Jact: A Playground Tool for Comparison of JavaScript Frameworks

Nozomi Nakajima, Shinsuke Matsumoto and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
{n-nakajm, shinsuke, kusumoto}@ist.osaka-u.ac.jp

Abstract—**Comparing and selecting the most appropriate JavaScript Framework (JSF) is an important activity for web application development. However, it is difficult to collect much information for comparison of JSFs. In this paper, we present a playground tool named Jact to support comprehension of individual JSF characteristics. By the concept of playground, users can freely run, edit, and share source code in their web browsers. Based on the concept, Jact enables developers to compare source codes and runtime performances of each JSF based on typical tasks in web development. Task-based comparison is useful for understanding the feature of each JSF. Furthermore, developers can submit tasks and source code which uses a specific JSF. By sharing tasks and source code written by not only administrators but also developers, Jact can continuously provide information relating to JSF, which includes benchmark and API usage. In order to evaluate the effectiveness of Jact, we conducted a subject experiment with 13 participants. Jact is currently available at http://13.231.18.92.**

Index Terms—**framework, web, playground**

I. Introduction

With an increase in demand for web technology, web application source code is always getting more complicated [1]. Currently, a large number of JavaScript frameworks (JSF) have appeared to ease such complexity. Some well-known JSFs include Vue.js[1], Angular[2], and React[3]. JSF contributes to facilitating development and maintenance by introducing design philosophy [2] [3] such as Model-View-Controller (MVC) and Model-View-ViewModel (MVVM). It is known that selecting an appropriate JSF assists the success of the development [4].

However, comparing and selecting the most appropriate JSF is not a simple problem due to the following three reasons;

(P_1) **Various choices:** There are many JSFs in the world. Individual JSF employs firm design philosophy and its own syntax rule. Developers need to have a broad understanding of these philosophical and syntax differences to meet the characteristics of their developing web application and their preferences.

(P_2) **Various runtime environments:** The runtime environment of a web application is composed of a user's web browser, operating system, and device. So, the possible number of the runtime environment will be vast. The difference of web browser is the same as the difference of JavaScript engines (e.g., Google's V8 and Mozilla's SpiderMonkey). Gizas et al. have pointed out that JSF performance is different depending on the environment [4]. Though the performance of an application strongly influences on its user experience [5], it is one of the important viewpoints for selecting a JSF. Therefore, web application developers should know the JSF characteristics depending on environmental differences.

(P_3) **Outdated information:** The number of JSFs and their versions have risen continuously and rapidly. Performance comparison of JSF itself has been conducted in the academic field [4] and on the web [6] [7]. Plenty of JSF usage information is also available on the web. However, such kind of **static information** will be deprecated with JSF's and browser's update. Outdated information may hinder understanding of JSF characteristics. JSF relating information including benchmark and API usage should be continuously maintained.

Besides, migration from a specific JSF to another JSF requires tremendous efforts. Unlike a software library which aims to reuse features in source code, JSF prescribes its design philosophy to the overall structure of source code. Therefore, it is necessary for developers to precisely and carefully grasp the characteristics of each JSF at the beginning of development.

In this paper, we present a playground tool named Jact to support comprehension of individual JSF characteristics. The concept of **playground** enables users to freely run, edit, and share source code in their web browser. Based on the concept, Jact provides a feature of source code comparison to grasp individual JSF usages to deal with P_1. Also, Jact supports on-demand runtime performance measurement to respond to P_2. These two features are based on a typical and small task in web application development such as DOM manipulation (e.g., change text color) and Ajax (e.g., get JSON data). These task-based features allow developers to facilitate understanding of JSFs by dividing a complex problem into smaller pieces. As a solution to P_3, Jact allows developers to submit task and source code which uses a specific JSF. By accepting submission from a developer, Jact can enrich its contents and catch up with new information. At the time of source code submission, Jact executes source code testing. The feature of source code testing certifies the correctness of registered source code. Also, we conducted a subject experiment with 13 participants in order to evaluate the effectiveness of Jact. We believe that Jact contributes to the comprehension of JSFs and developer's decision making. Currently, Jact is available at **http://13.231.18.92**.

[1]https://vuejs.org
[2]https://angularjs.org/
[3]https://reactjs.org/

II. Preliminaries

A. Playground

In general, playground is a concept that enables users to run and edit their source code on a web browser. Developers can readily check source code and its demonstration without development environment setup. Also, playground enables source code sharing via URL. JSFiddle[4] is one of the most popular playground services of JavaScript. As well as JavaScript, playgrounds are also available in various languages.

B. JavaScript framework (JSF)

JSF is a skeleton of source code which abstract developing system. Vue.js and Angular are popular JSFs. Figure 1 shows source code with and without Vue.js. Both source codes change text "World" to "JS"on a web page. Whereas library gives functions to the source code, JSF gives a structure to the developing source code according to its design philosophy as shown in Figure 1.

Each JSF adopts its own syntax rule. Even if two JSFs adopt the same design philosophy, their syntax rules are different. As such, migration from a specific JSF to another JSF requires much effort. Developers need to carefully grasp the characteristics of each JSF at the beginning of development.

III. Jact

A. Overview

We propose Jact which supports comparison of JSFs. Figure 2 shows the overall architecture and its usage flow of Jact. Jact provides to compare differences of source code and runtime performance. Through these comparisons, a developer can understand various characteristics of individual JSFs. Also, Jact adopts a concept named playground. A developer can run, edit, and submit their source code and tasks through Jact. By submission from a developer, Jact can enrich information for comparison.

```html
<html>
  <div id="app">
    <p>Hello
     <span id="msg">World
     </span>
    </p>
  </div>
  <script>
    document
    .getElementById('msg')
    .innerText = 'JS';
  </script>
</html>
```

(a) JavaScript(without JSF)

```html
<html>
  <div id="app">
    <p>Hello {{ msg }}</p>
  </div>
  <script>
    var app = new Vue({
      el: '#app',
      data: {
        msg: 'World'
      }
    })
    app.msg = 'JS';
  </script>
</html>
```

(b) Vue.js

Fig. 1: Difference of implementation with and without JSF

[4]https://jsfiddle.net/

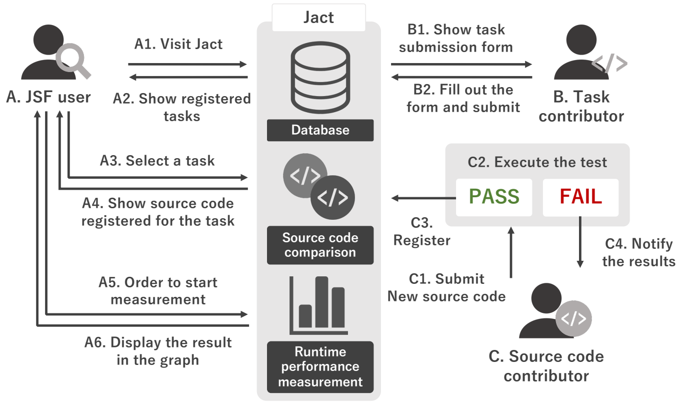

Jact
A. JSF user
A1. Visit Jact
A2. Show registered tasks
A3. Select a task
A4. Show source code registered for the task
A5. Order to start measurement
A6. Display the result in the graph
Database
Source code comparison
Runtime performance measurement
B1. Show task submission form
B2. Fill out the form and submit
B. Task contributor
C2. Execute the test
PASS
FAIL
C3. Register
C4. Notify the results
C1. Submit New source code
C. Source code contributor

Fig. 2: Architecture of Jact and its usage flow

B. Actors

As shown in Figure 2, Jact supposes three kinds of actors: **JSF user**, **Task contributor**, and **Source code contributor**. The usage flow of Jact depends on each actor.

JSF user. Those who compare some JSFs. As shown in A_1 to A_6 in the figure, this actor compares JSFs in terms of syntax rules and runtime performance. By selecting a task, Jact shows source code registered for the selected task and enables syntax comparison of JSF. Besides, this actor can start runtime performance measurement and check the result.

Task contributor. Those who submit useful tasks. As shown in B_1 and B_2, this actor fills out a submission form presented by Jact and submits a new task.

Source code contributor. Those who submit their source code using JSF. As shown in C_1 to C_4, this actor submits source code implementing the selected task. If the test were successful, it would be registered formally.

C. Features

In this section, we describe Jact's features. Table I shows how Jact's features correspond to P_1, P_2, and P_3. The detail of these features is described as follows.

1) Task-based comparison: In Jact, an actor compares JSFs based on small tasks which are typical in client-side development, such as DOM manipulation and Ajax [4]. These tasks are easy to understand and expandable to more complicated

TABLE I: Features of Jact corresponding to the three problems

Features	P_1	P_2	P_3
Source code comparison	✓		
Runtime performance measurement		✓	
Playground	✓		✓
Submission by actors			✓
Source code testing			✓[5]

P_1: Various choices
P_2: Various runtime environments
P_3: Outdated information

[5]Source code testing contributes to submission by actors.This feature does not respond to P_3 directly.

functions. Table II shows examples of typical tasks used in Jact. For picking out tasks, we refer to jQuery Examples [8].

2) Source code comparison: This feature responds to P_1. After selecting a task, an actor will be able to compare source code which implements the selected task. Figure 3 is a screenshot of Jact. In the left side, an actor can select a task and source code. Jact also shows information about the task and source code under each select box. The numbers of watches, likes, and copies show the degree of contribution to the comprehension of JSFs. An actor can compare JSFs in reference to these numbers. In the right side, Jact shows selected source code and check its behavior with the preview frame. An actor can edit new source code in the code editor in the upper right. Also, an actor can copy source code from code viewer and rewrite freely. Rewritten source code is also checked its behavior with the preview frame.

3) Runtime performance measurement: This feature responds to P_2. For each source code of the selected task, Jact measures the processing time to achieve the task 100 times. By measurements in various environments, an actor will get performance information on-demand. An actor can start measurement by clicking the button in the lower right in Figure 3. A measurement flow is described below.

a. Generate one hundred iframes. To avoid the effect of cache, measurements are executed in different iframes.
b. Embed source code to iframe. An iframe can embed source code by setting its src attribute. Jact has an API which returns source code. Jact sets a URL of the API to the src attribute of each iframe.
c. Set a target of observation. To detect a mutation invoked by an event, Jact applies MutationObserver. MutationObserver API can execute a callback function when a mutation is observed.
d. Start measurement. Jact records the start time of measurement.
e. Trigger an event. Clicking a button is one of the example of Trigger.
f. Detect a change and Stop measurement. MutationObserver executes a callback function which records the finish time. The difference between start time and finish time is processing time.
g. Jact repeats c. to f. in one hundred times and calculates the total processing time.

This measurement will be made in each source code in series. After the measurement, Jact shows the result in a graph. Figure 4 is a screenshot of the window showing the result of runtime performance measurement. A measurement result is shown as a barplot.

4) Task submission: There is a limit to provide enough information by our registration. In order to enrich information for comparison, Jact has a feature of task submission.

An actor can move from the button shown in Figure 3 to the task submission form. Table III shows the items required for the task submission. As essential matters, **Task name, Contributor, Description, Source code**, and **its name** are required. In addition to them, Jact requires **Pre, Trigger, and Post** for source code testing. We describe the source code testing in the next section.

5) Source code submission: For the adaptation of the concept of a playground, Jact has a feature of source code submission. This feature enables Jact to enrich its contents and catch up with the latest information. What we expect to submit is below.

- Source code using unregistered JSF
- Source code using a new version of registered JSF
- Another source code using registered JSF

Source code will be registered when source code testing succeeds. The testing requires source code to satisfy three conditions: **Pre, Post**, and **Trigger**. **Pre** means precondition which prescribes the initial state of source code. **Trigger** induces the event in the source code. **Post** is postcondition of the event. With three conditions, source code testing is executed as below.

a. Embed the given source code into an iframe. This iframe is used to remove the effect of this testing as a sandbox.
b. Jact runs **Pre** to check precondition.
c. Jact runs **Trigger**. The test fails if **Trigger** cannot be executed due to lack of executed DOM element.
d. Jact runs **Post** to check postcondition.
e. Notify the test result to source code contributor.

Both precondition and postcondition are necessary to be satisfied to pass the testing. If the source code testing succeeded, Jact shows registration form to input JSF name, contributor name, and tags which describe the feature of the source code.

D. Implementation

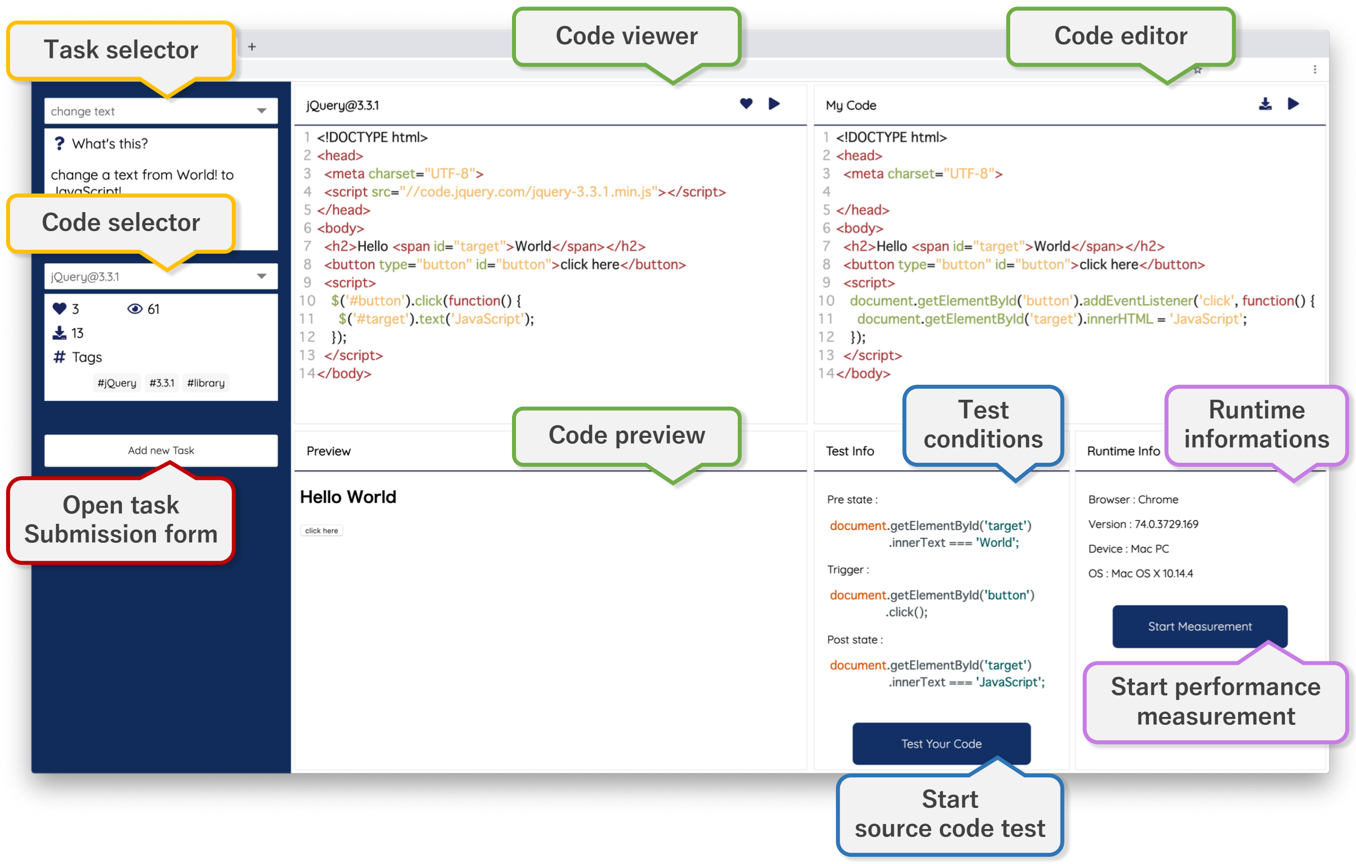
For the implementation of the user interface, Jact is made with Vue.js. Also, Jact employs CodeMirror as a syntax highlighter. Jact gets information about tasks and source code from the database through REST API. We use Node.js as

TABLE II: Examples of typical tasks used in Jact

Category	Typical task
DOM manipulation	Change text body
	Change text color
	Add class
	Remove class
Ajax	Get JSON data
	Post JSON data
Callback	Change text with retrieved JSON data
	Validate user's input

TABLE III: Required items for task submission

Item	Description
Task name	Task name to submit
Contributor	Task contributor's name
Description	Description of task to submit
Source code name	JSF name used for sample source code
Source code	Sample source code
Pre	Precondition of the task
Trigger	Trigger of the event in task
Post	Postcondition of the task



```html
<!DOCTYPE html>
<head>
 <meta charset="UTF-8">
 <script src="//code.jquery.com/jquery-3.3.1.min.js"></script>
</head>
<body>
 <h2>Hello <span id="target">World</span></h2>
 <button type="button" id="button">click here</button>
 <script>
  $('#button').click(function() {
   $('#target').text('JavaScript');
  });
 </script>
</body>
```

```html
<!DOCTYPE html>
<head>
 <meta charset="UTF-8">

</head>
<body>
 <h2>Hello <span id="target">World</span></h2>
 <button type="button" id="button">click here</button>
 <script>
  document.getElementById('button').addEventListener('click', function() {
   document.getElementById('target').innerHTML = 'JavaScript';
  });
 </script>
</body>
```

Pre state :

```
document.getElementById('target')
        .innerText === 'World';
```

Trigger :

```
document.getElementById('button')
        .click();
```

Post state :

```
document.getElementById('target')
        .innerText === 'JavaScript';
```

Browser : Chrome

Version : 74.0.3729.169

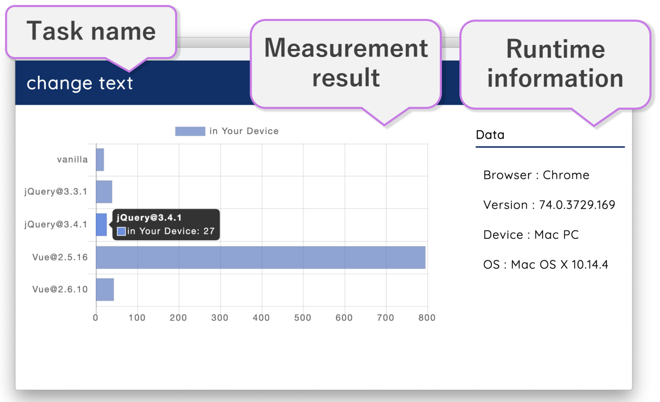
Device : Mac PC

OS : Mac OS X 10.14.4

Jact is available at **http://13.231.18.92**

🟨 Initial Selection
🟩 Source code comparison
🟦 Source code test & submission
🟪 Runtime performance measurement
🟥 Task submission

Fig. 3: Screenshot of Jact



Data

Browser : Chrome

Version : 74.0.3729.169

Device : Mac PC

OS : Mac OS X 10.14.4

Fig. 4: Screenshot of the result of runtime performance measurement

an API server and Express to make REST API. Jact uses MongoDB as a database.

IV. Preliminary Experiment

A. Overview

The purpose of the preliminary experiment is to indicate that there is a performance difference between JSFs. We select a task "Change text body" which changes the text on a web page. For comparison, we select three JSFs: vanilla, which is pure JavaScript source code and includes no library or JSF, jQuery (v3.3,v3.4), and Vue.js (v2.5,v2.6). jQuery is selected because of its usage rate. jQuery is used by 73.9% of all the websites [9]. We select Vue.js from the viewpoint of its popularity from developers. Vue.js is the most starred GitHub repository in all JSF repository [10]. Google Chrome and Firefox are selected as browsers. Also, iPhone, Android, Windows PC, and Mac PC are used as devices. By multiplying two browsers and four devices, we can measure on eight environments.

B. Results

Figure 5 shows the results of the measurement on Firefox. Figure 6 shows the results of the measurement on Google Chrome. We conduct each measurement in five times and average the results. In this bar chart, the vertical axis shows the runtime performance rate compared to the result of vanilla

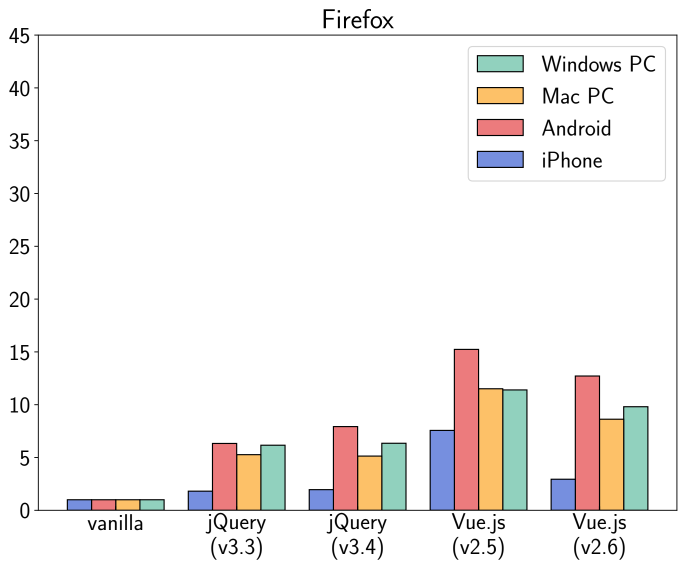


Fig. 5: Results of the runtime performance measurement on Firefox

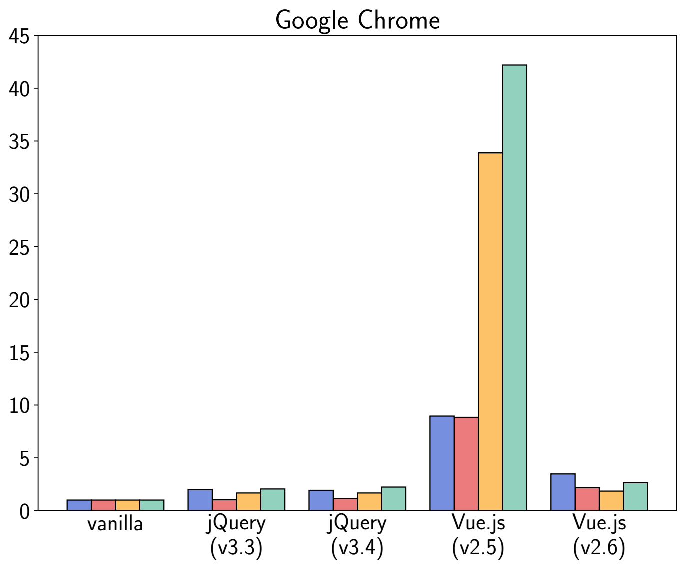


Fig. 6: Results of the runtime performance measurement on Google Chrome

on each environment. From left, four bars in the figure show the results on iPhone, Android, Mac PC, and Windows PC.

The results of jQuery are not so apart from the results of vanilla. However, there is a difference in the results of Firefox and Chrome. The performance rate on Firefox tends to be high compared to the results on Chrome. As to jQuery, there is not much difference between its versions.

Overall, Vue.js degrades runtime performance more significantly than jQuery. In the environment of Windows PC on Chrome, the performance rate of Vue.js version 2.5 reaches more than 40 times as long as vanilla. Also, the results of Vue.js version 2.6 is quite different from the one of Vue.js version 2.5. Notably, the results on Chrome is not almost different from jQuery. These results indicate that the later version of Vue.js has been improved in terms of performance.

C. Discussion

These results provide valuable information to understand the feature of JSFs' performance and to select an appropriate JSF. Generally, performance and utilities of JSF is a trade-off relationship. JSF degrades runtime performance compared to vanilla. Vanilla performs the best because it does not execute extra source code. Therefore, developers are required to select an appropriate JSF considering a balance between the convenience of JSF and its effect on performance. Also, the performance decline rate depends on JSFs, their versions, and their runtime environments. When developers select a JSF, they need to consider such influences. In that respect, Jact can provide valuable information. Jact enables runtime performance measurement on-demand, so developers do not have to spend time for performance measurement.

V. Evaluation

A. Overview

We conducted an evaluation experiment to confirm that Jact contributes to the comprehension of JSFs. As an experimental work, participants compare JSFs in terms of syntax rules and runtime performances. Through the work, we examine how exactly participants grasp the difference of JSFs by using Jact. After the work, we conduct a questionnaire about the usability of Jact. For a comprehension of JSFs, we prepare four tasks. We select three JSFs: vanilla, jQuery, and Vue.js. jQuery and Vue.js are selected for the same reason as a preliminary experiment. Vanilla is used as a standard of a comparison of JSFs.

B. Participants

We recruited 13 participants. The participants consisted of three undergraduate students, nine graduate students, and one associate professor. Figure 7 shows the usage experience of the participants. Five in participants have no experience of JavaScript. Almost half of the participants have no experience of jQuery and Vue.js. For JSF beginners, we select basic and

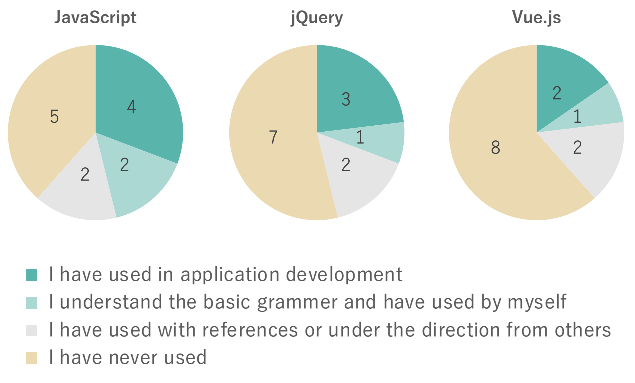


Fig. 7: Usage experience of participants

step-by-step tasks and insert comments to source code. These features and inserted comments help beginners to understand source code example [11].

C. Registered Tasks

For this experiment, we registered the following tasks.

- **Select DOM elements:** Selecting DOM elements in HTML by id name and class name.
- **Change an element:** Selecting a DOM element in HTML by id name and change a text body of the element.
- **Change some DOM elements:** Selecting DOM elements in HTML by class name and change text bodies of the elements.
- **Attaching events to button elements:** Selecting button elements in HTML and attaching click event and mouseover event to each element. This task overlaps the contents of the above three tasks.

D. Experiment Design

As experimental works, participants execute source code comprehension and runtime performance comprehension.

1) Source Code Comprehension: About four tasks mentioned in Section V-C, participants use source code comparison and playground of Jact. Participants describe the difference of the syntax rules of jQuery and Vue.js compared with vanilla.

2) Runtime Performance Comprehension: Participants measure the processing time of JSFs by using runtime performance measurement of Jact. Measurement is carried out for the task "Attaching events to button elements" mentioned in Section V-C. After the measurement, participant describes the difference of the runtime performance of jQuery and Vue.js from the following viewpoints:

- Features of each JSF compared with vanilla
- Features of each JSF's version
- Features of each syntax/API
- Features of each browser

Also, based on the measurement, we ask participants how do you consider the performance difference between JSFs.

3) Usability Evaluation: After the above two experimental works, we conducted a questionnaire about the usability of Jact. The questions in the questionnaire are following;

- Source code comparison contributes to the comprehension of syntax rules of JSFs.
- The concept of a playground is useful for the comprehension of syntax rules of JSFs.
- Performance measurement contributes to the comprehension of a runtime performance of JSFs.
- On-demand measurement is useful for the comprehension of runtime performance of JSFs.
- Jact contributes to the comprehension of JSFs.

Each question in the questionnaire is evaluated by Likert scale [12] [13]. Also, we collect additional comments about Jact by free description.

E. Results

1) Source Code Comprehension: We extract some descriptions of the experimental work of source code comprehension. Among the descriptions, the following comments were obtained for jQuery;

> In jQuery, $ function is used for getting DOM elements by id or class name. An argument of $ function starting from "." means getting DOM elements by class name. If an argument of $ function begins with "#", $ function gets a DOM element by id name.

> In vanilla, we change a text body by variable assignment. On the other hand, jQuery enables to change a text body by a function call.

> When using an event handler, vanilla needs "eventListener." The first argument of eventListener is an event name, and the second argument is a callback function. In contrast with vanilla, jQuery defines event functions on each event such as click, mouseover. Therefore, we can bind a callback function by using its event function..

From these descriptions, it is revealed that participants understand how to select DOM elements by using $ function. Also, some answers mention that jQuery use some functions in case of DOM manipulation. Function operation is the difference with vanilla.

Next, we extract some comments about Vue.js.

> Vue.js enables to define a method with Vue instance generation. We can declare data definitions and methods separately. This improves the visibility of source code.

> In Vue.js, if we write the same placeholder in the part that refers to the same data in HTML, the part can be replaced simultaneously with the data.

> Vue.js binds a target DOM node and data in advance. We can manipulate a DOM element by assignment to the data in a component.

Participants often refer to the feature that Vue.js needs to bind an instance to HTML element. Some participants realized that Vue.js could define data and methods separately.

2) Runtime Performance Comprehension: We excerpt some comments obtained after the runtime performance measurement and consideration from four viewpoints mentioned in Section V-D2.

> Although each source code implements the same function, I do not know there is a big difference between them. Also, it is hard to consider these performances when I develop a web application with a JSF. In that

> matter, Jact briefly visualizes a performance difference. So I think Jact is a helpful tool.

> I did not guess that there is several times difference between the performances of JSFs. When an application needs quick responses, it is important to select an appropriate JSF.

> I got an impression that: The simpler the source code is, the longer the processing time is.

Through runtime performance measurement, participants understand that there is a significant performance difference between JSFs. Also, there is a comment that refers to a trade-off between the simplicity of source code and runtime performance.

3) Usability Evaluation: Figure 8 shows the results of the usability questionnaire. From these results, it is revealed that participants grasp syntax rules and runtime performances of JSFs by using Jact. Also, the Jact's features of source code comparison, runtime performance measurement, and playground are useful for the comprehension of syntax rules and runtime performance. As a result, 12 out of 13 participants answer that Jact contributes to the comprehension of JSFs. We extract some favorable comments from an additional description about Jact.

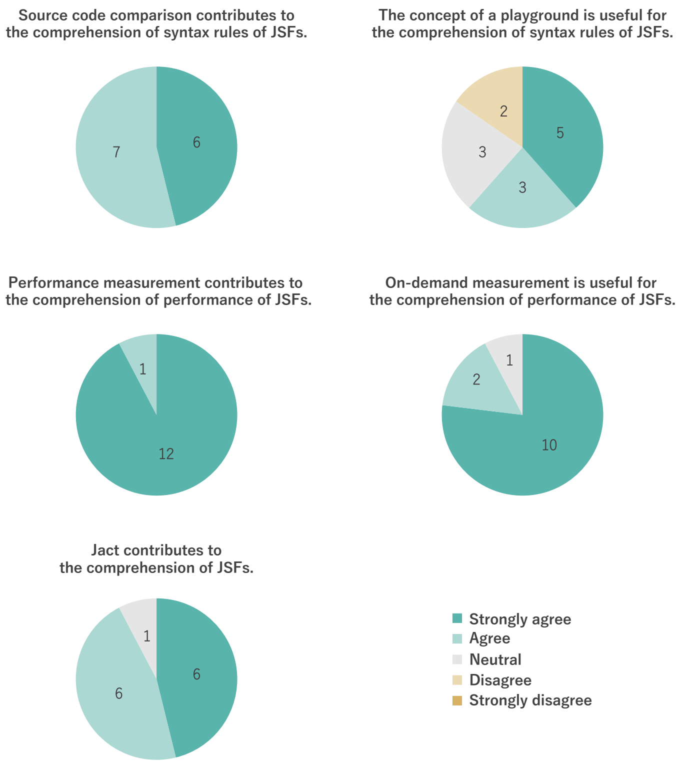


Fig. 8: Results of usability questionnaire

> It is excellent to understand syntax rules and performances easily. I want to use Jact when I select a JSF for web development. I want Jact to support other JSFs.

> It is useful to compare source code with my source code. Runtime performance measurement is also helpful to know the performance of JSFs.

> It is great to check the behavior of source code quickly.

On the other hand, there are some suggestions for improvement.

> I want to know the difference between source codes. It is just enough to highlight the difference.

> I want to use AltJS(AltJS stands for Alternative JavaScript. TypeScript is an example of AltJS.) in Jact.

F. Discussion

The result of this evaluation experiment enables to conclude that Jact contributes to the comprehension of JSFs. Some participants describe that Jact is an easy way to know syntax rules and runtime performance of JSFs. So it can be expected the use by developers who are looking into JSFs.

In this experiment, we provide four primary tasks. These tasks are based on basic syntax on JavaScript, and it is easy to understand for beginners. In practical development, more complicated functions such as DOM manipulation and Ajax are needed. In such a case, the difference in runtime performance is more significant. Also, developers are required to understand how to use JSF quickly. Therefore, Jact contributes to the comprehension of JSFs more effectively.

VI. Threats to Validity

In the evaluation experiment, there are some factors which may affect the validity of the results. First, most of the participants are not experienced in JavaScript. Jact expects to be used by more experienced developers. The evaluation by developers may be different from the result of the subject experiment. As future work, we are planning the evaluation by experienced developers. Second, four tasks used in the experimental works are extremely basic. In practice, developers use more complicated tasks shown in Table II. Such a difference in selecting tasks may influence the validity of the result. Finally, we conducted the only qualitative evaluation. Jact aims to support comprehension of JSFs. It is difficult to evaluate the degree of contribution by Jact quantitatively. Also, there is no existing tool supporting to understand JSFs in the same viewpoint as Jact. Therefore, all evaluation is qualitative, and the results are based on the subjectivities of participants.

VII. Related Works

It is an effective and practical approach to compare specific programming features based on typical program tasks. There have been many studies relating to comparison of programming languages using such typical tasks [14] [15] [16]. They used Rosetta Code[6] which provides many implementations of various languages for the same problem. Pano et al. have investigated factors to lead the decision making of JSF selection [2]. Implications of the study include that JSF documentation should include several examples for implementing common tasks. Our fundamental idea is to adopt such a practical approach to JSF comparison.

There have been various JSF comparison reports. Gizas et al. have compared several JSFs from views of traditional software metrics, of a security vulnerability, and of runtime performance [4]. One of their conclusions is that the performance of JSFs strongly depends on a web browser and on JSF version. TodoMVC[7] offers the same Todo applications implemented by major JSFs. It demonstrates the behavior of the application and publishes the source code in GitHub repository. For performance comparison, js-framework-benchmark[8] has been published. By cloning the repository, it enables to measure the performance of JSFs. Both of existing tools need environment setup for comparison while Jact is available on browsers. Also, compared to these reports, our tool adopts the concept of a playground to keep maintain these JSF relating information.

VIII. Conclusion

In this paper, we proposed a playground tool to support understanding characteristics of JSFs. Our tool provides to compare usage and runtime performance of various JSFs. In order to evaluate the usability of our tool, we conducted an evaluation experiment. The result indicated that our tool could contribute to the comprehension of JSFs.

Although our tool responds to the problems in selecting a JSF, there are some shortcomings. In syntax rules, our tool provides to compare only two JSFs at the same time. It is better for an actor to show the differences between all registered JSFs. Also, the feature of runtime performance measurement is not enough to know the performance of JSF completely. With our tool, an actor measure the performance in all expected environments. Though several measurements are the clue to the choice of an appropriate JSF, it is not enough to understand performance characteristics of JSFs perfectly. Therefore, we should look into the implementation of some functions such as submission of a measurement result, comparison with the results of others' measurement. Currently, our tool strongly depends on voluntary contributions from JavaScript developers. So, motivating and engaging them are the remaining important challenges. To meet the challenges, Jact should introduce a concept of gamification like Stack Overflow. Another future work includes supporting production build on server-side. Almost JSFs assume to apply production build to minify and optimize written web resources. Supporting production build may provide more practical performance comparison.

[6]http://www.rosettacode.org
[7]http://todomvc.com/
[8]https://github.com/krausest/js-framework-benchmark

Acknowledgements

This work was supported by JSPS/MEXT KAKENHI Grant Number 18H03222.

References

[1] S. Murugesan, Y. Deshpande, S. Hansen, and A. Ginige, *Web Engineering: a New Discipline for Development of Web-Based Systems*. Springer Berlin Heidelberg, 2001, pp. 3–13.

[2] A. Pano, D. Graziotin, and P. Abrahamsson, "Factors and actors leading to the adoption of a JavaScript framework," *Journal on Empirical Software Engineering*, vol. 23, no. 6, pp. 3503–3534, 2018.

[3] D. Graziotin and P. Abrahamsson, "Making Sense Out of a Jungle of JavaScript Frameworks," in *Product-Focused Software Process Improvement*, 2013, pp. 334–337.

[4] A. Gizas, S. Christodoulou, and T. Papatheodorou, "Comparative Evaluation of JavaScript Frameworks," in *Proceedings of the International Conference on World Wide Web*, 2012, pp. 513–514.

[5] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications," in *Proceedings of the 2010 USENIX Conference on Web Application Development*, 2010, pp. 3–3.

[6] P. Lewis. The Cost of Frameworks. https://aerotwist.com/blog/the-cost-of-frameworks/. Accessed at 10th July 2019.

[7] J. Schae. A Real-World Comparison of Front-End Frameworks with Benchmarks (2018 update). https://www.freecodecamp.org/news/e5760fb4a962/. Accessed at 10th July 2019.

[8] jQuery Examples. https://www.quackit.com/jquery/examples/. Accessed at 10th July 2019.

[9] Usage Statistics and Market Share of JavaScript Libraries for Websites, July 2019. https://w3techs.com/technologies/overview/javascript_library/all. Accessed at 10th July 2019.

[10] Collection: Front-end JavaScript frameworks. https://github.com/collections/front-end-javascript-frameworks. Accessed at 10th July 2019.

[11] J. Sillito, F. Maurer, S. M. Nasehi, and C. Burns, "What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 25–34.

[12] R. Likert, "A Technique for Measurement of Attitudes," *Archives of Psychology*, vol. 22, 1932.

[13] M. S. Matell and J. Jacoby, "Is There an Optimal Number of Alternatives for Likert-Scale Items? Study I," *Educational and Psychological Measurement*, vol. 31, pp. 657–674, 1971.

[14] S. Nanz and C. A. Furia, "A Comparative Study of Programming Languages in Rosetta Code," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 778–788.

[15] S. Georgiou, M. Kechagia, P. Louridas, and D. Spinellis, "What Are Your Programming Language's Energy-delay Implications?" in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 303–313.

[16] W. Oliveira, R. Oliveira, and F. Castor, "A Study on the Energy Consumption of Android App Development Approaches," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017, pp. 42–52.