IDE into a self-confident programming prompter. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM (2014)
15. Aho, T., Ashraf, A., Englund, M., Katajamäki, J., Koskinen, J., Lautamäki, J., Nieminen, A., Porres, I., Turunen, I.: Designing IDE as a service. Commun. Cloud Softw. **1**(1) (2011)
16. van Deursen, A., Mesbah, A., Cornelissen, B., Zaidman, A., Pinzger, M., Guzzi, A.: Adinda: a knowledgeable, browser-based IDE. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 2. ACM (2010)
17. Eaton, C., Memon, A.M.: An empirical approach to evaluating web application compliance across diverse client platform configurations. Int. J. Web Eng. **3**(3), 227–253 (2007)
18. Choudhary, S.R., Prasad, M.R., Orso, A.: X-PERT: accurate identification of cross-browser issues in web applications. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press (2013)
19. Choudhary, S.R., Versee, H., Orso, A.: WEBDIFF: automated identification of cross-browser issues in web applications. In: International Conference on Software Maintenance (ICSM). IEEE (2010)
20. Choudhary, S.R., Prasad, M.R., Orso, A.: Crosscheck: combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation. IEEE (2012)
21. Mesbah, A., Prasad, M.R.: Automated cross-browser compatibility testing. In: Proceedings of the 33rd International Conference on Software Engineering. ACM (2011)
22. Husmann, M., Spiegel, M., Murolo, A., Norrie, M.C.: UI testing cross-device applications. In: Proceedings of the ACM Conference on Interactive Surfaces and Spaces. ACM (2016)
23. Xu, S., Zeng, H.: Static analysis technique of cross-browser compatibility detecting. In: Proceedings of the International Conference on Applied Computing and Information Technology/International Conference on Computational Science and Intelligence. IEEE (2015)