Angular 2 Server Side Rendering

See video presentation about this initiative here: <https://www.youtube.com/watch?v=0wvZ7gakqV4>

Jeff Whelpley ( [jeffwhelpley@gmail.com](mailto:jeffwhelpley@gmail.com) )

PatrickJS ( [patrick@gdi2290.com](mailto:patrick@gdi2290.com) )

Tobias Bosch ( [tbosch@google.com](mailto:tbosch@google.com) )

[Objective](#h.8jcgnap8f3r6)

[Prior Art](#h.3dy3jy8uvydg)

[Rendr](#h.396ojgyzq3cs)

[Derby](#h.kdd2w6fumwp0)

[Meteor](#h.3444uzlxa1ca)

[React](#h.vk2ypqlbako8)

[Ember FastBoot](#h.oqdvilemouy0)

[Angular 2](#h.9ktkkwfkrinz)

[Use cases](#h.je8lg3vxthdp)

[1. Perceived load time](#h.tqfcammmig70)

[2. Actual load time](#h.kb1huta7868e)

[3. Client side performance](#h.19rvyetmpg3p)

[4. SEO](#h.qgx786fxlhn)

[5. Browser support](#h.46u1wihlvbao)

[6. Link preview](#h.n8z3e4oswtow)

[Technical Requirements - Primary](#h.olltj24wlid3)

[“It just works”](#h.442ppinnqx9a)

[Seamless state transfer](#h.qb7o6mgg3iha)

[Performance](#h.k91zcotpwnr)

[Technical Requirements - Secondary](#h.oaa2c7p7q8pj)

[Extensibility](#h.53nx85voy9cg)

[Component Routing](#h.1pb54xfr3qol)

[Optimizations](#h.hc7avqg7qbam)

[Technical Design](#h.kwdqafwkjctw)

[Approach](#h.c0of4oavsbzt)

[Overall Flow](#h.ufkq8ncys8zn)

[Module - ServerDomRenderer](#h.fyrsnic17yoq)

[Module - Server Plugin](#h.8j6bv8mkb3ws)

[Module - Preboot](#h.b3qdlffhbqmu)

[Key Features](#h.88oy92klxm6v)

[Installation](#h.ub92bphwuqgo)

[Options](#h.wt7ein25y6a5)

[Listen Strategies](#h.jcvyw758t31g)

[Replay Strategies](#h.x0wlr1ydxkru)

[Freeze Strategies](#h.2detaith1wvu)

[Future Items](#h.3b63hfyxr35q)

[Non-JS server rendering](#h.smugg9t1gji)

[Packaging service](#h.5zgpb7is3kta)

[Notes](#h.nbkj9q8npvu5)

# Objective

The new rendering architecture of Angular 2 will separate the Angular runtime into two layers: the application layer and the rendering layer. The application layer contains APIs and runtime that the application code interacts with directly. The rendering layer provides a common protocol for performing UI updates. The split will allow Angular 2 applications to run in different environments/platforms while providing the same set of abstractions to application developers. The default rendering environment is the browser, but other environments could include web workers, web servers or other devices. The primary purpose of this document is to design a solution for rendering Angular 2 views on a web server as well as the process by which server rendered views seamlessly transfer state down to an Angular 2 client app running in a browser.

# Prior Art

Spike Brehm from AirBnb popularized the term [Isomorphic JavaScript](http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/). Some people use this term just to refer to shared code used for view rendering while other people use the term for any shared code between the client and server. More recently, Michael Jackson suggested calling this [Universal JavaScript](https://medium.com/@mjackson/universal-javascript-4761051b7ae9). The following existing frameworks support isomorphic/universal view rendering at some level.

## [Rendr](https://github.com/rendrjs/rendr)

**About**

* *Background* - Rendr was created by Spike Brehm at AirBnb to render Backbone views on the server.
* *Popularity* - ~3,800 GitHub stars, but note that [AirBnb is switching from Rendr to React](https://medium.com/code-stories/dev-chats-spike-brehm-of-airbnb-87e155f3475d)
* *How it works*- The basic strategy of Rendr is to create wrappers around all the core Backbone objects. An app uses Rendr.view instead of Backbone.view, Rendr.collection instead of Backbone.collection, etc. There are also a set of server specific libraries for running a Rendr app on the server.

**Details**

* *Templates* - Uses Handlebars which is a string-based templating engine. These types of templating engines are actually good for isomorphic rendering because there is no need to analyze the contents of the template on the client or server. Also, rendering for string-based templates is typically very fast.
* *Routing* - Rendr.Router will feed the Express router on the server side and the Backbone.Router on the client side
* *State transfer* - The server put the serialized state in the rendered view and then the client will [hydrate the server rendered view](http://rendrjs.github.io/fetcher/#hydrate) on the client using that serialized state.

**Notes**

* Backbone subviews and jQuery plugins often modify the app state after a view gets rendered and thereby prevent a full re-render without losing some of that newly added state and incurring a performance hit
* Tightly coupled to Express on the server side

## 

## [Derby](https://github.com/derbyjs/derby)

**About**

* *Background* - Derby was created in 2011 by Nate Smith, the current CTO of Lever.
* *Popularity* - ~3,750 GitHub stars (compared to 1,600 two years ago)
* *How it works*- Derby is a full stack JavaScript framework. Whereas all of the other frameworks listed here have the feel of a client side framework with server side functionality, Derby is more server focused. It was built on Express and uses standard node.js libraries.

**Details**

* *Templates* - By default uses its own reactive templating system that can render on the client and server.
* *Routing* - Routes that are defined in Derby are fed into the Express router on the server side and a custom router built off Express’s router on the client side.
* *State transfer* - After the server generates HTML and sends it down to the client, the client will [“attach” to the server rendered view](https://github.com/derbyjs/derby/blob/master/lib/App.js#L58). This means that the client view starts the generation process as it normally would but instead of inserting new DOM elements, it checks to see what is already in the DOM from the server rendered view and then does an update on that existing element.

**Notes**

* Tightly coupled to Express

## [Meteor](https://github.com/meteor/meteor)

**About**

* *Background* - Built and maintained by a venture-based startup.
* *Popularity* - [7th most popular JavaScript project](https://github.com/search?l=JavaScript&o=desc&q=stars%3A%3E1&s=stars&type=Repositories) on GitHub and has increased in popularity over time. ~26,000 stars on GitHub (compared to 6,000 two years ago).
* *How it works*- Meteor, an ultra-simple, database-everywhere, data-on-the-wire, pure-Javascript web framework. The primary focus on Meteor is on real-time web and mobile apps.

**Details**

* *Templates* - By default it uses [Blaze](https://www.meteor.com/blaze) for it’s template engine. This only works by default on the client, however there is [a project from the community](https://github.com/meteorhacks/meteor-ssr) to adapt Blaze for server rendering.
* *Routing* - Meteor core does not have any routing, but there are a number of projects from the community such as the [Meteor Iron Router](https://github.com/iron-meteor/iron-router).
* *State transfer* - Meteor core does not have server rendering, so accordingly there is no mechanism to transfer state from server rendered views to the client.

**Notes**

* While it is possible to set up server rendering with Meteor, the [very first core principle of Meteor](http://docs.meteor.com/#/basic/sevenprinciples) explicitly states “Meteor doesn't send HTML over the network” (they use [DDP](https://www.meteor.com/ddp) instead of HTTP for client/server communication).

## [React](https://github.com/facebook/react)

**About**

* *Background* - Created by Facebook and released to the public last year.
* *Popularity* - 8th most popular JavaScript project on GitHub (~24,000 stars) despite only being around for about a year.
* *How it works*- React focuses only on the view layer and uses a virtual DOM diff implementation for high performance. During initial rendering and any subsequent changes, the entire page is re-rendered within the virtual DOM, but only the specific pieces of the virtual DOM that change are actually pushed to the real DOM.

**Details**

* *Templates* - At a low level, there are no templates in React. React uses JavaScript code to generate a virtual DOM object. However, there is an abstraction on top of this low level code called JSX which looks like HTML mixed into JavaScript. The only difference between client and server rendering is that while the virtual DOM changes are pushed to the DOM on the client side, the virtual DOM is simply serialized into an HTML string on the server before it is sent back to the client.
* *Routing* - React is only a view layer and does not contain anything else, including any router. However, the [react-router](https://github.com/rackt/react-router) built by the community is very popular and does client and server routing.
* *State transfer* - There is no need to serialize extra state within a React server rendered view because the first thing React does on the client side is diff the initial virtual DOM against the server rendered HTML. Thus, state is the server rendered HTML. Note, however, that this does mean the client side has to re-retrieve data from the API.

**Notes**

* Currently the server rendering solution is relatively slow because the server has to do a lot of work at run time to concatenate HTML. Most template engines will pre-compile templates and then do some caching once a view has been rendered the first time. React does not currently use either of these server side optimizations.

## [Ember FastBoot](https://github.com/tildeio/ember-cli-fastboot)

NOTE: Although Ember has released FastBoot and it is being used in the wild, they are in the process of developing another server rendering solution using the [Glimmer engine](https://github.com/emberjs/ember.js/pull/10501) which will use a virtual DOM similar to React (so, for example will use virtual DOM diff for strate transfer), but will have some additional enhancements like template precompilation. The following info is only applicable to the current FastBoot solution:

**About**

* *Background* - Created by Tom Dale a Yehuda Katz from Tilde.
* *Popularity* - 13,173 GitHub stars with a relatively small but extremely passionate user base.
* *How it works*- While Ember has been around for awhile, FastBoot was just recently created and released. The basic strategy of FastBoot is to use the [simple-dom](https://github.com/krisselden/simple-dom) library on the server side

**Details**

* *Templates* - Ember previously used Handlebars, but has more recently moved to a new library called [HTMLBars](https://github.com/tildeio/htmlbars). In both cases they rely on some sort of DOM to exist which is why they have to use simple-dom.
* *Routing* - The Ember Router works on both the client and server.
* *State transfer* - State is serialized on the server side in the HTML and then hydrated by the client app in the browser.

**Notes**

* FastBoot was just released a couple months ago.

## Angular 2

The following Angular 2 features exist today and will help support server rendering:

* [Angular 2 Rendering Architecture](https://docs.google.com/document/d/1M9FmT05Q6qpsjgvH1XvCm840yn2eWEg0PMskSQz7k4E/edit)
* [Angular 2 Router](https://github.com/angular/router)

# Use cases

Each of these use cases provide a possible answer to the question: Why is server rendering important for your client side web app? This section contains the primary use cases in the order that we believe are most important to the community.

## 1. Perceived load time

The initial load of a client side web application is typically very slow. The [Filament Group released a study](http://www.filamentgroup.com/lab/mv-initial-load-times.html) recently that says the average initial page load time for simple Angular 1.x apps on mobile devices is 3 - 4 seconds. It can be even worse for more complex apps. This is most often an issue for consumer facing apps, especially those that are typically accessed on a mobile device, but can be a problem for any app. The goal of rendering on the server for this use case is to lower the user’s perceived performance of the initial page load so that they see real content in under 1 second regardless of device or connection speed. This goal is much easier to achieve consistently with server rendering than client side rendering.

## 2. Actual load time

The initial load for the client web app is generally faster if the client code file is smaller and only executes what is needed to render the initial views. Then, other parts of the application are lazily loaded when needed or in the background. For the most part the solution for this occurs at build time. However, there may be additional optimizations that can be made and/or adjusted at runtime based off additional insight gleaned from actually executing the code on the server.

## 3. Client side performance

The typical story for server rendering is that it covers the initial page load and then the client handles rendering from that point forward. However, there are situations while the client has control of the app where it would be faster to fetch a fully rendered page snippet from the server rather than render it on the client. For example, let’s say there is a mostly static, but complex product page. Consider the following situations:

1. A section of the page takes a long time to render regardless of the best optimization efforts since it requires pulling down data from a very slow API call. The data for this partial does not change often, so we can cache a fully rendered version of the partial on the server and instead of pulling the data, we pull the cached, rendered partial HTML.
2. Same as before except the API call isn’t quite that bad. It is debateable whether there is a benefit to do the partial server rendering at run time. However, you can set up a process where the rendered partial is either pulled asynchronously and cached on the client or, in some situations, even render the partial at build time.
3. For certain visualizations, it may be more efficient to render on the server than the client. This is especially true for complex graphs or charts which require a lot of data and heavy computations, but the final output is relatively small.

It should be noted that Twitter has used this type of optimization for around 2 years. It’s benefits may be most clear on lower powered mobile devices.

## 4. SEO

The Google search crawler continues to get better at indexing client side rendered content, but there are challenges.

1. First, the crawler isn’t perfect (yet). There are a number of situations where the crawler may not index exactly what is rendered. This is often due to either JavaScript incapabilities or timing issues with async loading.
2. With server rendering, the crawler can determine exactly how long it takes before the user sees content (i.e. Document Complete). It is not as easy on the client side (and, as mentioned in the previous use case, when it is measured, it is often much slower than server rendering).
3. There is no success story out there for a client side only web app that beats server rendered websites for competitive keyword searches (ex. think any major purchase item like “flat screen tv” or “best sedan 2015”).

There likely will be a future where server rendering is not needed for SEO, but today consumer facing apps that really care about their search ranking need server rendering.

## 5. Browser support

The downside of using more advanced web technologies like Web Components is that it is hard to keep support for older browsers. This is why Angular 2 only targets evergreen browsers. However, depending on the app being built, it may be possible to replace certain rich-client behaviors with server-side behaviors to support older browsers while letting app developers take advantage of the latest web platform. A couple examples:

1. The app is mostly informational and it is OK for users on older browsers to just see the information without any of the client side functionality. In this case, it may be alright to give legacy browser users a completely server rendered website while evergreen browser users get the full client side app.
2. An app must support IE9 (not an evergreen browser, but not as bad as IE8). Most of the client side web app functionality works, but there is one component that uses functionality not supported by IE9. For that one component, the app could potentially fetch the fully rendered partial HTML from the server.

Note that this use case is NOT so that we can say Angular 2 actually supports something like IE6 because it would depend on the specific use case. A given app may require features in modern browsers and there is no server rendered version that would help. This is more for use cases where it is OK to have the app provide a static server rendered version of a component or page or website for legacy browsers.

## 6. Link preview

Programs that show website previews for provided links rely on server rendering. Due to the complexity involved with capturing client side rendered web pages, these programs will likely continue to rely on server rendering for the foreseeable future. The most well known examples involve social media platforms like Facebook, G+ or LinkedIn. Similar to the SEO use case, this is only relevant for consumer facing apps.

# Technical Requirements - Primary

The purpose of this section is to lay out the primary requirements for a server rendering solution in Angular.

## “It just works”

Right out of the box, Angular should be able to render an Angular 2 web app on a JavaScript backend server. It does not have to render perfectly and it does not need to be optimized right out of the box, but it does need to render. The goal of the functionality out of the box is to give developers a baseline for a server rendered solution. As long as an Angular 2 app does not directly touch the DOM (ex. reference the window or document object outside of the Angular 2 APIs), the component should render on the server just like how it renders on the client.

## Seamless state transfer

The client should be able to take over the server rendered view seamlessly. So, there shouldn’t be any jank during the transfer and user interactions that occur with the server rendered page (ex. user entering data into a form input) should not be interrupted in any way. So, if the user types in a search box, both the characters in the search box and the focus of the cursor in the search box should remain throughout the transfer process.

## Performance

As mentioned in the Use Cases above, the primary reason developers want server rendering is to improve performance. So, it will be important that a solution in Angular meet certain performance goals (as measured in Benchpress). Given the following conditions:

* Desktop browser on latest Macbook Pro
* Latency < 10ms
* Server is equivalent to one AWS t2 instance
* Single request
* Internet connection with > 11 Mbps download speed

Then the performance of this solution should hit the following targets:

1. Server render time < 100ms
2. Perceived load time < 1s
3. Actual load time < 3s

# Technical Requirements - Secondary

Once the primary requirements have been fulfilled, these are additional technical considerations.

## Extensibility

There are two major areas of extensibility that are needed with the Angular server rendering solution. First, the server side rendering engine must be able to integrate with any existing Node.js app. Second, the solution must be able to be adapted to non-JavaScript backends.

## Component Routing

While most routing will depend on how an app is implemented (i.e. it is up to the UI Router or Angular Router), the server rendering solution will need to have a generic way routing to specific components. This will allow Angular apps to ask the server to render partials.

## Optimizations

While there are common performance bottlenecks, each web app has their own set of unique challenges. It will be important that the Angular server rendering solution includes the following optional features which can be used for performance optimizations:

1. **Server/client** - The developer should be able to specify at the route, page or component level whether a part of the app should be rendered just on the client, just on the server or both.
2. **Server caching** - Page level caching can be handled by the web server, but within the app, we will want to enable caching at the component level. This should be configurable so that any medium can be used for the cache backend.
3. **Lazy loading** - The server rendering solution should be able to easily integrate with a lazy loading solution in order to cut down on the actual initial load time of the app.
4. **Anonymous server rendering** - In some cases it will be beneficial to allow the server to render anonymously (i.e. without user context) so that pages can be more aggressively cached. This is most often used by apps that have mostly public content with a couple small pieces of user-specific content (ex. user profile pic in upper right) for which it is OK to render client-only.

# Technical Design

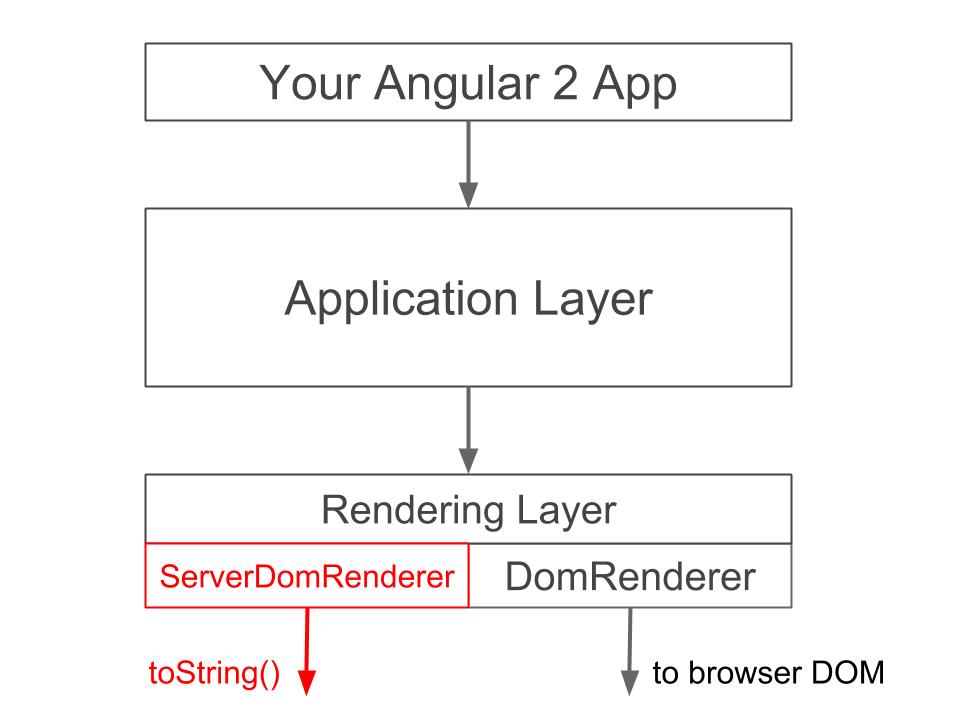
This section contains the design for implementing server rendering in Angular.

## Approach

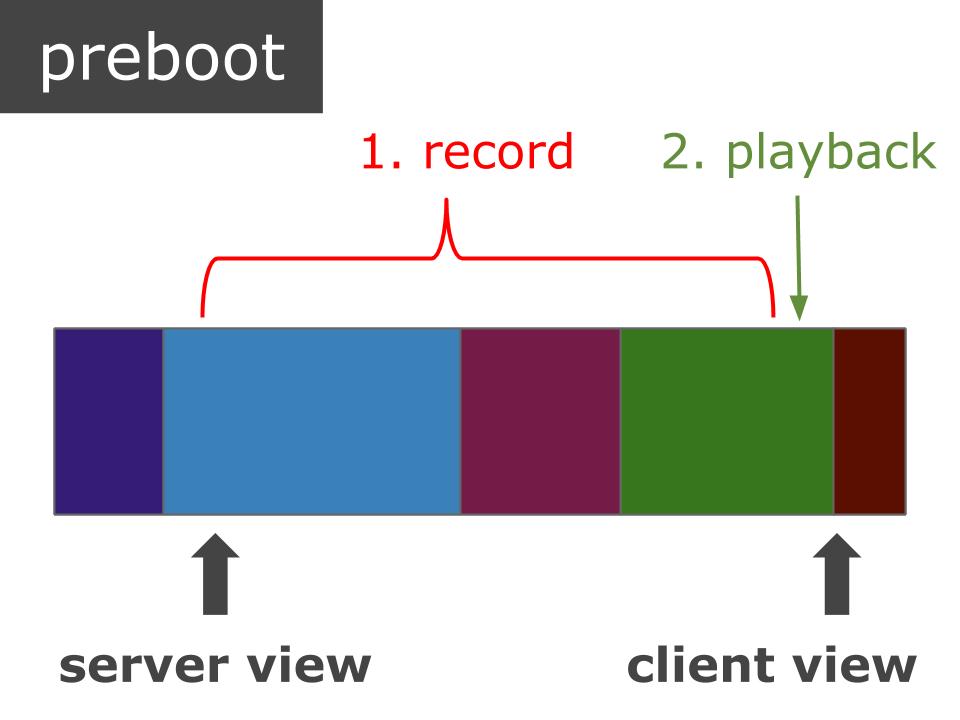
At a high level, there are two major pieces to the Angular 2 server rendering solution:

1. Rendering on the server
2. Transitioning from the server view to the client view

For #1, this solutions relies on the rendering architecture of Angular 2 with one additional piece which is explained further below, the ServerDomRenderer:



For #2, the transfer from the server view to the client view is facilitated by preboot which is also explained further below:



## Overall Flow

Let’s go over a few variations of the Angular rendering solution flow. The first one is the out of the box, basic rendering flow that will occur by default:

1. HTTP GET request sent to the server
2. Server renders page that contains the following:
   1. Rendered HTML from ServerDomRenderer that the user will see initially
   2. Inline JavaScript for preboot (see ‘preboot’ section below)
3. Browser receives initial payload from server
4. User sees server view
5. Preboot starts recording events
6. Requests made for additional external images, JS, CSS, etc.
7. Once external resources loaded, Angular client bootstrapping begins
8. Client view rendered
9. Bootstrap complete, so Angular client calls preboot.done()
10. Preboot events replayed in order to adjust the application state to reflect changes made by the user before Angular bootstrapped (i.e. typing in textbox, clicking button, etc.)

## Module - ServerDomRenderer

This module implements the [render api interface](https://github.com/angular/angular/blob/master/modules/angular2/src/render/api.ts) and will meet the requirements [laid out here](https://github.com/angular/angular/issues/2657). It will build and maintain a data structure which correlates to the browser DOM. We can easily toString() this internal object to generate a string of HTML which can be returned from a web server or used for testing purposes.

(details of this module to follow)

## Module - Server Plugin

This module is the core library that would be included on the server side to run an Angular app. It basically consists of a custom bootstrap implementation along with some convenience APIs to make it easier to plug into existing node.js web server frameworks such as Hapi or Express.

(details of this module to follow)

## Module - Preboot

The purpose of this module is to transition a server-generated view to a client web app. This module has no dependencies on Angular and can actually be used by any framework, even if it is completely outside the context of Angular altogether. In general, the desired behavior of a page before the client takes over will be different for different use cases. So, this library was created in such a way that developers can configure it to meet their needs.

### Key Features

1. **Record and playback events** - Which events are listened to and how they are replayed are specified through the preboot options.
2. **Respond immediately to events** - In some cases, you want an immediate action to take place (i.e. not wait for the client to kick in).
3. **Maintain focus even page is re-rendered** - If the page has to be re-rendered and the focus is in a textbox, we want to refocus in the corresponding textbox in the client view.
4. **Buffer client-side re-rendering for smoother transition** - Although Angular 2 already updates in batches, there may be use cases where we want to buffer our client updates and let preboot smoothly switch the fully rendered client view for the server view. Using this should ensure that the switch from server to client view can occur in 1 frame.
5. **Freeze page until bootstrap complete if user clicks button** - If selected as an option, we can “freeze” the page when the user clicks a button so that no further actions can be made until the client processes that button click.

### Installation

This is a server-side library that generates client-side code. To use this library, you would first install it through npm:

|  |
| --- |
| npm install preboot |

Then in your server-side code you would do something like this:

|  |
| --- |
| var preboot = require('preboot');  var prebootOptions = {}; // see options section below  var clientCode = preboot(prebootOptions); |

You then inject clientCode into the HEAD section of your server-side template. We want preboot to ONLY start recording once the web app root exists in the DOM. We are still playing with the best way to do this (NOTE: we have tried onLoad and

it does not work because the callback does not get executed quickly enough). For now, try putting the following

`preboot.start()` call immediately after your web app root in your server side template:

|  |
| --- |
| <web-app-root-here>  </web-app-root-here>  <script>  preboot.start();  </script> |

Finally, once your client-side web app is "alive" it has to tell preboot that it is OK to replay events.

|  |
| --- |
| preboot.complete(); |

### Options

There are 5 different types of options that can be passed into preboot. In addition to these options, we may in the future add the concept of “presets” so that someone could simply say preset=angular and would get all the default options that are typically used with an Angular app.

**Selectors**

* appRoot - A selector that can be used to find the root element for the view (default is 'body')

**Strategies**

These can either be string values if you want to use a pre-built strategy that comes with the framework or you can implement your own strategy and pass it in here as a function or object. The idea behind a strategy in the preboot library is to allow the developer to customize the library to suit their needs. There are typically a couple built-in options for each type of strategy, but the developer can always write their own. The client-side code generated by preboot will only contain the strategies that are actually being used. So, in other words, if you don't specify 'attributes' as a listen strategy in the preboot options, then none of the list\_by\_attributes.js code will be included in the generated client-side code.

* *listen* - How preboot listens for events. See Listen Strategies section below for more details
* *replay* - How preboot replays captured events on client view. See Replay Strategies section below for more details
* *freeze* - How preboot freezes the screen when certain events occur. See Freeze Strategies section below for more details

**Flags**

All flags false by default.

* *focus* - If true, will track and maintain focus even if page re-rendered
* *buffer* - If true, client will write to a hidden div which is only displayed after bootstrap complete
* *keyPress* - If true, all keystrokes in a textbox or textarea will be transferred from the server
* view to the client view
* *buttonPress* - If true, button presses will be recorded and the UI will freeze until bootstrap complete
* *pauseOnTyping* - If true, the preboot will not complete until user focus out of text input elements
* *doNotReplay* - If true, none of the events recorded will be replayed

**Workflow Events**

These are the names of global events that can affect the preboot workflow.

* *pauseEvent* - When this is raised, preboot will delay the play back of recorded events (default 'PrebootPause')
* *resumeEvent* - When this is raised, preboot will resume the playback of events (default 'PrebootResume')

**Build Params**

* *uglify* - You can always uglify the output of the client code stream yourself, but if you set this option to true preboot will do it for you.

### Listen Strategies

The 'listen' option can either be a string (the name of the pre-built strategy), configuration object (see below) or an array of configuration objects. Each configuration object can contain the following values:

* *name* - The name of one of the following pre-built strategies:
  + *selectors* - An object that maps a selector to an array of events
  + *attributes* - This strategy will inspect the server view for any element that contains a particular attribute name (default is 'preboot-events'). So if there is a `<input preboot-events="keypress,focus">` then all keypress and focus events on that input element would be tracked.
  + *event\_bindings* - Use Angular 2 event bindings defined in the server view HTML
* *getNodeEvents* - A custom strategy implementation
* *preventDefault* - Stop event propagation
* *dispatchEvent* - The name of an custom event that should be dispatched when event occurs
* *trackFocus* - Track current focus if on the target node
* *doNotReplay* - No not replay these events
* *attributeName* - Only used by the attributes strategy to identify the name of the attribute that will have all the events that need to be recorded (default value is 'preboot-events')
* *eventsBySelector* - This is only used by the selectors strategy. It should be an object that maps a selector
* string to an array of events.
* *action* - Custom function executed when events occur

Each strategy will return an array of objects with two values: eventName (name of the event that occurred) and node (the DOM element node on which the event occurred).

### Replay Strategies

The 'replay' option can either be a string (the name of the pre-built strategy), configuration object (see below) or an array of configuration objects.

Each configuration object must have one of two values:

* *name* - The name of the strategy (see below)
* *replayEvents* - A custom strategy implementation

The pre-built strategy or custom replayEvents implementation will attempt to replay all events passed in.

If, however, any events can't be replayed for any reason, they can be returned in an array. If there is

an array of replay strategies defined for preboot, the remaining events from one replay strategy will be

feed into the next.

The pre-built replay strategies are:

* *hydrate* - This assumes that the server view and the client view are the same. In other words, the DOM elements in memory for the server view are still there with the client view. So, when replaying events, we can simply use the existing nodes in memory. Easy peasy.
* *rerender* - During a re-render, the client view typically blows the server view away. This would mean that the nodes we have in memory are no longer valid. However, we can use the nodes in memory to help find the new client rendered nodes. Once we do, we replay the events, set focus, etc.

### Freeze Strategies

Although we want to give the user some functionality before the client-side web app bootstrap is complete, it makes sense to freeze the UI when certain things happen on the page. For example, if the user fills in a form and hits 'submit' we don't want the user to be able to do anything on the page until the client-side web app has the opportunity to process the form. Typically this type of thing includes an overlay and/or a spinner. The idea behind the freeze strategy is to customize how an overlay and/or spinner

works and how it looks.

NOTE: You can also customize the style of the overlay and spinner by simply overriding/replacing the CSS file that comes with this library in the dist folder.

## 

# Future Items

This section contains items that are part of this initiative but will not be explored until after the initial solution as described above is delivered and widely available.

## Non-JS server rendering

We will get into this after the design for other components are more baked, but the key for non-JS rendering will be either statically analyzing a web app or potentially using something like [Nashorn](http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html) to execute JavaScript in Java.

## Packaging service

Once the main rendering solution design is in place, we can dive into the packaging service which would be used to implement lazy loading and lower the actual load time of the web app.

# Notes

This section contains notes and brainstorming from meetings that have not been fully integrated into the rest of the design doc yet. This section will eventually be removed.

Restrictions:

* support non JS backends, as they are the hardest and make the solution general
* should be online, assuming that the result is depending on request parameters that cannot be enumerated
  + e.g. because they contain user input, ...
* support pre rendering not only for initial page but for any view later on
* support lazy loading of the component code that will take over the pre rendered areas of the application
  + needs to capture and replay user events until the component has woken up
  + will reduce network size of the app
* support delayed wakeup of components
  + e.g. for a long thread of messages, we might only need to wake up the component that the user clicked on
  + triggers:
    - wake up on specified events
    - wake up on change of a specified expression
    - wake up when code for component has been loaded

Benchmark details

* make as little requests as possible -> concatenate source files
* estimate network time based on transferred amount to get stable numbers that also incorporate the network traffic
* to get the data, create benchmarks in Angular 2 and add metrics to benchpress

Main ideas:

* reuse template, but not application logic on the server
* from the loaded component's perspective it should be the same whether it was loaded via the router or prerendered
  + use same mechanism as the router for lazy loading (i.e. dynamic components)
  + use same mechanism for passing data to the component (e.g. programmatic databinding (not there yet), or a service)

Implications

* For every component that should be pre rendered there needs to be non JS code as well
* Non JS backend needs to generate 2 sets of data
  + one set of data that should be sent to the client for wakeup
  + one set of data that represents the correctly formatted data for all values in the view that should be rendered

Offline: extract metadata from directives and precompile templates

* this needs to run JS to calculate the values of the annotations (as they might contain List concatenation, …)
* this needs to be implemented in JS as it needs to evaluate the module to get the correct values for the annotations
* references to other directives (e.g. from the @Template annotation) need to be replaced with paths to these modules
  + imports of the module need to be analyzed...

Online: render using a generic render server

* Non JS backend loads data, sends directive metadata + precompiled templates + data to generic render server
* As render server is generic, it can be implemented in JS (e.g. can run in the cloud, parallelized, …)
  + as the logic is simple, it is possible to re implement it also in a Non JS language, although this would be more work
* Downside: Non JS backend needs to generate 2 sets of data
  + one set of data that should be sent to the client for wakeup
  + one set of data that represents the correctly formatted data for all values in the view that should be rendered

Development mode: verify that the 2 datasets from the non JS backend match up

* compare the values that the component sends to the renderer with the values that the non JS backend sent to the renderer during pre-rendering