WebWorkers in Angular: Documentation

Angular includes native support for writing applications which live in a WebWorker. This document describes how to write applications that take advantage of this feature. It also provides a detailed description of the underlying messaging infrastructure that angular uses to communicate between the main process and the worker. This infrastructure can be modified by an application developer to enable driving an Angular application from an iFrame, different window / tab, server, etc..

Introduction

WebWorker support in Angular is designed to make it easy to leverage parallelization in your web application. When you choose to run your application in a WebWorker angular runs both your application's logic and the majority of the core angular framework in a WebWorker. By offloading as much code as possible to the WebWorker we keep the UI thread free to handle events, manipulate the DOM, and run animations. This provides a better framerate and UX for applications.

Bootstrapping a WebWorker Application

Bootstrapping a WebWorker application is not much different than bootstrapping a normal application. The main difference is that you need to do the bootstrap process on both the worker and render thread. Unlike in a standard Angular application you don't bootstrap your main component on the render thread. Instead you initialize a new application injector with the WORKER_APP_PLATFORM providers and provide the name of your WebWorker script. See the example below for details:

Example

To bootstrap Hello World in a WebWorker we do the following in TypeScript

```
<html>
  <head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.33.3/es6-shim.js"></scri
    <script src="https://jspm.io/system@0.16.js"></script>
    <script src="angular2/web_worker/ui.js"></script>
  </head>
  <body>
    <hello-world></hello-world>
    <script>System.import("index")</script>
  </body>
</html>
```

```
// index.js
```

```
import {WORKER_RENDER_PLATFORM, WORKER_RENDER_APPLICATION, WORKER_SCRIPT} from "angular2/pla
import {platform} from "angular2/core";

platform([WORKER_RENDER_PLATFORM])
.application([WORKER_RENDER_APPLICATION, {provide: WORKER_SCRIPT, useValue: "loader.js"};

// loader.js
importScripts("https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.33.3/es6-shim.js", "https:/
System.import("app");

// app.ts
import {Component, View, platform} from "angular2/core";
import {WORKER_APP_PLATFORM, WORKER_APP_APPLICATION} from "angular2/platform/worker_app";

@Component({
  selector: "hello-world"
})
@View({
  template: "<h1>Hello {{name}}</h1>"
})
export class HelloWorld {
  name: string = "Jane";
}

platform([WORKER_APP_PLATFORM])
.application([WORKER_APP_APPLICATION])
.then((ref) => ref.bootstrap(RootComponent));
```

There's a few important things to note here: * The UI loads angular from the file `angular2/web_worker/ui.js` and the Worker loads angular from `angular2/web_worker/worker.js`. These bundles are created specifically for using WebWorkers and should be used instead of the normal angular2.js file. Both files contain subsets of the angular2 codebase that is designed to run specifically on the UI or Worker. Additionally, they contain the core messaging infrastructure used to communicate between the Worker and the UI. This messaging code is not in the standard angular2.js file. * We pass `loader.js` to our application injector using the WORKER_SCRIPT symbol. This tells angular that our WebWorkers's init script is located at `loader.js`. You can think of `loader.js` as the index.html file for the WebWorker. Since WebWorkers share no memory with the UI we need to reload the Angular dependencies before bootstrapping our application. We do this with `importScripts`. Additionally, we need to do this in a different file than `app.ts` because our module loader (System.js in this example) has not been loaded yet, and `app.ts` will be compiled with a `System.define` call at the top. * The HelloWorld Component looks exactly like a normal Angular HelloWorld Component! The goal of WebWorker support was to allow as much of Angular to live in the worker as possible. As such, *most* Angular components can be bootstrapped in a WebWorker with minimal to no changes required.

For reference, here's the same HelloWorld example in Dart.

```
<html>
  <body>
    <script type="application/dart" src="index.dart"></script>
    <script src="packages/browser.dart.js"></script>
  </body>
</html>
```

```
// index.dart
import "angular2/core.dart";
import "angular2/platform/worker_render.dart";

main() {
  platform([WORKER_RENDER_PLATFORM])
  .asyncApplication(initIsolate("my_worker.dart"));
}
```

```
// background_index.dart
import "angular2/core.dart";
import "angular2/platform/worker_app.dart";
import "package:angular2/src/core/reflection/reflection.dart";
import "package:angular2/src/core/reflection/reflection_capabilities.dart";

@Component(
  selector: "hello-world"
)
@View(
  template: "<h1>Hello {{name}}</h1>"
)
class HelloWorld {
  String name = "Jane";
}

main(List<String> args, SendPort replyTo) {
  reflector.reflectionCapabilities = new ReflectionCapabilities();
  platform([WORKER_APP_PLATFORM, {provide: RENDER_SEND_PORT, useValue: replyTo}])
  .application([WORKER_APP_APPLICATION])
  .bootstrap(RootComponent);
}
```

This code is nearly the same as the TypeScript version with just a couple key differences: * We don't have a `loader.js` file. Dart applications don't need this file because you don't need a module loader. * We provide a `SendPort` through DI using the token `RENDER_SEND_PORT`. Dart applications use the Isolate API, which communicates via Dart's Port abstraction. When you call `setupIsolate` from the UI thread, angular starts a new Isolate to run your application logic. When Dart starts a new Isolate it passes a `SendPort` to that Isolate so that it can communicate with the Isolate that spawned it. You need to provide this `SendPort` through DI so that Angular can communicate with the UI. * You need to set up `ReflectionCapabilities` on both the UI and Worker. Just like writing non-concurrent Angular2 Dart applications you need to set up the reflector. You should not use Reflection in production, but should use the Angular transformer to remove it in your final JS code. Note there's currently a bug with running the transformer on your UI code (#3971). You can (and should) pass the file where you call `bootstrap` as an entry point to the transformer, but you should not pass your UI index file to the transformer until that bug is fixed.

Writing WebWorker Compatible Components

You can do almost everything in a WebWorker component that you can do in a typical Angular Component. The main exception is that there is **no** DOM access from a WebWorker component. In Dart this means you can't import anything from `dart:html` and in JavaScript it means you can't use `document` or `window`. Instead you should use data bindings and if needed you can inject `Renderer2` along with your component's `ElementRef` directly into your component and use methods such as `setProperty`, `setAttribute`, `addClass`, `setStyle`, and `setValue`. Note that you **cannot** call `getNativeElementSync`. Doing so will always return `null` when running in a WebWorker. If you need DOM access see Running Code on the UI.

WebWorker Design Overview

When running your application in a WebWorker, the majority of the angular core along with your application logic runs on the worker. The two main components that run on the UI are the `Renderer2` and the `RenderCompiler`. When running angular in a WebWorker the bindings for these two components are replaced by the `WebWorkerRenderer` and the `WebWorkerRenderCompiler`. When these components are used at runtime, they pass messages through the MessageBroker instructing the UI to run the actual method and return the result. The MessageBroker abstraction allows either side of the WebWorker boundary to schedule code to run on the opposite side and receive the result. You can use the MessageBroker Additionally, the MessageBroker sits on top of the MessageBus. MessageBus is a low level abstraction that provides a language agnostic API for communicating with angular components across any runtime boundary such as `WebWorker <--> UI` communication, `UI <--> Server` communication, or `Window <--> Window` communication.

Running Code on the UI

If your application needs to run code on the UI, there are a few options. The easiest way is to use a CustomElement in your view. You can then register this custom element from your html file and run code in response to the element's lifecycle hooks. Note, Custom Elements are still experimental. See MDN for the latest details on how to use them.

If you require more robust communication between the WebWorker and the UI you can use the MessageBroker or MessageBus directly.

MessageBus

The MessageBus is a low level abstraction that provides a language agnostic API for communicating with angular components across any runtime boundary. It supports multiplex communication through the use of a channel abstraction.

Angular currently includes two stable MessageBus implementations, which are used by default when you run your application inside a WebWorker.

1. The `PostMessageBus` is used by JavaScript applications to communicate between a WebWorker and the UI.
2. The `IsolateMessageBus` is used by Dart applications to communicate between a background Isolate and the UI.

Angular also includes three experimental MessageBus implementations:

1. The `WebSocketMessageBus` is a Dart MessageBus that lives on the UI and communicates with an angular application running on a server. It's intended to be used with either the `SingleClientServerMessageBus` or the `MultiClientServerMessageBus`.
2. The `SingleClientServerMessageBus` is a Dart MessageBus that lives on a Dart Server. It allows an angular application to run on a server and communicate with a single browser that's running the `WebSocketMessageBus`.
3. The `MultiClientServerMessageBus` is like the `SingleClientServerMessageBus` except it allows an arbitrary number of clients to connect to the server. It keeps all connected browsers in sync and if an event fires in any connected browser it propagates the result to all connected clients. This can be especially useful as a debugging tool, by allowing you to connect multiple browsers / devices to the same angular application, change the state of that application, and ensure that all the clients render the view correctly. Using these tools can make it easy to catch tricky browser compatibility issues.

Using the MessageBus in Your Application

Note: If you want to pass custom messages between the UI and WebWorker, it's recommended you use the MessageBroker. However, if you want to control the messaging protocol yourself you can use the MessageBus directly.

You can obtain the MessageBus on both the render and worker thread through DI. To use the MessageBus you need to initialize a new channel on both the UI and WebWorker. In TypeScript that would look like this:

```
// public_api.ts, which is running on the UI.
import {WORKER_RENDER_PLATFORM, WORKER_RENDER_APPLICATION, WORKER_SCRIPT, MessageBus} from '
import {platform} from "angular2/core";

let appRef = platform([WORKER_RENDER_PLATFORM])
.application([WORKER_RENDER_APPLICATION, {provide: WORKER_SCRIPT, useValue: "loader.js"};
let bus = appRef.injector.get(MessageBus);
bus.initChannel("My Custom Channel");
```

```
// background_index.ts, which is running on the WebWorker
import {MessageBus} from 'angular2/platform/worker_app';
@Component({...})
@View({...})
export class MyComponent {
  constructor (bus: MessageBus) {
    bus.initChannel("My Custom Channel");
  }
}
```

Once the channel has been initialized either side can use the `from` and `to` methods on the MessageBus to send and receive messages. Both methods return EventEmitter. Expanding on the example from earlier:

```
// public_api.ts, which is running on the UI.
import {WORKER_RENDER_PLATFORM, WORKER_RENDER_APPLICATION, WORKER_SCRIPT, MessageBus} from '
import {platform} from "angular2/core";

let appRef = platform([WORKER_RENDER_PLATFORM])
.application([WORKER_RENDER_APPLICATION, {provide: WORKER_SCRIPT, useValue: "loader.js"};
let bus = appRef.injector.get(MessageBus);
bus.initChannel("My Custom Channel");
bus.to("My Custom Channel").emit("Hello from the UI");
```

```
// background_index.ts, which is running on the WebWorker
import {Component, View} from 'angular2/core';
import {MessageBus} from 'angular2/platform/worker_app';
@Component({...})
@View({...})
export class MyComponent {
  constructor (bus: MessageBus) {
    bus.initChannel("My Custom Channel");
    bus.from("My Custom Channel").observer((message) => {
      console.log(message); // will print "hello from the UI"
    });
  }
}
```

This example is nearly identical in Dart, and is included below for reference:

```
// index.dart, which is running on the UI.
import 'package:angular2/web_workers/ui.dart';

main() {
  import "angular2/core.dart";
  import "angular2/platform/worker_render.dart";

  platform([WORKER_RENDER_PLATFORM])
  .asyncApplication(initIsolate("my_worker.dart")).then((ref) {
    var bus = ref.injector.get(MessageBus);
    bus.initChannel("My Custom Channel");
    bus.to("My Custom Channel").add("hello from the UI");
  });
}
```

```
// background_index.dart, which is running on the WebWorker
import 'package:angular2/platform/worker_app.dart';
@Component(...)
@View(...)
class MyComponent {
  MyComponent (MessageBus bus) {
    bus.initChannel("My Custom Channel");
    bus.from("My Custom Channel").listen((message) {
      print(message); // will print "hello from the UI"
    });
  }
}
```

The only substantial difference between these APIs in Dart and TypeScript is the different APIs for the `EventEmitter`.

Note: Because the messages passed through the MessageBus cross a WebWorker boundary, they must be serializable. If you use the MessageBus directly, you are responsible for serializing your messages. In JavaScript / TypeScript this means they must be serializable via JavaScript's structured cloning algorithim.

MessageBus and Zones

The MessageBus API includes support for zones. A MessageBus can be attached to a specific zone (by calling `attachToZone`). Then specific channels can be specified to run in the zone when they are initialized. If a channel is running in the zone, that means that any events emitted from that channel will be executed within the given zone. For example, by default angular runs the EventDispatch channel inside the angular zone. That means when an event is fired from the DOM and received on the WebWorker the event handler automatically runs inside the angular zone. This is desired because after the event handler exits we want to exit the zone so that we trigger change detection. Generally, you want your channels to run inside the zone unless you have a good reason for why they need to run outside the zone.

Implementing and Using a Custom MessageBus

Note: Implementing and using a Custom MessageBus is experimental and the APIs may change.

If you want to drive your application from something other than a WebWorker you can implement a custom message bus. Implementing a custom message bus just means creating a class that fulfills the API specified by the abstract MessageBus class.

If you're implementing your MessageBus in Dart you can extend the `GenericMessageBus` class included in angular. if you do this, you don't need to implement zone or channel support yourself. You only need to implement a `MessageBusSink` that extends `GenericMessageBusSink` and a `MessageBusSource` that extends `GenericMessageBusSource`. The `MessageBusSink` must override the `sendMessages` method. This method is given a list of serialized messages that it is required to send through the sink.

Once you've implemented your custom MessageBus in either TypeScript, you must provide it through DI during bootstrap like so:

In TypeScript:

```
// public_api.ts, running on the UI side
import {platform, APP_INITIALIZER, Injector} from 'angular2/core';
import {
    WORKER_RENDER_PLATFORM,
    WORKER_RENDER_APPLICATION_COMMON,
    initializeGenericWorkerRenderer,
    MessageBus
} from 'angular2/platform/worker_render';

var bus = new MyAwesomeMessageBus();
platform([WORKER_RENDER_PLATFORM])
.application([WORKER_RENDER_APPLICATION_COMMON, {provide: MessageBus, useValue: bus},
  { provide: APP_INITIALIZER,
    useFactory: (injector) => () => initializeGenericWorkerRenderer(injector),
    deps: [Injector],
    multi: true
  }
]);
```

```
// background_index.ts, running on the application side
import {WORKER_APP_PLATFORM, genericWorkerAppProviders} from "angular2/platform/worker_app"
import {NgZone, platform} from "angular/core";
import {MyApp} from './app';
```

```
/**
 * Do initialization work here to set up the app thread and MessageBus;
 * Once you have a working MessageBus you should bootstrap your app.
 */

platform([WORKER_APP_PLATFORM_PROVIDERS])
.application([WORKER_APP_APPLICATION_COMMON, {provide: MessageBus, useValue: bus},
{provide: APP_INITIALIZER, useFactory: (zone, bus) => () => initAppThread(zone, bus), multi
.bootstrap(MyApp);

function initAppThread(zone: NgZone, bus: MyAwesomeMessageBus): void{
  /**
   * Here you can do any initialization work that requires the app providers to be initiali
   * At a minimum, you must attach your bus to the zone and setup a DOM adapter.
   * Depending on your environment you may choose to do more work here.
  */
}
```

Notice how we use the `WORKER_RENDER_APPLICATION_COMMON` providers instead of the `WORKER_RENDER_APPLICATION` providers on the render thread. This is because the `WORKER_RENDER_APPLICATION` providers include an application initializer that starts a new WebWorker/Isolate. The `WORKER_RENDER_APPLICATION_COMMON` providers make no assumption about where your application code lives. However, we now need to provide our own app initializer. At the very least this initializer needs to call `initializeGenericWorkerRenderer`.

MessageBroker

The MessageBroker is a higher level messaging abstraction that sits on top of the MessageBus. It is used when you want to execute code on the other side of a runtime boundary and may want to receive the result. There are two types of MessageBrokers:

1. The `ServiceMessageBroker` is used by the side that actually performs an operation and may return a result;
2. The `ClientMessageBroker` is used by the side that requests that an operation be performed and may want to receive the result.

Using the MessageBroker In Your Application

To use MessageBrokers in your application you must initialize both a `ClientMessageBroker` and a `ServiceMessageBroker` on the same channel. You can then register methods with the `ServiceMessageBroker` and instruct the `ClientMessageBroker` to run those methods. Below is a lightweight example of using MessageBrokers in an application. For a more complete example, check out the `WebWorkerRenderer` and `MessageBasedRenderer` inside the Angular WebWorker code.

Using the MessageBroker in TypeScript

```
// public_api.ts, which is running on the UI with a method that we want to expose to a WebWo
import {WORKER_RENDER_PLATFORM, WORKER_RENDER_APPLICATION, WORKER_SCRIPT, ServiceMessageBrok
import {platform} from "angular2/core";

let appRef = platform([WORKER_RENDER_PLATFORM])
.application([WORKER_RENDER_APPLICATION, {provide: WORKER_SCRIPT, useValue: "loader.js"};
let injector = instance.injector;
var broker = injector.get(ServiceMessageBrokerFactory).createMessageBroker("My Broker Channe

// assume we have some function doCoolThings that takes a string argument and returns a Pro
broker.registerMethod("awesomeMethod", [PRIMITIVE], (arg1: string) => doCoolThing(arg1), PRI
// background.ts, which is running on a WebWorker and wants to execute a method on the UI
import {Component, View} from 'angular2/core';
import {ClientMessageBrokerFactory, PRIMITIVE, UiArguments, FnArgs} from 'angular2/platform/

@Component(...)
@View(...)
export class MyComponent {
  constructor(brokerFactory: ClientMessageBrokerFactory) {
    var broker = brokerFactory.createMessageBroker("My Broker Channel");

    var arguments = [new FnArg(value, PRIMITIVE)];
    var methodInfo = new UiArguments("awesomeMethod", arguments);
    broker.runOnService(methodInfo, PRIMITIVE).then((result: string) => {
      // result will be equal to the return value of doCoolThing(value) that ran on the UI.
    });
  }
}
```

Using the MessageBroker in Dart

```
// index.dart, which is running on the UI with a method that we want to expose to a WebWorke
import "angular2/core.dart";
import "angular2/platform/worker_render.dart";

main() {
  platform([WORKER_RENDER_PLATFORM])
  .asyncApplication(initIsolate("my_worker.dart")).then((ref) {
    var broker = ref.injector.get(ServiceMessageBrokerFactory).createMessageBroker("My Broke

    // assume we have some function doCoolThings that takes a String argument and returns a
    broker.registerMethod("awesomeMethod", [PRIMITIVE], (String arg1) => doCoolThing(arg1),
  });
}

// background.dart, which is running on a WebWorker and wants to execute a method on the UI
import 'package:angular2/core.dart';
import 'package:angular2/platform/worker_app.dart';

@Component(...)
@View(...)
class MyComponent {
  MyComponent(ClientMessageBrokerFactory brokerFactory) {
    var broker = brokerFactory.createMessageBroker("My Broker Channel");

    var arguments = [new FnArg(value, PRIMITIVE)];
    var methodInfo = new UiArguments("awesomeMethod", arguments);
    broker.runOnService(methodInfo, PRIMTIVE).then((String result) {
      // result will be equal to the return value of doCoolThing(value) that ran on the UI.
    });
  }
}
```

Both the client and the service create new MessageBrokers and attach them to the same channel. The service then calls `registerMethod` to register the method that it wants to listen to. Register method takes four arguments. The first is the name of the method, the second is the Types of that method's parameters, the third is the method itself, and the fourth (which is optional) is the return Type of that method. The MessageBroker handles serializing / deserializing your parameters and return types using angular's serializer. However, at the moment the serializer only knows how to serialize angular classes like those used by `Renderer2`. If you're passing anything other than those types around in your application you can handle serialization yourself and then use the `PRIMITIVE` type to tell the MessageBroker to avoid serializing your data.

The last thing that happens is that the client calls `runOnService` with the name of the method it wants to run, a list of that method's arguments and their types, and (optionally) the expected return type.