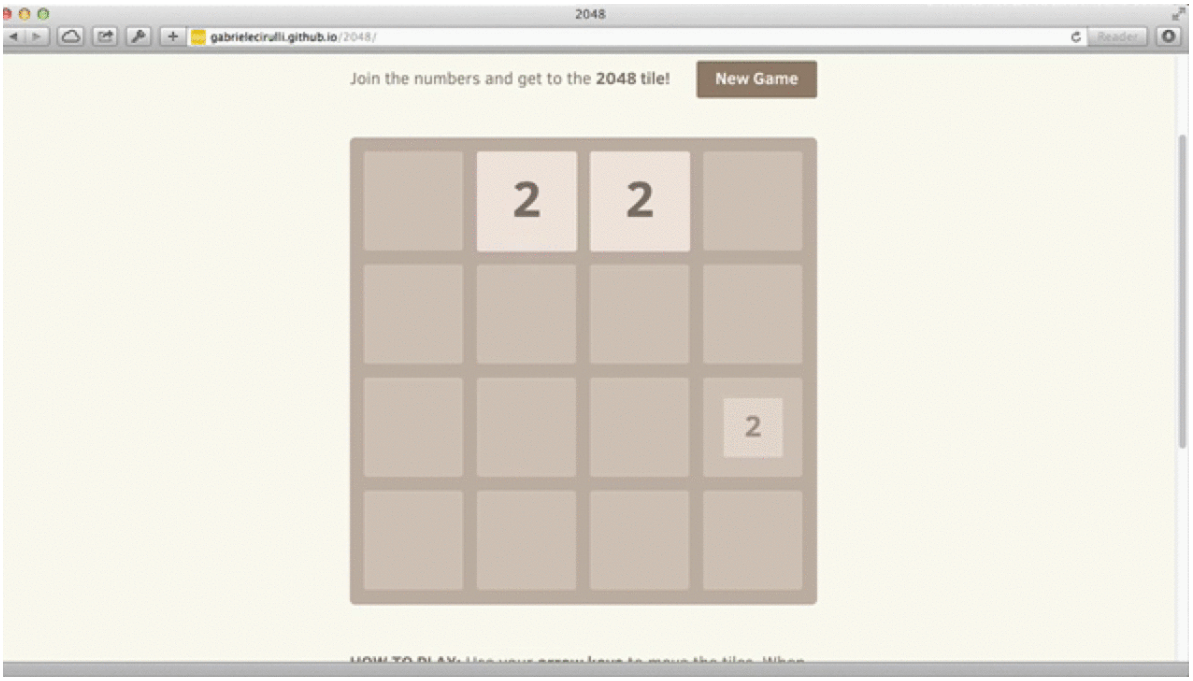
Building the 2048 game in AngularJS

Published on April 08, 2014

One of the most frequently asked questions we get is when would Angular be a poor choice to use as a framework. Our default answer is usually when writing games as Angular has it's own event loop handling (the `$digest` loop) and games usually require lots of low-level DOM manipulation. This is an inaccurate description as there are *many* types of games where Angular can support. Even games that require heavy DOM manipulation can use the angular framework for the static parts, such as tracking high scores and game menus.



Join the numbers and get to the **2048 tile!**

HOW TO PLAY: Use your arrow keys to move the tiles. When

If you are anything like me (and the rest of the tech industry), you may be addicted to the popular 2048 (http://gabrielecirulli.github.io/2048/) game. The objective of the game is to get to the 2048's tile by matching like-value tiles.

Discuss on HackerNews (https://news.ycombinator.com/item?id=7554348)

In today's post, we're going to build a clone of it in AngularJS, from start-to-finish, explaining the entire process of building the app. As this app is a fairly complex application, we intend this post to describe how to build complex AngularJS applications as well.

Here's the demo (http://d.pr/SnWD) of the app we're going to build in Angular.

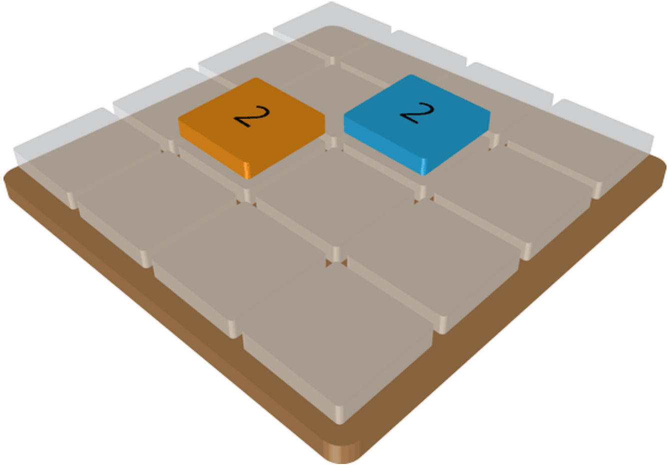
Buckle up and let's get going!

> **TL;DR:** the entire source for this application is available at on github in the link at the end of the article.

Index

First steps: planning the app

First steps: planning the app



The first step we like to do is to high-level design the application we are going to build. We do this no matter the size of the application, if we are cloning another application or creating one from scratch.

Looking at the game, we can see there is a game board with a bunch of *tiles* sitting atop it. Each one of the tiles themselves serves as a location for a numbered tile to be placed. We can use that fact to move the responsibility of placing the tiles to CSS3, rather than depend upon JavaScript needing to *know* where to place the tile. When we have a tile on the board, we'll simply make sure the tile is placed on top of the appropriate location.

Using CSS3 to layout the board gives us the ability to off-load the work of animations to CSS, but also use default AngularJS behaviors to keep track of the state of the board, the tiles, and the game logic.

Since we only have a single page, we will only need a single controller to manage the page.

Since there is only *one* game board for the duration of the app, additionally, we'll contain all of the grid logic in a single instance of a `GridService` service. Since services are singleton objects, this is an appropriate place to store the grid. We'll use the `GridService` to handle placing tiles, moving tiles, traversing the game grid for possible locations and managing the grid.

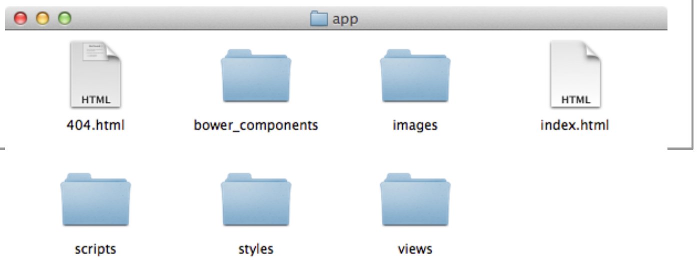
We'll store the game logic and processing inside of another service we'll call `GameManager` . The `GameManager` will be responsible for managing the stat of the game, handling the moves, and maintaining the scores (both current game score and high score).

Lastly, we'll need a component that will allow us to manage the keyboard. We'll use a service (we only need *one* action handler for the app) that we'll call `KeyboardService` . We'll implement handling in the app for the desktop in this article, but we can reuse this same service to manage touch actions to get it working on a mobile device.

Building the app

To build our app, we'll create a basic app (we used the yeoman (http://yeoman.io/) angular generator to generate the structure of our app, but it's not necessary. We only use it as a starting point, but quickly diverge from its structure). We'll create the app directory which will house the entire application. We'll place the `test/` directory as a sibling to the `app/` directory.

> The following instructions are for setting up the project using the yeoman tool. If you prefer to do it manually, you can skip installing the dependencies and move on to the next section.



Since we are using `yeoman` in our application, we'll first need to ensure it is installed. Yeoman is dependent upon NodeJS and npm being installed. Installation of NodeJS is outside the scope of this tutorial, but there are great instructions on the NodeJS.org (http://nodejs.org) site.

After `npm` has been installed, we can install the yeoman tool `yo` and the angular generator (the generator that will be used by the `yo` tool to create our Angular app):

```
$ npm install -g yo
$ npm install -g generator-angular
```

With these installed, we can create our application using the yeoman tool, as follows:

```
$ cd ~/Development && mkdir 2048
$ yo angular twentyfourtyeight
```

The tool will ask a few questions. We say yes to everything, except only select the `angular-cookies` as a dependency, since we won't need any of the other default dependencies it requires.

> Note that using the Angular generator, it will expect you have the compass gem installed along with a ruby environment. See the complete source for a way to get away without using ruby and compass below.

Our angular module

We'll create the `scripts/app.js` file to hold on to our application. Let's create the beginning of our application:

```
angular.module('twentyfourtyeightApp', [])
```

Modular structure

The recommended structure of laying out Angular apps is now by function, rather than by type. That is, instead of separating our components by controllers, services, directives, etc., we'll define our module structure based upon function. For instance, in our application we'll define a `Game` module and a `Keyboard` module.

The modular structure gives us a clean separation of responsibilities that match to the file structure. Not only does this help us build large, complex angular applications, it also helps us easily share function across our apps.

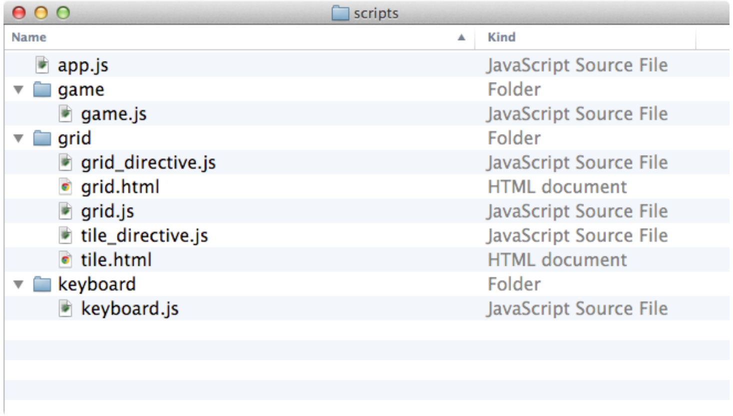
Later, we'll set up our testing environment to match the file directory structure.

The view

The easiest place to start in our application will be the view. Looking at the view itself, we only have one view/template. We won't need multiple views in this application, so we'll create a single `<div>` element that will hold the content of our application.

In our main `app/index.html` file, we'll need to include all of our dependencies (including `angular.js` itself as well as our JavaScript files – for now, this is simply `scripts/app.js`), like so:

```
<!-- index.html -->
<doctype html>
<html>
  <head>
    <title>2048</title>
    <link rel="stylesheet" href="style
  </head>
  <body ng-app="twentyfourtyeightApp"
    <!-- header -->
    <div class="container" ng-include="
    <!-- script tags -->
    <script src="bower_components/angu
    <script src="scripts/app.js"></scr
  </body>
</html>
```



Name	Kind
app.js	JavaScript Source File
▼ game	Folder
game.js	JavaScript Source File
▼ grid	Folder
grid_directive.js	JavaScript Source File
grid.html	HTML document
grid.js	JavaScript Source File
tile_directive.js	JavaScript Source File
tile.html	HTML document
▼ keyboard	Folder
keyboard.js	JavaScript Source File

> Feel free to make a more complex version of the game with multiple views – please leave a comment below if you do. We'd love to see what you create.

With our `app/index.html` file set, we will only need to detail with the views inside of our `app/views/main.html` for application-level views. We'll only need to modify the `index.html` file when we need to import a new resource in the application.

Cracking open our `app/views/main.html` , we'll place all of our game-specific views. Using the `controllerAs` syntax, we'll be able to be explicit where we expect to find data on our `$scope` and which controller is responsible for handling which component.

```
<!-- app/views/main.html -->
<div id="content" ng-controller='GameController as ctrl'>
  <!-- Now the variable: ctrl refers to the GameController -->
</div>
```

> The `controllerAs` syntax is a relatively *new* syntax that comes with version 1.2. It is useful when dealing with many controllers on the page as it allows us to be specific about the controllers where we expect functions and data to be defined.

In our view, at a minimum we'll want to surface a few items:

1. The static header of our game
2. The current game's score and the local user's highest score
3. The game board

The static header of the game can be as simple as:

```
<!-- heading inside app/views/main.html -->
<div id="content" ng-controller='GameController as ctrl'>
  <div id="heading" class="row">
    <h1 class="title">ng-2048</h1>
    <div class="scores-container">
      <div class="score-container">{{ ctrl.game.currentScore }}</div>
      <div class="best-container">{{ ctrl.game.highScore }}</div>
    </div>
  </div>
  <!-- ... -->
</div>
```

Notice that we're referencing the `GameController` when we're referencing the `currentScore` and `highScore` in the view. The `controllerAs` syntax enables us to explicitly referencing the controller we're interested in.

The GameController

Now that we have a reasonable project structure, let's create the `GameController` to hold on to the values we'll surface in the view. Inside `app/scripts/app.js` , we can create the controller on the main `twentyfourtyeightApp` module:

```
angular
.module('twentyfourtyeightApp', [])
.controller('GameController', function() {
});
```

In the view, we referenced a `game` object that we'll set on the `GameController` . The `game` object will referencing the main *game object*. We'll create this main game object in a new module we'll create that holds all references to the game.

Since this module isn't created yet, the app won't load in the browser quite yet. Inside the controller, we can add the `GameManager` dependency:

```
.controller('GameController', function(GameManager) {
  this.game = GameManager;
});
```

Remember, we are creating a modular level dependency with this different portion of our application, so in order to ensure it is loaded in our app, we'll need to list it as a dependency of our Angular module. To make the `Game` module a dependency for our `twentyfourtyeightApp` , we will list it in the array where we defined the module.

Our entire `app/scripts/app.js` file should now look like this:

```
angular
.module('twentyfourtyeightApp', ['Game'])
.controller('GameController', function(GameManager) {
  this.game = GameManager;
});
```

The Game

Now that we have the view partially hooked up to the view, we can start building the logic behind the game itself. To create a new game module, let's create our module in the `app/scripts/` directory as `app/scripts/game/game.js` :

```
angular.module('Game', []);
```

> When building modules, we like to write them in their own directory named after the module. We'll implement the module initialization in a file by the name of the module. For instance, we're building a *game* module, so we'll build our game module inside the `app/scripts/game` directory in a file named `game.js` . This methodology has provided to be scalable and logical in production.

The `Game` module will provide a single core component: the `GameManager` .

We'll write the `GameManager` such that it's responsible for keeping the state of the game, the different movements the user can make, keeping track of the scores, as well as determining when the game is over and if the user has beaten the game or the game has beaten the user.

When we develop our applications, we like to stub out the methods that we *know* we'll need, write tests for those methods and then fill in the blanks.

> For the purposes of this article, we'll run through this process for this module. When we write the next several modules, we'll only mention the core components we should be testing.

We know there are several features the `GameManager` we *know* at this point will support:

1. Creating a new game
2. Handling the game loop/move action
3. Updating the score
4. Keeping track if the game is done

With these features in mind, we can build the basic outline of the `GameManager` service that we can start writing tests against:

```
angular.module('Game', [])
.service('GameManager', function() {
  // Create a new game
  this.newGame = function() {};
  // Handle the move action
  this.move = function() {};
  // Update the score
  this.updateScore = function(newScore) {};
  // Are there moves left?
  this.movesAvailable = function() {};
});
```

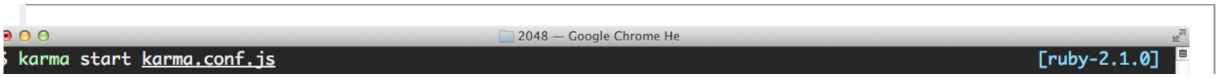
With the basic functionality complete, let's move over and write tests that define the barest of functionality we *know* our `GameManager` needs to support.

Test Driven Development (TDD)

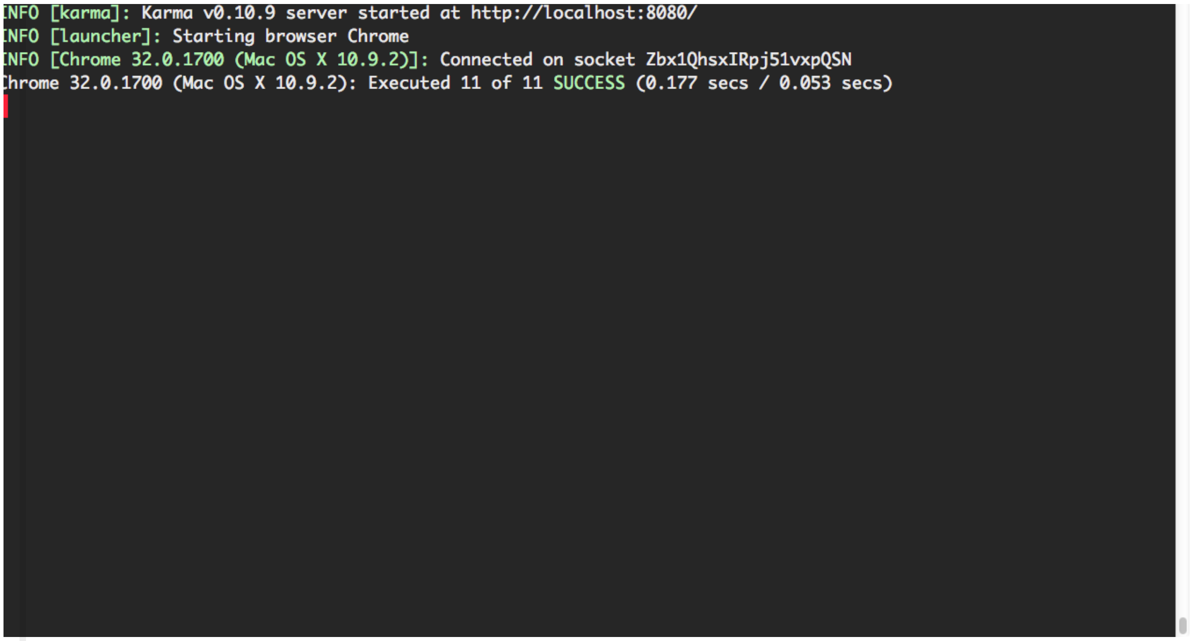
Before we can start implementing our tests, we'll need to set up karma to drive our tests. If you're unfamiliar with karma, it's simply a test runner that allows us to efficiently automate front-end tests from the comfort of our terminal and code.

To use Karma, we'll need to ensure it's installed. In order to use Karma, we're dependent upon NodeJS as it's distributed as an npm package. To install Karma, run the installation command:

```
$ npm install -g karma
```



```
$ karma start karma.conf.js                                    [ruby-2.1.0]
```



```
INFO [karma]: Karma v0.10.9 server started at http://localhost:8080/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 32.0.1700 (Mac OS X 10.9.2)]: Connected on socket Zbx1QhsxIRpj51vxpQSN
Chrome 32.0.1700 (Mac OS X 10.9.2): Executed 11 of 11 SUCCESS (0.177 secs / 0.053 secs)
```

> The `-g` flag tells npm to install the package globally. Without this flag, the package would only be installed locally in the current working directory.

If you built the app using the yeoman angular generator, you can skip this next part.

To use karma, we'll need to make a config file. Although we won't go in-depth into how to configure Karma here (check out ng-book (https://www.ng-book.com) for a highly detailed book on the options for configuring Karma), the crucial part of the process is setting Karma to load all of the files we're interested in testing.

To create a configuration file, we can use the `karma init` command to generate a basic one for us.

```
$ karma init karma.conf.js
```

The command will ask a few questions and generate the `karma.conf.js` file. From here, we'll change two configuration options: the `files` array and we'll turn on `autoWatch` :

```

  // ...
  files: [
    'app/bower_components/angular/angular.js',
    'app/bower_components/angular-mocks/angular-mocks.js',
    'app/bower_components/angular-cookies/angular-cookies.js',
    'app/scripts/**/*.js',
    'test/unit/**/*.js'
  ],
  autoWatch: true,
  // ...
```

With this configuration file set up, we can run our tests (which we will write in the `test/unit/` directory) anytime that we save any file in the directory.

To run the tests, we'll run the `karma start` command, like so:

```
$ karma start karma.conf.js
```

Writing our first tests

Now that we have karma set up and configured, we can write our basic tests for our `GameManager` . Since we do not yet know of the entire functionality of the application, we can only write a limited number of tests.

> Often times, we find that our API changes as we develop the application, so rather than introduce a lot of work ahead of time that we'll likely change, we set up our tests to test basic functionality and fill them in deeper as we uncover the eventual API.

A good candidate for the first test is the test that tells us if there are possible moves left. In order to test if there are moves left, we'll simply stub a few calls that we *know* we'll need, returning true/false to test the behavior of our application's logic.

We'll create a `test/unit/game/game_spec.js` file and start creating our test context:

```
describe('Game module', function() {
  describe('GameManager', function() {
    // Inject the Game module into this test
    beforeEach(module('Game'));

    // Our tests will go below here
  });
});
```

> In this test, we're using Jasmine (http://jasmine.github.io/2.0/introduction.html) syntax.

Like any other unit test, we'll need to create the instance of our `GameManager` object. We can do this in the usual manner (when testing services), by injecting it into our test:

```
  // ...
  // Inject the Game module into this test
  beforeEach(module('Game'));

  var gameManager; // instance of the GameManager
  beforeEach(inject(function(GameManager) {
    gameManager = GameManager;
  });

  // ...
```

With this instance of the `gameManager`, we can set up expectations of our `movesAvailable()` function.

We'll define the `movesAvailable()` function as a method that will check to see if there are any available squares left as well as to check to see if there are any possible merges available. Since this is a condition for the game to be over, we'll keep this method in the `GameManager`, but implement most of the gory details in the `GridService` that we'll create next.

For there to be any moves available on the board, we have to satisfy two conditions:

1. There are locations available on the board
2. There are match locations available

With these two conditions, we can write our test to check if these are satisfied.

The basic idea is that we'll write our tests such that we are setting up a condition that we want to see how our unit test performs under the circumstances. Since we will be relying upon the `GridService` to report the condition of the game board, we'll mock the condition and check to ensure our logic is correct in our `GameManager`.

Mocking the GridService

To mock the `GridService`, we will simply *override* the default Angular behavior and *provide* our mocked service, instead of the *real* one, so we can set up controlled conditions on the service.

To fill in some details, we'll simply be creating a fake object with mocked methods and then telling Angular that these fake objects are the *real* objects by switching them out with the `$provide` service.

```
  // ...
  var _gridService;
  beforeEach(module(function($provide) {
    _gridService = {
      anyCellsAvailable: angular.noop,
      tileMatchesAvailable: angular.noop
    };

    // Switch out the real GridService for our
    // fake version
    $provide.value('GridService', _gridService);
  }));
  // ...
```

Now we can use this fake `_gridService` instance to set up our conditions.

We'll want to ensure that the the `movesAvailable()` function reports true when there are cells available. Let's mock the `anyCellsAvailable()` method (that we haven't yet written) in our `GridService`. We expect this method to report that cells are available in the `Gridservice`.

```
// ...
describe('.movesAvailable', function() {
  it('should report true if there are cells available', function() {
    spyOn(_gridService, 'anyCellsAvailable').andReturn(true);
    expect(gameManager.movesAvailable()).toBeTruthy();
  });
  // ...
```

Now that the groundwork has been set, we can set up expectations for the second condition. If there are matches available, then we'll want to ensure that the `movesAvailable()` functions reports true. The converse is true as well that we'll want to ensure there are no moves available when there are neither cells nor matches available.

The other two tests confirming this are:

```
// ...
it('should report true if there are matches available', function() {
  spyOn(_gridService, 'anyCellsAvailable').andReturn(false);
  spyOn(_gridService, 'tileMatchesAvailable').andReturn(true);
  expect(gameManager.movesAvailable()).toBeTruthy();
});
it('should report false if there are no cells nor matches available', function() {
  spyOn(_gridService, 'anyCellsAvailable').andReturn(false);
  spyOn(_gridService, 'tileMatchesAvailable').andReturn(false);
  expect(gameManager.movesAvailable()).toBeFalsy();
});
// ...
```

Now we have the groundwork laid so that we can write our tests before we implement the behavior we expect.

> Although we aren't going to continue with TDD in this post, for the sake of overall completion, we suggest you should continue with it. Check out the full source code below for more tests.

Back to the GameManager

Now we have the task of implementing the `movesAvailable()` function. Since we can already verify that the code is working **and** we've identified the conditions it requires, implementing the function is easy.

```
// ...
this.movesAvailable = function() {
  return GridService.anyCellsAvailable() ||
          GridService.tileMatchesAvailable();
};
// ...
```

Building the game grid

Now that we have the `GameManager` up and running, we'll need to create the `GridService` to handle all the conditions of the board.

Recalling how we'll deal with the board itself, with two arrays, a base `grid` and a base `tiles` array, we can set up our `GridService`

Recalling how we'll deal with the board itself, with two arrays, a base `grid` and a base `tiles` array, we can set up our `GridService` with these two local variables. Inside our `app/scripts/grid/grid.js` file, let's create the service:

```
angular.module('Grid', [])
.service('GridService', function() {
  this.grid   = [];
  this.tiles  = [];
  // Size of the board
  this.size   = 4;
  // ...
});
```

When we want to create a new game, we'll need to populate these arrays to contain null elements. Our `grid` array can be static, as it's really only DOM elements used to place squares on the game board.

The `tiles` array, on the other hand will be dynamic and will keep track of the current tiles in-play. To use these different conditions, let's build the grid on the page so that we can start to see how they will lay out.

Back to our `app/views/main.html` , we'll need to start to lay out the grid. Since it's dynamic and we'll have our own logic inside the grid, it is only logical that we should place it in its own directive. Using a directive, will allow us to keep the main template clean as well as to encapsulate functionality inside the directive and keep our main controller clean.

Inside the `app/index.html` , we'll place the grid directive and pass it the `GameManager` instance on our controller:

```
  <!-- instructions -->
  <div id="game-container">
    <div grid ng-model='ctrl.game' class="row"></div>
    <!-- ... -->
```

We'll write this directive such that it's contained in the `Grid` module, so inside our `app/scripts/grid/` folder, let's create a `grid_directive.js` file to house our `grid` directive.

Inside the `grid` directive, we'll only need a few variables as it's responsibility is quite limited to encapsulating the view.

The directive will need to have an instance of the `GameManager` (or at least, a model that has the `grid` and `tiles` arrays), so we'll set it as a requirement for the directive. Additionally, we don't want our directive messing with any of the rest of the page or the GameManager instance itself on the page, so we'll create it using an `isolate` scope.

> Check out our post on custom directives (/posts/directives.html) for a deep dive into directives, or check out ng-book (https://www.ng-book.com) for advanced details on directives.

```
angular.module('Grid')
.directive('grid', function() {
  return {
    restrict: 'A',
    require: 'ngModel',
    scope: {
      ngModel: '='
    },
    templateUrl: 'scripts/grid/grid.html'
  };
});
```

The primary responsibility this directive will take on is setting up the grid view, so we won't need any custom logic in the directive.

grid.html

Inside the directive's template, we'll run through two `ngRepeat` to show both the grid and the tiles array and (for now) track each using their `$index` in the repeat.

```
<div id="game">
  <div class="grid-container">
    <div class="grid-cell"
      ng-repeat="cell in ngModel.grid track by $index">
      </div>
  </div>
  <div class="tile-container">
  <div tile
   ng-model='tile'
   ng-repeat='tile in ngModel.tiles track by $index'>
  </div>
</div>
</div>
```

The first `ng-repeat` is fairly straight-forward in that it simply iterates over the grid array and places a single, empty div with a class of `grid-cell`.

In the second `ng-repeat`, we'll create a secondary directive for each element placed on the screen we'll call `tile`. The `tile` directive will be responsible for creating each of the tile element's visual display. We'll create our `tile` directive shortly…

The astute reader might see that we're only using a one-dimensional array to display a two-dimensional grid. When we render our view, we'll get a single column of 'tiles', rather than a grid.

To make them grids, we'll need to dive into the CSS.

Enter SCSS

For this project, we'll be using the modern variant of SASS: scss. Not only is scss a much more powerful CSS, we'll be building our CSS in a dynamic way.

The main portion of this app's visual elements will be done using CSS, including animations as well as layout and visual elements (colors of the tiles, etc).

To create the board, in a two-dimensional way, we'll use the CSS3 keyword: `transform` to handle where on the board each specific tile will be placed.

CSS3 transform property

The CSS3 transform property is a property that allows us to apply 2D or 3D transformations on to an element. It allows us to move, skew, rotate, scale, and more to elements (and is animateable). Using this property, we can simply place the tile on the board and apply the proper `transform` to the element.

For instance, in the following demo, we have a box with a width of 40px and a height of 40px.

```
.box {
  width:40px;
  height:40px;
  background-color: blue;
}
```

If we apply a `transform` property of `translateX(300px)`, we'll have moved the box 300px to the left, as the demo below demonstrates:

```
.box.transformed {
  -webkit-transform: translateX(300px);
  transform: translateX(300px);
}
```

Using this translate property, we can move our tiles around the board simply by applying a CSS class to them. Now, the tricky part is how do we build our dynamic classes such that they correspond to a proper grid location using CSS when we variable locations on our page?

This is where the power of SCSS starts to shine. We'll set up some variables (such as how many tiles we want per row) and build our

SCSS around these variables, using some math to do the calculations for us.

Let’s look at the variables we’ll need to properly calculate positioning on the board:

```

$width: 400px;          // The width of the whole board
$tile-count: 4;         // The number of tiles per row/column
$tile-padding: 15px;    // The padding between tiles
```

With these three variables, we can let SCSS dynamically calculate the positioning for us. First, we’ll need to calculate the size of each tile. This is fairly trivial to do with SCSS variables:

```
$tile-size: ($width - $tile-padding * ($tile-count + 1)) / $tile-count;
```

Now we can set up the `#game` container with the appropriate height and width. We’ll also set the positioning on the `#game` container such that we can absolutely position our child elements contained inside it. We’ll position our `.grid-container` and `.tile-container` inside the `#game` container object.

We’ve included only the relevant parts of the scss here. The rest of the source can be found on github at the end of the article.

```
#game {
  position: relative;
  width: $width;
  height: $width; // The gameboard is a square

  .grid-container {
    position: absolute;   // the grid is absolutely positioned
    z-index: 1;           // IMPORTANT to set the z-index for layering
    margin: 0 auto;       // center

    .grid-cell {
      width: $tile-size;              // set the cell width
      height: $tile-size;             // set the cell height
      margin-bottom: $tile-padding;   // the padding between lower cells
      margin-right: $tile-padding;    // the padding between the right cell
      // ...
    }
  }
  .tile-container {
    position: absolute;
    z-index: 2;

    .tile {
      width: $tile-size;        // tile width
      height: $tile-size;       // tile height
      // ...
    }
  }
}
```

Note that in order for the `.tile-container` to sit above the `.grid-container` , we **must** set the `z-index` as a higher value for the `.tile-container` . If we don’t set the `z-index` , the browser would place them at the same height and it wouldn’t look very pretty.

With these set, we can now dynamically generate the positioning of the tiles. What we need is a `.position-{x}-{y}` class that we can assign to a tile so that the browser will see the positioning of the tile and dynamically place it there. Since we’ll be calculating the transformation properties relative to the grid container, we will use `0,0` as the initial location for the first tile.

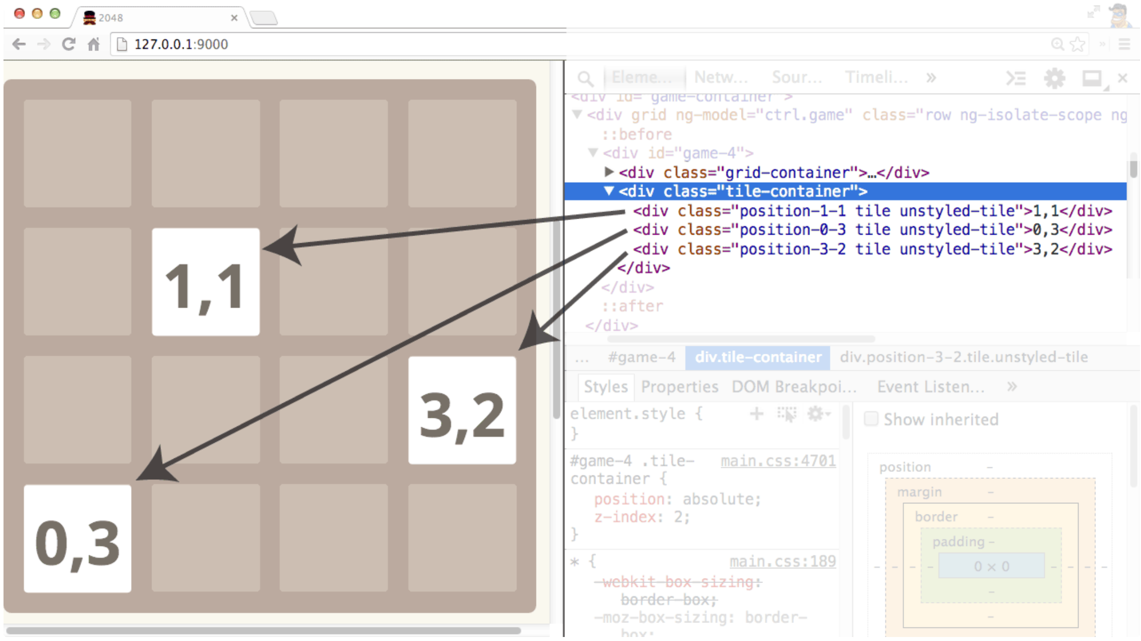
We’ll iterate over the square and dynamically create each class, based upon the expected offset we calculate:

```
.tile {
  // ...
  // Dynamically create .position-#{x}-#{y} classes to mark
  // where each tile will be placed
  @for $x from 1 through $tile-count {
    @for $y from 1 through $tile-count {
      $zeroOffsetX: $x - 1;
      $zeroOFfsetY: $y - 1;
      $newX: ($tile-size) * ($zeroOffsetX) + ($tile-padding * $zeroOffsetX);
      $newY: ($tile-size) * ($zeroOffsetY) + ($tile-padding * $zeroOffsetY);

      &.position-#{$zeroOffsetX}-#{$zeroOffsetY} {
        -webkit-transform: translate($newX, $newY);
        transform: translate($newX, $newY);
      }
    }
  }
  // ...
}
```

> Note that we have to calculate our positioning using 1-based offsets, instead of the traditional 0-indexed. This is a limitation imposed by SASS itself. We easily handle it by subtracting 1 from the index.

Now that we have created dynamic `.position-#{x}-#{y}` CSS classes, we can lay out our tiles on screen.


2048
127.0.0.1:9000
1,1
3,2
0,3
Eleme... Netw... Sour... Timeli... »
... #game-4 div.tile-container div.position-3-2.tile.unstyled-tile
Styles Properties DOM Breakpoi... Event Listen... »
Show inherited
position
margin
border
padding
0 × 0

```
<div id="game-container">
▼<div grid ng-model="ctrl.game" class="row ng-isolate-scope ng
   ::before
  ▼<div id="game-4">
    ▶<div class="grid-container">…</div>
    ▼<div class="tile-container">
        <div class="position-1-1 tile unstyled-tile">1,1</div>
        <div class="position-0-3 tile unstyled-tile">0,3</div>
        <div class="position-3-2 tile unstyled-tile">3,2</div>
     </div>
   </div>
   ::after
 </div>
```

```
element.style {
}
#game-4 .tile-container {        main.css:4701
  position: absolute;
  z-index: 2;
}
* {                              main.css:189
  ~~-webkit-box-sizing: border-box;~~
  -moz-box-sizing: border-box;
```

Coloring the different tiles

Notice that when different tiles appear on the screen, each tile is a different color. These different colors identify the *value* which the tile itself holds. It's an easy way for players to see the different states at which the tiles are at. Using the same technique as we did when we iterated over the number of tiles to create a tile color scheme.

To accomplish creating the color scheme, we'll first need to create a SCSS array that holds the background colors for each of the colors we'll use on the screen. Each one of these colors will

```

$colors:  #EEE4DA, // 2
          #EAE0C8, // 4
          #F59563, // 8
          #3399ff, // 16
          #ffa333, // 32
          #cef030, // 64
          #E8D8CE, // 128
          #990303, // 256
          #6BA5DE, // 512
          #DCAD60, // 1024
          #B60022; // 2048
```

With our `$colors` array variable, we can simply iterate over each of the colors and dynamically create a class based upon the tile value. That is, when a tile has a value of 2, we'll add the CSS class of `.tile-2`, which will have the background color of `#EEE4DA`. Rather than hard-coding this for every tile, we'll use the magic of SCSS to handle it for us:

```
@for $i from 1 through length($colors) {
  &.tile-#{power(2, $i)} .tile-inner {
    background: nth($colors, $i)
  }
}
```

Of course, we’ll need to define the `power()` mixin. That’s defined as:

```
@function power ($x, $n) {
  $ret: 1;

  @if $n >= 0 {
    @for $i from 1 through $n {
      $ret: $ret * $x;
    }
  } @else {
    @for $i from $n to 0 {
      $ret: $ret / $x;
    }
  }

  @return $ret;
}
```

The Tile directive

With the gritty work of the SASS done, we can return to our tile directive and lay out each tile with the dynamic positioning, allowing CSS to work the way it was designed and let the tiles fall into place.

Since the `tile` directive is a container for a custom view, we don’t need it to do very much. We will need to have access to the cell it’s responsible for displaying. Beyond that, there isn’t any other functionality that we’ll need to place in the directive. The code is self explanatory:

```
angular.module('Grid')
.directive('tile', function() {
  return {
    restrict: 'A',
    scope: {
      ngModel: '='
    },
    templateUrl: 'scripts/grid/tile.html'
  };
});
```

Now, the interesting part of the `tile` directive is how we’re laying out the grid dynamically. This is all taken care of in the template using the `ngModel` variable that’s on our isolate scope. As we can see above, this refers to the tile object that comes from our `tiles` array.

```
<div ng-if='ngModel' class="tile position-{{ ngModel.x }}-{{ ngModel.y }} tile-{{ ngModel.value }}">
  <div class="tile-inner">
    {{ ngModel.value }}
  </div>
</div>
```

With this basic directive, we’re almost ready to show it on-screen. For all tiles with an `x` and a `y` coordinate, they will *automatically* get assigned their corresponding `.position-#{x}-#{y}` class and the browser will *automatically* place them in the expected location.

This means that our tile objects will need an `x` , `y` , and a `value` available for the directive to function. For that, we’ll need to create a new object for each tile we’ll lay on the screen.

The TileModel

Rather than creating a *dumb* object, we’ll create a smarter one that not only contains data, but also functionality.

Since we’ll want to be able to use Angular’s dependency injection, we’ll create a service that will house our data model. We’ll create a `TileModel` service in the `Grid` module as it will only be necessary to use the low-level `TileModel` when in relation to the game board.

Using the `.factory` method, we can simply create a function that we’ll assign as the factory. Unlike the `service()` function which assumes the function we’re using it to define the service is the constructor for that service, the `factory()` method will assign the return value of the function to be the service object. Thus, using the `factory()` method, we can assign any object as a service to *inject* around our Angular apps.

In our `app/scripts/grid/grid.js` file, we can create our `TileModel` factory:

```
angular.module('Grid')
.factory('TileModel', function() {
  var Tile = function(pos, val) {
    this.x = pos.x;
    this.y = pos.y;
    this.value = val || 2;
  };

  return Tile;
})
// ...
```

Now, anywhere in our Angular app, we can *inject* the `TileModel` and use it as though it were a global object. Pretty nifty, right?

> Don’t forget to write tests for any functionality we place on the `TileModel`.

Our first grid

Now that we have our `TileModel`, we can start placing instances of the `TileModel` in our `tiles` array and they will *magically* appear in the right place on the grid.

Let’s try adding a few Tile instances to the `tiles` array in our `GridService`:

```
angular.module('Grid', [])
.factory('TileModel', function() {
  // ...
})
.service('GridService', function(TileModel) {
  this.tiles  = [];
  this.tiles.push(new TileModel({x: 1, y: 1}, 2));
  this.tiles.push(new TileModel({x: 1, y: 2}, 2));
  // ...
});
```

The Board’s ready for the game

Now that we can place tiles on the screen, we need to create the functionality that will prepare the board for us in our `GridService`. When we first load the page, we’ll want to create an empty game board. We’ll also want to do the same action when the user presses on the `New Game` or `Try again` buttons during game place.

To clear out the board, we’ll create a new function that we’ll call `buildEmptyGameBoard()` on our `GridService`. This method will be responsible for populating the `grid` and the `tiles` array in the `GridService` with null values.

Before we write our code, we’ll write our tests to confirm the behavior of the `buildEmptyGameBoard()` function. As we’ve covered this process above, we won’t discuss the process, but the result. The tests might look something like:

```
// In test/unit/grid/grid_spec.js
// ...
describe('.buildEmptyGameBoard', function() {
  var nullArr;

  beforeEach(function() {
    nullArr = [];
    for (var x = 0; x < 16; x++) {
      nullArr.push(null);
    }
  })
  it('should clear out the grid array with nulls', function() {
    var grid = [];
    for (var x = 0; x < 16; x++) {
      grid.push(x);
    }
    gridService.grid = grid;
    gridService.buildEmptyGameBoard();
    expect(gridService.grid).toEqual(nullArr);
  });
  it('should clear out the tiles array with nulls', function() {
    var tiles = [];
    for (var x = 0; x < 16; x++) {
      tiles.push(x);
    }
    gridService.tiles = tiles;
    gridService.buildEmptyGameBoard();
    expect(gridService.tiles).toEqual(nullArr);
  });
});
```

Now that we have the test, we can implement the functionality of our `buildEmptyGameBoard()` function.

The function is small enough, where the code itself is self-documenting enough. In `app/scripts/grid/grid.js`

```
.service('GridService', function(TileModel) {
  // ...
  this.buildEmptyGameBoard = function() {
    var self = this;
    // Initialize our grid
    for (var x = 0; x < service.size * service.size; x++) {
      this.grid[x] = null;
    }

    // Initialize our tile array
    // with a bunch of null objects
    this.forEach(function(x,y) {
      self.setCellAt({x:x,y:y}, null);
    });
  };
  // ...
```

The code above uses a few helper methods that are self-explanatory enough as to their intended use. A few helper functions we'll use throughout our entire program are listed here and are all self-explanatory:

```
// Run a method for each element in the tiles array
this.forEach = function(cb) {
  var totalSize = this.size * this.size;
  for (var i = 0; i < totalSize; i++) {
    var pos = this._positionToCoordinates(i);
    cb(pos.x, pos.y, this.tiles[i]);
  }
};
```

```

// Set a cell at position
this.setCellAt = function(pos, tile) {
  if (this.withinGrid(pos)) {
    var xPos = this._coordinatesToPosition(pos);
    this.tiles[xPos] = tile;
  }
};

// Fetch a cell at a given position
this.getCellAt = function(pos) {
  if (this.withinGrid(pos)) {
    var x = this._coordinatesToPosition(pos);
    return this.tiles[x];
  } else {
    return null;
  }
};

// A small helper function to determine if a position is
// within the boundaries of our grid
this.withinGrid = function(cell) {
  return cell.x >= 0 && cell.x < this.size &&
          cell.y >= 0 && cell.y < this.size;
};
```

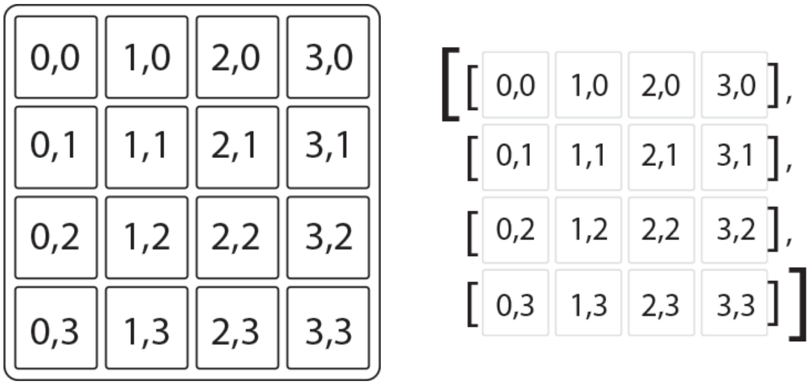
What in the world?!??

What are these two functions we used `this._positionToCoordinates()` and `this._coordinatesToPosition()` ?

Recall above we discussed that we'll use a one-dimensional array to lay out our grid. This is preferable for both performance reasons as well as for handling the complex animations of the app. We will explore animations later in this section. For the time being, we only benefit from the complexities of using a single array to represent a multi-dimensional array.

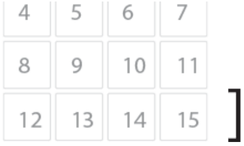
Multi-dimensional array in one dimension

How can we represent a multi-dimensional array in a single array? Let's look at the grid that represents out game board without any color, with their cell locations as the value. In code, this multi-dimensional array breaks down into an array of arrays:



Looking at each of the cell locations, we can see that a pattern emerges when looking at it from a single array view:

We can see that the first cell, at `(0,0)` maps to the cell location of `0`. The second array location at `1` points to the cell location at `(1,0)`. Moving to the next row, the cell location at `(0,1)` points to the 4th element in the one-dimensional array, whereas the element at the index of 5



4	5	6	7
8	9	10	11
12	13	14	15

points to the 4th element in the one-dimensional array, whereas the element at the index of 5 points to `(1,1)`.

Extrapolating the relationship between the the locations, we can see that an equation emerges about the relationships between the two.

i = x + ny

Where `i` is the cell index, `x` and `y` are the locations in the multi-dimensional array, and `n` is the number of cells per-row/column.

The two helper functions we defined above convert the cell location to an x-y coordinate system and visa versa. Conceptually, it is easier to deal with cell locations as x-y coordinates, but functionality we'll be setting every tile in our single-dimensional array.

```
// Helper to convert x to x,y
this._positionToCoordinates = function(i) {
  var x = i % service.size,
      y = (i - x) / service.size;
  return {
    x: x,
    y: y
  };
};

// Helper to convert coordinates to position
this._coordinatesToPosition = function(pos) {
  return (pos.y * service.size) + pos.x;
};
```

Initial player positions

Now, at the outset of a new game we'll want to set some starting pieces. We'll *randomly* choose these starting places on the game board for our players.

```
.service('GridService', function(TileModel) {
  this.startingTileNumber = 2;
  // ...
  this.buildStartingPosition = function() {
    for (var x = 0; x < this.startingTileNumber; x++) {
      this.randomlyInsertNewTile();
    }
  };
  // ...
```

Building a starting position is pretty easy as it only calls the `randomlyInsertNewTile()` function for the number of starting tiles we want to place. The `randomlyInsertNewTile()` function requires us to know about all of the available possible locations we can randomly place a tile. This functionality is easy to implement as all we will need to do is walk through our single array and keep track of the positions in the array that do not already have a tile placed in it.

```
.service('GridService', function(TileModel) {
  // ...
  // Get all the available tiles
  this.availableCells = function() {
    var cells = [],
        self = this;

    this.forEach(function(x,y) {
      var foundTile = self.getCellAt({x:x, y:y});
      if (!foundTile) {
        cells.push({x:x,y:y});
      }
    });

    return cells;
  };
  // ...
  // ...
```

With a list of all available coordinates on the board, we can simply pick a random location from within the array. Our `randomAvailableCell()` function will handle doing this for us. We can implement this a few different ways. Here's how we implemented it in 2048:

```
.service('GridService', function(TileModel) {
  // ...
  this.randomAvailableCell = function() {
    var cells = this.availableCells();
    if (cells.length > 0) {
      return cells[Math.floor(Math.random() * cells.length)];
    }
  };
  // ...
```

From here, we can simply create a new TileModel instance and insert it into our `this.tiles` array.

```

.service('GridService', function(TileModel) {
  // ...
  this.randomlyInsertNewTile = function() {
    var cell = this.randomAvailableCell(),
        tile = new TileModel(cell, 2);
    this.insertTile(tile);
  };

  // Add a tile to the tiles array
  this.insertTile = function(tile) {
    var pos = this._coordinatesToPosition(tile);
    this.tiles[pos] = tile;
  };

  // Remove a tile from the tiles array
  this.removeTile = function(pos) {
    var pos = this._coordinatesToPosition(tile);
    delete this.tiles[pos];
  }
  // ...
});
```

Now, by virtue of the fact that we're using Angular, our grid piece will just magically show up as a tile on the game board in our view.

> Remember, the smart thing to do next is write tests to test our assumptions about the functionality. We found several bugs through the process of writing tests for this project and so will you.

Keyboard interaction

Great, now we have our tile pieces on the board. What fun is a game that you can't actually play? Let's shift our attention to adding interaction to the game.

> For the purposes of this post, we're only going to focus on keyboard interaction and leave touch actions aside. However, adding touch actions shouldn't be hard to do, especially because we're only interested in swipe actions, which `ngTouch` provides. We leave this up for you the implement.

The game itself is played through the use of the arrow keys (or the a, w, s, d keys). In our game, we want to allow the user to simply be on the page to interact with the game. As opposed to requiring the user to focus on the game board element (or any other element on the page, for that matter). This will allow the user to interact with the game with only having to have the document in focus.

In order to allow this type of interaction for the user is to attach an event listener to the document. In Angular, we will `bind` our event listener to the `$document` service provided by Angular. To handle defining the user interactions, we'll wrap our keyboard event bindings in a service. Remember, we only need one keyboard handler on the page, so a service is perfect.

Additionally, we'll also want to set custom actions to happen whenever we detect a keyboard action from our user. Using a service will allow us to naturally inject it into our angular objects and act upon user input.

To begin with, we'll create a new module (as we are doing modular-based development) that we'll call `Keyboard` in the `app/scripts/keyboard/keyboard.js` file (we'll need to create it, if it doesn't already exist).

```
// app/scripts/keyboard/keyboard.js
angular.module('Keyboard', []);
```

As with any new JavaScript we create, we'll need to reference it in our `index.html` file. The list of `<script>` tags now looks like:

```
  <!-- body -->
  <script src="scripts/app.js"></script>
  <script src="scripts/grid/grid.js"></script>
  <script src="scripts/grid/grid_directive.js"></script>
  <script src="scripts/grid/tile_directive.js"></script>
  <script src="scripts/keyboard/keyboard.js"></script>
  <script src="scripts/game/game.js"></script>
</body>
</html>
```

And, as we are creating a new module, we'll also need to tell our Angular module that we want to use this new module as a dependency to our own application:

```
.module('twentyfourtyeightApp', ['Game', 'Grid', 'Keyboard'])
```

The idea behind the `Keyboard` service, we'll `bind` the `keydown` event on the `$document` to capture the user interaction component from the document. On the other end, in our angular objects, we'll register handler functions that will be called when there is a user interaction.

Let's get to it.

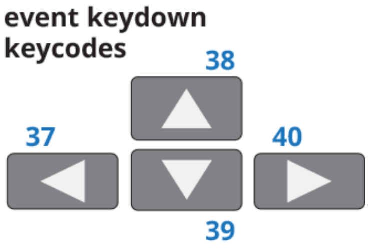
```
// app/scripts/keyboard/keyboard.js
angular.module('Keyboard', [])
.service('KeyboardService', function($document) {

  // Initialize the keyboard event binding
  this.init = function() {
  };

  // Bind event handlers to get called
  // when an event is fired
  this.keyEventHandlers = [];
  this.on = function(cb) {
  };
});
```

The `init()` function will kick off the `KeyboardService` to start listening for keyboard events. We'll filter out any `keydown` events fired that we are uninterested in.

For any events that get fired that are interesting to us, we'll prevent the default action to run and send it off to our `keyEventHandlers`.



How do we know what events are interesting to us? Since we are interested in only a limited number of keyboard actions, we can check to see if the event has been fired by one of these *interesting* keyboard events.

When the arrow keys are pressed, the document will receive an event with the key code for the keyboard key that was pressed.

We can create a map of these events and check for the existence of the offending key action to see if it is in this *interesting* map.

```
// app/scripts/keyboard/keyboard.js
angular.module('Keyboard', [])
.service('KeyboardService', function($document) {

  var UP    = 'up',
      RIGHT = 'right',
      DOWN  = 'down',
      LEFT  = 'left';

  var keyboardMap = {
    37: LEFT,
    38: UP,
    39: RIGHT,
    40: DOWN
  };

  // Initialize the keyboard event binding
  this.init = function() {
    var self = this;
    this.keyEventHandlers = [];
    $document.bind('keydown', function(evt) {
      var key = keyboardMap[evt.which];

      if (key) {
        // An interesting key was pressed
        evt.preventDefault();
        self._handleKeyEvent(key, evt);
      }
    });
  };
  // ...
});
```

Anytime a key in our `keyboardMap` fires a `keydown` event, the `KeyboardService` will run the `this._handleKeyEvent` function.

This function's entire responsibility is to call every event handler per registered key handler. It will simply iterate over the array of key handlers and call each one with both the key as well as the raw event:

```
// ...
this._handleKeyEvent = function(key, evt) {
  var callbacks = this.keyEventHandlers;
  if (!callbacks) {
    return;
  }

  evt.preventDefault();
  if (callbacks) {
    for (var x = 0; x < callbacks.length; x++) {
      var cb = callbacks[x];
      cb(key, evt);
    }
  }
};
// ...
```

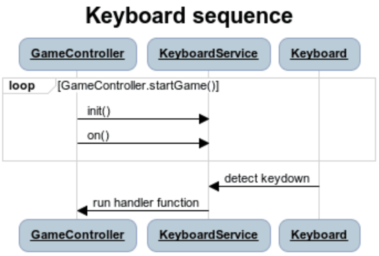
On the other side, we only need to push our handler function into our list of handlers.

```
// ...
this.on = function(cb) {
  this.keyEventHandlers.push(cb);
};
// ...
```

Using the Keyboard service

Now that we have the ability to watch for keyboard events from the user, we need to kick it off when our app starts. Since we built it

Now that we have the ability to watch for keyboard events from the user, we need to kick it off when our app starts. Since we built it as a service, we'll simply do this inside of the main controller.



First, we'll need to call the `init()` function to start listening on the keyboard. Second, we'll register our function handler to call to the `GameManager` to call the `move()` function.

Back in our `GameController`, we'll add the `newGame()` function and the `startGame()` functions. The `newGame()` function will simply call to the game service to create a new game and kick off the keyboard event handler.

Let's get to the code! We need to *inject* the `Keyboard` module as a new modular dependency for our application:

```
angular.module('twentyfourtyeightApp', ['Game', 'Keyboard'])
// ...
```

Now we can inject the `KeyboardService` into our `GameController` and start off the user interaction. First, the `newGame()` method:

```
// ... (from above)
.controller('GameController', function(GameManager, KeyboardService) {
  this.game = GameManager;

  // Create a new game
  this.newGame = function() {
    KeyboardService.init();
    this.game.newGame();
    this.startGame();
  };

  // ...
```

We haven't yet defined the `newGame()` method on the `GameManager`, we'll flesh it out shortly.

Once we've created our new game, we'll call `startGame()`. The `startGame()` function will set up the keyboard service event handler:

```
.controller('GameController', function(GameManager, KeyboardService) {
  // ...
  this.startGame = function() {
    var self = this;
    KeyboardService.on(function(key) {
      self.game.move(key);
    });
  };

  // Create a new game on boot
  this.newGame();
});
```

Press the start button

We've done a lot of work to get us to this point: **starting the game**. The last method we need to implement is the `newGame()` method in our `GameManager` that will:

1. build an empty game board
2. set up the starting positions
3. initialize the game

We've already implemented this logic inside of our `GridService`, so now it's a matter of hooking it up!

We've already implemented this logic inside of our `GridService`, so now it's a matter of hooking it up:

In our `app/scripts/game/game.js` file, let's add the `newGame()` function. This function will reset our game stats back to the expected start conditions:

```
angular.module('Game', [])
.service('GameManager', function(GridService) {
  // Create a new game
  this.newGame = function() {
    GridService.buildEmptyGameBoard();
    GridService.buildStartingPosition();
    this.reinit();
  };

  // Reset game state
  this.reinit = function() {
    this.gameOver = false;
    this.win = false;
    this.currentScore = 0;
    this.highScore = 0; // we'll come back to this
  };
});
```

Loading this page up in our browser, we'll have our functional grid… it's pretty boring at this point as we haven't defined any of the move functionality.

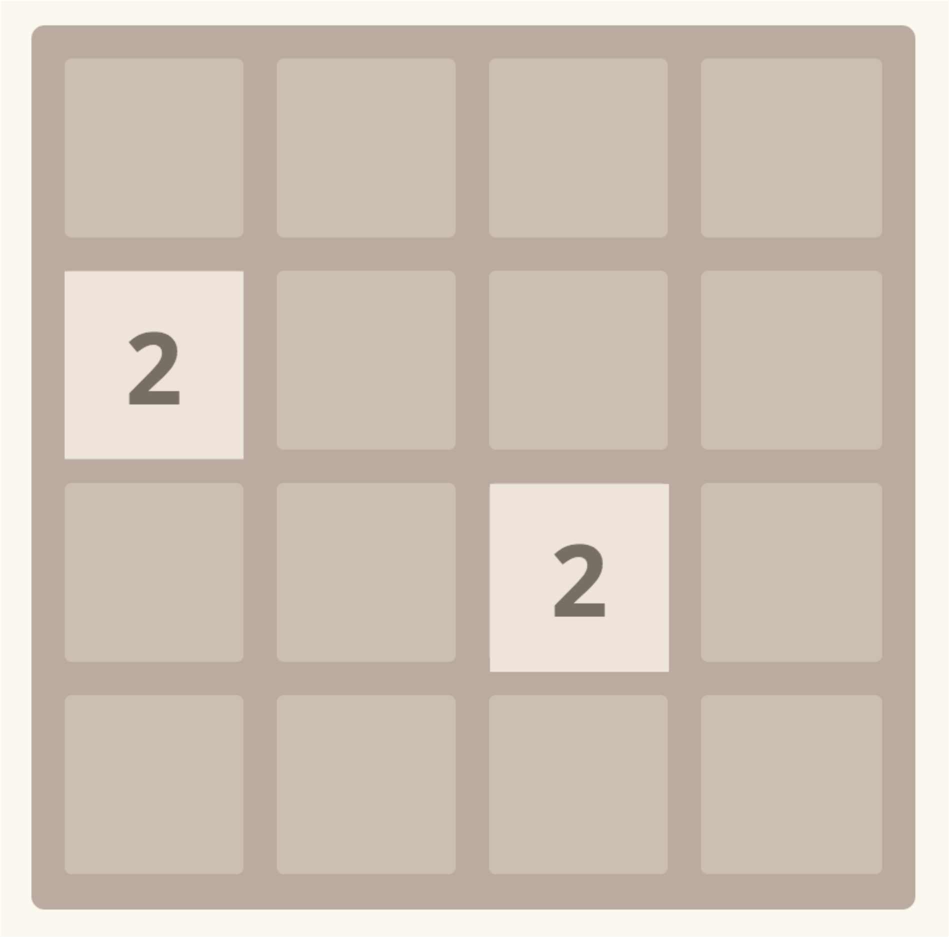
Get your move on (the game loop)

Now let's get into the meat of the functionality of our game. When the user presses on any of the arrow keys, we will call the `move()` function on the `GridService` (we built this within the `GameController`).

To build the `move()` function, we'll need to define the constraints of our game. That is, we'll need to define how our game will act at every given move.

For every move, we'll need to:

1. Determine the *vector* that the user's arrow key indicates.
2. Find all of the furthest possible locations for every tile on the board. At the same time, grab the tile in the next location to see if we can *merge*.
3. For every tile, we'll want to determine if there is a *next* tile with the same value.
    1. If there is not a next tile, then we'll simply move the tile to the furthest-away possible position. (This means the further location is on the edge of the board).
    2. If there is a next tile:
    3. and the value is a different value than the current



tile, then we'll move the tile to the furthest away location (the next tile is the boundary of movement for our current tile).

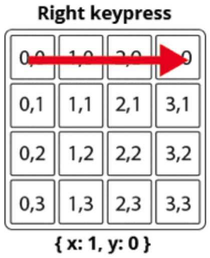
4. and the value is equal to the current tile, we've found a possible merge.
    1. If the tile has already been the result of a merge, then we'll skip it and consider it *used*.
    2. If the tile has not been merged yet, then we'll consider this a merge.

Now that we've defined the functionality, we can lay out the strategy for building the `move()` function.

```
angular.module('Game', [])
.service('GameManager', function(GridService) {
  // ...
  this.move = function(key) {
    var self = this; // Hold a reference to the GameManager, for later
    // define move here
    if (self.win) { return false; }
  };
  // ...
});
```

A few conditions for movement: if the game is over and we've somehow ended up in the game loop, we'll simply return and move on.

Next we'll need to traverse the grid and find all of the possible locations. As this is the responsibility of the grid to *know* where it's open locations are, we'll create a new function on the `GridService` that will help us find all the possible traversal locations.



In order to find the direction, we'll need to pick out the *vector* that the user's keypress indicates. For instance, if the user presses the right arrow key, then the user will want to move in *increasing* `x` locations.

If the user presses the up button, then the user wants tiles to move in the *decreasing* `y` location. We can map our vectors against the key the user pressed (which we get back from our `KeyboardService`) using a JavaScript object, like so:

```
// In our `GridService` app/scripts/grid/grid.js
var vectors = {
  'left': { x: -1, y: 0 },
  'right': { x: 1, y: 0 },
  'up': { x: 0, y: -1 },
  'down': { x: 0, y: 1 }
};
```

Now we'll simply iterate over the possible positions, using the vector to determine which direction we'll want to iterate over our potential positions:

```

.service('GridService', function(TileModel) {
  // ...
  this.traversalDirections = function(key) {
    var vector = vectors[key];
    var positions = {x: [], y: []};
    for (var x = 0; x < this.size; x++) {
      positions.x.push(x);
      positions.y.push(x);
    }
    // Reorder if we're going right
    if (vector.x > 0) {
      positions.x = positions.x.reverse();
    }
    // Reorder the y positions if we're going down
    if (vector.y > 0) {
      positions.y = positions.y.reverse();
    }
    return positions;
  };
  // ...
```

Now, with our new `traversalDirections()` defined, we can iterate over the possible movements in the `move()` function. Back in our `GameManager` , we'll use these potential positions to start walking the grid.

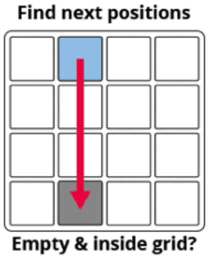
```
// ...
this.move = function(key) {
  var self = this;
  // define move here
  if (self.win) { return false; }
  var positions = GridService.traversalDirections(key);

  positions.x.forEach(function(x) {
    positions.y.forEach(function(y) {
      // For every position
    });
  });
};
// ...
```

Now, inside of our position loop, we will iterate over the possible locations and look for existing tiles in the positions. From here we'll attack the second part of our functionality and find all of the further locations from the tile:

```
// ...
// For every position
// save the tile's original position
var originalPosition = {x:x,y:y};
var tile = GridService.getCellAt(originalPosition);

if (tile) {
  // if we have a tile here
  var cell = GridService.calculateNextPosition(tile, key);
  // ...
}
```



If we *do* find a tile, we'll start to look for the furthest possible positions away from the tile. To do this, we'll walk through the next positions in the grid and check if the next cell is within the boundary of the grid **and** if the grid cell location is empty.

If the grid cell is empty and within the bounds of the grid, then we will look at the next cell and check the same conditions.

If either of these two conditions fail, then we have found either the boundary of the grid or we found the next cell. We'll save the next position as `newPosition` and grab the *next* cell (regardless if it exists or not).

As this process deals with the grid, we'll place this function inside of the `GridService` :

```
// in GridService
// ...
this.calculateNextPosition = function(cell, key) {
  var vector = vectors[key];
  var previous;

  do {
    previous = cell;
    cell = {
      x: previous.x + vector.x,
      y: previous.y + vector.y
    };
  } while (this.withinGrid(cell) && this.cellAvailable(cell));

  return {
    newPosition: previous,
    next: this.getCellAt(cell)
  };
};
```

Now that we can calculate the next possible locations for our tiles, we can check for potential merges.

A *merge* is defined as one tile colliding into another of the same value. We'll check to see if the `next` position has a tile of the same value and that it has not yet previously been *merged* into.

```
// ...
// For every position
// save the tile's original position
var originalPosition = {x:x,y:y};
var tile = GridService.getCellAt(originalPosition);

if (tile) {
  // if we have a tile here
  var cell = GridService.calculateNextPosition(tile, key),
      next = cell.next;

  if (next &&
      next.value === tile.value &&
      !next.merged) {
    // Handle merged
  } else {
    // Handle moving tile
  }
  // ...
}
```

Now, if the next position does **not** satisfy the condition, then we're just going to simply move the tile from the current location to it's next position (the else statement).

This is the easier-to-deal with condition in that all we need to do is move the tile to the newPosition location.

```
// ...
if (next &&
    next.value === tile.value &&
    !next.merged) {
  // Handle merged
} else {
  GridService.moveTile(tile, cell.newPosition);
}
```

Moving the tile

As you likely may have guessed, the `moveTile()` method is an operation that is most appropriately defined in the `GridService`.

Moving a tile is simply updating the location of the tile in the array as well as updating the `TileModel`.

As we have defined it, these are two distinct operations for two separate purposes. When we are:

Moving the tile in the array

The `GridService` array maps where the tile is located for the back-end. The location of the tile in the array is **NOT** tied to the location on the grid.

Updating the position on the TileModel

We are updating the coordinates for the front-end CSS to place the tile.

In short: in order to keep track of the tiles on the back-end, we'll need to update the `this.tiles` array in the `GridService` **and** update the tile object's position.

The `moveTile()` then becomes a simple two-step operation:

```
// GridService
// ...
this.moveTile = function(tile, newPosition) {
  var oldPos = {
    x: tile.x,
    y: tile.y
  };

  // Update array location
  this.setCellAt(oldPos, null);
  this.setCellAt(newPosition, tile);
  // Update tile model
  tile.updatePosition(newPosition);
};
```

Now, we'll need to define our `tile.updatePosition()` method. The method does exactly what it sounds like, it simply updates the `x` and `y` coordinates of the model itself:

```
.factory('TileModel', function() {
  // ...

  Tile.prototype.updatePosition = function(newPos) {
    this.x = newPos.x;
    this.y = newPos.y;
  };
  // ...
});
```

Back in our `GridService`, we can simply call `.moveTile()` to both update the position in our `GridService.tiles` array and on the tile itself.

Merging a tile

Now that we've handled the *easier* case, merging a tile becomes the next trick we need to handle. Merging is defined as:

A merge is when one tile encounters another tile of the same value in it's next potential position.

When a tile is merged, it shifts the board as well as updates the scoring of both the current game as well as the high score (if necessary).

Merging takes a few steps:

1. Adding a new tile in the final location with the merged score value
2. Removing the old tile
3. Updating the game score
4. Checking for a winning value

Broken down, the merge operation is an easy operation to handle.

```
// ...
var hasWon = false;
// ...
if (next &&
    next.value === tile.value &&
    !next.merged) {
  // Handle merged
  var newValue = tile.value * 2;
  // Create a new tile
  var mergedTile = GridService.newTile(tile, newValue);
  mergedTile.merged = [tile, cell.next];

  // Insert the new tile
  GridService.insertTile(mergedTile);
  // Remove the old tile
  GridService.removeTile(tile);
  // Move the location of the mergedTile into the next position
  GridService.moveTile(merged, next);
  // Update the score of the game
  self.updateScore(self.currentScore + newValue);
  // Check for the winning value
  if (merged.value >= self.winningValue) {
    hasWon = true;
  }
} else {
// ...
```

Since we only want to support a single tile movement per-row (that is if we have two possible merges, only one will happen per-row), we have to keep track of tiles that have already been *merged*. We do this by setting the `.merged` flag to something other than `undefined`.

Before we leave this function, we’ve used two functions here that we haven’t yet defined.

The `GridService.newTile()` method simply creates a new `TileModel` object. We’ll keep this operation in the `GridService` simply to contain the place where we create a new tile:

```
// GridService
this.newTile = function(pos, value) {
  return new TileModel(pos, value);
};
// ...
```

We’ll come back to the `self.updateScore()` method shortly. For now, it’s sufficient enough to know that it updates the game score (as the method name indicates).

After tile movement

We only want to add new tiles only after a valid movement has been made, we’ll need to check to see if there actually has been any movement from one tile to the next.

```
var hasMoved = false;
// ...
  hasMoved = true; // we moved with a merge
} else {
  GridService.moveTile(tile, cell.newPosition);
}

if (!GridService.samePositions(originalPos, cell.newPosition)) {
  hasMoved = true;
}
// ...
```

After all of the tiles have been moved (or tried to be moved), we’ll check to see if the game has been defeated. If the game is, in fact over then we’ll set the `self.win` flag on the game.

> We moved when we had a tile collision, so in the merge condition, we’ll simply set the `hasMoved` variable to true.

Lastly, we’ll check to see if there has been any movement in the board. If there has been movement, we’ll:

1. Add a new pieces to the board
2. Check if we need to show the gameOver screen

```
if (!GridService.samePositions(originalPos, cell.newPosition)) {
  hasMoved = true;
}

if (hasMoved) {
  GridService.randomlyInsertNewTile();

  if (self.win || !self.movesAvailable()) {
    self.gameOver = true;
  }
}
// ...
```

Reset the tiles

Before we run any of this main game loop, we'll need to *reset* each of the tiles such that we no longer keep track of their merge state. That is, every single move we make, we wipe the slate clean and we consider that every tile can run again. To do this, at the beginning of the move loop, we'll call:

```
GridService.prepareTiles();
```

The `prepareTiles()` method in our `GridService` simply iterates over each tile and *resets* it's status:

```
this.prepareTiles = function() {
  this.forEach(function(x,y,tile) {
    if (tile) {
      tile.reset();
    }
  });
};
```

Keeping the score

Back to the `updateScore()` method; the game itself needs to keep track of two scores:

1. The current game's score
2. The player's overall high score

The `currentScore` is simply a single variable we'll keep track of in-memory on a per-game basis. That is, we don't need to handle it in any special manner.

The `highScore`, on the other hand is a variable that we'll want to persist across each game. We have a couple of ways to handle this, using localstorage, cookies, or a combination of both.

As cookies are the easiest and most cross-browser safe of the two methods, we'll stick with setting our highScore in a cookie.

The easiest way to get access to cookies in Angular is to use the `angular-cookies` module.

To use the module, we'll need to download it either from angularjs.org (http://angularjs.org) or use a package manager, like bower to install it.

```
$ bower install --save angular-cookies
```

As usual, we'll have to reference the script in our `index.html` and set the module-level dependency of `ngCookies` on our app.

We'll update our `app/index.html` like so:

```
<script src="bower_components/angular-cookies/angular-cookies.js"></script>
```

Now to add the `ngCookies` module as a module-level dependency (on our `Game` module, where we'll reference cookies):

```
angular.module('Game', ['Grid', 'ngCookies'])
// ...
```

With the `ngCookies` as a dependency, we can *inject* the `$cookieStore` service into our `GameManager` service. We can now get and set cookies for our user's browser.

For instance, to *get* the user's latest highscore, we'll write a function to fetch it for us from the user's cookie:

```
this.getHighScore = function() {
  return parseInt($cookieStore.get('highScore')) || 0;
}
```

Back in our `updateScore()` method on our `GameManager` class, we'll update the local current score. If the score itself is higher than our previous highscore, we'll update the high score cookie for the future.

```
this.updateScore = function(newScore) {
  this.currentScore = newScore;
  if (this.currentScore > this.getHighScore()) {
    this.highScore = newScore;
    // Set on the cookie
    $cookieStore.put('highScore', newScopre);
  }
};
```

Wrath of track by

Now that we have tiles popping up on our screen, a bug will start popping up on screen in that we'll get duplicate tiles with some odd behavior. Additionally, our tiles may end up in unexpected places.

The reason for this is that Angular only *knows* what tiles are in the `tiles` array based upon a unique identifier. We set this unique identifier in the view as the `$index` of the tile in the array (aka where it is in the array). Since we are moving our tiles in the array around, the `$index` is no longer able to track our tiles as unique. We'll need a different tracking scheme.

```
<div id="game">
  <!-- grid-container -->
  <div class="tile-container">
    <div tile
      ng-model='tile'
      ng-repeat='tile in ngModel.tiles track by $index'></div>
  </div>
</div>
```

Rather than depend upon the array to identify the location of the tile, we'll track each tile by their own unique uuid. Creating our own unique identifier will guarantee that angular will respect each of the tiles in our tiles array will be treated as their own unique object. Angular will respect our unique identifiers and view each tile as it's own object, so long as it's unique uuid doesn't change.

We can easily implement a unique identifying scheme for our tiles using the `TileModel` when we create a new instance. We can come up with our own scheme for creating unique identifiers.

> It doesn't matter how we generate unique ids so long as they are unique for every `TileModel` instance we create.

To create a unique id, we jumped over to StackOverflow (http://stackoverflow.com/questions/105034/how-to-create-a-guid-uuid-in-javascript) to find ourselves a rfc4122-compliant (http://www.ietf.org/rfc/rfc4122.txt) globally unique identifier generator and wrapped it into a factory with a single method: `next()` :

```
 .factory('GenerateUniqueId', function() {
   var generateUid = function() {
     // http://www.ietf.org/rfc/rfc4122.txt
     var d = new Date().getTime();
     var uuid = 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
       var r = (d + Math.random()*16)%16 | 0;
       d = Math.floor(d/16);
       return (c === 'x' ? r : (r&0x7|0x8)).toString(16);
     });
     return uuid;
   };
   return {
     next: function() { return generateUid(); }
   };
 })
```

To *use* the `GenerateUniqueId` factory, we can *inject* it and call `GenerateUniqueId.next()` to create us a new uuid. Back in our `TileModel` , we can create a unique id for the instance (in the constructor):

```
// In app/scripts/grid/grid.js
// ...
.factory('TileModel', function(GenerateUniqueId) {
  var Tile = function(pos, val) {
    this.x      = pos.x;
    this.y      = pos.y;
    this.value  = val || 2;
    // Generate a unique id for this tile
    this.id = GenerateUniqueId.next();
    this.merged = null;
  };
  // ...
});
```

Now that we have a unique identifier for each tile, we can tell Angular to track by the id, rather than by `$index` .

```
<!-- ... -->
<div tile
      ng-model='tile'
      ng-repeat='tile in ngModel.tiles track by $id(tile.id)'></div>
<!-- ... -->
```

There is only one problem with this. Since we initialized our array with nulls (to be explicit) and we reset the array to contain nulls (rather than sort & resize the array), angular will try to track nulls as objects regardless. As null values do not have unique ids, this will cause our browser to throw an error and have no idea how to handle the duplicate objects.

Thus, we can use a built-in angular tool to track by either the unique id or the `$index` position of the object (null objects can be tracked by their position in the array as there will only be one in each position). We can change our grid_directive's view to account for null objects, like so:

```
<!-- ... -->
<div tile
      ng-model='tile'
      ng-repeat='tile in ngModel.tiles track by $id(tile.id || $index)'></div>
<!-- ... -->
```

> This issue can be solved with a different implementation of the underlying data structure, such as looking up the position in each `TileModel` with an iterator, rather than depending upon the index of the `tiles` array or by reshuffling the array every time we make a change (or on `$digest()`). For simplicity and clarity, we've implemented it using the array as this is the only side-effect we'll need to handle with this implementation.

We won?!?? Game over

When we lose the original 2048 game, a *game over* screen slides in and allows us to restart the game and follow the creator on twitter. Not only is this a cool effect that the game gives to the player, it also presents a nice way to interrupt the game play.

We can easily create this using some basic angular techniques. We're already keeping track of when the game ends in the `GameManager` with the `gameOver` variable. We can simply create a `<div>` object that contains our game over screen and absolutely position it over the game grid. The magic of this technique (and Angular) is that it's so simple to implement without any trickery:

We can simply create a new `<div>` element that contains the game over or winning message and show it depending upon the state of the game. For instance, the game over screen may look something like:

```
<!-- ... -->
<div id="game-container">
  <div grid ng-model='ctrl.game' class="row"></div>
    <div id="game-over"
        ng-if="ctrl.game.gameOver"
        class="row game-overlay">
      Game over
      <div class="lower">
        <a class="retry-button" ng-click='ctrl.newGame()'>Try again</a>
      </div>
    </div>
  <!-- ... -->
```

The *tough* part is handling the styling/CSS. Effectively we'll just place the element at an absolute positioning over the game grid and let the browser do the work of positioning it. Here are the *relevant* parts of the css (note, the full CSS is available on the github link below):

```
.game-overlay {
  width: $width;
  height: $width;
  background-color: rgba(255, 255, 255, 0.47);
  position: absolute;
  top: 0;
  left: 0;
  z-index: 10;
  text-align: center;
  padding-top: 35%;
  overflow: hidden;
  box-sizing: border-box;

  .lower {
    display: block;
    margin-top: 29px;
    font-size: 16px;
  }
}
```

> We can use the exact same technique with a win screen, we just create a winning `.game-overlay` element.

Animation

One of the impressive features of the original 2048 game is that the tiles seem to magically slide from one location to the next and the game over/winning screen naturally appear on the screen. As we are using Angular, we can achieve the **exact same effects almost for free** (thanks to CSS).

In fact, we've set up the game so that we can create the sliding, appearing, revealing, etc. animations are easy to implement. We won't even touch the JavaScript (just barely) to implement them.

Animating the CSS positioning (aka adding sliding tiles)

As we are positioning our tiles via CSS using the class `position-[x]-[y]`, when a new position is set on the tile, this DOM element will have the class `position-[newX]-[newY]` added and the old class of `position-[oldX]-[oldY]` removed. In this case, we can simply define the default slide behavior to happen on the CSS class itself by defining a CSS transition on the `.tile` class.

The relevant SCSS is:

```
.tile {
  @include border-radius($tile-radius);
  @include transition($transition-time ease-in-out);
  -webkit-transition-property: -webkit-transform;
  -moz-transition-property: -moz-transform;
  transition-property: transform;
  z-index: 2;
}
```

With the CSS transition defined, the tiles will now easily slide between one location and the next (yes, it's seriously **that easy**).

Animating the game over screen

Now, if we want to get more *fancy* with our animations, we can do so with the `ngAnimate` module. The module itself works out-of-the-box with angular.

Before we can use it, we'll need to install the `ngAnimate` module. We can either grab the module from angularjs.org (http://angularjs.org) or use a package manager (such as bower) to install it.

```
$ bower install --save angular-animate
```

As usual, we'll then need to reference the script in our HTML so that the browser can load it. Let's modify our main `index.html` to include the `angular-animate.js` file:

```
<script src="bower_components/angular-animate/angular-animate.js"></script>
```

Finally, like any other angular module, we'll need to tell angular that we're dependent upon the module being available to invoke our module. We can do this in the module dependencies array in our `app/app.js` file:

```
angular
.module('twentyfourtyeightApp', ['Game', 'Grid', 'Keyboard', 'ngAnimate', 'ngCookies'])
// ...
```

ngAnimate

Although an in-depth discussion of `ngAnimate` is outside of the scope of this tutorial (see ng-book (https://www.ng-book.com) for a deep look into how it works), we'll loosely look at how it works to understand how we can implement animations for our game.

With the `ngAnimate` module included as a module-level dependency, anytime that angular adds a new object in one of the relevant (to our game) directives it will assign a CSS class (for free). We can use these classes to assign CSS animations for our different components of the game:

Directive	Added class	Leaving class
ng-repeat	ng-enter	ng-leave
ng-if	ng-enter	ng-leave
ng-class	[className]-add	[className]-remove

When an element is added to the `ng-repeat` scope, the new DOM element will be automatically assigned the `ng-enter` CSS class. Then, when it's actually added to the view, the `ng-enter-active` CSS class will be added. This is important as it will allow us to set up how we want our animations to look in the `ng-enter` CSS class and set the animation style in the `ng-enter-active` class. This functionality works the same with the `ng-leave` class when elements are removed from the `ng-repeat` iterator.

When a new CSS class is added (or removed) from a DOM element, the corresponding CSS class of `[classname]-add` and `[classname]-add-active` will be added to the DOM element. Again, here we can set up our CSS animations in the corresponding classes.

Animating the game over screen

We can animate the game over and winning screens by using the `ng-enter` class. Remember, the `.game-overlay` class is hidden and shown using the `ng-if` directive. When the `ng-if` conditions changes, `ngAnimate` will add the `.ng-enter` and `.ng-enter-active` when the expression results in a truthy value (or `.ng-leave` and `.ng-leave-active` , when angular removes the element).

We'll set up the animation in the `.ng-enter` class and then start it within the `.ng-enter-active` class. The relevant SCSS:

```
.game-overlay {
  // ...
  &.ng-enter {
    @include transition(all 1000ms ease-in);
    @include transform(translate(0, 100%));
    opacity: 0;
  }
  &.ng-enter-active {
    @include transform(translate(0, 0));
    opacity: 1;
  }
  // ...
}
```

All of the SCSS is available on github at the link at the bottom of the article.

Location customizations

Suppose we want to create a different board size. For instance, the original 2048 game has a 4x4 grid. What if we want to create a 3x3 or a 6x6 board? We can easily make this a possibility without needing to change much code at all.

The board itself is created and positioned by SCSS and the grid is managed in the `GridService` . Thus, we'll need to make modifications to these two locations to allow us to create customized boards.

Dynamic CSS

Okay, so we aren't *really* going to have dynamic CSS, but we can create more CSS than we'll actually need. Instead of creating a single `#game` tag, we can dynamically create the DOM element tag that we'll allow the grid to set dynamically. In other words, we'll create a version of the board with 3x3 nested under the DOM element with an ID of `#game-3` and one for a 6x6 board with an id tag of `#game-6` .

We can create a mixin (http://sass-lang.com/documentation/file.SASS_REFERENCE.html#mixins) out of our already dynamic SCSS. Simply find the `#game` css ID tag and wrap it in a mixin. For instance:

```
@mixin game-board($tile-count: 4) {
  $tile-size: ($width - $tile-padding * ($tile-count + 1)) / $tile-count;
  #game-#{$tile-count} {
    position: relative;
    padding: $tile-padding;
    cursor: default;
    background: #bbaaa0;
    // ...
}
```

Now we can include the `game-board` mixin to dynamically create a stylesheet that contains multiple versions of the gameboard, each isolated by their respective `#game-[n]` tag.

To build multiple versions of this, we'll simply iterate over the number of game boards we'd like to create and call the mixin.

```
$min-tile-count: 3;       // lowest tile count
$max-tile-count: 6;       // highest tile count
@for $i from $min-tile-count through $max-tile-count {
  @include game-board($i);
}
```

Dynamic GridService

Now that we have our CSS set up such that we will create multiple sized boards, we'll need to modify our `GridService` so that we can set the grid size when we boot the app.

Angular makes this process pretty easy. First, we'll need to change our `GridService` to be a `provider` , rather than a direct `service` . If you're unfamiliar with the differences between services and providers, check out ng-book (https://www.ng-book.com) for a deep discussion. In short, a provider allows us to configure it before it's launched.

In addition, we'll need to change the constructor function to be set as the `$get` method on the provider:

```
.provider('GridService', function() {
  this.size = 4; // Default size
  this.setSize = function(sz) {
    this.size = sz ? sz : 0;
  };

  var service = this;

  this.$get = function(TileModel) {
    // ...
```

Any method on the provider that is **not** in the `$get` method will be available in the `.config()` function on our module. Anything that is **inside** the `$get()` function is available in the running app, but not in the `.config()` method on the module.

That's all we need to do to make the board sizing dynamic. Now, let's say that we want to make a 6x6 board, instead of the default 4x4 grid. In the `.config()` function on our app module, we can call out to the `GridServiceProvider` to set the size:

```
angular
.module('twentyfourtyeightApp', ['Game', 'Grid', 'Keyboard', 'ngAnimate', 'ngCookies'])
.config(function(GridServiceProvider) {
  GridServiceProvider.setSize(4);
})
```

> When creating a provider, Angular **automatically** creates a config-time module we can inject in with the name: [serviceName]Provider.

Demo demo

The complete demo for the entire application is available at http://ng2048.github.io/ (http://d.pr/SnWD).

Conclusion

Phew! We hope you've enjoyed building the 2048 puzzle in Angular. We've covered quite a bit in this post. If you've enjoyed it, leave a comment below. If you're interested in learning more about Angular, check out our Complete Book on AngularJS (https://www.ng-book.com). It's the **only** constantly updated book on AngularJS available and covers everything you need to know about AngularJS.

Discuss on HackerNews (https://news.ycombinator.com/item?id=7554348)

Thanks

A BIG thanks to Gabriele Cirulli (http://gabrielecirulli.com/) for the awesome (and addicting) 2048 game and the inspiration for this post. Many of the ideas in this post were gleened from the game itself and extracted to demonstrate how to build it in Angular.

Complete source

The complete source of the game can be found on Github at http://d.pr/pNtX (http://d.pr/pNtX). To build the game locally, clone the source and run:

```
$ npm install
$ bower install
$ grunt serve
```

Troubleshooting

If you're having trouble doing an `npm install`, make sure you have a recent version of node.js and `npm`.

This repo was tested on node `v0.10.26` and `npm 1.4.3`.

Here's a nice way to get a recent version of node using the `n` node version manager:

```
$ sudo npm cache clean -f
$ sudo npm install -g n
$ sudo n stable
```

← Back to articles (/posts)

Share on Twitter (http://twitter.com/home/?status=Building the 2048 game in AngularJS - http://ng-newsletter.com/posts/building-2048-in-angularjs.html via @ngnewsletter)

Follow us on Google+ (https://plus.google.com/+AriLerner?rel=author)

Get the weekly email all focused on AngularJS. **Sign up below** to receive the weekly email and exclusive content.

Email address | Signup

We will **never** send you spam and it's a cinch to unsubscribe.

Download a free sample of the ng-book: The Complete Book on AngularJS

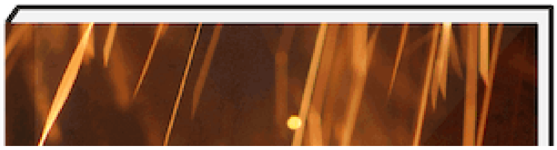
ng-book: The Complete Book on AngularJS is the canonical AngularJS book available today.

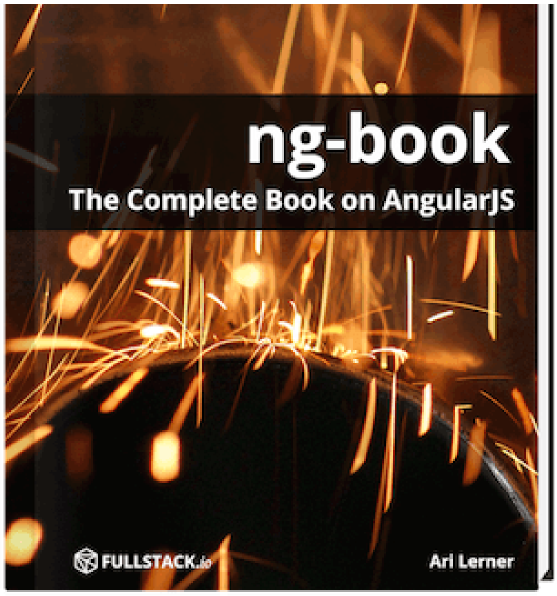
It's free, so just enter your email address and the PDF will be sent directly to your inbox. Mailchimp can take up to an hour to deliver the free sample chapter, but if you don't receive it within the hour, send us an email and we'll manually send them to you!

We'll send you updates about the book, when it updates and other free content.

Email address | Get the chapter

We will **never** send you spam and it's a cinch to unsubscribe.





Comments

0 Comments ng-newsletter Login

Sort by Best Share Favorite ★

Start the discussion…

Be the first to comment.

Subscribe Add Disqus to your site

PRODUCTS

ng-book (https://ng-book.com)
Newsletter Sponsorship (/sponsor)

CURATED BY

Ari Lerner (http://twitter.com/auser)
Q (http://twitter.com/qookins)
Nate Murray (http://twitter.com/eigenjoy)

RESOURCES

How to Learn AngularJS (/posts/how-to-learn-angular.html)
How to Write Directives (/posts/directives.html)
Using AngularJS on Mobile (/posts/angular-on-mobile.html)

FOLLOW US ON

Twitter (https://twitter.com/ngnewsletter)
Github (https://github.com/fullstackio)