Design: Running Angular Apps in WebWorker

*Status: (Draft)*

*Authors: misko@google.com*

*This document is published to the web as part of the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk) *folder*

# Objective

Explore the idea of running an Angular application in a web-worker thread and let angular marshal and render on the UI thread in order to improve performance.

# Background

WebWorkers are secondary (non-UI) threads which can be executed in the browser. To simplify the threading model web-workers are not allowed to share any memory between threads and are only allowed to communicate in an asynchronous way.

The basic idea is to fully execute the application in the WebWorker thread and hence unblocking the main thread to deal with purely rendering operations. This, in theory, should increase the perceived latency of the application. This is in theory, since it is not clear what kinds of costs will be involved in marshaling the rendering instructions.

# Prior Art

* [WebWorkers](http://www.html5rocks.com/en/tutorials/workers/basics/)
* [Transferable Objects](http://www.w3.org/html/wg/drafts/html/master/infrastructure.html#transferable-objects)

# Detailed Design

## Constraints

The design will be governed by the kind of constraints which exist around web workers. Let's explore them.

### DOM

Web workers can not access DOM nodes. This means that all DOM manipulation must stay in the UI thread. The implications are:

* HTML Compiler executes on the UI Thread. (Needs access to actual DOM)
* ProtoView and View need to be split into DOM manipulation and directive creation code
* Directives need to be given Element Proxy objects. This is actually desirable since we need to intercept DOM manipulation for two reasons:
  1. Animation: Intercept class setting and element insertion and removal.
  2. View reuse: needs to guarantee that DOM was not modified without our knowledge and hence can no longer be reused.

### Asynchronous Communication

Communication between threads is asynchronous which complicates making decision on the DOM which must be synchronous. IE intercepting events and preventing default. I believe this can be solved by knowing response ahead of time or by always preventing and then simulating propagation.

### Web Components

Must run on the UI thread. Communicating between web-component and angular directives becomes trickier. I think this can be solved by assuming that any Web Components used are purely for the rendering purposes (ie calendar widget) and not part of the application behavior. ie dumb rendering black boxes.

## ProtoView / View

A natural place to place the boundary is on the View. A View is indivisible block of HTML which can be added to other Views. As such the only operations which need to be marshaled to it are which text node needs to be updated with which value. (optionally we may have to marshal property updates on elements.) View is already set up to do all of this due to the way the change detection communicates changes to it.

The issue is that the View currently has both the App (change detection, directive instances) and Elements. So this section is really about how the two can be split, and how we can marshal communication messages between them.

|  |
| --- |
| class DOMView {  // needed to match up/identify views during marshaling.  var id:string;  var textNodes: Array<Text>;  var elements: Array<Element>;  var viewPorts: Array<DOMViewPorts>;  } |
| class WorkerView implements WatchGroupDispatcher {  // needed to match up/identify views during marshaling.  var id:string;  var elementInjectors: Array<ElementInjector>;  var watchGroup: WatchGroup;  // Elements may be injected into Directives. These are proxies.  var elements: Array<NgElement>;  var viewPorts: Array<WorkerViewPort>;  } |

Marshaling operations

* FORWARD: from App to DOM
  + write string to text node
  + write primitive to Element property
  + invoke method on Element with primitive arguments
  + Instantiate/Release View
  + Add/Remove/Move View to ViewPort
* REVERSE: from DOM to App
  + Forward event to App
  + DOM read queue: callback would have to be injected with the read values once DOM stable and the values have been marshaled to the Worker Thread.

## Open Questions

* Do we need special class of directives which run on the UI Thread?
  + Implication is that they would not be able to participate in the change detection.
  + We could have second change detection loop in the UI thread. Probably not worth it. What would it be detecting?
  + Some DOM directives with no change detection just to do simple UI related work. Maybe animation?
* If the application fully runs in WebWorker, do we even need zones in UI thread?
  + Maybe for debugging and better understanding of event propagation
  + Maybe for integrating with third party code. Need to think more about it.
* Do we need zones in WebWorker?
  + Probably for XHR

# Design Brainstorming

There are three areas which need to be solved:

* bootstrapping
* HTML compilation and marshaling
* View marshaling

In general the problem resolves around correctly placing the right piece of code on the right side of the boundary as well as creating the right kind of marshaling abstractions to minimize the amount of communications between the two layers.

## General Thoughts

* Worker Thread
  + Directives: the code is loaded and stays in the worker thread.   
    Implication: the HTML compiler does not get a reference to it. This means that we need to be able to somehow marshal the KEY IDs for dependency injection so that the compiler can configure ProtoView/ProtoElementInjector. We also need to marshal the selectors.
* UI Thread
  + A stripped down version of [Proto]View/[Proto]ViewPort need to stay on the UI thread. It contains DOM but not the element Injector. Remove the injector bits.
  + NgElement a proxy to the element needs to be injected into Directives. Will need to have DOM read/write async queues here.

## Bootstrapping

1. UI Thread launches Angular Slave (Slave because it does not initiate anything by itself. The worker thread is in charge.)
2. Worker Thread starts up and handshakes with the slave. At this point Worker thread is in charge for the remaining of the application.
3. Worker Thread then bootstraps like normal application.

## Compilation

1. WORKER requests UI to compile DOM.
   1. marshal all directive IDs, injection IDs and Selectors
   2. Compiler builds SelectorMatcher using the directive selectors. Matching returns the directive ID, injection IDs
   3. HTML Compiler requires
      1. HTML Template. Either marshaled as string or Compiler fetches it over the network
      2. List of directive selectors: easy to mashal just array of strings
         1. If it directive is component than this is recursive
      3. Directive ID: this would be a Type on App side but just an ID (int) on UI side.
2. Normal compilation executes
   1. Result in UIProtoView
   2. UIProtoView is marshaled to AppProtoView. In the process ProtoElementInjector is materialized.

## Runtime

1. WORKER: AppProtoView.createView() results in AppView
   1. AppProtoView sends command to UIProtoView to create UIView
   2. UIView clones DOM, sets up DOM listeners
2. WORKER: ChangeDetection sets bindings on AppView
   1. AppView marshals bindings to UIView which then sets them on DOM.

## Marshaling

Marshaling is the general problem of getting data from one thread to the other. I think the approach which would work best for Angular is to have two objects which are tied together on both side representing the same concept. ie have UIView and WorkerView. The marshaling layer would than be able to send commands between the two objects back and forth.



The marshaling system would have to understand that two objects represent the same concept on two different locations. It would also have to be able to serialize commands and deliver them to the other side.

Out of the box the marshaling system can only marshal primitives. There would have to be some kind of pluggable architecture which would allow the marshaler to serialize and deserialize non primitive objects across the threads. This pluggable architecture would also allow communication with more complex web-components which require special types.

# Caveats

You may need to describe what you did not do or why simpler approaches don't work. Mention other things to watch out for (if any).

# Security Considerations

How you’ll be secure

# Performance Considerations / Test Strategy

How you’ll be fast.

# Work Breakdown

## Naming

Part of reasoning about the code requires that we have proper mental model and with it proper names

|  |  |  |
| --- | --- | --- |
| **Description** | **UI Thread (Interface)** | **Worker Thread** |
| n/a | TemplateCompiler | Compiler |
| Result of template compilation is the ProtoBlock. There is additional information which is created which is combined with ProtoView | ProtoBlock | ProtoView |
| Basic building blocks for building the UI. Blocks are purely DOM based. Views contain injectors, and change detection | Block | View |
|  | BlockPort | ViewPort |
|  | n/a | ProtoElementInjector |
|  | n/a | ElementInjector |

|  |
| --- |
| class Compiler {  var templateCompiler: TemplateCompiler;  compile(component:Type) {  var componentInfo: ComponentInfo;  var compilationResult:CompilationResult;  templateCompiler.compile(templateInfo).then(function(result) {  compilationResult = result;  });  }  }  // Interface which has no dependencies on DOM  class TemplateCompiler {  compile(componentInfo:ComponentInfo):Promise<CompilationResult> {  }  }  // Remote Proxy which has no dependencies on DOM  class RemoteTemplateCompiler extends TemplateCompiler {  }  // Actual compiler which has direct dependencies on DOM  class LocalTemplateCompiler extends TemplateCompiler {  }  class TemplateInfo {  url:string;  html:string;  selectors: Array<string>;  directiveIds: Array<string>; }  class CompilationResult {  textExpr:Array<string>;  propertyExpr: Array<ElementPropertyBinding>;  eventExpr: Array<?>;  }  class ElementPropertyBinding {  name:string;  directive:int;  element:bool;  } |