Layer Activation Mechanism for Asynchronous Executions in JavaScript

Hiroaki Fukuda
hiroaki@shibaura-it.ac.jp
Shibaura Institute of Technology
Toyosu, Tokyo, Japan

Paul Leger
pleger@ucn.cl
Universidad Católica del Norte
Coquimbo, Chile

Nicolás Cardozo
n.cardozo@uniandes.edu.co
Universidad de los Andes, Bogotá,
Colombia

ABSTRACT

In modern software development with JavaScript, an asynchronous execution model is often adopted to prevent freezing execution triggered by the blocking operations. JavaScript is now used in various types of applications for the Web, smartphones, and server-side due to its rich ecosystem. In such applications, programmers implement several concerns that should perform different behavior according to the current identified context. Context-Oriented Programming (COP) posits *layers* as an abstraction to manage such concerns. With COP, programmers can implement context dependent application behavior in a layer, then (de)activate such layers when the context changes, leading to a change in the system behavior. Additionally, COP offers different scoping strategies which define when and how layers should be (de)activated. The dynamic extent of layers is one of such scoping strategies, which encapsulates the duration of a layer within a block, then deactivates the layer when the block execution ends. However, applying an asynchronous execution model breaks the semantics of dynamic extent because the result of an asynchronous execution generally returns when the caller of the asynchronous execution goes through the block. Existing work proposes a variant of the dynamic extent that activates a layer for a block and its logically-connected asynchronous operations by keeping information across them. However, that proposal only supports one of three kinds of asynchronous operations used in JavaScript (MacroTask, EventTask, and MicroTask). This paper extends on the existing work to support a layer activation mechanism with a scoping strategy that fulfills all three kinds of asynchronous operations in JavaScript. We show the benefit of our proposal through the implementation of a real world application for smartphones.

CCS CONCEPTS

• **Software and its engineering → Software system models**; *Software design engineering*.

KEYWORDS

Context-Oriented Programming, asynchronous execution, JavaScript

ACM Reference Format:
Hiroaki Fukuda, Paul Leger, and Nicolás Cardozo. 2022. Layer Activation Mechanism for Asynchronous Executions in JavaScript. In *COP '22: International Workshop on Context-Oriented Programming and Advanced Modularity, June 06–10, 2022, Berlin, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3570353.3570354

1 INTRODUCTION

Context-awareness is becoming ever more important due to the rising variety of computing devices (*e.g.,* tablets, smartphones), and their execution environment conditions (*e.g.,* location or users' preferences). Context-oriented Programming (COP) [7] allows programmers to change software behavior at run time, in response to an identified context [1]. COP allows programmers to define partial methods [4, 7] to refine the behavior of the base system. These partial methods can be grouped and encapsulated in a *layer* abstraction. Additionally, COP provides mechanisms to activate and deactivate layers when a context is identified. Several activation mechanisms have been propose in COP, which can be imperative [7, 10, 19], event-based [15], or implicit [14, 16, 23, 25]. One of the most frequently used scoping strategies is *dynamic extent* [12]. Dynamic extent limits the scope of activating layers within a block, meaning that programmers do not need to (de)activate layers explicitly, avoiding unexpected behavior due to leaking the active layers to other parts of the system. COP features are usually provided as library extensions of programming languages, as is, for example, the case of ContextJS [20], a COP library for JavaScript.

In JavaScript, an asynchronous execution model is often adopted to prevent freezing the execution of applications, given that this execution model divides a sequence of operations into an asynchronous operations and their corresponding callbacks to get the result. For example, Web applications use asynchronous operations to obtain necessary data without freezing users' interactions with browsers. We note the asynchronous execution model does not directly fit to the semantics of dynamic extent scoping in COP, as the result of asynchronous operations is postponed while the caller thread goes through the block, leading to unexpected behavior. To ensure modularity, the implementation of a concern with a context should be executed inside the concern. However, the combination of COP and an asynchronous execution model breaks this concept. A previous proposal points out this problem and gives a solution for MicroTasks using a concept called Zones [26]. Zones keep context information across logically-connected asynchronous operations.

In this paper we propose a layer activation mechanism with a scoping strategy called *async dynamic extent* for JavaScript (ADEjs), as an extension of existing work [24], that fulfills the three kinds of asynchronous operations in JavaScript, MicroTask, MacroTask, and EventTask.

To position our solution we first offer a background on COP and the different kinds of asynchronous operations in JavaScript in Section 2. Additionally, we describe the motivation for ADEjs. In developing our solution we explore two zone libraries for JavaScript, building an API that observes all asynchronous operations and executes postponed callbacks with the appropriate layers, following the layer activation mechanisms implemented for ContextJS (Section 3). The usefulness of ADEjs is illustrated through an example implementation using multiple kinds of asynchronous operations in combination of the zone libraries (Section 5). Section 6 presents COP approaches related to our proposal, and finally, Section 7 closes the paper with the conclusion and avenues of future work.

Availability. The proposal implementation is accesible at our Github repository: https://github.com/kensan914/context-zone. The examples presented used the revision aa28b4e.

2 COP AND ASYNCHRONOUS OPERATIONS IN JAVASCRIPT

This section presents the COP paradigm and its activation and scoping mechanisms, especially focusing on dynamic extent scoping for layer activation. We then introduce the asynchronous operation types in JavaScript, and point out the problems that may arise when combining COP activation and scoping in asynchronous operations.

2.1 COP in a Nutshell

COP extends systems' base cross-cutting behavior dynamically in a modular manner. Programmers can use partial methods to refine the behavior of the base system, where partial methods are grouped and encapsulated in an abstraction called a *layer*.

```
1  class ButtonGroupManager {
2    show() {
3      //base code
4    }
5  }
6
7  const LandscapeLayer = layer("landscape");
8  LandscapeLayer.refineClass(ButtonGroupManager, {
9    show() {
10     //refine the behavior of show
11   }
12 });
13
14 const buttonGroupManager = new ButtonGroupManager();
15 buttonGroupManager.show();
```

Listing 1: Using ContextJS for the smartphone application

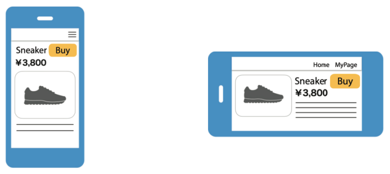


Figure 1: Two different contexts in a smartphone application

Figure 1 shows a smartphone application working in two different contexts (*i.e.,* `Portrait` and `Landscape`). Listing 1 illustrates the ContextJS code for the base behavior, and the partial methods extending the application in a layer. We define a class, named ButtonGroupManager, as the main core module of the application. This class contains the base method show (Lines 1-5). To define context-based behavior, we create the LandscapeLayer layer, using the layer function provided by ContextJS (Line 7). In the layer we refine the ButtonGroupManager behavior by providing an alternative definition of the show method. Finally, we create an instance of the ButtonGroupManager class and invoke the base implementation of show as LandscapeLayer is not active yet (Lines 14-15).

Layer activation mechanisms establish when a layer is activated. There have been three main proposals for activating layers:

(1) *imperative* mechanisms activate layers explicitly using constructs such as with or activate,
(2) *implicit* mechanisms activate layers whenever a specified condition is satisfied, and
(3) *event-based* mechanisms use event matching triggers.

```
1 withLayers([LandscapeLayer], function() {
2   buttonGroupManager.show(); //refine
3 });
4 buttonGroupManager.show();    //base code
```

Listing 2: Dynamic extent in layer activations

Moreover, layers have effect within a given scope when activated. In the dynamic extent scoping mechanism, refined behavior is available throughout the extend of the with context activation blocks. ContextJS provides the withLayers function that receives two arguments, an array of layers to be activated, and a function that is invoked with these layers active. For example, Listing 2 shows that LandscapeLayer has effect over the behavior called in Line 2, executing the refinement of the show method as LandscapeLayer becomes active in the withLayers function. In contrast, the call to show in Line 4 corresponds to the base code because the withLayers block is no longer in scope, deactivating the layer LandscapeLayer. Using dynamic extent, programmers can activate layers only within a block's scope, avoiding leaking behavior adaptations throughout other parts of the program.

2.2 Asynchronous Operations in JavaScript

In modern JavaScript system development, programmers often use asynchronous operations. Suppose we use synchronous operations for the development of user interfaces with JavaScript. When we invoke a synchronous operation to send a message to a Web server, the Web browser stops the execution, including rendering the user interface until the response is received because JavaScript does not support multi-threading. To prevent such situations, programmers use asynchronous operations with current JavaScript libraries that enable programmers to write asynchronous operations such as Promise [21], async/await, or Sync/cc [17, 18]. In this paper, we call operations executed in callbacks as *async tasks*, which includes event handlers such as functions that handle click events dispatched from a button component. Async tasks in JavaScript can be divided into three categories: *MicroTask*, *EventTask*, and *MacroTask* [?]. We briefly explain these categories.

MicroTask. MicroTasks are executed only once, after the corresponding asynchronous operation is invoked, and cannot be canceled. In the following code snippet, an anonymous function passed to the function onreadystatechange corresponds to a MicroTask, which will be invoked when the corresponding asynchronous operation finishes (*e.g.,* send finishes).

```
let req = new XMLHttpRequest();
req.open(GET, "https://aaa.com/", true);
req.onreadystatechange = function(e) {
  //handle asynchronous callback
}
req.send();
```

MacroTask. MacroTasks may be executed more than once and can be canceled. These tasks delay time is decided when it is scheduled. In the following, an anonymous function passed to setTimeout as a first argument corresponds to the MacroTask. This task is invoked every 100 milliseconds and can be canceled.

```
setTimeout(function() {
  //handle timeout events
  console.log(count);
}, 100);
```

EventTask. EventTasks are executed whenever an event happens, such as a button click. These events occur at arbitrary times, so they cannot be foreseen. In the following, an anonymous function is passed to addEventListener, which corresponds to an EventTask. This task can be removed by invoking the opposite method such as removeEventListener.

```
button.addEventListener("click",function () {
  //handle event
});
```

2.3 COP Problems with Asynchronous Tasks

To illustrate the semantics mismatch between asynchronous execution models and dynamic extent, we use COP to extend the behavior of an EventTask for a Downloader as shown in Listing 3. The Downloader class provides a download method that downloads data from a specified URL. The Downloader also tries to download the data three times whenever an error in the download occurs (*e.g.,* it cannot download the data for unstable network conditions). The downloader generates a dlcomplete event when the download is finally complete. We can delegate complicated download issues in this class.

Suppose a given URL requires authentication, which is commonly achieved by adding the saved cookie to the URL. Listing 3 creates the authLayer and refines the download method to add cookies (Line 4). In Listing 3, we need to get a set of confidential information (*e.g.,* name and age) and display them sequentially for a rich user interface using a single Downloader object for saving resources. Thereby, we apply authLayer with the withLayers function (Line 10). Inside the function applied as a second argument to the withLayers function, we access two different URLs reusing the downloader object; therefore we expect that we can access both with the authentication layer.

```
1  const authLayer = layer("authLayer");
2  authLayer.refineClass(DownLoader, {
3    download(url) {
4      addAuthenticateCookie();
5      return proceed();
6    }
7  });
8
9  downloader = new Downloader();
10 withLayers([authLayer], function() {
11   downloader.addEventListener("dlcomplete", function(e) {
12     downloader.removeEventListener("dlcomplete");
13     //show name
14     downloader.addEventListener("dlcomplete", function(e) {
15       downloader.removeEventListener("dlcomplete");
16         //show age
17     });
18     downloader.download("https://yourinfo.com/?age");
19   }
20   downloader.download("https://yourinfo.com/?name");
21   });
```

Listing 3: A problem in COP with asynchronous programming

In Listing 3, the second access (https://yourinfo.com?age) is accessed without the authentication due to the semantics of dynamic extent adopted by withLayers. Even though scheduling the second URL is written in the block scope of withLayer, it will be done outside the block due to the asynchronous behavior. The proposal presented by Ramson et al. [24] solved this problem for MicroTask, however, the problem of semantics mismatch for MacroTask and EventTask still remains which is a motivation of our ADEjs.

3 CONTEXTJS AND ZONES

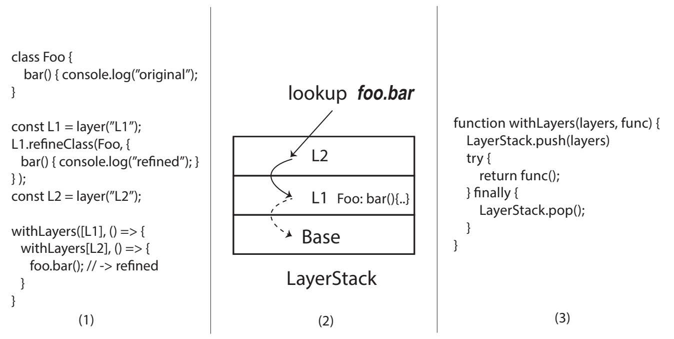
We propose a solution to solve the semantics mismatch between asynchronous execution model and dynamic extent for the three kinds of async tasks in JavaScript: MicroTask, MacroTask, and EventTask. We extend ContextJS and introduce Zones [26] following the proposal of existing work [24]. Thereby this section firstly describes how ContextJS dynamically (de)activates layers, then, we briefly explain the concept of Zones.

3.1 Layer Activation Mechanism

ContextJS uses a stack, called *LayerStack*, to manage the (de)activation of layers. Figure 2 shows:

(1) code snippets nested with withLayers in ContextJS,
(2) the LayerStack and the way the method lookup in ContextJS works, and
(3) the essential implementation of the withLayers function respectively.

In Figure 2-(1), the Foo class contains a bar method, and two layers, L1 and L2. We refine the Foo.bar method with refineClass in L1. Additionally, we use nested withLayers functions (applying L1 first, and then L2), to invoke the Foo.bar method, showing the "refined" message defined in L1.



```
class Foo {
  bar() { console.log("original");
}

const L1 = layer("L1");
L1.refineClass(Foo, {
  bar() { console.log("refined"); }
} );
const L2 = layer("L2");

withLayers([L1], () => {
  withLayers[L2], () => {
    foo.bar(); // -> refined
  }
}
```

(1)

```
function withLayers(layers, func) {
  LayerStack.push(layers)
  try {
    return func();
  } finally {
    LayerStack.pop();
  }
}
```

(3)

Figure 2: Layer activation in ContextJS

In ContextJS, as shown in Figure 2-(2), a base layer is always pushed on to the LayerStack containing all base classes and methods. A layer L is pushed on to the LayerStack when the withLayer function is invoked with this L as a parameter. In our example, L1 and L2 are pushed following their definition of the nested withLayer, as shown in Figure 2-(2). When we invoke a method (*e.g.,* Foo.bar), ContextJS starts looking up the method from the top layer on the LayerStack. In this example, ContextJS first looks up the bar method in L2. Since L2 does not contain a definition for bar, the lookup request is forwarded to the next layer (*i.e.,* L1), where it is defined. Note that if no extensions are defined, ContextJS reverts to the base definitions in the base layer. The order in which layers are pushed into the stack follows the function in Figure 2-(3). Furthermore, as we can see, after the execution of the func, we pop the layer, reason why the execution of callbacks in asynchronous operations are invoked without the required layer.

3.2 Zones

Zones are originally provided by the Dart language to keep information across logically-connected asynchronous operations. Additionally, zones control asynchronous behavior by observing and intercepting the execution of async tasks.

```
1 runZoned(() {
2   print(Zone.current[#key]); //--> 499
3   Zone.current[#name] = Mike;
4   button.addEventListener("click",function() {
5     console.log(Zone.current[#name]); //--> Mike
6   });
7 }, zoneValues: { #key: 499 });
```

Listing 4: Zones keep information across logically-connected operations in Dart

Listing 4 shows how to keep information into logically-connected asynchronous operations in a zone; a block containing any operations as a second argument of runZoned. Moreover, properties attached to a zone persist regardless of whether the invoked behavior is synchronous or asynchronous. As shown in Listing 4, runZoned creates a scope in which we can access the corresponding properties from both synchronous and asynchronous operations using Zone.current. For example, we set a value "Mike" with key "#name" at Line 3, then at Line 5 we get the value "Mike" inside a callback. Zones also expose a variety of life-cycle callbacks for example when entering a zone, leaving a zone or invoking an asynchronous operation and a callback.

4 ADEJS: ASYNC DYNAMIC EXTENT FOR JAVASCRIPT

This section presents ADEjs, our proposed solution to support all kinds of asynchronous operations used in JavaScript. We start presenting the crucial point of our proposal which is the storing and restoring of layers, then describe two Zone libraries used to create ADEjs. Finally, we present a concrete ADEjs implementation.

4.1 Store and Restore Layers

Based on the layer activation mechanism in ContextJS (Section 3.1), the essential problem to solve in using dynamic extent scoping and asynchronous operations is the way layers are pushed/popped on/from to the LayerStack. For instance, using Listing 3, when the first download is invoked for https://yourinfo.com/?name, authLayer has been pushed on to the LayerStack (Line 20). When the second download method is invoked, inside the callback in Line 18, the authLayer is no longer on the LayerStack because of the asynchronous behavior. Figure 3 shows ADEjs' approach for the adequate push/pop behavior for layers related to asynchronous operations to (re)store layers on the LayerStack in Figure 2. In Figure 3, we describe an asynchronous operation as *async schedule*, and its corresponding callback as *async callback*. When a programmer invokes an asynchronous operation, ADEjs keeps all layers on the stack, except the Base. For example, Figure 3-(a) shows L1 and L2 in the stored stack. When the corresponding callback is being invoked, ADEjs firstly pops and stores the currently existing L3 on the LayerStack. Then ADEjs temporary restores the stored L1 and L2 on the LayerStack and stores L3. After handling the callback, ADEjs finally restores the stored L3 on the LayerStack (Figure 3-(c)).

To carry out the layer storing/restoring strategy, ADEjs needs to detect invocations of asynchronous operations and their corresponding callbacks without explicit programmers' support, leading to the use of the Zone libraries.

4.2 Zone libraries

In JavaScript, we can find several libraries that provide Zones' functionality. This section reviews the two libraries used to build ADEjs.

Zone.js is part of the Angular framework and supports the three kinds of async tasks. This library allows programmers to expose the life-cycle of callbacks such as invoking an asynchronous operation and its callback. Using life-cycles, programmers can modify the behavior of a system on the fly.

Dexie.Promise is a wrapper for *IndexedDB*[1] and *Dexie.Promise*[2] to include a zone-like utility. Dexie allows programmers to write database transactions as a sequence of asynchronous operations. Zones are used to keep track of transaction scopes. Although *Dexie.Promise* only supports MicroTask, it

[1]https://www.w3.org/TR/IndexedDB/#idl-def-IDBEnvironment, accessed on Mar. 31th, 2022

[2]https://dexie.org/docs/Promise/Promise.PSD, accessed on Mar. 31th, 2022

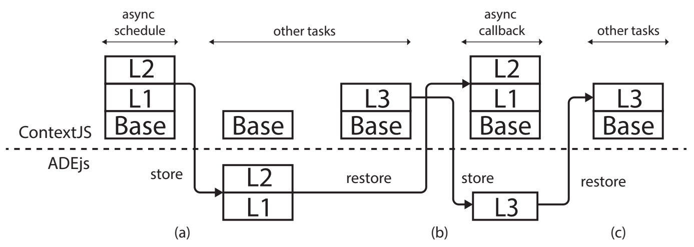


Figure 3: Storing and restoring layers before callbacks

can detect native async/await expressions. However, different from *Zone.js*, it does not expose life-cycle of callbacks.

The reason to combine these libraries is twofold. On the one hand, even though *Zone.js* claims that it supports all kinds of asynchronous operations, it cannot detect the asynchronous operations using async/await, which is categorized as a MicroTask. This means that programmers cannot modify the behavior of a system that uses async/await. On the other hand, *Dexie.Promise* only supports MicroTask, meaning that this library cannot be used for MacroTasks and EventTasks. Thereby we decide to use both libraries to fully support the three kinds of JavaScript asynchronous operations.

4.3 Detecting Asynchronous Operation and Callbacks

To detect the three kinds of asynchronous operations implicitly, we use *Zone.js* and *Dexie.Promise* simultaneously. The following sections explain the process of detecting invocations of asynchronous operations and their corresponding callbacks in detail.

4.3.1 MicroTask. To detect callbacks that are invoked for MicroTask, including native async/await, we provide a CustomPromise object that is an extension of a Dexie.Promise Promise object. This is because JavaScript provides async/await as syntactic sugar on top of Promises.

```
1  const then = function(onFullfilled) {
2    //temporary pop current layers and restore stored layers
3    restoreLayers(...);
4    try {
5      return onFulfilled(this, arguments);
6    } finally {
7      //pop layers and restore current layers
8      removeAndRestoreLayers(...);
9    }
10 }
```

Listing 5: then method in CustomPromise

Listing 5 shows the then method, refined using the CustomPromise object, which is an extension of a promise. The then method is invoked when an asynchronous operation is completed, which means that refining the then method can change the behavior of a system. In Listing 5, onFullfilled represents a callback, therefore we restore layers before invoking the callback (Line 3). We then restore existing layers after invoking the callback (Line 8), which is the same as the implementation using *Zone.js*. Finally, to apply this CustomPromise to the JavaScript virtual machine, we replace the Promise in a native window to this CustomPromise.

4.3.2 MacroTask/EventTask. We use *Zone.js* to detect asynchronous operations and corresponding callbacks for MacroTask and EventTask. We show pseudo-code of this implementation using *Zone.js* in Listing 6. The Zone.current function returns the reference of the current zone running on the JavaScript virtual machine, and this zone has a fork method to create a new zone. This fork method receives an object that follows a ZoneSpec interface in which callback functions such as onFork, onScheduleTask, and onInvokeTask. We use the following two callback functions:

onScheduleTask: This function is invoked just before an async task is scheduled (*i.e.,* before invoking an asynchronous operation), so we can use this function to store existing layers on the LayerStack.

onInvokeTask: This function is invoked just before a callback is invoked. We use this function to restore the layers that are used when the corresponding asynchronous operation is invoked.

Zone objects provide a run method that receives a function to execute operations such as loops, branches and invoking methods in the zone.

```
1  const zone = Zone.currrent.fork({
2    onScheduleTask: function(...) {
3      storeLayer(); //store layers
4    },
5    onInvokeTask: function(...) {
6      //temporary pop current layers and restore stored layers
7      restoreLayers(...);
8      //execute a callback
9      invokeTask(...);
10     //pop layers and restore current layers
11     removeAndRestoreLayers(...);
12     },
13   });
14
15 zone.run(function() {
16   //execute any operations
17 });
```

Listing 6: Store/Restore layers with Zone.js

Using *Zone.js*, we show the intuitive implementation of our proposal in Listing 6. In this implementation, we create a new zone with the fork method passing an object that implements the required ZoneSpec interface (Line1). In the onScheduleTask function, we store all layers on LayerStack (Line 3). In the onInvokeTask method, which is invoked before executing callbacks, we first restore the layers stored in onScheduleTask after removing and storing the currently existing layers on the LayerStack (Line 7), as for example L3 in Figure 3-(b). Second, we execute the callback invoking the invokeTask method (Line 9). Finally, we remove stored layers (*e.g.,* L1 and L2 in Figure 3) from LayerStack and restore the existing layers (L3) (Line 11) to it. Note that, onInvokeTask will not be invoked when we use native async/await expressions, which explains why we cannot use *Zone.js* for MicroTasks.

4.3.3 *Combine libraries.*

Using these two zone libraries and ContextJS, we show an extract of the implementation for our new layer activation abstraction, withLayersZone. The withLayersZone function consists of nested functions in which we store layers on the LayerStack (Line 2 in Listing 7) before invoking the withLayers function of ContextJS. Then, we apply aforementioned functions using *Zone.js* and CustomPromise (Line 3-5). Finally, we invoke the function that should be executed within a layer (Line 6).

```
1  const withLayersZone = function(layers, func) {
2    storeLayers();                  //store layers on LayerStack
3    withLayers(layers, function() {    //provided by ContextJS
4      macroEventZone.run(function() { //using Zone.js
5        microTaskZone(function() {     //using CustomPromise
6          func.call();
7        });
8      });
9    });
10 }
```

Listing 7: Implementation of withLayersZone

Note that, we modify the implementation of Listing 6 at two points to prevent multiple stores of layers and invoking callbacks for MicroTask due to the combination of *Zone.js* and CustomPromise:

(1) we omit storeLayer in Line 2 of Listing 6 because withLayersZone stores layers already in Line 2 of Listing 7, and
(2) we do not execute Lines 7 trough 11 of Listing 6 when tasks belong to MicroTask because we delegate this case to the CustomPromise in microTaskZone (Line 5 of Listing 7) as shown in the following code snippets.

```
onInvokeTask: function(..., task,...) {
  if (task.type != "microTask") {...}
}
```

With our proposed withLayersZone, we can use all operation types (*i.e.,* synchronous or asynchronous) with active layer blocks, as shown in Listing 8.

```
withLayersZone([authlayer],function() {
  //write synchronous and asynchronous operations
  ...
});
```

Listing 8: Usage of withLayersZone

4.3.4 *Summary.*

In the implementation of ADEjs, we use Zones to keep layers across logically-connected asynchronous operations and their corresponding callbacks, which are implicitly detected by using zone libraries. A zone is created when an asynchronous operation is invoked at runtime, storing all information related to layers and callbacks, which leads to the correct execution of context and base behavior under all possible situations.

5 VALIDATION

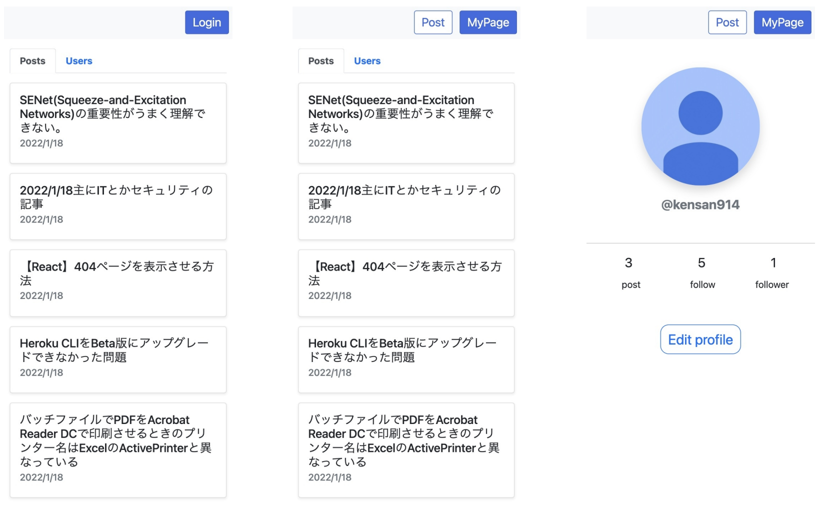
To check the feasibility of ADEjs and its implementation, we develop a smartphone application called *QiitaClient* that uses the APIs provided by Qiita [13]. Qiita is a web service where IT engineers can share useful knowledge, reusable code, and connect with each other. For example, engineers publishing articles about how to use new features of ECMA 6 [9]. Figure 4 shows three screenshots of the QiitaClient application to manage the user login status under different contexts. Figure 4a shows when the user is not yet log in. Figure 4b shows when the user has logged in. Both screens show abstracts of articles as a list. Whereas Figure 4a shows a "login" button, Figure 4b shows "post" and "MyPage" buttons on the top respectively. When we click the "MyPage" button, the application shows the user information (Figure 4c). Listing 9 shows essential pieces of QiitaClient and an authentication layer that refines the authenticated behavior. Listing 10 presents the use of ADEjs, the withLayersZone, in which EventTasks and MicroTasks are used within the authentication layer.

Listing 9 shows the QiitaClient with two methods generateHeader and request. The former method synchronously generates HTTP headers that are required by Qiita APIs, the latter method asynchronously sends a request to the given URL with a GET method. In this listing, the request invokes generateHeader (Line 6). We refine the generateHeader of the QiitaClient in which we add an access token to the HTTP header in order to be authenticated (Line 13).

```
1  class QiitaApiClient {
2    generateHeader() {return {"Content-Type":"application/json"}}
3    getName() { return "Guest";}
4    request(url) { return fetch(url,{
5        method: "GET",
6        headers: self.generateHeader(), //invoke generateHeader
7      });}
8  }
9  const authLayer = layer("authLayer");
10 authLayer.refineClass(QiitaApiClient, {
11   generateHeader() { return {
12       ...proceed(),
13       Authorization:`Bearer ${ACCESS_TOKEN}`, //access token
14     };},
15   getName() { return window.localStrage.getItem("name");}
16 });
```

Listing 9: Definition of QiitaClient

When a user has been already authenticated, ADEjs activates authLayer using withLayersZone, then create the "MyPage" button and adds an event listener, scheduling an EventTask. Additionally, we schedule a MicroTask invoking request, which returns a CustomPromise. Then we invoke then with an anonymous function as a callback (Line 9-10 of Listing 10). This anonymous function is



SENet(Squeeze-and-Excitation Networks)の重要性がうまく理解できない。
2022/1/18

2022/1/18主にITとかセキュリティの記事
2022/1/18

【React】404ページを表示させる方法
2022/1/18

Heroku CLIをBeta版にアップグレードできなかった問題
2022/1/18

バッチファイルでPDFをAcrobat Reader DCで印刷させるときのプリンター名はExcelのActivePrinterと異なっている
2022/1/18

SENet(Squeeze-and-Excitation Networks)の重要性がうまく理解できない。
2022/1/18

2022/1/18主にITとかセキュリティの記事
2022/1/18

【React】404ページを表示させる方法
2022/1/18

Heroku CLIをBeta版にアップグレードできなかった問題
2022/1/18

バッチファイルでPDFをAcrobat Reader DCで印刷させるときのプリンター名はExcelのActivePrinterと異なっている
2022/1/18

(a) Before login **(b) After login** **(c) MyPage**

Figure 4: Screenshots of QiitaClient with different contexts

invoked asynchronously when the MicroTask is completed. Using withLayersZone, we can correctly show the personal information when a user clicks the "MyPage" button. If we use the basic use withLayers function in ContextJS, it is not possible to observe the user name correctly ("Guest" is shown instead) and obtain the personal information, as the authentication information in the layer is not available.

```
if (isAuthenticated) {
  const mypageButton = createAndaddButton("MyPage");
  withLayersZone([authLayer],function() {
    //schedule EventTask
    mypageButton.addEventListener("click",function() {
    const qc = new QiitaClient();
      //schedule MicroTask
      qc.request("/api/v2/authenticated_user/")
        .then(function(profile) {
          //show personal information
          renderMypage(qc.getName(), profile);
        });
    });
  });
} else { .... }
```

Listing 10: Usage of ADEjs using withLayersZone

6 RELATED WORK

The work of ADEjs is related to two areas. First, the combination of COP and event-based programming, considered as different types of asynchronous models. Second, the management of thread-local layer (de)activation that encapsulates thread-related information with the thread similar to Zones.

The combination of an asynchronous model and COP is discussed in ECaesarJ [22] and JCop [6] as event-based programming. In these, event handlers are invoked when corresponding events happen that are generally independent of the main control flow, considered as another asynchronous model. ECaesarJ supports the definition of context as a class implementing context entry and exit functions. ECaesarJ also allows programmers to define compositions of contexts and event handlers that may handle context specific operations. However, ECaesarJ does not provide layer style compositions of partial methods that will dynamically change the behavior of a system. JCop provides layers and their compositions in addition to definitions and compositions of contexts for event-based layer activation. However, JCop does not consider the connection between scheduling asynchronous operations and their callbacks. Therefore, programmers must explicitly (de)activate layers if they want to use them in scheduling async taskss and their callbacks.

Thread-local layer (de)activation is a feature supported by various COP languages [3] such as ContextS [11], ContextJ [5], and Subjective-C [8]. The thread-local concept influenced the design of Zones [24], which is a predecessor of our research. This work first considers the connection between scheduling an asynchronous operation and its callback to maintain the consistency of (de)activating layers across logically-connected asynchronous operations using Zones. On top of this, they extend Dexie.Promise to provide life-cycle callbacks, and use them to support MicroTask that uses async/await. Due to using *Dexie.Promise*, it does not support all types of asynchronous operation, namely MacroTask and EventTask. Our approach also maintains the consistency of layers across all types of logically-connected asynchronous operations combining *Zone.js* and our own extension of *CustomPromises* to support all kinds of async tasks.

7 CONCLUSION AND FUTURE WORK

In response to the appearance of small and mobile computing devices and systems that dynamically adapt their behavior based on the surrounding execution environment (*e.g.,* users' location), COP provides partial method extensions associated to layers and layer activation to manage behavior adaptations to the context. Dynamic extent is one of the most used scoping strategies for behavior adaptations which limits the extend of a layer to its activation block, deactivating the layer at the end of the block. The semantics of dynamic extent implicitly assumes that all operations are executed synchronously. This assumption is not guaranteed when we use asynchronous operations that are commonly used for Web development using JavaScript. ContextJS is a library that supplies COP features including dynamic extent using withLayer. Therefore the execution model of ContextJS does not guarantee the correct behavior when we combine synchronous and asynchronous operations with layers.

This paper proposes ADEjs, a layer activation mechanism with scoping strategies that fulfill the three kinds of asynchronous operations in JavaScript using Zones. More precisely, we use *Zone.js* and *CustomPromise* to support MicroTasks, MacroTasks, and EventTasks. ADEjs provides withLayersZone abstraction that ensures all active layers in an asynchronous call throughout the execution of its corresponding callback. To validate the feasibility and use of ADEjs, we implement a smartphone application, QiitaClient, using withLayersZone.

With ADEjs, programmers can keep layers activate not only for synchronous operations within a code block but also logically-connected asynchronous operations and their callbacks without any manual management of the kinds of operations used. To compliment this, as future work we will evaluate existing issues related to the scope of layer and compare ADEjs to different kinds of activation mechanisms auch as implicit layer activation.

Even though we can always keep layers through the execution of asynchronous calls, it might not be always necessary. For example, when we add an event handler to a button with a layer (*e.g.,* landscape), the event handler should be invoked along with the current environment (*e.g.,* portrait). This requirement cannot be solved with ADEjs because the withLayersZone always applies the same layers that are used when the asynchronous operations are invoked to execute callbacks. We need to expose adequate mechanisms for programmers to change the semantics of withLayersZone to adjust to their requirements.

REFERENCES

[1] Unai Alegre, Juan Carlos Augusto, and Tony Clark. 2016. Engineering context-aware systems and applications: A survey. *Journal of Systems and Software* 117 (2016), 55–83. https://doi.org/10.1016/j.jss.2016.02.010

[2]]tasks Angular. [n. d.]. Task lifecycle. https://github.com/angular/zone.js/blob/master/doc/task.md. Accessed: 2022-05-22.

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. 2009. A Comparison of Context-Oriented Programming Languages. In *International Workshop on Context-Oriented Programming* (Genova, Italy) *(COP '09)*. Association for Computing Machinery, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/1562112.1562118

[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. 2011. ContextJ: Context-oriented Programming with Java. *Information and Media Technologies* 6, 2 (2011), 399–419. https://doi.org/10.11185/imt.6.399

[5] Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. 2009. Improving the Development of Context-Dependent Java Applications with ContextJ. In *International Workshop on Context-Oriented Programming* (Genova, Italy) *(COP '09)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/1562112.1562117

[6] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. 2010. Event-Specific Software Composition in Context-Oriented Programming. In *Software Composition*, Benoît Baudry and Eric Wohlstadter (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–65.

[7] Robert H. Pascal C. and Oscar N. 2008. Implementing protocols via declarative event patterns. *Journal of Object Technology* 7, 3 (2008), 125–151.

[8] Nicolás Cardozo, Sebastián González, Kim Mens, and Theo D'Hondt. 2012. Uniting Global and Local Context Behavior with Context Petri Nets. In *International Workshop on Context-Oriented Programming (COP'12, 3)*. ACM, New York, NY, USA, 1 – 6.

[9] ECMA. 2016. ECMAScript 6: A scripting-language specification for JavaScript - https://www.ecma-international.org/ecma-262/6.0. https://www.ecma-international.org/ecma-262/6.0 Accessed: 2020-01-20.

[10] Sebastián González, Nicolas Cardozo, Kim Mens, Alfredo Cádiz, Jean-Cristophe Libbrecht, and Julien Goffaux. 2010. Subjective-C: Bringing Context to Mobile Platform Programming. In *Proceedings of the International Conference on Proceedings of the International Conference on Software Language Engineering* (Eindhoven, The Netherlands) *(series-lncs, Vol. 6563)*, Brian Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer, 246 – 265.

[11] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. 2007. An Introduction to Context-Oriented Programming with ContextS, Vol. 5235. 396–407. https://doi.org/10.1007/978-3-540-88643-3_9

[12] Robert Hirschfeld, Hidehiko Masuhara, Atsushi Igarashi, and Tim Felgentreff. 2016. Visibility of Context-oriented Behavior and State in L. *Information and Media Technologies* 11 (2016), 11–20. https://doi.org/10.11185/imt.11.11

[13] Qiita Inc. 2014. How developers code is here. https://qiita.com. Accessed: 2022-03-23.

[14] Tetsuo Kamina and Tomoyuki Aotani. 2019. TinyCORP: A Calculus for Context-Oriented Reactive Programming. In *Proceedings of the Workshop on Context-Oriented Programming* (London, United Kingdom) *(COP '19)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3340671.3343356

[15] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2011. EventCJ: A Context-Oriented Programming Language with Declarative Event-Based Context Transition. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development* (Porto de Galinhas, Brazil) *(AOSD '11)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/1960275.1960305

[16] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2017. Push-Based Reactive Layer Activation in Context-Oriented Programming. In *Proceedings of the 9th International Workshop on Context-Oriented Programming* (Barcelona, Spain) *(COP '17)*. Association for Computing Machinery, New York, NY, USA, 17–21. https://doi.org/10.1145/3117802.3117805

[17] Paul Leger and Hiroaki Fukuda. 2017. Sync/CC: Continuations and Aspects to Tame Callback Dependencies on JavaScript Handlers. In *Proceedings of the 32nd Annual ACM Symposium on Applied Computing (SAC 2017)*. Marrakech, Morocco, 1245–1250. https://doi.org/10.1145/3019612.3019783

[18] Paul Leger, Hiroaki Fukuda, and Ismael Figueroa. 2021. Continuations and Aspects to Tame Callback Hell on the Web. *Journal of Universal Computer Science* 27, 9 (Sept. 2021).

[19] Paul Leger, Hidehiko Masuhara, and Ismael Figueroa. 2020. Interfaces for Modular Reasoning in Context-Oriented Programming. In *Proceedings of the 12th International Workshop on Context-Oriented Programming and Advanced Modularity (COP 20)*. Virtual Event, USA, 1–7. https://doi.org/10.1145/3422584.3423152

[20] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2011. An open implementation for context-oriented layer composition in ContextJS. *Sci. Comput. Program.* 76 (12 2011), 1194–1209. https://doi.org/10.1016/j.scico.2010.11.013

[21] Forbes Lindesay. 2012. Promise: A library for promises in JavaScript. https://www.promisejs.org. Accessed: 2022-01-20.

[22] Angel Núñez, Jacques Noyé, and Vaidas Gasiūnas. 2009. Declarative Definition of Contexts with Polymorphic Events. In *International Workshop on Context-Oriented Programming* (Genova, Italy) *(COP '09)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/1562112.1562114

[23] Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2017. The Declarative Nature of Implicit Layer Activation. In *Proceedings of the 9th International Workshop on Context-Oriented Programming* (Barcelona, Spain) *(COP '17)*. Association for Computing Machinery, New York, NY, USA, 7–16. https://doi.org/10.1145/3117802.3117804

[24] Stefan Ramson, Jens Lincke, Harumi Watanabe, and Robert Hirschfeld. 2020. Zone-Based Layer Activation: Context-Specific Behavior Adaptations across Logically-Connected Asynchronous Operations. In *Proceedings of the 12th International Workshop on Context-Oriented Programming and Advanced Modularity* (Virtual, USA) *(COP '20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/3422584.3422764

[25] Takuo Watanabe. 2018. A Simple Context-Oriented Programming Extension to an FRP Language for Small-Scale Embedded Systems. In *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition* (Amsterdam, Netherlands) *(COP '18)*. 23–30. https://doi.org/10.1145/3242921.3242925

[26] Zones. 2014. Asynchronous dynamic extents. https://dart.dev/articles/archive/zones. Accessed: 2022-01-03.