Добро пожаловать в Руководство разработчика по Angular. Если вы хотите узнать в подробностях о том, как использовать Angular для разработки веб-приложений, то вы попали в нужное место.

Обзор

Что такое Angular?

AngularJS является структурной основой для динамических веб-приложений. Эта технология позволяет использовать HTML в качестве языка шаблона и позволяет расширить синтаксис HTML, чтобы выразить компоненты вашего приложения ясно и кратко. Из коробки, он содержит большую часть кода будущего приложения, и вы в настоящее время можете кодировать с помощью привязки данных и зависимостей. И все это происходит в браузере с использованием JavaScript, что делает его идеальным партнером с любой серверной технологией.
Angular это то, чем был бы HTML, если бы он был разработан для разработки приложений. HTML является прекрасным, декларативным языком для статических документов. Он не содержит много из того, что нужно для создания приложений, и в результате создания веб-приложений это попытка объяснить браузеру то, что мне нужно сделать.
Несоответствия между динамическими и статическими приложениями часто решается, так:

- Библиотека - набор функций, которые полезны при написании веб-приложений. Ваше приложение изменяется и вызывает библиотеку, когда оно считает нужным. Например, Jquery.
- Фреймворки - конкретная реализация веб-приложения, где ваш код заботится о деталях. Фреймворки изменяют и вызывают код, когда им необходимо что-то от конкретного приложения. Например, knockout, SproutCore и т.д.

В Angular другой подход. Он пытается свести к минимуму несоответствия между документом HTML и требованиями приложения путем создания новых HTML конструкций. Angular учит браузер новому синтаксису, используя конструкцию, которую мы называем директива. Примеры:

- Привязка данных, как в {{}}.
- Управляющие структуры DOM для повторения / скрытия DOM фрагментов.
- Поддержка формы и ее проверка.
- Установка привязок DOM элементов к данным.
- Группировка HTML в многократно используемые компоненты.
- Конечное комплексное решение
-

Комплексный подход

Angular – это конечное комплексное решение для создания веб-приложений. Это означает, что это ни один компонент в общем построении веб-приложения, а полное решение. Angular имеет собственные требования к тому, как приложение CRUD должны быть построены. Но эти требования не обязательные, и вы можете легко им не следовать. Angular из коробки имеет:

- Все, что Вам нужно для создания CRUD приложения: привязки данных, основные директивы шаблонов, проверки форм, маршрутизация, глубокие связи, повторно используемые компоненты, внедрение зависимостей.
- Тестируемость: юнит-тестирование, end-to-end тестирование, элементы заменители, средства тестирования.
- Стартовое приложение с описанием содержимого каталогов и сценариев тестирования в качестве отправной точки.

Предназначение Angular

Angular упрощает разработку приложений, представляя абстракции высокого уровня для разработчика. Как и любая абстракция, это связано с определенной потерей гибкости. Другими словами, не каждое приложение хорошо подходит для Angular. Angular был построен для приложения типа CRUD. К счастью CRUD приложения представляют, по крайней мере, 90% веб-приложений. Но чтобы понять, где Angular хорош, мы должны понимать, когда приложение не подходит для Angular.
Игры и графические редакторы, примеры очень интенсивного и сложного манипулирования DOM. Эти виды приложений отличаются от CRUD приложений, и, как следствие, не подходит для Angular. В этих случаях, используйте что-то близкое к голому JavaScript, может быть JQuery здесь лучше подходит.

Вводный пример Angular

Ниже приводится типичное CRUD приложение, которое содержит форму. Значения элементов формы будут проверяться, и используются для расчета общей суммы, которая форматируется в формат конкретной

местности. Здесь использованы некоторые общие понятия, с которыми разработчик приложения может столкнуться:

- привязка данных модели к пользовательскому интерфейсу.
- запись, чтения и проверка пользовательского ввода.
- вычисления новых значений на основе модели.
- Форматирование выходных данных в удобном виде для пользователей определенного региона.

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="InvoiceCntl">
      <b>Invoice:</b>
      <br>
      <br>
      <table>
       <tr><td>Quantity</td><td>Cost</td></tr>
       <tr>
         <td><input type="number" ng-pattern="/\d+/" step="1" min="0" ng-model="qty" required ></td>
         <td><input type="number" ng-model="cost" required ></td>
       </tr>
      </table>
      <hr>
      <b>Total:</b> {{qty * cost | currency}}
    </div>
  </body>
</html>
```

Script.js

```
function InvoiceCntl($scope) {
  $scope.qty = 1;
  $scope.cost = 19.95;
}
```

End to end test

```
it('should show of angular binding', function() {
  expect(binding('qty * cost')).toEqual('$19.95');
  input('qty').enter('2');
  input('cost').enter('5.00');
  expect(binding('qty * cost')).toEqual('$10.00');
});
```

Попробуйте выполнить пример выше, а затем давайте его разберем и опишем то, что происходит.

В теге <html>, мы указываем, что это Angular приложение с помощью директивы ng-app. Ng-app будет вызывать инициализировать приложение Angular автоматически.

```
<html ng-app>
```

Используя тег <script>, загружается код Angular.

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/?.?.?/angular.min.js"></script>
```

Атрибут ng-model в элементе <input> автоматически устанавливает двунаправленную привязку данных, а также используется простая валидация вводимых данных.

```
Quantity: <input type="number" ng-pattern="/\d+/" step="1" min="0" ng-model="qty" required >

Cost: <input type="number" ng-model="cost" required >
```

Эти поля ввода будут выглядеть вполне нормально, но учтите следующие моменты:

- После загрузки страницы, Angular связывает имена входных полей ввода (qty и cost) с одноименными переменными. Воспринимайте переменные, как элементы "Модели" шаблона проектирования Model-View-Controller.
- Обратите внимание, что HTML поле ввода имеет особые возможности. Когда вы вводите неверные данные или оставляете поле ввода пустым, он окрашивается в красный цвет. Это новое поведение поля ввода позволяет легче осуществить проверку полей в приложениях CRUD.

И, наконец, таинственные {{двойные фигурные скобки}}:

```
Total: {{qty * cost | currency}}
```

Нотация {{_expression_}} в Angular указывает на привязку данных. Содержащееся внутри выражение может быть комбинацией выражения и фильтра: `{{ expression | filter }}`. Angular применяет фильтры для форматирования отображаемых данных.

В примере выше, выражение в двойных фигурных скобках говорит Angular «привяжи данные, полученные из полей ввода, умножь их и выведи полученное число в денежном формате».

Обратите внимание, мы достигли требуемого результата, не вычисляя ни каких методов, и не применяя ни какого специфического поведения Фреймворка. Мы достигли этого, просто подсказав браузеру, что хотим, чтобы он вел себя как с динамическим приложением, а не как со статическим. Angular позволил нам это сделать.

Angular постулаты

Angular построен вокруг убеждения, что декларативные код лучше, и это тем важнее, когда дело доходит до построения пользовательских интерфейсов и компонентов программного обеспечения работающих вместе, а императивный код отлично подходит для выражения бизнес-логики.

- Это очень хорошая идея, чтобы отделить DOM манипуляции от логики приложения. Это значительно улучшает читабельность кода.
- Это очень, очень хорошая идея, чтобы считать тестирование приложения равным по значению его написанию. Трудности тестирования сильно повлияли на способ структурирования кода.
- Это отличная идея, чтобы отделить клиентскую сторону приложения от стороны сервера. Это позволяет вести разработку параллельно, и позволяет повторно использовать код обеих сторон.
- Это очень полезно, если Фреймворк помогает разработчикам на всем пути построения приложений: от проектирования пользовательского интерфейса, через написание бизнес-логики, к тестированию.
- Это всегда хорошо, когда общие задачи, а иногда и сложные, становятся.

Angular освобождает вас от головной боли в следующих случаях:

- Регистрация обратных вызовов: регистрация обратных вызовов загромождает ваш код, что делает его трудным для понимания. Удаление общего, стереотипного кода таких обратных вызовов является хорошая вещью. Это значительно уменьшает количество JavaScript кода, который вам нужно написать, и это позволяет легче увидеть, что делает ваше приложение.
- Манипулирования HTML DOM программно: манипулирования DOM HTML является краеугольным камнем приложений AJAX, но громоздким и подверженным ошибкам. Декларативно описывая, как следует изменить пользовательский интерфейс при изменении состояния приложения, вы освобождены от задач манипуляции DOM низкого уровня. Большинство приложений, написанных с Angular, никогда не должны программно манипулировать DOM, хотя это можно делать, если хотите.
- Маршалинг данных в и из пользовательского интерфейса: операции CRUD составляют большинство в приложениях AJAX. Поток маршалирует данных от сервера во внутренний объект и к HTML-форме, позволяя пользователям изменять форму, проверять ее, отображать ошибки проверки, сохраняя данные во внутренней модели и затем, отправляя обратно на сервер, создает много стереотипного

кода. Angular устраняет почти все из этого шаблона, оставив код, описывающий общий поток приложения, а не детали реализации.

Начальная загрузка

Обзор

Здесь объясняется, как вручную инициализировать Angular, когда это необходимо.

Инициализация Angular с помощью тега <script>

Этот пример показывает рекомендуемый способ включения Angular в ваше приложение.

```
<!doctype html>
<html xmlns:ng="http://angularjs.org" ng-app>
  <body>
    ...
    <script src="angular.js">
  </body>
</html>
```

- Поместите тег <script> в нижней части страницы. Это позволит уменьшить время загрузки приложения, т.к. загрузка HTML не будет откладываться до конца загрузки файлов сценария. Вы можете получить последнюю версию agular по адресу http://code.angularjs.org. Не рекомендуется использовать эту ссылку в рабочем коде у себя на сайте по соображениям безопасности. Есть две версии файлов:
  - Для разработки и отладки выбирайте версию, которая именуется по следующему шаблону - `angular-[version].js`. Для
  - Рабочая минимизированная версия файлов - `angular-[version].min.js`
- Впишите атрибут ng-app в корневой элемент вашего приложения, обычно это элемент <html>, если вы хотите чтобы angular инициализировался автоматически.
  `<html ng-app>`
- В IE7 также требуется добавить атрибут id, со значение ng-app.
  `<html ng-app id="ng-app">`
- Также есть старый, не рекомендуемый к использованию стиль. Вы можете использовать синтаксис пространства имен XML. Делается это следующим образом:
  `<html xmlns:ng="http://angularjs.org">`

Автоматическая инициализация

При возникновении события `DOMContentLoaded` Angular инициализируется автоматически. В этот момент он ищет директиву ng-app, которая указывает на корень вашего приложения. Если директива будет найдена, тогда происходит следующее:

- Загружается модуль, связанный с директивой.
- Для приложения создается инжектор.
- Начиная от найденной директивы компилируется дерево DOM. Это позволяет обрабатывать только часть дерева DOM как приложение angular.

```
<!doctype html>
<html ng-app="optionalModuleName">
```

```
    <body>
      I can add: {{ 1+2 }}.
      <script src="angular.js"></script>
    </body>
  </html>
```

Ручная инициализация

Если вы хотите контролировать процесс инициализации, тогда можете использовать ручную загрузку. Например, вам может это понадобится при асинхронной загрузке страницы, или если перед инициализацией требуется выполнить какой-либо скрипт. Ниже приведен пример ручной инициализации. Этот пример эквивалентен применению директивы ng-app.

```
<!doctype html>
<html xmlns:ng="http://angularjs.org">
  <body>
    Hello {{'World'}}!
    <script src="http://code.angularjs.org/angular.js"></script>
    <script>
       angular.element(document).ready(function() {
         angular.bootstrap(document);
       });
    </script>
  </body>
</html>
```

Этот код делает следующее:

- После загрузки всего кода, находит корневой элемент приложения angular (обычно это html).
- Вызывает метод API angular.bootstrap чтобы скомпилировать приложение и вычислить все требуемые двунаправленные привязки в приложении.

Компилятор HTML

Обзор

Компилятор HTML в angular позволяет разработчику расширять синтаксис HTML так как это ему нужно. Компилятор позволяет прикрепить поведение к любому элементу HTML, и даже позволяет создавать новые элементы с пользовательскими атрибутами. В angular это реализуется в директивах.

HTML уже имеет множество конструкций для форматирования статического документа в декларативном стиле. К примеру для того чтобы задать расположение элемента по центру, нет необходимости давать команды браузеру разделить окно пополам, определить центр и совместить это т центр с центром текста. Просто добавляется атрибут align=”center”, и все. В этом вся мощь и красота декларативного стиля.

Но этот декларативный язык также ограничен, т.к. он не позволяет учить браузер новому синтаксису. К примеру нет простого способа заставить браузер выровнять текст не по ½ а по 1/3

части экрана. Отсюда вывод – необходимо добавить возможность браузеру обучатся новому синтаксису.

Angular добавляет эту возможность в комплекте с общими директивами, которые будут полезны при построении любого приложения. Также вы можете создавать собственные директивы, специфичные для вашего приложения, что существенно расширяет язык понятиями из вашей предметной области.

Вся работа по компиляции происходит на стороне клиента, не загружая сервер.

Компилятор

Сам angular и является компилятором. Он проходит дерево DOM в поиске компилируемых атрибутов. Процесс компиляции проходит в два этапа:

1. Компиляция – проходим DOM и находим все директивы. В результате получаем связующую функцию.
2. Связывание – связываются директивы, области видимости(scope) и соответствующие представления (view). Любые изменения в области видимости отражаются на соответствующем представлении, а любые взаимодействия пользователя проходят через представления в рамках модели. Это делает модель области видимости единственным источником информации для представлений.

Некоторые директивы, такие как ng-repeat клонируют элементы DOM для каждого элемента входной коллекции. Раздельная компиляция и связывание позволяют повысить производительность, поскольку компиляция проводится только один раз, а связывание для каждого элемента в коллекции. Иначе бы и компиляция проводилась для каждого элемента коллекции.

Директивы

Директивы обрабатываются компилятором по мере их нахождения в процессе компиляции. Директивы можно задавать четырьмя разными способами: как элемент, атрибут, значение атрибута class, а также в виде комментариев. В примере показаны эти способы для директивы ng-bind.

```
1. <span ng-bind="exp"></span>
2. <span class="ng-bind: exp;"></span>
3. <ng-bind></ng-bind>
4. <!-- directive: ng-bind exp -->
```

Директива – это просто функция, которая выполняется, когда компилятор находит директиву в DOM. Для углубленного изучения директив, смотрите соответствующие разделы документации.

Пример директивы

index.html:

```
1. <!doctype html>
2. <html ng-app="drag">
3.   <head>
```

```
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <span draggable>Drag ME</span>
  </body>
</html>
```

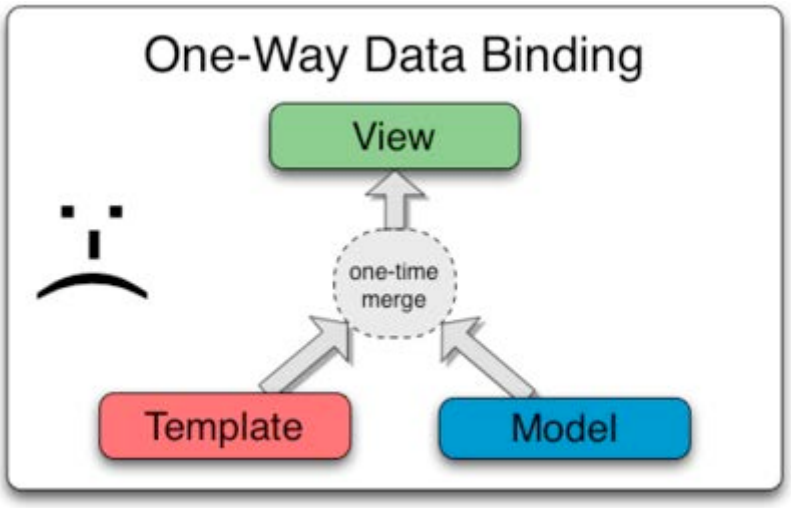
script.js

```
angular.module('drag', []).
  directive('draggable', function($document) {
    return function(scope, element, attr) {
      var startX = 0, startY = 0, x = 0, y = 0;
      element.css({
       position: 'relative',
       border: '1px solid red',
       backgroundColor: 'lightgrey',
       cursor: 'pointer'
      });
      element.bind('mousedown', function(event) {
        // Prevent default dragging of selected content
        event.preventDefault();
        startX = event.screenX - x;
        startY = event.screenY - y;
        $document.bind('mousemove', mousemove);
        $document.bind('mouseup', mouseup);
      });

      function mousemove(event) {
        y = event.screenY - startY;
        x = event.screenX - startX;
        element.css({
          top: y + 'px',
          left:  x + 'px'
        });
      }

      function mouseup() {
        $document.unbind('mousemove', mousemove);
        $document.unbind('mouseup', mouseup);
      }
    }
  });
```

Поведение перетаскивания дает директива draggable. Красота этого подхода в том, что мы добавили нужное поведение элементу в браузере. Можно сказать, что мы расширили список слов, который понимает браузер, добавив еще одно свойство в HTML.

Понимание view.

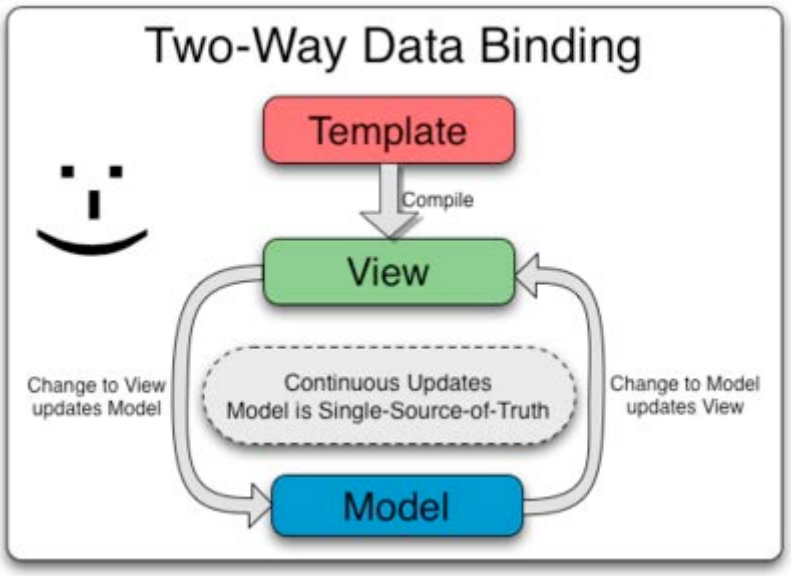
Существует множество систем для работы с шаблонами. Большинство из них просто берет строку шаблона, объединяет ее с данными и возвращает в результате новую строку. Затем полученная строка вставляется в HTML с помощью свойства innerHTML.



Этот означает что любые изменения в данных должны быть повторно объединены с шаблоном и повторно вставлены с помощью innerHTML в DOM. Вот некоторые из проблемм, которые возникают при таком подходе:

- Чтение пользовательского ввода и слияние его с данными.
- Удаление пользовательского ввода в случае его обновления.
- Управление всем процессом обновления.
- Отсутствие выразительности намерений.

Angular работает по другому. Он работает непосредственно с DOM, через директивы, а не шаблонные строки. В результате появляется связующая функция, которая связывает воедино модель (scope) и представление. Представление и привязка к модели является прозрачной и легкой для понимания. Разработчику не нужно предпринимать ни каких действий для обновления представления. А т.к. innerHTML не используется, то и нет проблем с удалением старых пользовательских данных при обновлении. Кроме всего этого, angular директивы могут содержать не только текст привязки, но и добавлять определенное поведение.



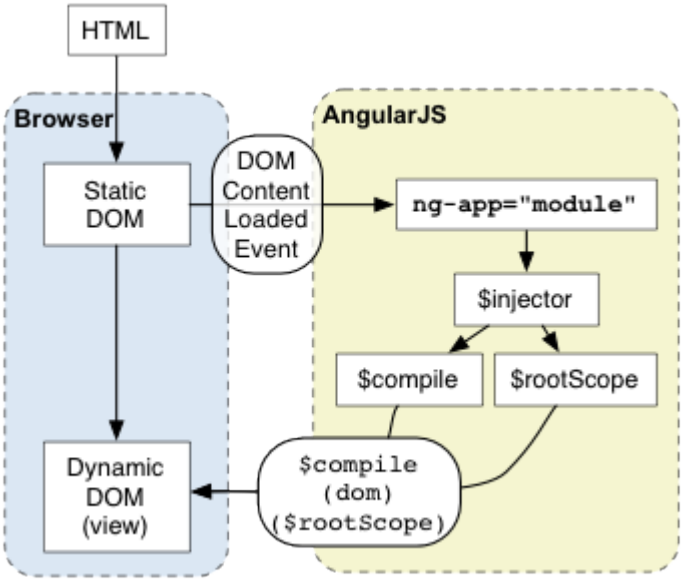
Такой подход позволяет получить стабильное дерево DOM, что означает что экземпляр элемента DOM связан с экземпляром элемента модели и не изменяется в течении всего своего жизненного цикла. Также это значит что в коде можно получить элемент и зарегистрировать событие, и при этом не опасаться, что ссылка на него будет уничтожена при следующем изменении данных.

Обзор концепций

Обзор

Этот раздел дает описание основных компонентов angular, а также позволяет понять как они взаимодействуют в процессе работы. Это компоненты:

- Запуск (startup) – который является дорогой в мир.
- Время выполнения (runtime) – обзор angular во время выполнения.
- Область видимости (scope) – слой между представлением и контроллером.
- Контроллер – поведение приложения.
- Модель – данные вашего приложения.
- Представление – это то, что видит пользователь.
- Директивы – расширения языка HTML.
- Фильтры – форматеры данных и локальные настройки приложения.
- Инжектор – собирает ваше приложение в целое.
- Модуль – является конфигуратором для инжектора.
- $ - angular пространство имен.



Запуск

Итак, мы сдвинулись с мертвой точки. Как это произошло расскажем дальше, а также смотрите схему.

1. Браузер загружает HTML и анализирует его DOM.

2. Браузер грузит файл angular.js с движком angular.
3. Angular ожидает события DOMContentLoaded.
4. Angular ищет директиву ng-app, которая задает границы приложения.
5. Модуль, указанный в ng-app (если есть) используется для настройки инжектора.
6. Инжектор используется для создания сервиса $compile, а также $rootScope.
7. Сервис $compile используется для компиляции дерева DOM и связывания его с $rootScope.
8. Директива ng-init присваивает свойствам в $scope заданные значения.
9. {{name}} используется для вставки в данное место значение свойства name из $scope или $rootScope.

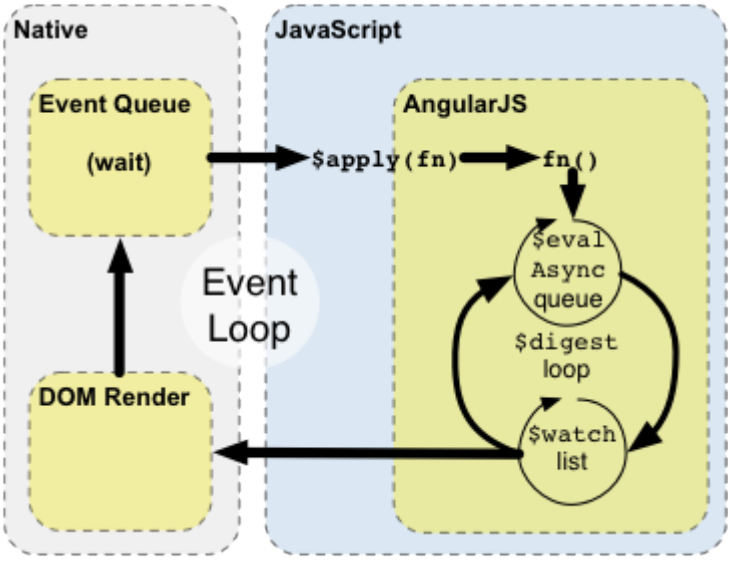
Пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
  </head>
  <body>
    <p ng-init=" name='World' ">Hello {{name}}!</p>
  </body>
</html>
```

Время выполнения

Схема и пример ниже описывают, как angular взаимодействует с циклом событий браузера.



1. Браузер возбуждает определенное событие в ответ на действия пользователя, истечения времени или поступление данных по сети.
2. По возникновению события запускается функция обратного вызова, в которой можно изменить DOM.
3. После выполнения функции обратного вызова браузер выходит из контекста javascript и отображает изменения DOM на экране.

Angular изменяет нормальное течение javascript, представляя собственный цикл обработки событий. Это разбивает javascript на классический и angular контекст выполнения. Только операции, выполняемые в контексте выполнения angular, получают доступ к его привязкам данных, обработке исключений, изменениям свойств и т.д. Можно использовать функцию $apply() для перехода в контекст выполнения angular из javascript. Имейте ввиду, что в большинстве мест (контроллеры, сервисы) функция $apply() уже была вызвана за вас, для ваших директив, которые обрабатывают события. Явный вызов $apply() применяется при реализации пользовательских функций обратного вызова

для событий или при работе со сторонними библиотечными функциями обратного вызова для событий браузера.

1. Войдите в контекст выполнения angular, вызвав $scope.$apply(stimulusFn). Где stimulusFn является функцией обратного вызова, которая должна быть выполнена к контексте angular.
2. Angular выполняет stimulusFn, которая обычно изменяет состояние приложения.
3. Angular входит в цикл $digest. $digest цикл проходит по двум небольшим петлям, это $evalAsync очередь и список $watch. Итерации в цикле $digest проходят до тех пор, пока модель не стабилизируется, что означает $evalAsync очередь пуста, и список $watch не обнаруживает ни каких изменений.
4. $evalAsync очередь используется для планирования работы, которая должна проходить вне текущего кадра стека, но перед отображением представления пользователю. Обычно это делает с помощью setTimeout(0), но этот подход страдает от медлительности, и может привести к мерцанию в окне браузера, т.к. браузер обновляет представление после каждого события.
5. Список $watch содержит набор выражений, которые могут изменяться после каждой итерации. Если обнаруживаются изменения, то $watch вызывает функцию, которая изменяет отображаемое представление.
6. После окончания цикла $digest браузер покидает контекст Angular и javascript. Это сопровождается повторным отображением дерева DOM, чтобы отобразить любые изменения.

Сейчас переходим к объяснению привязки данных на примере приложения “hello world”, которое изменяет вывод в ответ на ввод данных пользователем.

1. Во время фазы компиляции:
   1.1. Ng-model и директива input устанавливают обработчик события keydown для элемента input.
   1.2. {{name}} отображает установленное значение name, а $watch уведомляет о любом изменении name.
2. Во время этапа выполнения:
   2.1. Нажатие пользователем на клавишу ‘x’ в элементе input приводит к генерации браузером события keydown.
   2.2. Директива input отражает изменения в элементе input и вызывает $apply(name=’x’) для обновления модели данных внутри контекста Angular.
   2.3. Angular изменяет свойство модели name на ‘x’.
   2.4. Стартует цикл $digest.
   2.5. $watch список отслеживает изменения свойства name и уведомляет об этом привязки, которые в свою очередь обновляют DOM.
   2.6. Приложение выходит из контекста Angular, затем завершается обработка события keydown, и приложение покидает контекст javascript.
   2.7. Браузер повторно отображает представление с новыми данными.

Пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
  </head>
  <body>
    <input ng-model="name">
    <p>Hello {{name}}!</p>
  </body>
</html>
```

Область видимости (scope)

Scope отвечает за обнаружение изменений в модели и обеспечивает контекст выполнения выражений. Scope вкладываются друг друга в иерархическую структуру и следят за всеми изменениями в DOM. (Смотрите документацию конкретной директивы, чтобы узнать какие из них создают собственную scope).

Следующий пример демонстрирует, что выражение name будет принимать разные значения, в зависимости от области видимости, в которой оно будет вызвано.

Пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="GreetCtrl">
      Hello {{name}}!
    </div>
    <div ng-controller="ListCtrl">
      <ol>
        <li ng-repeat="name in names">{{name}}</li>
      </ol>
    </div>
  </body>
</html>
```

Style.css

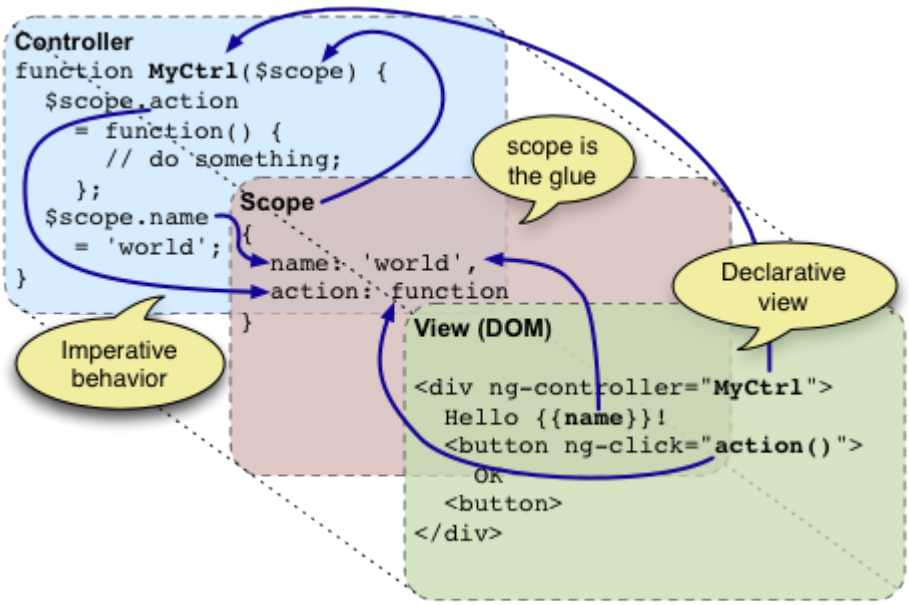
```
.show-scope .doc-example-live.ng-scope,
.show-scope .doc-example-live .ng-scope  {
  border: 1px solid red;
  margin: 3px;
}
```

Script.js

```
function GreetCtrl($scope) {
  $scope.name = 'World';
}

function ListCtrl($scope) {
  $scope.names = ['Igor', 'Misko', 'Vojta'];
}
```

Контроллер

Контролер связывает данные с представлением. Его задача заключается в построении модели и опубликовании ее в представлении расширяемом методами обратного вызова. Представление является проекцией scope на шаблон. Scope является слоем, который отображает модель на представление и направляет события в контроллер. Разделение контроллера и представления важно по следующим причинам:



- Контроллер пишется на javascript, и это обязательно. Он хорошо подходит для определения поведения приложения. Контроллер не должен содержать ни какой информации об отображении (DOM ссылки или HTML фрагменты).
- Представление написано на HTML, который является декларативным языком. Декларативный стиль хорошо подходит для определения представления. Представление не должно содержать ни какого поведения.
- Так как контроллер ничего не знает о представлении, может быть множество представлений на один и тот же контроллер. Это позволяет разрабатывать разные представления для отображения (re-skinning), для различных устройств (например мобильная версия и десктопная), а также позволяет легко их тестировать.

Пример:

Index.html

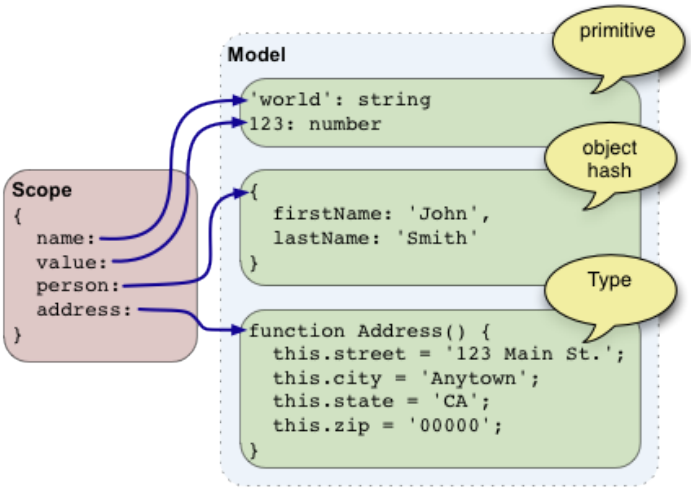
```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
```

```
  </head>
  <body>
    <div ng-controller="MyCtrl">
      Hello {{name}}!
      <button ng-click="action()">
        OK
      </button>
    </div>
  </body>
</html>
```

script.js

```
function MyCtrl($scope) {
  $scope.action = function() {
    $scope.name = 'OK';
  }

  $scope.name = 'World';
}
```

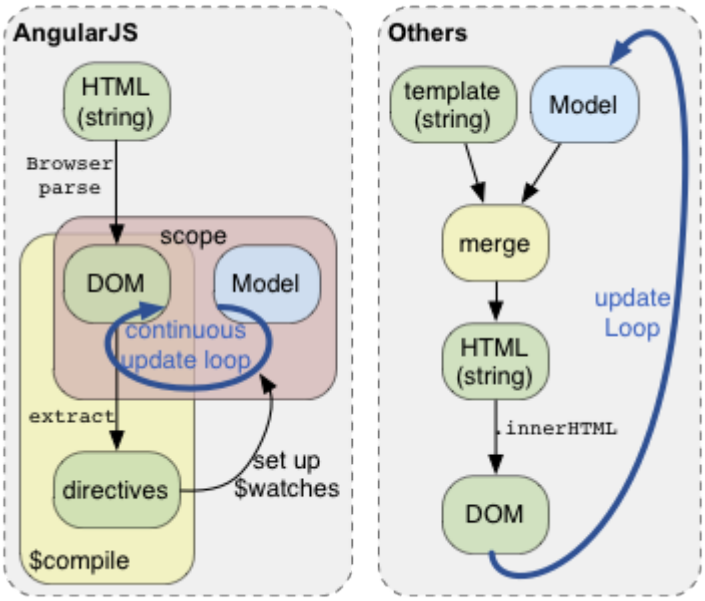
Модель



Модель – это данные, которые объединяются с шаблоном чтобы отобразить представление. Чтобы отобразить модель на представление, нужно иметь на нее ссылку в scope. В отличии от других фреймворков Angular не ограничивает модель ни чем и не предъявляет к ней ни каких требований. Не предлагается ни каких классов для реализации наследования, нет специальных методов для получения доступа к элементам модели. Модель может состоять из примитивных свойств, быть хэшем или полным классом. Обобщая сказанное – модель является простым объектом javascript.

Представление

Представление, это то, что пользователь видит на экране компьютера. Представление начинается с шаблона, который сливается с моделью данных и в конце, отображается на экране браузером.



Angular проповедует другой подход по реализаций представлений, не такой как многие другие фраймеворки:

- Другие фраймеворки – шаблоном является строка, с некоторым синтаксисом. Которая в процессе сливается с данными и на выходе получается строка с кодом HTML, которая и вставляется в код с помощью innerHTML. При изменении модели весь процесс повторяется заново. Ключевым здесь является то, что манипуляции с DOM проходят через строки.
- Angular – ключевое отличие в том, что система представлений работает непосредственно с DOM, а не со строками. Шаблоны по прежнему пишутся как строки, но это строки с валидным HTML, которые анализируются браузером, и на базе которых может быть построено дерево DOM, с которым и работает Angular через систему компиляции ($compile). Компилятор ищет в DOM директивы, создающие привязки к данным через объект $watch. В результате мы получаем постоянное обновление представления, для показа которого не требуется заново проводить слияние шаблона и модели.

Пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
  </head>
  <body>
    <div ng-init="list = ['Chrome', 'Safari', 'Firefox', 'IE'] ">
      <input ng-model="list" ng-list> <br>
      <input ng-model="list" ng-list> <br>
      <pre>list={{list}}</pre> <br>
      <ol>
        <li ng-repeat="item in list">
          {{item}}
        </li>
      </ol>
    </div>
  </body>
</html>
```

Директивы

Директивы – это поведение или трансформации DOM, которые вызываются когда в коде HTML присутствуют определенные элементы, атрибуты элементов или установлен определенный класс.

Директивы позволяют расширить HTML, чтобы в последующем из можно было использовать в декларативном стиле. Следующий пример демонстрирует привязку данных через директиву contentEditable.

Пример:

Index.html

```
<!doctype html>
<html ng-app="directive">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div contentEditable="true" ng-model="content">Edit Me</div>
    <pre>model = {{content}}</pre>
  </body>
</html>
```

Style.css

```
div[contentEditable] {
  cursor: pointer;
  background-color: #D0D0D0;
  margin-bottom: 1em;
  padding: 1em;
}
```

Script.js

```
angular.module('directive', []).directive('contenteditable', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      // view -> model
      elm.bind('blur', function() {
        scope.$apply(function() {
          ctrl.$setViewValue(elm.html());
        });
      });

      // model -> view
      ctrl.$render = function(value) {
        elm.html(value);
      };

      // load init value from DOM
      ctrl.$setViewValue(elm.html());
    }
```

```
20.   };
21. });
```

Фильтры

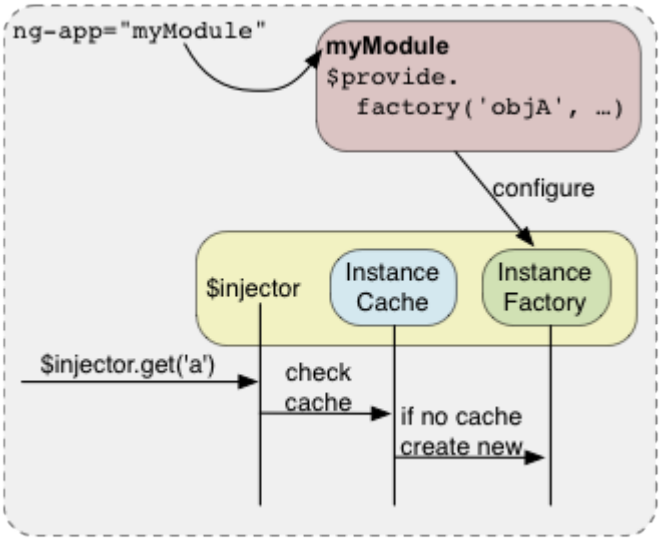
Фильтры предназначены для трансформаций данных. Обычно они используются для форматирования данных в зависимости от текущих локальных настроек пользователя (например языка). Они следуют стиля UNIX фильтров и имеют примерно такой же синтаксис.

Пример:

Index.html

```
1.  <!doctype html>
2.  <html ng-app>
3.    <head>
4.      <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
5.    </head>
6.    <body>
7.      <div ng-init="list = ['Chrome', 'Safari', 'Firefox', 'IE'] ">
8.        Number formatting: {{ 1234567890 | number }} <br>
9.        array filtering <input ng-model="predicate">
10.       {{ list | filter:predicate | json }}
11.     </div>
12.   </body>
13. </html>
```

Модули и инжектор



Инжектор – это локальный сервис, который существует в одном экземпляре для каждого приложения Angular. Инжектор позволяет проводить поиск экземпляра объекта по его имени. Он сохраняет в внутреннем кэше все вызовы объектов, чтобы обеспечить быстрый вызов объекта и эмитировать поведение одного экземпляра каждого объекта. Если объекта в кэше нет, тогда инжектор вызывает фабричный метод, который создает требуемый объект.

Модуль представляет собой конфигурацию для инжектора, известную как провайдер.

```
1.  // создание модуля
2.  var myModule = angular.module('myModule', [])
3.
4.  // конфигурация инжектора
5.  myModule.factory('serviceA', function() {
6.    return {
```

```
    // вместо {}, задайте здесь свой объект
  };
});

// создание инжектора и конфигурирование его модулем 'myModule'
var $injector = angular.injector(['myModule']);

// получение объекта у инжектора по его имени
var serviceA = $injector.get('serviceA');

// это вернет истину, т.к. экземпляр объекта повторно не создается.
$injector.get('serviceA') === $injector.get('serviceA');
```

Но настоящая магия инжектора в том, что можно вызвать методы у созданных экземпляров. Эта тонкая особенность позволяет запрашивать методы и типы через зависимости, а не ища их.

```
// напишите вашу функцию, которая требует двух объектов.
function doSomething(serviceA, serviceB) {
  // do something here.
}

// Angular предоставляет инжектора для вашего приложения
var $injector = ...;

////////////////////////////////////////////////
// старый стиль для получения зависимостей.
var serviceA = $injector.get('serviceA');
var serviceB = $injector.get('serviceB');

// сейчас вызываем функцию
doSomething(serviceA, serviceB);

////////////////////////////////////////////////
// а так можно сделать по новому.
// $injector будет вычислять аргументы функции автоматически
$injector.invoke(doSomething); // это новый стиль вызова функции
```

Обратите внимание, что единственное что вам нужно было сделать, это написать функцию со списком зависимостей в аргументах. Когда Angular будет выполнять функцию он автоматически разрешит зависимости и подставит их в требуемые аргументы функции.

Изучите ClockCtrl ниже, и обратите внимание, что он указывает список зависимостей в виде аргументов функции. Когда директива ng-controller работает, автоматически разрешаются все зависимости, требуемые контроллеру, и подставляются в его конструктор. Для вас нет необходимости создания зависимых объектов, их поиска и даже работы с инжектором.

Пример:

Index.html

```
<!doctype html>
<html ng-app="timeExampleModule">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="ClockCtrl">
      Current time is: {{ time.now }}
    </div>
  </body>
</html>
```

Script.js

```
angular.module('timeExampleModule', []).
  // Объявляется новый объект, вычисляющий время,
  // который будет доступен в инжекторе
  factory('time', function($timeout) {
    var time = {};

    (function tick() {
      time.now = new Date().toString();
      $timeout(tick, 1000);
    })();
    return time;
  });

// Обратите внимание, что вы просто запрашиваете
// нужные ресурсы. Их не нужно создавать или искать
function ClockCtrl($scope, time) {
  $scope.time = time;
}
```

Пространство имен Angular

Для предотвращения конфликта имен в Angular используется префикс $. Пожалуйста, не используйте его в своих приложениях, чтобы случайно не создать конфликт имен.

Директивы

Директивы – это способ добавить новое поведение при помощи новых элементов HTML. Во время компиляции DOM, директивы находятся в коде HTML и затем исполняются. Это позволяет с помощью директив реализовывать новое поведения или преобразовывать DOM.

Angular уже включает в себя множество директив, которые будут полезны при создании любого приложения, но легко также добавить и собственные директивы, в результате чего HTML может быть превращен в декларативный язык предметной области (DSL).

Выполнение директивы из кода HTML.

Директивы именуются с помощью верблюжей нотации, например ngBind. Директиву можно также вызывать и с помощью других имен, которые получаются из основного имени, путем добавления специальных символов, таких как ':', ' –' или '_'. Директивы могут иметь также префиксы 'x-' или 'data-', для того, чтобы код, содержащий новые директивы был валидным со стороны компилятор HTML. Сейчас покажем несколько эквивалентных имен одной и той же директивы: ng:bind, ng_bind, ng-bind, x-ng-bind или data-ng-bind.

Директивой может быть элемент, атрибут, класс или комментарий со специальным синтаксисом. Ниже приводятся все варианты расположения в коде директивы myDir (однако, большинство директив обычно ограничиваются синтаксисом атрибута).

```
<span my-dir="exp"></span>
<span class="my-dir: exp;"></span>
<my-dir></my-dir>
<!-- directive: my-dir exp -->
```

Директивы могут указываться множеством способов, но результат их выполнения всегда будет идентичным, что демонстрируется в следующем примере.

Пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl1">
      Hello <input ng-model='name'> <hr/>
      &lt;span ng:bind="name"&gt; <span ng:bind="name"></span> <br/>
      &lt;span ng_bind="name"&gt; <span ng_bind="name"></span> <br/>
      &lt;span ng-bind="name"&gt; <span ng-bind="name"></span> <br/>
      &lt;span data-ng-bind="name"&gt; <span data-ng-bind="name"></span> <br/>
      &lt;span x-ng-bind="name"&gt; <span x-ng-bind="name"></span> <br/>
    </div>
  </body>
</html>
```

Script.js

```
function Ctrl1($scope) {
  $scope.name = 'angular';
}
```

End to end test

```
1. it('should show off bindings', function() {
2.   expect(element('div[ng-controller="Ctrl1"] span[ng-bind]').text()).toBe('angular
   ');
3. });
```

Интерполяция строк

Во время компиляции компилятор находит текст и атрибуты использую службу $interpolate и определяет, есть ли в найденном внедренные выражения. Найденные выражения регистрируются с помощью сервиса $watch и будут обновляться в результате цикла $digest. Пример интерполяции вы можете увидеть в следующей строке кода.

```
<a href="img/{{username}}.jpg">Hello {{username}}!</a>
```

Процесс компиляции и нахождение директив.

Компиляция HTML проходит в три этапа:

1. На первом этапе строится дерево DOM используя стандартный синтаксис HTML. Исходя из этого нужно помнить, что шаблоны должны быть интерпретируемыми. Это отличие от большинства других систем шаблонов, которые работают со строками, а не с DOM элементами.
2. После составления дерева DOM вызывается метод $compile(). Метод обходит дерево DOM, и ищет директивы. Если директива будет найдена, она будет добавлена в список директив, связанных с данным элементом DOM. После того как будут найдены все директивы для элемента DOM, они сортируются по приоритету и выполняется компиляция каждой из них по порядку. Компиляция директивы может изменить дерево DOM, а также вызывает функцию link директивы, о которой мы поговорим позднее. Метод $compile возвращает связующую функцию, которая состоит из всех отдельных связующих функции, возвращенных в результате компиляции каждой директивы.
3. Связываются между собой шаблон и scope, для вычисления связующих функций, полученных на предыдущем этапе. Это в свою очередь вызывает связующие функции каждой директивы, что позволяет им зарегистрировать слушателей для событий, а также наблюдать за изменениями в scope. В результате получается связь в реальном времени между DOM и scope. Изменение в scope сразу же отображается в DOM.

```
4. var $compile = ...; // вставленный вами код
5. var scope = ...;
6.
7. var html = '<div ng-bind="exp"></div>';
8.
9. // Шаг 1. Разбор HTML в DOM
10. var template = angular.element(html);
11.
12. // Шаг 2. Компиляция шаблона
13. var linkFn = $compile(template);
14.
15. // Шаг 3. Связывание скомпилированного шаблона со scope.
16. linkFn(scope);
```

Причины разделения фазы компиляции и связывания

На данном этапе вы можете удивиться, почему процесс компиляции разделен на 2 процесса, собственно компиляции и связывания. Чтобы это понять, давайте проанализируем следующий пример с директивой повторителем (которая повторяет некоторый шаблон).

```
1. Hello {{user}}, you have these actions:
2. <ul>
3.   <li ng-repeat="action in user.actions">
4.     {{action.description}}
5.   </li>
6. </ul>
```

Краткий ответ заключается в том, что это необходимо для того, чтобы исключить частое изменение DOM для директив повторителей.

В приведенном выше примере в процессе компиляции, компилятор посещает каждый узел и ищет директивы. {{user}} - это пример интерполяции, ngRepeat – это другая директива. Но в ngRepeat есть проблема. Она должна быстро создавать элемент li для каждого элемента в user.actions. Это значит что она должна сохранить копию шаблона li и в последующем вставлять данные в эту копию, а затем вставить результат в элемент ul. Однако простое клонирование элемента li не помогает решить проблему. Нужно также скомпилировать его, чтобы разрешить элементы внутри него, в нашем примере это {{action.description}}. Наивно полагать, что простая вставка копии элемента li, и последующая компиляция решит проблему. Но компиляция на каждом клоне является медленной операцией, так как при этом затрагивается DOM, при поиске директив и их выполнении. Если мы применим подобный подход к директиве повторителю, которая будет работать примерно со 100 элементами, мы быстро столкнемся с проблемами производительности.

Решением является разбиение на стадии компиляции и связывания, в результате в фазе компиляции находятся директивы и сортируются по приоритету, а на стадии связывания конкретные экземпляры li получают свои актуальные значения из scope.

ngRepeat работает без компиляции вложенного в li содержимого. ngRepeat компилирует li отдельно. В результате для элемента li получаем связующую функцию, которая содержит указатели на все элементы li, а также читает и присваивает им значения из scope. Во время выполнения ngRepeat следит за выражениями, отслеживает добавление клонированных элементов, создает для них новую scope, а также вызывает связующую функцию для каждого li элемента.

Резюме:

- Функция компиляции – применяется довольно редко в директивах, так как большинство директив работают с конкретным элементом DOM, а не занимаются преобразованием шаблона элемента DOM. Любые операции, которые могут быть разделены между разными директивами, должны быть перенесены в связующую функцию для повышения производительности.

- Связующая функция – редко какая директива не имеет связующей функции. Она позволяет директиве регистрировать слушателей для конкретных клонированных экземпляров DOM элементов, а также копировать содержимое из scope в DOM.

Написание директив (короткая версия)

В этом примере мы будем строить директиву, которая отображает текущее время.

Пример:

Index.html

```
<!doctype html>
<html ng-app="time">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl2">
      Date format: <input ng-model="format"> <hr/>
      Current time is: <span my-current-time="format"></span>
    </div>
  </body>
</html>
```

Script.js

```
function Ctrl2($scope) {
  $scope.format = 'M/d/yy h:mm:ss a';
}

angular.module('time', [])
  // регистрация фабричного метода для директивы 'myCurrentTime'.
  // объявление зависимостей $timeout и dateFilter в фабричном методе.
  .directive('myCurrentTime', function($timeout, dateFilter) {
    // возврат связующей функции. (функция компиляции не нужна)
    return function(scope, element, attrs) {
      var format,  // формат данных
          timeoutId; // период обновления данных об времени

      // используется для обновления интерфейса пользователя
      function updateTime() {
        element.text(dateFilter(new Date(), format));
      }

      // отслеживание изменений выражений и обновление UI.
      scope.$watch(attrs.myCurrentTime, function(value) {
        format = value;
        updateTime();
      });

```

```
      // обновление времени раз в секунду
      function updateLater() {
        // сохранение для последующего использования
        timeoutId = $timeout(function() {
          updateTime(); // обновление DOM
          updateLater(); // обновление времени
        }, 1000);
      }

      // подписка на событие destroy для прекращения обновления UI
      // после удаления элемента.
      element.bind('$destroy', function() {
        $timeout.cancel(timeoutId);
      });

      updateLater(); // запуск процесса обновления UI.
    }
  });
```

Написание директив (длинная версия)

Существуют различные способы объявления директивы. Различие заключается в возвращаемом из фабричной функции значении. Можно вернуть либо объект определения директивы, который определяет свойства директивы (см. ниже), либо только функцию postLink такого объекта, все остальные свойства примут значения по умолчанию.

Вот пример директивы, которая объявляется с объектом определения директивы.

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  var directiveDefinitionObject = {
    priority: 0,
    template: '<div></div>',
    templateUrl: 'directive.html',
    replace: false,
    transclude: false,
    restrict: 'A',
    scope: false,
    compile: function compile(tElement, tAttrs, transclude) {
      return {
        pre: function preLink(scope, iElement, iAttrs, controller) { ... },
        post: function postLink(scope, iElement, iAttrs, controller) { ... }
      }
    },
    link: function postLink(scope, iElement, iAttrs) { ... }
  };
  return directiveDefinitionObject;
});
```

В большинстве случае вам не нужно будет так много свойств объекта директивы, тогда их можно упустить. Можно возвращать объект определения директивы, но в нем обязательно должна быть определена функция compile, а остальные члены объекта определения директивы получать значения по умолчанию. Таким образом выше указанная функция может быть упрощена до следующей:

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  var directiveDefinitionObject = {
    compile: function compile(tElement, tAttrs) {
      return function postLink(scope, iElement, iAttrs) { ... }
    }
  };
  return directiveDefinitionObject;
});
```

И наконец, большинство директив работает только с одним элементом, а не работает с шаблоном преобразований. В этом случае достаточно просто определить функцию postLink.

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  return function postLink(scope, iElement, iAttrs) { ... }
});
```

Фабричный метод

Фабричный метод отвечает за создание директивы. Он вызывается один раз, когда компилятор впервые находит директиву. В нем можно выполнить любую работу по инициализации директивы. Фабричный метод вызывается с помощью $injector.invoke, что в дальнейшем позволяет делать его инъекции в код с соблюдением всех аннотационных правил для инъекции.

Объект определения директивы

Объект определение директивы содержит команды для компилятора. Его атрибуты:

- name – имя текущей области видимости (scope). Не обязательный и по умолчанию это имя заданное при регистрации директивы.
- priority – приоритет, когда существуют несколько директив для одного элемента DOM, определяет порядок выполнения директив. Приоритет используется при определения порядка их исполнения на этапе компиляции. Директивы, обладающие большим приоритетом выполняются в первую очередь. Порядок выполнения директив с одинаковым приоритетом не определен.
- terminal – если установлен в true, то текущий приоритет будет установлен на последующие директивы, которые необходимо выполнить (любая директива с установленным приоритетом будет выполняться как будто приоритет не определен).
- scope – если установлено:
  - true – будет создана новая область видимости. Если несколько директив запросят создание новой области видимости, будет создана только одна такая область,

которую эти директивы и разделят. Новая область видимости не применяется для корневого шаблона, который всегда получает новую область видимости.

- {}(хэш объект) – создается новая **изолированная** область видимости. Изолированная область видимости отличается от обычной тем, что прототипное наследование от родительской области данных не применяется. Это полезно при создании повторно используемых компонентов, которые не должны читать или изменять данные родительской области видимости. Объект хэша принимает набор локальных свойств области видимости, которые должны быть унаследованы от родительской области видимости. Эти локальные свойства полезны для однотипной интерпретации шаблонов. Локально определенные свойства в хэше определяются и в области видимости.
  - @ или @attr – связывает локальные свойства со значением атрибута DOM. Результатом всегда является строка, т.к. DOM атрибуты это строки. Если имя атрибута не указанно, тогда считается что оно одинаковое с локальным именем. Директива `<widget my-attr="hello {{name}}">` в определении области видимости которой присутствует `scope: { localName:'@myAttr' }` то область видимости директивы widget будет отражать интерполированное значение “hello {{name}}”. При изменении значения `name`, изменится и значение свойства `localName`. Значение `name` считывается из родительской области видимости (а не с области компонента).
  - = или =attr – создает двунаправленную привязку свойства с именем attr в области видимости директивы с одноименным свойством родительской области видимости. Если значение attr не указанно, то предполагается что оно совпадает с локальным именем свойства. Директива `<widget my-attr="parentModel">` область видимости которой определяется как `scope: { localModel:'=myAttr' },` тогда свойство локальной области видимости директивы с именем `localModel` будет содержать значение свойства с именем `parenModel` родительской области видимости. Любые изменения свойства `parentModel` в родительской области видимости отображаются на свойство `localModel` локальной области видимости директивы, и наоборот.
  - & или &attr – выражение для выполнения в контексте родительской области. Если attr не задан, тогда считается что его значение совпадает с локальным именем свойства. Директива `<widget my-attr="count = count + value">`, область видимости которой определяется как `scope: { localFn:'&myAttr' }`, тогда свойство локальной области видимости localFn будет указывать на функцию обертку над `count = count + value.` Часто возникает необходимость передать данные из изолированной области данных директивы в выражение, выполняемое в контексте родительской области данных, то это можно сделать используя ссылку в локальной переменной `localFn` на обернутую функцию. Например, если выражение это `increment(amount),` и на него ссылается `localFn`, тогда можно вызвать это выражение внутри области видимости директивы с помощью `localFn(amount:22)`.

- controller – функция конструктор контроллера. Контроллер создается перед фазой связывания и разделяется с другими директивами, если они запросят его по имени. (см. атрибут require). Это позволяет директивам связываться друг с другом и дополнять поведение друг друга. В контроллер могут быть вставлены следующие сервисы:
  - $scope – текущая область видимости, ассоциированная с элементом.

    - $element – текущий элемент.
    - $attrs – объект с текущими атрибутами элемента.
    - $transclude – связующая функция включения, предварительно связанная с правильной областью видимости включение: `function(cloneLinkingFn)`.
- require – требовать от других контроллеров передавать в текущую директиву корректную связующую функцию. Require – принимает имя директивы контроллера для просмотра. Если контроллер не найден, генерируется ошибка. Название контроллера может иметь префикс:
    - ? – не вызывает ошибку. Это делается когда требуются дополнительные зависимости.
    - ^ - просмотреть родительские контроллеры и сделать так же.
- restrict – строка, которая может содержать один или несколько символов из подстроки ‘EACM’, которая ограничивает директиву определенным стилем декларации директивы в коде.
    - E – элемент: `<my-directive></my-directive>`
    - A – атрибут элемента: `<div my-directive="exp"> </div>`
    - C – класс элемента: `<div class="my-directive: exp;"></div>`
    - M – комментарий: `<!-- directive: my-directive exp -->`
- template – шаблон HTML для текущего элемента, которым заменяется определение директивы. В процессе замены сохраняются все атрибуты старого элемента соответствующими атрибутами нового. Смотрите ниже раздел «создание компонентов» для получения дополнительной информации.
- templateUrl – то же, что и шаблон, но шаблон загружается из указанного url. Поскольку загрузка шаблона является асинхронной операцией, то до ее завершения приостанавливаются фазы компиляции и связывания.
- replace – если установлен в true, тогда шаблон будет заменять собой определение директивы, иначе он будет добавляться к элементу.
- transclude – компилировать содержимое элемента и сделать его доступным для директивы. Обычно используется с ngTransclude. Функция включения получает связующую функцию, предварительно связанную с правильной областью видимости. При типичных параметрах директива создается с локальной областью видимости, но при transclude эта область видимости не наследуется, а является как-бы родственной родительской области данных. Это дает директиве возможность включать приватное состояние и связываться с родительской областью.
    - true – привязаться к содержимому директивы
    - ‘element’ – привязаться ко всему элементу, включая любые директивы, определенные с низким приоритетом.
- compile – это функция компиляции, которая описана в следующем разделе.
- link – связующая функция, которая описывается в следующем разделе. Это свойство используется только тогда, когда свойство compile не определено.

Функция компиляции(compile)

```
function compile(tElement, tAttrs, transclude) { ... }
```

Функция компиляции используется для изменения шаблона DOM. Большинство директив не затрагивают шаблона DOM, поэтому эта функция используется редко. Примерами директив, которым требуется функция компиляции, это директивы которые меняют DOM, такие как ngRepeat, или асинхронно загружают данные и отображают их, такие как ngView. Функция компиляции принимает следующие аргументы:

- tElement – шаблон элемента – это шаблон элемента при декларировании директивы. Это позволяет безопасно изменять только сам элемент и его дочерние элементы.
- tAttrs – шаблон атрибутов – это нормализованный список атрибутов, который разделяется между всеми директивами, требующими компиляции шаблона. См. атрибуты.
- transclude – включение связующей функции: `function(scope, cloneLinkingFn)`.

Пояснение: создание нового элемента по шаблону и создание его связи с данными не могут быть в одной функции, если шаблон клонируется. Это нужно по причине безопасности, в функции компиляции не нужно делать ничего, кроме преобразований DOM, которые применяются ко всем клонам. В частности регистрация слушателей для событий DOM, должна делаться в связующей функции, а не в функции компиляции.

Функция компиляции может возвращать либо функцию, либо объект.

- Возврат функции – равнозначен заданию связующей функции через свойство link, при создании директивы через объект определения директивы, которое при задании свойства compile должно быть пустым.
- Возвращение объекта со свойствами pre и post, которые ссылаются на связующие функции позволяет контролировать, какая функция должна быть вызвана во время фазы связывания первой (pre) или последней (post). Информация об пре-связывании и пост-связывании дана ниже.

Связующая функция(link)

```
function link(scope, iElement, iAttrs, controller) { ... }
```

Связующая функция применяется для регистрации слушателей событий DOM, и для его обновления, не затрагивая саму структуру. Она выполняется после клонирования шаблона. На этом этапе вводиться большинство логики для директив.

- scope – это область видимости для директивы, к которой будут привязываться отслеживание изменений с помощью $watch.
- iElement - экземпляр элемента – это элемент, на котором будет использована директива. Это позволяет безопасно манипулировать дочерними элементами только в postLink функции, так как дочернии элементы уже были связаны.
- iAttrs – атрибуты элемента – нормализованный список атрибутов элемента, разделяется между всеми директивами, подключенными к этому элементу. См. атрибуты.
- controller – экземпляр контроллера – экземпляр контроллера, если хотя бы одна директива на текущий момент его определяет. Контроллер является общим для всех директив, что позволяет использовать его как коммуникационных канал.

пре-связывания функция (pre-linking)

Выполняется до того, как будут связаны между собой дочерние элементы. Не безопасно для преобразований DOM, т.к. функция компиляции не сможет найти правильные элементы для связывания.

пост-связывания функция (post-linking)

Выполняется после связи дочерних элементов между собой. Это делает безопасным преобразование дерева дом в этой функции.

Атрибуты

Объект с атрибутами передается как параметр в функции link или compile, что позволяет им иметь доступ:

1. по нормализованным именам атрибутов – директивы могут присутствовать в коде во многих видах, эта возможность дает доступ к их содержимому через нормализованное имя атрибута.
2. Внешние связи директив – все директивы разделяют один объект с атрибутами, что позволяет им взаимодействовать между собой через него.
3. Поддержка интерполяции – позволяет через атрибут читать его значение, даже если оно содержит интерполяцию. При этом значение всегда будет актуальным.
4. Наблюдение за интерполируемыми атрибутами – для наблюдения за изменениями значения атрибутов, содержащих интерполяцию используется метод $observe. Это единственный и эффективный способ получать актуальные значения, поскольку во время стадии связывания интерполяция еще не разрешена и значения интерполируемых свойств в это время не определено.

```
function linkingFn(scope, elm, attrs, ctrl) {
  // получение значения атрибута
  console.log(attrs.ngModel);

  // изменение значения атрибута
  attrs.$set('ngModel', 'new value');

  // наблюдение за изменениями интерполиемого атрибута
  attrs.$observe('ngModel', function(value) {
    console.log('ngModel has changed value to ' + value);
  });
}
```

Подробнее о включениях и области видимости.

Часто требуется иметь повторно используемые компоненты. Ниже приведен псевдокод, который упрощенно показывает, как мог бы работать компонент диалогового окна.

```
<div>
  <button ng-click="show=true">show</button>
  <dialog title="Hello {{username}}."
          visible="show"
          on-cancel="show = false"
          on-ok="show = false; doSomething()">
     Body goes here: {{username}} is {{title}}.
  </dialog>
</div>
```

Клик на кнопке “show” открывает диалог. Диалог будет иметь заголовок, который связан с `username`, и он также имеет тело, которое мы хотели бы включать в диалог. Вот пример того, как могло бы выглядеть определение шаблона для виджета диалога.

```
<div ng-show="visible">
  <h3>{{title}}</h3>
  <div class="body" ng-transclude></div>
  <div class="footer">
    <button ng-click="onOk()">Save changes</button>
    <button ng-click="onCancel()">Close</button>
  </div>
</div>
```

Это не заработает, пока мы не поколдуем над областью видимости

Первый вопрос, который мы должны решить, наш диалог ожидает получить название title, которое к тому же привязано к имени пользователя username. Кроме того, в области видимости должны присутствовать функции onOk и onCancel, которые нужны для работы кнопок. Это ограничивает полезность виджета. Для создания локальных переменных, ожидаемых шаблоном, мы поступим следующим образом, привязав их значения к локальным переменным шаблона.

```
scope: {
  title: '@',       // использовать привязку к одноименному родительскому свойству
  onOk: '&',       // создать ссылку на одноименную функцию родителя
  onCancel: '&', // создать ссылку на одноименную функцию родителя
  visible: '=' // установка двунаправленной связи с одноименным свойством родителя
}
```

Создание локальных свойств виджета создает две проблеммы:

1. Изоляция – если пользователь забудет установить значение свойства title для виджета, шаблон будет привязываться к его родительскому элементу.
2. Включение – DOM может видеть внутренние свойства виджета, а это может привести к изменения свойств, используемых для привязки данных. В нашем примере свойство title виджета может переопределять свойство title включения.

Для решения проблемы изоляции директива объявляет новую обособленную область видимости. Действия изолированной области видимости не прототипически наследуются от дочерней области видимости, поэтому мы можем не беспокоится, если случайно удалим любые свойства.

Однако изолированная область видимости создает новую проблему, если включаемый в DOM виджет, является дочерним виджетом с изолированной областью видимости, то он не сможет привязаться ни к чему. По этой причине при включении дочерняя область видимости наследуется от начальной, (а не от родителя) перед включением изолированной области видимости. Это делает включение и изолированную область видимости, как-бы дочерними (они имеют доступ к членам друг друга).

Это может вызвать некоторое удивление, но это работает.

Поэтому окончательное определение директивы выглядит примерно так:

```
transclude: true,
scope: {
```

```
    title: '@',    // использовать привязку к одноименному родительскому свойству
    onOk: '&',              // создать ссылку на одноименную функцию родителя
    onCancel: '&',          // создать ссылку на одноименную функцию родителя
    visible: '=' //установка двунаправленной связи с одноименным свойством род-ля
},
restrict: 'E',
replace: true
```

Создание компонентов

Часто есть необходимость заменить одну директиву более сложной структурой DOM. Директивы могут стать сокращение для компонентов, из которых строятся приложения.

Ниже приведен пример создания виджета для многократного использования.

Пример:

Index.html

```
<!doctype html>
<html ng-app="zippyModule">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl3">
      Title: <input ng-model="title"> <br>
      Text: <textarea ng-model="text"></textarea>
      <hr>
      <div class="zippy" zippy-title="Details: {{title}}...">{{text}}</div>
    </div>
  </body>
</html>
```

Style.css

```
.zippy {
  border: 1px solid black;
  display: inline-block;
  width: 250px;
}
.zippy.opened > .title:before { content: '▼ '; }
.zippy.opened > .body { display: block; }
.zippy.closed > .title:before { content: '► '; }
.zippy.closed > .body { display: none; }
.zippy > .title {
  background-color: black;
  color: white;
  padding: .1em .3em;
  cursor: pointer;
```

```
}
.zippy > .body {
  padding: .1em .3em;
}
```

Script.js

```
function Ctrl3($scope) {
  $scope.title = 'Lorem Ipsum';
  $scope.text = 'Neque porro quisquam est qui dolorem ipsum quia dolor...';
}

angular.module('zippyModule', [])
  .directive('zippy', function(){
    return {
      restrict: 'C',
      // шаблон заменит директиву zippy.
      replace: true,
      transclude: true,
      scope: { title:'@zippyTitle' },
      template: '<div>' +
                  '<div class="title">{{title}}</div>' +
                  '<div class="body" ng-transclude></div>' +
                '</div>',
      // связующая функция добавляет поведение к шаблону
      link: function(scope, element, attrs) {
            // элемент Title
        var title = angular.element(element.children()[0]),
            // Opened / closed состояния
            opened = true;

        // Клик на названии показывает/скрывает подробности
        title.bind('click', toggle);

        // обработка изменений состояния
        function toggle() {
          opened = !opened;
          element.removeClass(opened ? 'closed' : 'opened');
          element.addClass(opened ? 'opened' : 'closed');
        }

        // инициализация zippy
        toggle();
      }
    }
  });
```

End to end test

```
it('should bind and open / close', function() {
  input('title').enter('TITLE');
```

```
3.    input('text').enter('TEXT');
4.    expect(element('.title').text()).toEqual('Details: TITLE...');
5.    expect(binding('text')).toEqual('TEXT');
6.
7.    expect(element('.zippy').prop('className')).toMatch(/closed/);
8.    element('.zippy > .title').click();
9.    expect(element('.zippy').prop('className')).toMatch(/opened/);
10. });
```

Выражения

Выражения – это JavaScript подобные фрагменты кода, которые обычно размещаются внутри двойных фигурных скобок. (пример {{expression}}). Выражения вычисляются при помощи сервиса $parse.

Пример допустимых выражений в Angular:

- 1+2
- 3*10 | currency
- user.name

Angular выражения и JavaScript выражения

Не думайте, что выражения Angular это просто код JavaScript. Angular не использует функцию eval(), для вычисления выражений. Вот отличия Angular выражений от JavaScript выражений:

- вычисление атрибутов – angular вычисляет атрибуты в контексте области видимости, а JavaScript в контексте глобального окна.
- Не строгость – angular при вычислении свойства на неопределенном объекте вернет null, а JavaScript сгенерирует исключение `NullPointerExceptions`.
- Ни каких операторов управления потоком – в angular внутри выражений нельзя использовать такие операторы как циклы, условия или выброс исключения.
- Фильтры – в angular можно пропустить результат вычисления выражения через цепочку фильтров. Например для преобразования даты с учетом местной специфики.

Если вы хотите выполнить произвольный код JavaScript, то вам нужно определить соответствующий метод в контроллере и вызвать его. Если вам необходимо использовать eval() метод из JavaScript внутри выражений angular, используйте метод $eval().

Пример:

Index.html

```
1. <!doctype html>
2. <html ng-app>
3.   <head>
4.     <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
5.   </head>
6.   <body>
7.     1+2={{1+2}}
8.   </body>
```

```
</html>
```

Ent to end test

```
it('should calculate expression in binding', function() {
  expect(binding('1+2')).toEqual('3');
});
```

Вы может попробовать различные варианты выражения, используя следующий пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Cntl2" class="expressions">
      Expression:
      <input type='text' ng-model="expr" size="80"/>
      <button ng-click="addExp(expr)">Evaluate</button>
      <ul>
       <li ng-repeat="expr in exprs">
         [ <a href="" ng-click="removeExp($index)">X</a> ]
         <tt>{{expr}}</tt> => <span ng-bind="$parent.$eval(expr)"></span>
        </li>
      </ul>
    </div>
  </body>
</html>
```

Script.js

```
function Cntl2($scope) {
  var exprs = $scope.exprs = [];
  $scope.expr = '3*10|currency';
  $scope.addExp = function(expr) {
     exprs.push(expr);
  };

  $scope.removeExp = function(index) {
    exprs.splice(index, 1);
  };
}
```

End to end test

```
it('should allow user expression testing', function() {
   element('.expressions :button').click();
   var li = using('.expressions ul').repeater('li');
   expect(li.count()).toBe(1);
   expect(li.row(0)).toEqual(["3*10|currency", "$30.00"]);
});
```

Вычисление свойства

Вычисление всех свойств проводиться в области видимости. В отличие от JavaScript, для того чтобы получить глобальные свойства окна, нужно использовать сервис $window. К примеру, вы хотите вызвать функцию alert() глобального окна, для этого нужно сделать следующий вызов $window.alert(). Это сделано намеренно, чтобы предотвратить не намеренное изменение или чтение глобальных свойств (которые часто являются источник трудно отлавливаемых ошибок).

Пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div class="example2" ng-controller="Cntl1">
      Name: <input ng-model="name" type="text"/>
      <button ng-click="greet()">Greet</button>
    </div>
  </body>
</html>
```

Script.js

```
function Cntl1($window, $scope){
  $scope.name = 'World';

  $scope.greet = function() {
    ($window.mockWindow || $window).alert('Hello ' + $scope.name);
  }
}
```

End to end test

```
it('should calculate expression in binding', function() {
  var alertText;
  this.addFutureAction('set mock', function($window, $document, done) {
    $window.mockWindow = {
      alert: function(text){ alertText = text; }
```

```
    };
    done();
  });
  element(':button:contains(Greet)').click();
  expect(this.addFuture('alert text', function(done) {
    done(null, alertText);
  })).toBe('Hello World');
});
```

Прощение ошибок

Вычисление выражений прощает попытки доступа к свойствам на неопределенном объекте или на null. В JavaScript вычисление выражения a.b.c приведет к ошибке, если объектом не является a или b. Хотя это имеет смысл в общем случае, в angular вычисление выражений часто используется для привязки данных, которые часто выглядят следующим образом:

```
{{a.b.c}}
```

Это имеет больше смысла при показе объектов со значение ничего, так как во время привязки еще может не быть требуемых данных, а в последующем они будут определены. Если бы генерировалось исключение, тогда приходилось бы писать громоздкие выражения для привязки данных, которые бы загромождали код, например: `{{((a||{}).b||{}).c}}`.

В итоге обращение к свойству неопределенного объекта или объекта со значением null, просто вернет undefined.

Нет операторов управления потоком

Вы не можете писать операторы управления потоком в выражениях. Причиной этого является философия angular, где логика приложения должна находится в контроллерах, а не в представлениях. Если вам нужно ветвление, цикл или выброс исключения из выражения в представлении, просто определите метод в контроллере и сошлитесь на него.

Фильтры

При представлении данных пользователю возможно нужно будет их преобразовать из исходного формата, в формат удобный для показа пользователю. Например, у вас могут быть данные, которые необходимо отформатировать в соответствии с языковыми настройками пользователя, прежде чем их отобразить. Вы можете передать в выражение цепочку фильтров для этого, вот так:

```
name | uppercase
```

Значение name просто передается на вход фильтра uppercase, который переводит его в верхний регистр. Для использования цепочки фильтров используется следующий синтаксис:

```
value | filter1 | filter2
```

Вы можете также передавать в фильтры параметры, например для отображение числа с двумя знаками после запятой нужно сделать следующее:

```
123 | number:2
```

$

Не удивляйтесь, это просто значение префикса. Применяя его angular пытается отделить свое api от других. Если бы angular не использовал бы префикс, то вычисление a.length(), вернуло бы undefined, поскольку ни ‘a’, ни angular не определяют такое свойство.

Возможно, в будущей версии angular будет возможность добавлять собственные свойства, что бы изменило поведения выражения выше. Что еще хуже, у разработчика есть возможность создавать множество методов и свойств, в результате чего может возникнуть конфликт имен. Чтобы это предотвратить, как раз и используется этот префикс.

Формы

Элементы управления (input, select, textarea) позволяют пользователю вводить данные. Форма – это коллекция элементов управления с целью группирования связанных элементов управления в одном месте. Имейте ввиду, что проверка ввода на стороне пользователя является хорошей практикой, но ее можно легко обойти, следовательно не стоит ей доверять. Проверка на стороне сервера необходима для безопасной работы приложения.

Простая форма

Основная директива для создания двухсторонней привязки – ngModel. Она привязывает данные отображаемые в представлении к данным в области видимости, и одновременно отображает на эти данные ввод пользователя. Кроме этого, она обладает собственным api, для того, чтобы другие директивы могли изменить ее поведение.

Пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form novalidate class="simple-form">
        Name: <input type="text" ng-model="user.name" /><br />
        E-mail: <input type="email" ng-model="user.email" /><br />
        Gender: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />
        <button ng-click="reset()">RESET</button>
        <button ng-click="update(user)">SAVE</button>
      </form>
      <pre>form = {{user | json}}</pre>
      <pre>master = {{master | json}}</pre>
    </div>
  </body>
</html>
```

Script.js

```
function Controller($scope) {
  $scope.master= {};

  $scope.update = function(user) {
    $scope.master= angular.copy(user);
  };

  $scope.reset = function() {
    $scope.user = angular.copy($scope.master);
  };

  $scope.reset();
}
```

Примечание: novalidate используется для отключения родной процедуры валидации форм браузером.

Использование CSS классов

Чтобы различать вид формы, а также элементов управления ngModel добавляет следующие классы:

- ng-valid
- ng-invalid
- ng-pristine
- ng-dirty

В следующем примере используются классы CSS, для отображение результатов валидации каждого элемента формы. В примере user.name и user.email являются обязательными, но они рисуются с красным фоном, когда это не выполняется. Это гарантирует, что пользователь не увидит ошибки до своего действия с элементом управления, и не пройдет валидацию, пока не введет значение в поле.

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form novalidate class="css-form">
        Name:
          <input type="text" ng-model="user.name" required /><br />
        E-mail: <input type="email" ng-model="user.email" required /><br />
        Gender: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />
```

```
15.         <button ng-click="reset()">RESET</button>
16.         <button ng-click="update(user)">SAVE</button>
17.       </form>
18.     </div>
19.
20.     <style type="text/css">
21.       .css-form input.ng-invalid.ng-dirty {
22.         background-color: #FA787E;
23.       }
24.
25.       .css-form input.ng-valid.ng-dirty {
26.         background-color: #78FA89;
27.       }
28.     </style>
29.   </body>
30. </html>
```

Script.js

```
1.  function Controller($scope) {
2.    $scope.master = {};
3.
4.    $scope.update = function(user) {
5.      $scope.master = angular.copy(user);
6.    };
7.
8.    $scope.reset = function() {
9.      $scope.user = angular.copy($scope.master);
10.   };
11.
12.   $scope.reset();
13. }
```

Связь формы и состояния элементов управления

Форма – это экземпляр FormController. Экземпляр формы, если нужно, можно получить в области видимости, использую атрибут формы name. Аналогично, элементы управления – это экземпляры NgModelController. Экземпляры элементов управления также можно получить внутри формы, используя их атрибут ‘name’. Это значит, внутренне состояние формы может быть связанно с внутренним состоянием любого элемента управления, используя стандартное связывание.

Все это позволяет расширить приведенный ранее пример следующими функциями:

- Кнопка сброса включена, если форма имеет некоторые значения.
- Кнопка сохранить доступна тогда, когда форма валидна и в ней есть изменения.
- Вывод определенных пояснений, в случае ошибок в user.email и user.agree.

Index.html

```
1.  <!doctype html>
2.  <html ng-app>
```

```
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form name="form" class="css-form" novalidate>
        Name:
          <input type="text" ng-model="user.name" name="uName" required /><br />
        E-mail:
          <input type="email" ng-model="user.email" name="uEmail" required/><br />
        <div ng-show="form.uEmail.$dirty && form.uEmail.$invalid">Invalid:
          <span ng-show="form.uEmail.$error.required">Tell us your email.</span>
          <span ng-show="form.uEmail.$error.email">This is not a valid email.</spa
n>
        </div>

        Gender: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />

        <input type="checkbox" ng-model="user.agree" name="userAgree" required />
        I agree: <input ng-show="user.agree" type="text" ng-model="user.agreeSign"
                 required /><br />
        <div ng-show="!user.agree || !user.agreeSign">Please agree and sign.</div>

        <button ng-click="reset()" ng-disabled="isUnchanged(user)">RESET</button>
        <button ng-click="update(user)"
                ng-disabled="form.$invalid || isUnchanged(user)">SAVE</button>
      </form>
    </div>
  </body>
</html>
```

Script.js

```
function Controller($scope) {
  $scope.master= {};

  $scope.update = function(user) {
    $scope.master= angular.copy(user);
  };

  $scope.reset = function() {
    $scope.user = angular.copy($scope.master);
  };

  $scope.isUnchanged = function(user) {
    return angular.equals(user, $scope.master);
  };

  $scope.reset();
```

```
}
```

Пользовательская валидация

Angular обеспечивает базовую реализацию для всех основных типов элементов управления HTML5 (text, number, url, email, radio, checkbox), а также обеспечивает некоторыми директивами для проверки ввода (required, pattern, minlength, maxlength, min, max). Определение собственного валидатора, это определение директивы, которая добавляет пользовательскую функцию проверки к контроллеру ngModel. Чтобы получить контроллер, директива декларирует зависимость, как показано в примере ниже. Проверку нужно проводить в двух местах:

1. Обновление представления из модели – всякий раз, когда изменяется связанное свойство модели, вызываются все функции в массиве NgModelController#$formatters, при этом каждая функция может изменить состояние валидации и отформатировать значение через вызов NgModelController#$setValidity.
2. Обновление модели из представления – аналогично, при взаимодействии пользователя с элементом управления, вызывается метод NgModelController#$setViewValue. Это в свою очередь вызывает все функции из массива NgModelController#$parsers, где каждая из них может конвертировать значение и изменить состояние валидации для формы через вызов NgModelController#$setValidity.

В этом примере мы создадим две директивы.

1. Директива integer – проверяет, является ли значение целым действительным числом. Например 1,23 не является целым числом, т.к. оно содержит дробную часть. Обратите внимание, что мы вызвали метод unshift на массиве parsers, для добавления нашего обработчика в начало этого массива. Это необходимо, так как мы хотим, чтобы наш анализатор стартовал первым, до конверсии строки в число.
2. Следующая директива – smart-float. Оба формата числа 1,2 или 1.2 считаются валидным числом и преобразуются к 1.2. Обратите внимание что в шаблоне мы не используем элемент управления с типом ‘number’, т.к. в этом случае браузер не позволит ввести не валидное по его мнению значение.

Index.html

```
<!doctype html>
<html ng-app="form-example1">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form name="form" class="css-form" novalidate>
        <div>
          Size (integer 0 - 10):
          <input type="number" ng-model="size" name="size"
                 min="0" max="10" integer />{{size}}<br />
          <span ng-show="form.size.$error.integer">This is not valid integer!</spa
n>
```

```
          <span ng-show="form.size.$error.min || form.size.$error.max">
            The value must be in range 0 to 10!</span>
        </div>

        <div>
          Length (float):
          <input type="text" ng-model="length" name="length" smart-float />
          {{length}}<br />
          <span ng-show="form.length.$error.float">
            This is not a valid float number!</span>
        </div>
      </form>
    </div>
  </body>
</html>
```

Script.js

```
var app = angular.module('form-example1', []);

var INTEGER_REGEXP = /^\-?\d*$/;
app.directive('integer', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$parsers.unshift(function(viewValue) {
        if (INTEGER_REGEXP.test(viewValue)) {
          // it is valid
          ctrl.$setValidity('integer', true);
          return viewValue;
        } else {
          // it is invalid, return undefined (no model update)
          ctrl.$setValidity('integer', false);
          return undefined;
        }
      });
    }
  };
});

var FLOAT_REGEXP = /^\-?\d+((\.|\,)\d+)?$/;
app.directive('smartFloat', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$parsers.unshift(function(viewValue) {
        if (FLOAT_REGEXP.test(viewValue)) {
          ctrl.$setValidity('float', true);
          return parseFloat(viewValue.replace(',', '.'));
        } else {
          ctrl.$setValidity('float', false);
          return undefined;
```

```
        }
      });
    }
  };
});
```

Реализация собственных элементов управления для форм (используя ngModel)

Angular реализует все основные элементы управления для форм (input, select, textarea), которых достаточно для большинства случаев. Тем не менее, если вам нужно больше гибкости, вы можете написать свой собственный элемент управления для формы используя директивы.

Для того, чтобы пользовательский элемент управления работал с ngModel для реализации двухсторонней привязки, нужно выполнить следующие условия:

- Реализовать метод $render, который отвечает за отрисовку данных, после получение их от NgModelController#$formatters.
- Вызвать NgModelController#$setViewValue метод, когда пользователь взаимодействует с элементом управления и модель нуждается в обновлении.

Смотрите $compileProvider.directive для получения дополнительной информации.

В следующем примере показано, как добавить привязку данных к редактирующим элементам управления.

Index.html

```
<!doctype html>
<html ng-app="form-example2">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div contentEditable="true" ng-model="content" title="Click to edit">Some</div
>
    <pre>model = {{content}}</pre>

    <style type="text/css">
      div[contentEditable] {
        cursor: pointer;
        background-color: #D0D0D0;
      }
    </style>
  </body>
</html>
```

Script.js

```
angular.module('form-example2', []).directive('contenteditable', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      // view -> model
      elm.bind('blur', function() {
        scope.$apply(function() {
          ctrl.$setViewValue(elm.html());
        });
      });

      // model -> view
      ctrl.$render = function() {
        elm.html(ctrl.$viewValue);
      };

      // load init value from DOM
      ctrl.$setViewValue(elm.html());
    }
  };
});
```

i18n и l10n

i18n и l10n в приложениях Angular.

Интернационализация, сокращенно i18n, это процесс разработки приложения, при котором оно легко может локализовано для различных языков и культур. Локализация, сокращенно l10n, процесс адаптации приложения под требования конкретной языковой или культурной группы. Для разработчиков интернационализация обозначает отделить от приложения все строки и другие зависимые от локали данные (форматы дат, валюты) от приложения. Локализация означает предоставить переводы, форматы дат и прочие данные для требуемой языковой среды.

Какова сейчас поддержка в angular i18n и l10n?

В настоящее время angular поддерживает i18n и l10n для фильтров даты и времени, чисел и валюты.

Дополнительно angular поддерживает локализацию через поддержку плюрализации с помощью директивы ngPluralize.

Все локализуемые angular компоненты зависят от набора правил языковых стандартов и управляются с помощью сервиса $locale.

Для читателей, которые хотят сразу перейти к примерам, есть несколько страниц, которые наглядно демонстрируют, как использовать фильтры Angular с различными наборами языковых правил. Вы может найти их либо на Github, либо в папке i18n/l10n в пакете для разработчиков Angular.

Что такое код региональных настроек?

Региональные настройки – это конкретный географический, политический или культурный регион. Наиболее часто используются коды региональной настройки, состоящие из двух частей: кода языка и кода страны. Например en-US, en-AU, zh-CH, они все имеют как код языка, так и код страны. Коды страны не являются обязательными, просто коды языка также применяются на практике, например en, zh. Зайдите на сайт ICU для получения более подробной информации о кодах региональной настройки.

Какие языки поддерживает angular?

Angular определяет форматы чисел и дат в отдельных файлах для каждого языка и конкретной местности. Списки поддерживаемых расположений можно найти здесь.

Обеспечение правил языка в Angular.

Angular предоставляет два подхода для подключения языковых правил:

Предварительно созданный набор правил Angular.

Можно предварительно связать требуемый файл с национальными настройками, загрузив его после загрузки основного скрипта angular.

Например на системах *nix чтобы создать файл, содержащий правила локализации для немецкого языка, нужно сделать следующее:

```
cat angular.js i18n/angular-locale_de-ge.js > angular_de-ge.js
```

Когда приложение стартует с загруженным файлом angular_de-ge.js, вместо универсального angular.js, Angular автоматически конфигурируется на поддержку немецкого языка.

Включение языкового js файла в index.html

Также позволяется включать локализованный файл в скрипты подгружаемые index.html. Например для немецких пользователей страница index.html выглядела бы примерно так:

```
1. <html ng-app>
2.  <head>
3. ….
4.    <script src="angular.js"></script>
5.    <script src="i18n/angular-locale_de-ge.js"></script>
6. ….
7.  </head>
8. </html>
```

Сравнение двух подходов

Оба описанных выше подходов требует, чтобы вы готовили локализованные версии файлов (в первом подходе index.html, во втором – angular-locale.js) для каждого языка, который вы хотите поддерживать. Кроме этого нужно настроить сервер, чтобы он отдавал нужный конкретному пользователю локализованный файл.

Тем не менее второй подход, скорее всего будет медленнее, т.к. требуется загрузка дополнительного файла.

«Ошибки»

Символ валюты ‘gotcha’

Angular предоставляет фильтр для отображения символа валюты, который использует символ валюты по умолчанию установленный в сервисе $locale, или вы можете передать ему любой символ валюты, который он должен использовать. Если ваше приложение будет использоваться только в одном месте, тогда можно спокойно положиться на поведение по умолчанию. Однако, если ваше приложение будет использоваться в разных странах, вы должны самостоятельно определить для них символы валют, чтобы они были адекватными.

Например, если вы хотите отобразить остаток на счете в размере 1000$, и будете его показывать в соединенных штатах, тогда можно просто вызвать фильтр {{1000 | currency}}, но если пользователь откроет это приложение в Японии, то остаток на счете будет показан как 1000 йен. Ваш клиент вас не поймет.

В этом случае вам нужно изменить символ валюты по умолчанию на символ доллара. Для этого просто установите параметр фильтра, и фильтр постоянно будет отображать значение в долларах.

Перевод длины ‘gotcha’

При переводе даты и времени в формат локали, нужно учитывать, что длина результата может сильно отличаться для разных стран. Нужно подстроить стили, чтобы убедиться что локализованная версия приложения работает правильно, в на страницах не перекрываются данные.

Часовые пояса

Имейте ввиду, что angular фильтр datetime использует настройку часового пояса из браузера. Ни angular, ни JavaScript в настоящее время не поддерживают задание часового пояса разработчиком.

Поддержка IE

Обзор

Этот раздел описывает особенности работы с пользовательскими атрибутами и тегами в IE. Прочитайте его если собираетесь развертывать приложение angular на IE8.0 или более раней.

Короткая версия

Для того, чтобы приложение Angular работало в IE, пожалуйста, убедитесь в следующем:

1. Что IE поддерживает метод JSON.stringify (IE7.0 его не поддерживает), и если не поддерживает, используйте JSON2 или JSON3 для включения его поддержки.

```
<!doctype html>
<html xmlns:ng="http://angularjs.org">
  <head>
    <!--[if lte IE 8]>
      <script src="/path/to/json2.js"></script>
    <![endif]-->
  </head>
```

```
  <body>
    ...
  </body>
</html>
```

2. Добавьте id="ng-app" для корневого элемента, в котором содержится атрибут ng-app.

```
<!doctype html>
<html xmlns:ng="http://angularjs.org" id="ng-app" ng-app="optionalModuleName">
  ...
</html>
```

3. Не используйте пользовательские теги, такие как <ng:view>, вместо этого используйте вариант с атрибутами - <div ng-view>, или

4. Если вы все же хотите применять пользовательские теги, тогда вам нужно предпринять следующие шаги, чтобы добавить их поддержку в IE:

```
<!doctype html>
<html xmlns:ng="http://angularjs.org" id="ng-app" ng-app="optionalModuleName">
  <head>
    <!--[if lte IE 8]>
      <script>
        document.createElement('ng-include');
        document.createElement('ng-pluralize');
        document.createElement('ng-view');

        // Optionally these for CSS
        document.createElement('ng:include');
        document.createElement('ng:pluralize');
        document.createElement('ng:view');
      </script>
    <![endif]-->
  </head>
  <body>
    ...
  </body>
</html>
```

Обратите внимание:

- Нужно указать `xmlns:ng` - *namespace*, которая указывает пространство имен для пользовательских тегов, которые вы хотите использовать.
- Для каждого пользовательского тега нужно вызвать метод `document.createElement(yourTagName)`, чтобы объявить его в IE.

Длинная версия

В IE есть проблемы с пользовательскими тегами, которые не являются стандартными именами тегов HTML. Они делятся на две категории, и каждая исправляется по разному:

- Если имя тега начинается с префикса `my:`, тогда считается что это объявление пространства имен XML, и оно должно быть объявлено как положено `<html xmlns:my="ignored">`
- Если тег не имеет в имени «:», но все же не является стандартным тегом HTML, его нужно предварительно создать с помощью `document.createElement('my-tag')`
- Если вы планируете стилизовать пользовательский тег с помощью селекторов CSS, тогда нужно предварительно его создать с помощью `document.createElement('my-tag')` независимо от того есть или нет пространство имен.

Хорошая новость

Хорошей новостью является то, что все ограничения касаются только пользовательских тегов, синтаксис с помощью атрибутов не требует ни каких действий и поддерживается изначально.

Что произойдет, если я этого не сделаю?

Предположим, у вас есть пользовательский тег myTag (ну или другой, результат будет тем же).

```
1. <html>
2.   <body>
3.     <mytag>some text</mytag>
4.   </body>
5. </html>
```

Который разбирается на следующий DOM.

```
1. #document
2.   +- HTML
3.      +- BODY
4.         +- mytag
5.            +- #text: some text
```

Ожидается, что в результате разбора у элемента BODY будет дочерний элемент mytag, который в свою очередь будет внутри содержать текстовый узел. Но в IE это не так, а вот так:

```
1. #document
2.   +- HTML
3.      +- BODY
4.         +- mytag
5.         +- #text: some text
6.         +- /mytag
```

Мы видим, что тег BODY имеет три дочерних элемента:

- Самозакрывающийся тег mytag. Еще один пример такого тега, это
. Такие теги не могут иметь дочерних элементов, и «/» является обязательной.
- Текстовый узел, который должен быть дочерним, для тега mytag, а в данном случае он является сестринским.

- Не корректный закрывающий тег “/ mytag”. Это ошибочный тег, так как в имени тега запрещается иметь знак «/». Если же это запирающий элемент, тогда он не должен быть частью структуры DOM, так как он просто очерчивает границы элемента.

CSS стили пользовательских тегов

Чтобы CSS стили работали с пользовательскими тегами нужно предварительно вызвать метод `document.createElement('my-tag')` независимо от того, есть пространство имен в имени, или его нет.

```
<html xmlns:ng="needed for ng: namespace">
  <head>
    <!--[if lte IE 8]>
      <script>
        // needed to make ng-include parse properly
        document.createElement('ng-include');

        // needed to enable CSS reference
        document.createElement('ng:view');
      </script>
    <![endif]-->
    <style>
      ng\:view {
        display: block;
        border: 1px solid red;
      }

      ng-include {
        display: block;
        border: 1px solid blue;
      }
    </style>
  </head>
  <body>
    <ng:view></ng:view>
    <ng-include></ng-include>
    ...
  </body>
</html>
```

Введение

Angular – это клиентская библиотека, полностью написанная на JavaScript. Она работает со старыми технологиями (HTML, CSS, JavaScript), чтобы сделать разработку приложений проще и быстрее чем когда-либо раньше.

Одной из важных особенностей является то, что angular предоставляет дополнительный уровень абстракции между разработчиком, и задачами низкого уровня. Angular автоматически берет на себя многие из этих низкоуровневых задач, такие как:

- Манипуляции с DOM
- Настройка слушателей и генерация событий

- Проверка ввода

Поэтому разработчики могут сконцентрироваться на бизнес логике приложения, а не работать над повторяющимися, подверженными ошибкам задачами более низкого уровня.

В то же время, хотя angular и упрощает разработку, он дает возможность использовать сравнительно сложные технологии на стороне клиента, в т.ч.:

- Разделение данных, логики приложения и представления
- Привязки данных между данными и представлениями
- Сервисы (общие операции веб-приложения, реализованные в виде замещающих объектов)
- Внедрение зависимостей (используется в основном для совместной работы с сервисами)
- Расширенный компилятор HTML (полностью написан на JavaScript)
- Простое тестирование

Одностраничные и обычные приложения.

Angular поддерживает оба типа веб-приложений, но все же он создан для создания одностраничных приложений. Он поддерживает историю посещения страниц, кнопки туда и обратно, а также закладки для одностраничных приложений.

Обычно вы не захотите грузить Angular каждый раз при загрузке страницы, как бы это было при загрузке страниц обычного клиент-серверного приложения. Тем не менее в этом появился бы смысл, если добавленная функциональность angular нужна вашему приложению. Лучше все же перейти к одностраничному приложению.

Модули

Что такое модуль?

Большинство приложений имеют основной метод, который создает экземпляр приложения и инициализирует приложение. Angular приложения не имеют основного метода. Вместо этого модуль декларативно определяет, как должно быть загружено приложение. Есть несколько преимуществ такого подхода:

- Этот процесс является более декларативной которые легче понять
- В юнит-тестирования нет необходимости загружать все модули, которые могут помочь в написании юнит-тестов.
- Дополнительные модули могут быть загружены в сценарии тестов, которые могут переопределить некоторые конфигурации и помогают end-to-end тестированию приложения
- Исходный код сторонних производителей может быть упакован в модуль для многоразового использования.
- Модули могут быть загружены в асинхронном / синхронном порядке (из-за задержки загрузки модуля выполнения).

Основы

Хорошо, я спешу. Как я могу сделать модуль Hello world рабочим?

Важно:

- `Module` API
- Обратите внимание на ссылки на модуль по имени myApp в ng-app="myApp", это позволяет загрузить все приложение сконфигурированное с помощью вашего модуля.

Пример:
Index.html

```
<!doctype html>
<html ng-app="myApp">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div>
      {{ 'World' | greet }}
    </div>
  </body>
</html>
```

Script.js

```
// определение модуля
var myAppModule = angular.module('myApp', []);

// конфигурация модуля.
// в этом примере создается фильтр
myAppModule.filter('greet', function() {
 return function(name) {
    return 'Hello, ' + name + '!';
  };
});
```

Рекомендуется установка

Хотя вышеприведенный пример прост, он не будет масштабироваться до больших приложений. Вместо этого мы рекомендуем вам разбить приложение на несколько модулей, например, так:

- Service модуль, для обслуживания сервисов.
- Directives модуль, для декларации директив
- Filters модуль, для декларации фильтров
- И модуль уровня приложения, который зависит от модулей выше, и который содержит код инициализации.

Причиной этого разделения является то, что в тестах, часто бывает необходимо игнорировать код инициализации, который имеет тенденцию быть трудно проверяемым. Выделив его в отдельный модуль, его можно легко игнорируется в тестах. Тесты также могут загружать только те из модулей, которые имеют отношение к тестовым испытаниям.

Структура выше, это лишь предложение, так что не стесняйтесь адаптировать ее к вашим потребностям.

Пример:
Index.html

```
<!doctype html>
<html ng-app="xmpl">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
```

```
    <div ng-controller="XmplController">
      {{ greeting }}!
    </div>
  </body>
</html>
```

Script.js

```
angular.module('xmpl.service', []).
  value('greeter', {
    salutation: 'Hello',
    localize: function(localization) {
      this.salutation = localization.salutation;
    },
    greet: function(name) {
      return this.salutation + ' ' + name + '!';
    }
  }).
  value('user', {
    load: function(name) {
      this.name = name;
    }
  });

angular.module('xmpl.directive', []);

angular.module('xmpl.filter', []);

angular.module('xmpl', ['xmpl.service', 'xmpl.directive', 'xmpl.filter']).
  run(function(greeter, user) {
    // This is effectively part of the main method initialization code
    greeter.localize({
      salutation: 'Bonjour'
    });
    user.load('World');
  })


// A Controller for your app
var XmplController = function($scope, greeter, user) {
  $scope.greeting = greeter.greet(user.name);
}
```

Загрузки модуля и зависимостей

Модуль представляет собой набор конфигурации и запускает блоки, которые быть сконфигурированы к применению во время загрузки машины. В своей простейшей форме модуль состоит из набора блоков двух видов:

1. Блок конфигурации – начинает выполняться во время фазы регистрации и конфигурации. Только поставщики и константы могут быть введены в конфигурационный блок. Это сделано для предотвращения случайного создания экземпляра сервиса, прежде чем он будет полностью настроен.

2. Блок запуска – начинает выполняться после создания инжектора, и используется для старта приложения. Только экземпляры и константы могут быть введены в блок запуска. Это сделано для предотвращения дальнейшей конфигурации системы во время выполнения приложения.

```
angular.module('myModule', []).
  config(function(injectables) { // provider-injector
    // Это экземпляр конфигурационного блока.
    // Их может быть несколько.
    // Вы можете также добавить зависимость от поставщика (не экземпляра)
    // в этот конфигурационных блок.
  }).
  run(function(injectables) { // instance-injector
    // Это пример блока запуска.
    // Их может быть несколько.
    // Вы можете добавить зависимость от экземпляра (не поставщика)
    // в этот блок
  });
```

Блоки конфигурации

Есть некоторые удобные методы модуля, эквивалентные блоку конфигурации. Например:

```
angular.module('myModule', []).
  value('a', 123).
  factory('a', function() { return 123; }).
  directive('directiveName', ...).
  filter('filterName', ...);

// то же самое

angular.module('myModule', []).
  config(function($provide, $compileProvider, $filterProvider) {
    $provide.value('a', 123);
    $provide.factory('a', function() { return 123; });
    $compileProvider.directive('directiveName', ...);
    $filterProvider.register('filterName', ...);
  });
```

Конфигурационные блоки будут применены в том порядке, в котором они зарегистрированы. Единственным исключением из этого являются постоянные определения, которые размещены в начале всех конфигурационных блоков.

Блокивыполнения

Блоки выполнения в Angular ближе всего к основному методу. Выполнения этого блока кода требуется для старта приложения. Он выполняется после того как службы были настроены и инжектора была создана. Блоки выполнения обычно содержат код, который трудно поддается юнит-тестированию, и по этой причине они должны быть объявлены изолированном модуле, после чего они могут быть проигнорированы при юнит-тестировании.

Зависимости

Модули могут перечислить другие модули, как свои зависимости. Зависимости от модуля означает, что требуемый модуль должен быть загружен перед загрузкой зависимого модуля. Другими словами конфигурационный блок необходимых модулей выполнится перед тем, как выполнится конфигурационный

блок требуемого модуля. То же самое справедливо для блоков выполнения. Каждый модуль может быть загружен только один раз, даже если несколько других модулей требуют этого.

Асинхронная загрузка

Модули - это способ управления конфигурацией $injector, и не имеют ничего общего с загрузкой скрипта в VM. Есть существующие проекты, которые касаются загрузки сценария, и которые могут быть использованы с Angular. Поскольку модули ничего не делают во время загрузки, они могут быть загружены в VM в любом порядке, и, таким образом сценарий загрузчики могут воспользоваться этим свойством для распараллеливания процесса загрузки.

Юнит тестирование

В своей простейшей форме тест представляет собой способ создания экземпляра и последующее применение к нему различных проверок. Важно понимать, что каждый модуль может быть загружен только один раз в инжекторе. Обычно приложение имеет только один инжектор. Но в тестах, каждый тест имеет свой собственный инжектор, что означает, что модули загружаются несколько раз в VM. Правильно структурированные модули могут помочь с модульным тестированием, как в этом примере:

Во всех этих примерах мы будем применять это определение модуля:

```
1. angular.module('greetMod', []).
2.
3.   factory('alert', function($window) {
4.     return function(text) {
5.       $window.alert(text);
6.     }
7.   }).
8.
9.   value('salutation', 'Hello').
10.
11.  factory('greet', function(alert, salutation) {
12.    return function(name) {
13.      alert(salutation + ' ' + name + '!');
14.    }
15.  });
```

Давайте напишем несколько тестов.

```
1.   describe('myApp', function() {
```

```
1.   // загрузка модуля приложения, затем загрузка специального модуля
2.   // который перекрывает объект $window его mock версией,
3.   // так что вызов $window.alert не будет блокировать выполнение теста
4.   // и выдавать реальное окно. Это пример переопределения конфигурации
5.   // в тестах.
6.   beforeEach(module('greetMod', function($provide) {
7.     $provide.value('$window', {
8.       alert: jasmine.createSpy('alert')
9.     });
10.  }));
11.
12.  // Вызов inject() создает injector и вставляет greet и
13.  // $window в тест. Тесты не должны заниматься работой,
14.  // они должны только тестировать.
```

```
15.   it('should alert on $window', inject(function(greet, $window) {
16.     greet('World');
17.     expect($window.alert).toHaveBeenCalledWith('Hello World!');
18.   }));
19.
20.   // это другая возможность переопределения конфигурации в
21.   // тестах используя включения модуля и инжектор.
22.   it('should alert using the alert service', function() {
23.     var alertSpy = jasmine.createSpy('alert');
24.     module(function($provide) {
25.       $provide.value('alert', alertSpy);
26.     });
27.     inject(function(greet) {
28.       greet('World');
29.       expect(alertSpy).toHaveBeenCalledWith('Hello World!');
30.     });
31.   });
32. });
```

Область видимости

Что такое область видимости?

Область видимости является объектом, который относится к модели приложения. Это контекст выполнения выражений. Области расположены в иерархической структуре, которые имитируют DOM структуру приложения. Области могут вычислять выражения и генерировать события.

Характеристики области видимости

- Области видимости обеспечивают интерфейс прикладного программирования ($watch) наблюдающий за изменением модели.
- Области видимости интерфейс прикладного программирования ($apply) распространяющий любые изменения модели через систему, в поле зрения за пределами «Angular области" (контроллеры, сервисы, Angular обработчики событий).
- Области видимости могут быть вложенными, чтобы изолировать компоненты приложений, обеспечивая при этом доступ к общим свойствам модели. Они (прототипически) наследуют свойства своего родителя.
- Области видимости обеспечить контекст, в котором вычисляются выражения. Например {{username}} выражение не имеет смысла, если оно не оценивается в определенной области, которая определяет свойство username.

Область видимости, как модель данных

Область видимости является связующим звеном между контроллером приложения и представлением. На стадии компоновки шаблона директивы создают сервис $watch, который следит за любыми изменениями в области видимости. $watch позволяет директивам получать уведомления об изменениях свойств в области видимости, что позволяет директивам обновлять значение в DOM.

Контроллер и директивы имеют ссылки на область видимости, но не имеют ссылок друг на друга. Это отделяет контроллер от директив, а также от DOM. Это важный момент, так как контроллер ничего не знает о представлении, что значительно улучшает тестирование приложения.
Пример:

Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="MyController">
      Your name:
        <input type="text" ng-model="username">
        <button ng-click='sayHello()'>greet</button>
      <hr>
      {{greeting}}
    </div>
  </body>
</html>
```

Script.js

```
function MyController($scope) {
  $scope.username = 'World';

  $scope.sayHello = function() {
    $scope.greeting = 'Hello ' + $scope.username + '!';
  };
}
```

В приведенном выше примере, что MyController присваивает свойству username значение world в своей области видимости. Область видимости затем уведомляет элемент управления об изменении значения, который устанавливает свое значение в соответствии с текущим значением. Это демонстрирует, как контроллер может записать данные в области видимости.
Точно так же контроллер может задать поведение для метода внутри области видимости, это видно по методу sayHello, который вызывается, когда пользователь нажимает на кнопку "приветствие". Метод sayHello может прочитать свойство username и создать поздравительную надпись. Это показывает, что свойства в области видимости обновляются автоматически, когда они связаны с HTML элементами управления для ввода данных.
Логически указание {{ greeting }} включает в себя:

- поиск области видимости, связанной с DOM узла, который содержит в своем шаблоне выражение {{ greeting }}. В данном примере это та же область видимости, что и область видимости, которая была передана в MyController. (Мы рассмотрим иерархии областей видимости позже.)
- Вычисляется выражение greeting в контексте области видимости полученной выше, и результат присваивается содержимому элемента управления вводом.

Вы можете думать об области видимости и ее свойствах, как о данных, которые используются для визуализации представления. Область видимости является единственным источником данных для всех представлений.
С точки зрения тестируемости, желательно разделить представления и контроллер, т.к. это позволит нам тестировать поведение, не отвлекаясь на детали отображения.

```
it('should say hello', function() {
  var scopeMock = {};
  var cntl = new MyController(scopeMock);

```

```
    // Assert that username is pre-filled
    expect(scopeMock.username).toEqual('World');

    // Assert that we read new username and greet
    scopeMock.username = 'angular';
    scopeMock.sayHello();
    expect(scopeMock.greeting).toEqual('Hello angular!');
});
```

Иерархия областей видимости

Каждое приложение angular имеет один корневую область видимости, но может иметь несколько дочерних областей видимости.
Приложение может иметь несколько областей видимости, потому что некоторые директивы создают новые дочерние области видимости (см. документацию по директивам, чтобы узнать, какие директивы создают новые области видимости). Когда новые области видимости будут созданы, они будут добавлены как дочерние к родительской области. Это создает древовидную структуру, которая параллельна DOM, к которому они прикреплены.
Когда Angular оценивает выражение {{username}}, то он сначала ищет значение в области видимости, связанной элементом, который содержит в себе выражение. Если такое свойство не найдено, он ищет в родительской области данных, и так далее, пока не будет достигнута корневая область видимости. В JavaScript это поведение называется прототипным наследованием, когда дочерних области видимости прототипически наследуются от своих родителей.
Этот пример иллюстрирует применение областей видимости, и прототипное наследование свойств.
Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="EmployeeController">
      Manager: {{employee.name}} [ {{department}} ]<br>
      Reports:
        <ul>
          <li ng-repeat="employee in employee.reports">
            {{employee.name}} [ {{department}} ]
          </li>
        </ul>
      <hr>
      {{greeting}}
    </div>
  </body>
</html>
```

Style.css

```
/* remove .doc-example-live in jsfiddle */
.doc-example-live .ng-scope {
  border: 1px dashed red;
}
```

Script.js

```
function EmployeeController($scope) {
  $scope.department = 'Engineering';
  $scope.employee = {
    name: 'Joe the Manager',
    reports: [
      {name: 'John Smith'},
      {name: 'Mary Run'}
    ]
  };
}
```

Обратите внимание, что Angular автоматически помещает класс ng-scope к элементам, которые присоединяются к области видимости. Определение <style> в данном примере подсвечивает красным цветом значения из новых областей видимости. Дочерние рамки в стиле необходимы для того, чтобы показать, как ретранслятор оценивает выражение {{ employee.name }} и как в зависимости от области видимости вычисление выражения приводит к разным результатам. Аналогично вычисляется выражение {{ `department` }}, которое прототипически наследуется от корневой области видимости, так как это единственное место, где определено свойство с таким именем.

Получение области видимости из DOM.

Области видимости крепятся к элементам DOM как свойство $scope, и могут быть получены в целях отладки. (Маловероятно, что возникнет необходимость получения областей видимости таким образом внутри приложения.) Место, где корневая область видимости крепится к DOM, определяется расположением директивы ng-app. Обычно ng-app размещается в элементе <html>, но она может быть помещена и в другие элементы, а также, если, например, только часть представления необходимо контролировать с помощью Angular.
Чтобы изучить область видимости в отладчике:

1. Щелкните правой кнопкой мыши на интересующем элементе в вашем браузере и выберите "проверить элемент». Вы должны увидеть браузерный отладчик с элементом который вы выделили.
2. Отладчик позволяет получить доступ к выбранному элементу через консоль, используя переменную $0.
3. Чтобы получить соответствующую область видимости в консоли выполните: angular.element ($0).scope().

Распространение событий области видимости

Области видимости могут распространять события аналогично события DOM. Событие можно передать в дочернюю область видимости или выбросить в родительскую область видимости.
Пример:
Index.html

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="EventController">
      Root scope <tt>MyEvent</tt> count: {{count}}
      <ul>
        <li ng-repeat="i in [1]" ng-controller="EventController">
          <button ng-click="$emit('MyEvent')">$emit('MyEvent')</button>
          <button ng-click="$broadcast('MyEvent')">$broadcast('MyEvent')</button>
```

```
          <br>
          Middle scope <tt>MyEvent</tt> count: {{count}}
          <ul>
            <li ng-repeat="item in [1, 2]" ng-controller="EventController">
              Leaf scope <tt>MyEvent</tt> count: {{count}}
            </li>
          </ul>
        </li>
      </ul>
    </div>
  </body>
</html>
```

Script.js

```
function EventController($scope) {
  $scope.count = 0;
  $scope.$on('MyEvent', function() {
    $scope.count++;
  });
}
```

Жизненный цикл области видимости

При получении события в нормальном потоке браузера вызывается соответствующая функция обратного вызова JavaScript. После завершения функции обратного вызова браузер переделывает DOM и возвращается к ожиданию возникновения других событий.
Когда браузер выполняет код вне контекста выполнения Angular, это означает что Angular модели не изменились. Чтобы правильно обработать изменение модели контекст должен быть переключен на Angular контекст выполнения, для чего используется метод $apply. Только модификаций модели, которые выполняются внутри метода $apply будут правильно извещать Angular. Например, если директива прослушивает события DOM, такие как ng-click она должна вычислить выражение внутри метода $apply. После вычисления выражения, метод $apply вызывает $digest. В методе $digest область видимости проверяет все $watch выражения и сравнивает их с предыдущим значением. Эта грязная проверка делается асинхронно. Это означает, что присваивания, такие как $ scope.username = "angular" не будет немедленно вызывать $watch, чтобы передать уведомление слушателям событий, вместо этого $watch уведомление откладывается до фазы $digest. Эта задержка нежелательна, поскольку в результате сливается несколько обновлений модели в одном $watch уведомлении, а также она гарантирует, что в течение метода $watch никаких других уведомлений $watch не сработают. Если $watch изменяет значение модели, это вызовет дополнительную итерацию цикла $digest.

1. Создание
Корневая область видимости создается во время загрузки приложений сервисом $injector. В шаблонах некоторые директивы создают новые дочерние области видимости.

2. Watcher регистрации
При связывании директивы в шаблонах регистрируются подписчики событий для области видимости. Эти подписчики будут использоваться для отражения значений модели в значения в DOM.

3. Изменения модели
Чтобы наблюдатели узнали об изменениях модели нужно сделать их только внутри метода $aplly. (Angular API делает это неявно, так что никаких дополнительных вызовов $aplly делать не нужно, но это необходимо при выполнении синхронных работ в контроллерах или работы с асинхронными сервисами $http или $timeout.

4. Наблюдения за изменениями
По окончанию метода $apply, Angular входит в цикл $digest в контексте корневой области видимости, и затем распространяется на все дочерние области видимости. Во время цикла $digest, все $watch выражения или функции проверяются на наличие изменений и если они обнаруживаются, $watch уведомляет слушателей событий об этом.

5. Разрушение области видимости
Когда дочерняя область видимости больше не нужна, ее родитель может ее уничтожить, вызвав метод API $destroy. Это позволит остановить распространение $digest уведомлений в дочернюю область видимости и позволяют сборщику мусора освободить память, используемую для удаленной области видимости.

Области и директивы

Во время фазы компиляции, компилятор находит директивы внутри шаблонов DOM. Директивы обычно попадают в одну из двух категорий:

- Директивы наблюдения, такие как выражения {{expression}}, регистрируют слушателей используя метод $watch (). Этот тип директива должен быть уведомлен при изменении выражения, после чего он обновит представление.
- Директивы слушатели, такие как ng-click, регистрируют слушателя для событий DOM. Когда происходит прослушиваемое событие DOM, директива выполняет ассоциированное выражение и обновления представление с использованием метода $apply().

Когда внешнее событие (например, действия пользователя, таймер или XHR) получено, связанное выражение должно быть применено к области видимости через метод $apply(), чтобы все слушатели были правильно уведомлены.

Директивы создают области видимости

В большинстве случаев, директивы с областями видимости взаимодействуют, а не создают новые экземпляры. Тем не менее, некоторые директивы, такие как ng-controller и ng-repeat, создают новые дочерние области видимости и присоединяют их к соответствующему элементу DOM. Вы можете получить область видимости любого элемента DOM с помощью вызова метода angular.element(aDomElement).scope().

Контроллеры и области видимости

Области видимости и контроллеры взаимодействуют друг с другом в следующих ситуациях:

- Контроллеры используют области видимости для вычисления методов контроллера в шаблонах (см. ng-controller).
- Контроллеры определять методы (поведение), которые могут изменять модели (свойства области видимости).
- Контроллеры могут регистрировать наблюдателей для модели. Эти наблюдатели будут уведомленны сразу после того, как метод контроллера будет выполнен.

См. ng-controller для получения дополнительной информации.

Вопросы производительности наблюдений $watch области видимости

Грязные проверки наличия изменений свойств в области видимости являются распространенной операцией в Angular и по этой причине грязную функция проверки должна быть эффективной. Следует позаботиться о том, что грязные функция проверки не работали с DOM, т.к. доступ к DOM на несколько порядков медленнее, чем доступ к свойствам объекта на JavaScript.

Внедрение зависимостей

Внедрение зависимостей (DI) это шаблон проектирования программного обеспечения, который обеспечивает передачу коду его зависимостей.
Для углубленного изучени DI Dependency Injection, или статью Inversion of Control Мартин Фаулер, или читайте о DI в вашей любимой книге по шаблонам проектирования программного обеспечения.

В двух словах о DI

Есть только три способа, которыми объект или функция может получить свои зависимости:

1. Зависимость может быть создана, обычно с использованием оператора new.
2. Зависимость может быть найдена путем обращения к глобальной переменной.
3. Зависимость может быть передана туда, где это необходимо.

Первые два варианта создания или поиск зависимости не являются оптимальными, поскольку они жестко кодируют зависимость. Это делает его трудным, или вообще невозможным, изменение зависимостей. Это особенно проблематично в тестах, где часто желательно обеспечить прочную изоляцию замещающего элемента зависимости.

Третий вариант является наиболее жизнеспособным, так как снимает ответственность за поиск зависимости с компонента. Зависимость просто передается компоненту.

```
function SomeClass(greeter) {
  this.greeter = greeter;
}

SomeClass.prototype.doSomething = function(name) {
  this.greeter.greet(name);
}
```

В примере выше `SomeClass` не знает, где размещена его зависимость `greeter`, она просто передается ему во время выполнения.
Это желательно, но это возлагает ответственность за разрешение зависимости на код, который строит `SomeClass`.
Для управления ответственность за разрешение и создание зависимостей, каждое приложение Angular имеет инжектор. Инжектор – это локальный сервис, который отвечает за создание и поиск зависимостей.
Ниже приведен пример использования сервиса инжектор:

```
// Предоставляет информацию в модуле
angular.module('myModule', []).

  // учит инжектор как построить 'greeter'
  // обратите внимание greeter уже зависит от '$window'
  factory('greeter', function($window) {
    // Эта фабричная функция применяется для
    // создания сервиса 'greet'.
    return {
      greet: function(text) {
        $window.alert(text);
      }
    };
  });

// Новый инжектор созданный из модуля.
// (Обычно это делается автоматически при загрузке Angular)
var injector = angular.injector(['myModule', 'ng']);

// Запрос зависимости у инжектора
var greeter = injector.get('greeter');
```

Запрос зависимостей решает вопрос жесткого кодирования, но это также означает, что инжектор должен быть передан для каждого метода во всем приложении. Передача инжектора нарушает закон Деметры. Чтобы исправить это, мы возлагаем ответственность за поиск зависимости на инжектор, объявив зависимости как в этом примере:

```
<!—это в HTML -->
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
// а это определение контроллера
```

```
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}

// директива 'ng-controller' делает это за кулисами
injector.instantiate(MyController);
```

Обратите внимание, что при наличии экземпляра класса ng-controller, он может удовлетворить все зависимости MyController без контроллера, ничего не зная о инжекторе. Это самый лучший результат. Код приложения просто запросит зависимости, которые ему нужны, без необходимости иметь дело с инжектором. Это указание не нарушать закон Деметры.

Объявление зависимости

Как инжектор узнает, что должна быть введена служба?
Разработчик приложения должен предоставить информацию, которую инжектор используется для разрешения зависимостей. При выполнении приложения Angular, некоторые функции API вызываются с помощью инжектора, в соответствии с документацией API. Инжектор должен знать, какие сервисы требуются передать в функцию. Ниже приведены три эквивалентных способов аннотирования кода с именем сервиса. Они могут быть использованы как взаимозаменяемые.

Ввод зависимости

Самый простой способ объявить зависимость, это вставить в качестве имен параметров функции имена зависимостей.

```
function MyController($scope, greeter) {
  ...
}
```

Для данной функции инжектор может определить требуемые имена служб для введения путем проверки объявления функции и извлечения имен параметров. В приведенном выше примере `$scope` и `greeter` два сервиса, которые должны быть введены в функцию.
В то время как этот метод самый простой, этот метод не будет работать с JavaScript минификаторами / обфускаторами, так как они переименовают имена параметров метода. Это делает данный способ аннотирования полезно только для прототипов и демо-приложений.

$inject аннотации

Чтобы разрешить процессу минификации переименовать параметры функции и по-прежнему иметь возможность генерировать нужные сервисы, функция должна иметь пояснение для $inject. Свойство $inject является массивом имен сервисов для введения.

```
var MyController = function(renamed$scope, renamedGreeter) {
  ...
}
MyController.$inject = ['$scope', 'greeter'];
```

Необходимо позаботиться о том, что $inject аннотации следовали с том же порядке, что и фактические аргументы в объявлении функции.
Метод аннотации полезен для декларации контроллера, поскольку назначает аннотационную информация для функции.

Встроенные аннотации

Иногда не удобно использовать стиль аннотации для $inject, например, когда аннотируются директивы. Например:

```
someModule.factory('greeter', function($window) {
  ...
});
```

В результате требуется переменная, что приводит к разрастанию кода:

```
var greeterFactory = function(renamed$window) {
  ...
};

greeterFactory.$inject = ['$window'];

someModule.factory('greeter', greeterFactory);
```

По этой причине так же предоставляется третий стиль:

```
someModule.factory('greeter', ['$window', function(renamed$window) {
  ...
}]);
```

Имейте в виду, что все стили аннотации эквивалентны и могут быть использованы в любом месте, где поддерживаются Angular инъекции.

Где я могу использовать DI?

DI распространена в Angular везде. Она обычно используется в контроллерах и фабричных методах.

DI в контроллерах

Контроллеры это классы, которые отвечают за поведением приложения. Рекомендуемый способ объявления контроллеров:

```
var MyController = function($scope, dep1, dep2) {
  ...
  $scope.aMethod = function() {
    ...
  }
}
MyController.$inject = ['$scope', 'dep1', 'dep2'];
```

Фабричные методы

Фабричные методы отвечают за создание большинства объектов в Angular. Примерами являются директивы, сервисы и фильтры. Фабричные методы, регистрируются в модуле, и вот рекомендуемый способ их декларирования:

```
angular.module('myModule', []).
  config(['depProvider', function(depProvider){
    ...
```

```
  }]).
  factory('serviceId', ['depService', function(depService) {
    ...
  }]).
  directive('directiveName', ['depService', function(depService) {
    ...
  }]).
  filter('filterName', ['depService', function(depService) {
    ...
  }]).
  run(['depService', function(depService) {
    ...
  }]);
```

Об MVC в Angular

Хотя модель-представление-контроллер (MVC) приобрел различные оттенки смысла с годами, по сравнению с тем, когда он впервые появился, Angular включает основные принципы проектировочного шаблона MVC в свой путь создания клиентских веб-приложений.

Шаблон MVC кратко:

- Разделение приложения в отдельные компоненты, представления, данные и логика
- Поощряется слабая связность между этими компонентами

Сервисы и введения зависимостей MVC делают Angular приложения лучше структурированнымы, легко поддерживаемыми и более контроллируемыми.

В следующих разделах объясняется, как Angular включает шаблон MVC в свой способ разработки веб-приложений:

- Объяснение модели
- Объяснение контроллера
- Объяснение представления

Объяснение модели

В зависимости от контекста обсуждения в Angular документации термин модель может относиться либо к одному объекту, представляющему одну сущность (например, модель под названием «телефоны» и его значением является массив телефонов) или всей модели данных для приложения (все сущности).
В Angular модель — любые данные, которые можно получить через свойство объекта области видимости Angular. Имя свойства, является идентификатором модели, а значение — это любой объект JavaScript (включая массивы и примитивовы).
Единственное требование, чтобы объект JavaScript стал моделью в Angular, что на него должна ссылаться область видимости Angular через собственное свойство. Эти свойства могут создаваться прямо или косвенно.
Можно создавать модели, явно задавая свойств области видимости и присваивая им ссылки на объекты JavaScript следующим образом:

- Назначить свойству в области видимости объект с JavaScript-кодом; Это чаще всего применяется в контроллерах:

```
function MyCtrl($scope) {

     // create property 'foo' on the MyCtrl's scope

     // and assign it an initial value 'bar'

     $scope.foo = 'bar';

 }
```

- Используя выражения Angular с оператором присваивания в шаблонах:

```
<button ng-click="{{foos='ball'}}">Click me</button>
```

- Используя директиву ng-init в шаблонах (не рекомендуется для использования в реальных приложениях).

```
<body ng-init=" foo = 'bar' ">
```

Angular создает модели неявно (свойству области видимости присваивается значение) когда обрабатываются следующие шаблонные конструкции:

- Элементы управления input, select, textarea и другие в формах.

  ```
  <input ng-model="query" value="fluffy cloud">
  ```

  Этот код создает область видимости со свойством query и присваивает ему значение “fluffy cloud”.
- Декларация директивы ng-repeat

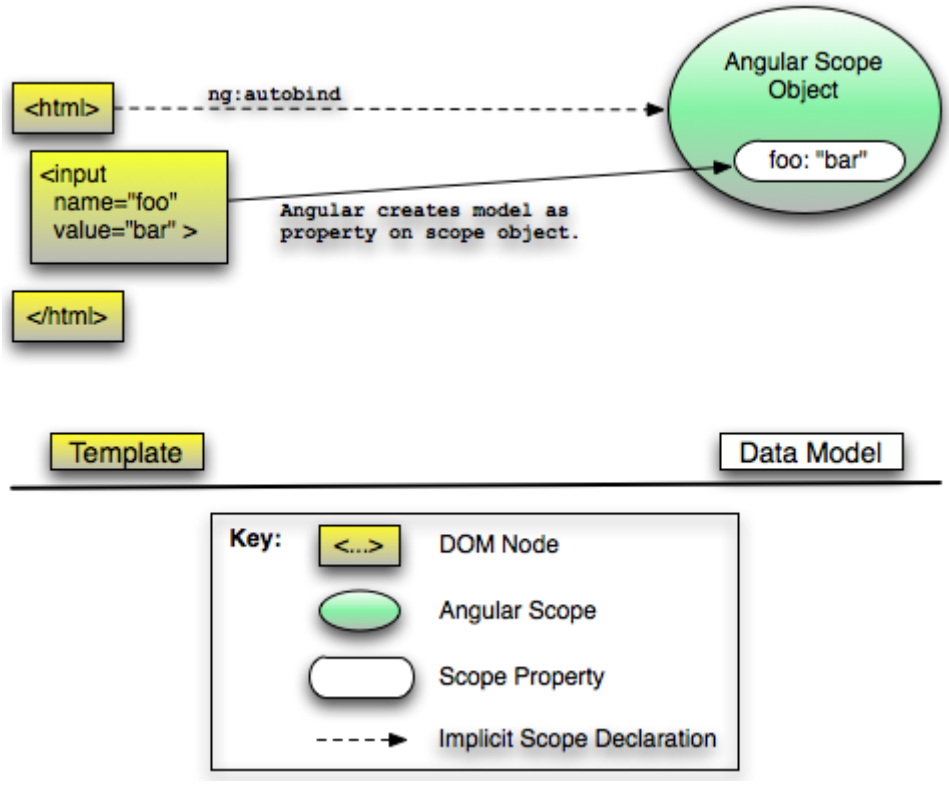
  ```
  <p ng-repeat="phone in phones"></p>
  ```

  Приведенный выше код создает одну дочернюю область видимости для каждого элемента в массиве «phones» и создает свойство «phone» (модель) для каждой из этих областей в значение «phone» в массиве.

В Angular, JavaScript объект перестает быть модель когда:

- Angular область видимости не содержит свойства, которое ссылается на объект.
- Все Angular области видимости, которые содержат свойства, ссылающейся на объект, выходят из области видимости и становятся мусором.

Следующий рисунок показывает простую модель данных, созданную неявно из простого шаблона.



Объяснение контроллера

В Angular контроллер является JavaScript функцией (типом/классом), которая используется для создания экземпляров областей видимости, исключая корневой областью видимости. Когда вы или Angular создаете новый дочерний объект области видимости через scope.$new() API, его потом можно передать в контроллер в качестве аргумента. Это скажет Angular, что нужно связать контроллер с новой областью видимости и добавить в него ее поведение.
Используйте контроллеры для:

- Настройки начального состояния объекта области видимости.
- Добавления поведения к объекту области видимости.

Настройка начальное состояние объекта области видимости

Как правило, при создании приложения необходимо настроить начальное состояние для Angular области видимости.
Angular применяет (в смысле JavaScript function#apply) функцию конструктора контроллера для нового объекта области видимости, который устанавливает первоначальное состояние. Это означает, что Angular никогда не создает экземпляры типа контроллера (путем вызова оператора new для конструктора контроллера). Конструкторы всегда применяются к существующему объекту области видимости.
Вы можете настроить начальное состояние области видимости путем задания свойств модели. Например:
function GreetingCtrl($scope) { $scope.greeting = 'Hola!'; }
Контроллер GreetingCtrl создает модель greeting, которая может быть передана в шаблон.
Примечание: Многие примеры в документации показывают создание функций в глобальной области видимости. Это только для демонстрационных целей - в реальном приложении следует использовать метод Angular модуля controller() для вашего приложения, следующим образом:

```
var myApp = angular.module('myApp',[]);
myApp.controller («GreetingCtrl», [«$scope», function($scope) {$scope.greeting = «Hola!»;}]);
```

Также обратите внимание, что мы использовать нотацию массива для явного указания зависимости контроллера от сервиса $scope, предоставляемого Angular.

Добавление поведения к объекту областивидимости

Поведение Angular объекта области видимости находится в форме свойств, ссылающихся на метод области видимости и доступных для шаблона представления. Это поведение взаимодействует с моделью приложения и изменяет ее.
Как обсуждалось в разделе «объяснение модели» данного руководства, любые объекты (или примитивы), присвоенные области видимости становятся свойствами модели. Любые функции, на которые есть ссылки в области видимости доступны в шаблоне представления и могут быть вызваны через Angular выражения и директивы, работающие с поведением (например, ng-click).

Правильное использование контроллеров

В общем, контроллер не должен пытаться делать слишком много. Он должен содержать только бизнес-логику для одного представления.
Наиболее распространенным способом держать контроллеры в облегченном состоянии, является инкапсуляция работы, которая не принадлежит контроллерам в службы и затем, использование этих служб в контроллерах через внедрение зависимостей. Этот вопрос обсуждается в разделе «внедрение зависимостей» данного руководства.
Не используйте контроллеры для:

- Любого рода манипуляции с DOM — контроллеры должны содержать только бизнес-логику. Манипуляции с DOM — логику представления приложения — трудно проверить. Ввод любой логики представления в контроллеры значительно влияет на возможности тестирования бизнес-логики. Angular предлагает привязки данных для автоматической манипуляции с DOM. Если нужно выполнить свою собственный манипуляцию DOM, инкапсулировать логику представления в директивах.
- Входной форматирование — вместо этого используйте Angular элементы управления в формах.
- Фильтрация выходных данных — вместо этого используйте Angular фильтры.
- Для запуска кода без состояния, общего для всех контроллеров используйте сервисы Angular.
- Создание экземпляров или управлять жизненным циклом других компонентов (например, для создания экземпляров службы).

Ассоциация контроллеров с Angular областью видимости

Вы можете ассоциировать контроллер с областью видимости явно, используя метод API scope.$new(), или неявно – при использовании директивы ngController или сервиса $route.

Конструктор контроллера и пример методов

Чтобы проиллюстрировать, как работает компонент Angular контроллер, давайте создадим маленькое приложение со следующими компонентами:

- Шаблон с двумя кнопками и простым сообщением
- Модель, состоящая из строки с именем spice
- Контроллер с двумя функциями, которые задают значение spice

Сообщение в нашем шаблоне содержит привязку к модели spice, которая по умолчанию установлена в строку «very». В зависимости от того, какая кнопка нажата, spice модель устанавливается для `chili` или `jalapeño`, и сообщение автоматически обновляется путем привязки данных.

Пример Spicy контроллера

```
<body ng-controller="SpicyCtrl">
  <button ng-click="chiliSpicy()">Chili</button>
  <button ng-click="jalapenoSpicy()">Jalapeño</button>
  <p>The food is {{spice}} spicy!</p>
</body>

function SpicyCtrl($scope) {
  $scope.spice = 'very';
  $scope.chiliSpicy = function() {
    $scope.spice = 'chili';
  }
  $scope.jalapenoSpicy = function() {
    $scope.spice = 'jalapeño';
  }
}
```

В приведенном выше примере следует обратить внимание на следующие вещи:

- NgController директива используется для (косвенно) создания области видимости для нашего шаблона, и добавляется (управляемо) область видимости SpicyCtrl контроллера.
- SpicyCtrl это просто функция JavaScript. Имя которой (опционально) начинается с буквы и заканчивается строкой «Ctrl» или «Controller».
- Назначение свойства $scope создает или обновляет модель.
- Методы контроллера могут быть созданы путем прямого их назначения в области видимости (метод chiliSpicy)
- Оба методы контроллера доступны в шаблоне (для элемента body и дочерних элементов).
- NB: Предыдущих версий Angular (pre 1.0 RC) позволило использовать this взаимозаменяемо со $scope, но это больше не так. Внутри методов, определенных для области видимости this и $scope являются взаимозаменяемыми (Angular устанавливает this = $scope), но это справедливо только внутри конструктора контроллера.
- NB: Предыдущих версиях Angular(pre 1.0 RC) добавлелялись методы прототипов в область видимости автоматически, но это больше не так; все методы должны быть добавлены вручную в область видимости.

Методы контроллера также могут принимать аргументы, как показано в следующем варианте предыдущего примера.

Пример метода контроллера с аргументом

```
<body ng-controller="SpicyCtrl">
  <input ng-model="customSpice" value="wasabi">
```

```
  <button ng-click="spicy('chili')">Chili</button>
  <button ng-click="spicy(customSpice)">Custom spice</button>
  <p>The food is {{spice}} spicy!</p>
</body>

function SpicyCtrl($scope) {
  $scope.spice = 'very';
  $scope.spicy = function(spice) {
    $scope.spice = spice;
  }
}
```

Обратите внимание, что контроллер `SpicyCtrl` сейчас определяет только один метод `spicy`, который принимает один аргумент `spice`. Шаблон ссылается на метод контроллера и передает ему строку `'chili'` в связывании для первой кнопки и свойство модели `customSpice` (привязанной к текстовому полю) для следующей кнопки.

Пример наследования контроллеров

Наследование контроллеров в Angular базируется на наследовании области видимости. Давайте посмотрим на пример:

```
<body ng-controller="MainCtrl">
  <p>Good {{timeOfDay}}, {{name}}!</p>
  <div ng-controller="ChildCtrl">
    <p>Good {{timeOfDay}}, {{name}}!</p>
    <p ng-controller="BabyCtrl">Good {{timeOfDay}}, {{name}}!</p>
  </div>
</body>

function MainCtrl($scope) {
  $scope.timeOfDay = 'morning';
  $scope.name = 'Nikki';
}

function ChildCtrl($scope) {
  $scope.name = 'Mattie';
}

function BabyCtrl($scope) {
  $scope.timeOfDay = 'evening';
  $scope.name = 'Gingerbreak Baby';
}
```

Обратите внимание как расположенны директивы `ngController` в дереве вашего шаблона. В результате это шаблон создает 4 области видимости для вашего представления:

- Корневая область видимости
- Область видимости для `MainCtrl`, которая содержит `timeOfDay` и `name` модели
- Область видимости для `ChildCtrl`, которая заменяет `name` модель из предыдущей области видимости и наследует модель `timeOfDay`.

- Область видимости для `BabyCtrl`, которая заменяет обе модели, `timeOfDay` определенную в `MainCtrl` и `name`, определенную ChildCtrl

Наследование работает между контроллерами таким же образом, как это происходит с моделями. Таким образом, в наших предыдущих примерах, все модели могут быть заменены на методы контроллера, возвращающие строковые значения.
Примечание: Стандартное прототипное наследование между двумя контроллерами не работает, как можно было бы ожидать, потому что, как мы упоминали ранее, контроллеры не создаются непосредственно Angular, а скорее применяются к объекту области видимости.

Тестирование контроллеров

Хотя есть много способов для тестирования контроллера, один из лучших, показанный ниже, включает инъекцию $rootScope и $controller.

Функция контроллера:

```
function myController($scope) {
   $scope.spices = [{"name":"pasilla", "spiciness":"mild"},
                    {"name":"jalapeno", "spiceiness":"hot hot hot!"},
                    {"name":"habanero", "spiceness":"LAVA HOT!!"}];

   $scope.spice = "habanero";
}
```

Тест для контроллера:

```
describe('myController function', function() {

  describe('myController', function() {
    var scope;

    beforeEach(inject(function($rootScope, $controller) {
      scope = $rootScope.$new();
      var ctrl = $controller(myController, {$scope: scope});
    }));

    it('should create "spices" model with 3 spices', function() {
      expect(scope.spices.length).toBe(3);
    });

    it('should set the default value of spice', function() {
      expect(scope.spice).toBe('habanero');
    });
  });
});
```

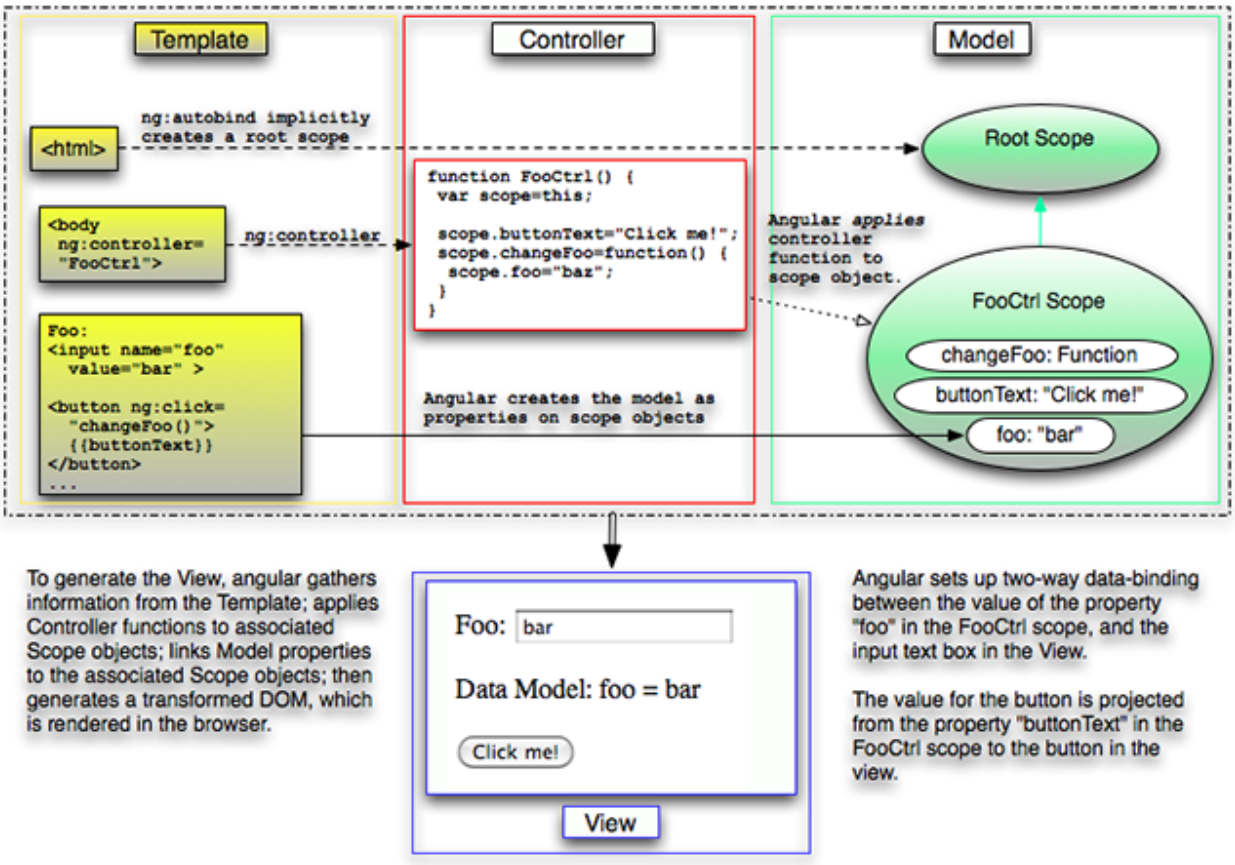
Если вам нужно протестировать вложенные контроллеры, тогда нужно создать иерархию областей видимости, таку же, как в DOM.

```
describe('state', function() {
    var mainScope, childScope, babyScope;

```

```
4.      beforeEach(inject(function($rootScope, $controller) {
5.          mainScope = $rootScope.$new();
6.          var mainCtrl = $controller(MainCtrl, {$scope: mainScope});
7.          childScope = mainScope.$new();
8.          var childCtrl = $controller(ChildCtrl, {$scope: childScope});
9.          babyScope = childCtrl.$new();
10.         var babyCtrl = $controller(BabyCtrl, {$scope: babyScope});
11.     }));
12.
13.     it('should have over and selected', function() {
14.         expect(mainScope.timeOfDay).toBe('morning');
15.         expect(mainScope.name).toBe('Nikki');
16.         expect(childScope.timeOfDay).toBe('morning');
17.         expect(childScope.name).toBe('Mattie');
18.         expect(babyScope.timeOfDay).toBe('evening');
19.         expect(babyScope.name).toBe('Gingerbreak Baby');
20.     });
21. });
```

Объяснение представления

В Angular представление это загруженный и отображенный в браузере DOM, после трансформаций, базирующихся на информации из шаблона, контроллера и модели.



В Angular реализован паттерн MVC, где представление знает о модели и контроллере. Представление знает о модели в случае двунаправленной привязки. Представление знает о

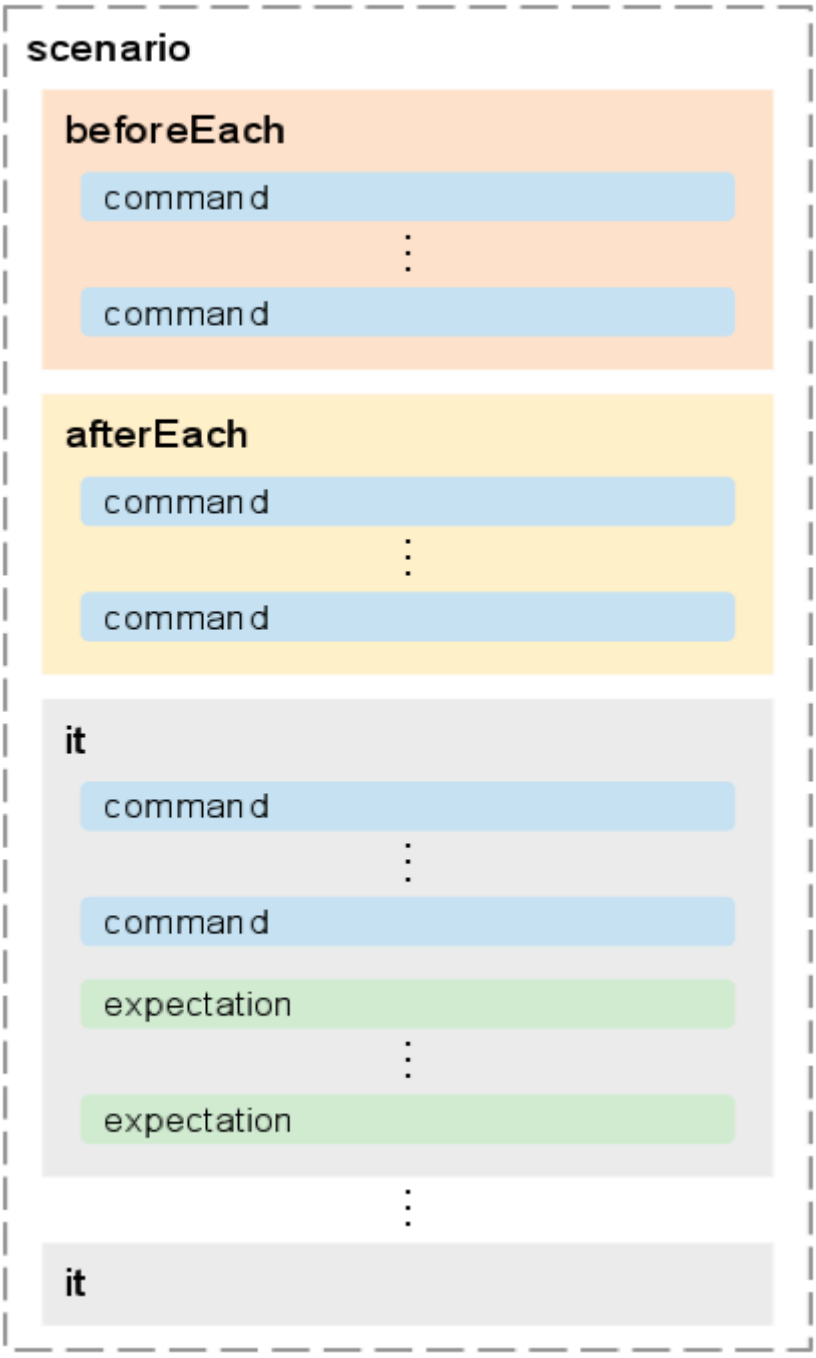
контроллере через Angular директивы, такие как `ngController` и `ngView`, и через привязки данных в форме: `{{someControllerFunction()}}`. В этом случае, представление может вызвать функцию в как ассоциированную функцию контроллера.

E2e тестирование

Со временем приложение растет в размерах и сложности, и становится нереально рассчитывать на ручное тестирование для проверки корректности новых функций, отлова ошибок и объяснение регрессий.
Для решения этой проблемы, мы сделали Angular исполнитель сценариев, который имитирует взаимодействие пользователей, и поможет вам проверить работоспособность приложения Angular.

Обзор

Вы будете писать сценарий тестов в JavaScript, которые описывают, как должно вести себя приложение, учитывая определенные взаимодействия в определенном состоянии. Сценарий состоит из одного или более блоков (вы можете думать о них как о требованиях вашего приложения), которые, в свою очередь состоят из команд и ожиданий. Команды говорит исполнителю сценария, что нужно сделать с приложением (например, перейдите на страницу или нажмите на кнопку), и ожидает, пока исполнитель не сообщит значение текущего состояния (например, значение поля или текущий URL-адрес). В случае несоответствия результата ожиданиям, исполнитель помечает тест как «failed» и продолжает следующий. Сценарии могут также иметь beforeEach и afterEach блоки, которые будут выполняться до (или после) каждого блока, независимо от того, пройден он или нет.



Дополнительно к вышеперечисленному сценарии могут также содержать вспомогательные функции, которые призваны исключить дублирование кода в блоках `it`.

Вот пример простого сценария:

```
describe('Buzz Client', function() {
it('should filter results', function() {
  input('user').enter('jacksparrow');
  element(':button').click();
  expect(repeater('ul li').count()).toEqual(10);
  input('filterText').enter('Bees');
  expect(repeater('ul li').count()).toEqual(1);
});
});
```

Сценарий определяет блок требований «Buzz Client», который проверяет работу фильтра для поля user. При старте вводится значение в поле ввода 'user' и после этого кликается кнопка на странице, а затем проверяется наличие 10 элементов li. Затем вводится 'Bees' в поле 'filterText' и проверяется, что остался один элемент.
В секции API приведен список доступных команд и ожиданий.

API

pause()
Пауза в выполнении теста до вызова функции `resume()` в консоли (или клик ссылки resume в Runner UI).

sleep(seconds)
Пауза в выполнении теста на указанное `seconds` количество секунд.

browser().navigateTo(url)
Загрузка содержимого `url` в тестовую область.

browser().navigateTo(url, fn)
Загрузка URL и который возвращается функцией `fn` в тестовое окружение. Полученные данные из `url` используются только для вывода теста. Используется когда URL создается динамичестки (может быть он просто не определен во время написания теста).

browser().reload()
Обновление текущей загруженной страницы в тестовой области.

browser().window().href()
Возвращает window.location.href для текущей страницы, загруженной в тестовую область.

browser().window().path()
Возвращает window.location.pathname для текущей страницы, загруженной в тестовую область.

browser().window().search()
Возвращает window.location.search для текущей страницы, загруженной в тестовую область.

browser().window().hash()
Возвращает window.location.hash для текущей страницы, загруженной в тестовую область.

browser().location().url()
Возвращает `$location.url()` для текущей страницы, загруженной в тестовую область.

browser().location().path()
Возвращает `$location.path()` для текущей страницы, загруженной в тестовую область.

browser().location().search()
Возвращает `$location.search()` для текущей страницы, загруженной в тестовую область.

browser().location().hash()
Возвращает `$location.hash()` для текущей страницы, загруженной в тестовую область.

expect(future).{matcher}
Утверждение, что значение `future` должно удовлетворять условию `matcher`. Все методы API возвращают объект `future`, который получает значение `value` после того как закончится выполнение. Вычислители определяются с помощью `angular.scenario.matcher`, и используются для проверки будущих значений. Для примера:
`expect(browser().location().href()).toEqual('http://www.google.com')`

expect(future).not().{matcher}
Утверждение, что значение `future` не должно удовлетворять `matcher`.

using(selector, label)
Выбор области для следующего DSL элемента.

binding(name)
Возвращает значение первого привязанного выражения с именем `name`.

input(name).enter(value)
Вводит значение `value` в текстовое поле с именем `name`.

input(name).check()
Ставит / убирает флажок с поля выбора с именем `name`.

input(name).select(value)
Выбирается текущим значение `value` в радио кнопках с именем `name`.

input(name).val()
Возвращается текущее значение элемента управления с именем `name`.

repeater(selector, label).count()
Возвращается количество строк, которые соответствуют переданному селектору jQuery в параметре `selector`. А `label` используется только внутри теста.

repeater(selector, label).row(index)
Возвращает из массива с привязками строку с индексом `index` в выражении повторителя с использованием селектора jQuery `selector`. А `label` используется только внутри теста.

repeater(selector, label).column(binding)
Возвращает из массива со значениями столбец `binding` для которого значение соответствует переданному jQuery селектору `selector`. А `label` используется только внутри теста.

select(name).option(value)
Проверка, что значение `value` выбрано в элементе select с именем `name`.

select(name).option(value1, value2...)
Проверка, что все значения `values` выбраны в элементе select `name`.

element(selector, label).count()
Возвращает количество элементов, которые соответствуют jQuery селектору `selector`. А `label` используется внутри теста.

element(selector, label).click()
Клик по элементу, найденному по jQuery селектору `selector`. А `label` используется только внутри теста.

element(selector, label).query(fn)
Выполнение функции `fn(selectedElements, done)`, с значением selectedElements которые находятся по jQuery селектору `selector` и done это функция, которая выполняется, когда завершается функция `fn`. А `label` используется только внутри теста.

element(selector, label).{method}()
Возвращает результат выполнения метода `method` на элемента, найденом с помощью jQuery селектор `selector`, где `method` может быть любой из jQuery методов: `val`, `text`, `html`, `height`, `innerHeight`, `outerHeight`, `width`, `innerWidth`,`outerWidth`, `position`, `scrollLeft`, `scrollTop`, `offset`. А `label` используется только внутри теста.

element(selector, label).{method}(value)
Выполняет `method` передавая в качестве параметра `value` на элементе, найденом с помощью jQuery селектора `selector`, где `method` может быть любым из следующих методов jQuery: `val`, `text`, `html`, `height`, `innerHeight`, `outerHeight`, `width`,`innerWidth`, `outerWidth`, `position`, `scrollLeft`, `scrollTop`, `offset`. А `label` используется только внутри теста.

element(selector, label).{method}(key)
Возвращает результат выполнения метода `method` которому в качестве параметра передаются ключ `key` на элементе, найденом с помощью jQuery селектора `selector`, где `method` может быть любым и следующих jQuery методов: `attr`, `prop`, `css`. А `label` используется только внутри теста.

element(selector, label).{method}(key, value)
Выполняет метод `method` которому в качестве параметров передаются ключ `key` и значение `value` на элементе, найденом с помощью jQuery селектора `selector`, где `method` может быть любым и следующих jQuery методов: `attr`, `prop`, `css`. А `label` используется только внутри теста.

JavaScript — это динамически типизированный язык, который включает множество выражений, но это почти нет помощь от компилятора. По этой причине мы считаем очень хорошей практикой, что любой код, написанный на JavaScript необходимо пропустить через хороший набор тестов. Мы построили множество функций в Angular, которые делают тестирование Angular приложений легким. Так что нет никакого оправдания для отсутствия тестирования.

Объяснение шаблонов Angular

Angular шаблон – это декларация того, как информация из модели и контроллера будет отображаться в представлении, которое пользователь видит в окне браузера. Это статический DOM, который содержит HTML, CSS, специфичные элементы и атрибуты angular. Angular элементы и атрибуты управляют angular для добавления поведения и трансформации шаблона DOM в динамическое представление DOM.

Эти Angular элементы и атрибуты можно использовать в шаблоне:

- Directive — атрибут или элемент увеличивающий функционал существующий в DOM или перерисовывающий используемый компонент DOM – по другому виджет.
- `Markup` — Нотация в двойных фигурных скобках для привязки выражений к элементам, встроенная в Angular.
- Filter — Форматируют ваши данные для отображения пользователю.
- Form controls — предоставляет проверку ввода пользователя.

Заметьте: В дополнение к декларации элементов в шаблоне, вы можете также получить доступ к элементам из JavaScript кода.

Следующий код показывает простой пример шаблона Angular сделанному из тегов HTML расширенному с помощью директив Angular и привязанному к данным с помощью выражения:

```
1. <html ng-app>
2.  <!—Тег Body с директивой ngController-->
3.  <body ng-controller="MyController">
4.    <input ng-model="foo" value="bar">
5.    <!—тег Button с директивой ng-click, и
6.           строковое выражение 'buttonText'
7.           Обернутое в "{{ }}" markup -->
8.    <button ng-click="changeFoo()">{{buttonText}}</button>
9.    <script src="angular.js">
10.  </body>
11.</html>
```

В простом одностраничном приложении, шаблон состоит из HTML, CSS, и директив angular содержащихся в одном файле HTML (обычно `index.html`). В более сложных приложениях, вы можете отобразить множество представлений с одной главной страницы используя «части», которые являются сегментами шаблона и расположенны в разных HTML файлах. Вы "включаете" части в главную страницу используя сервис `$route` и отображаете их с помощью директивы `ngView`. Пример этой техники можете увидить в angular туториале, в шагах семь и восемь.

Работа с CSS в Angular

Angular устанавливает CSS классы. Используя это вы можете стилизовать ваше приложение.

CSS classes used by angular

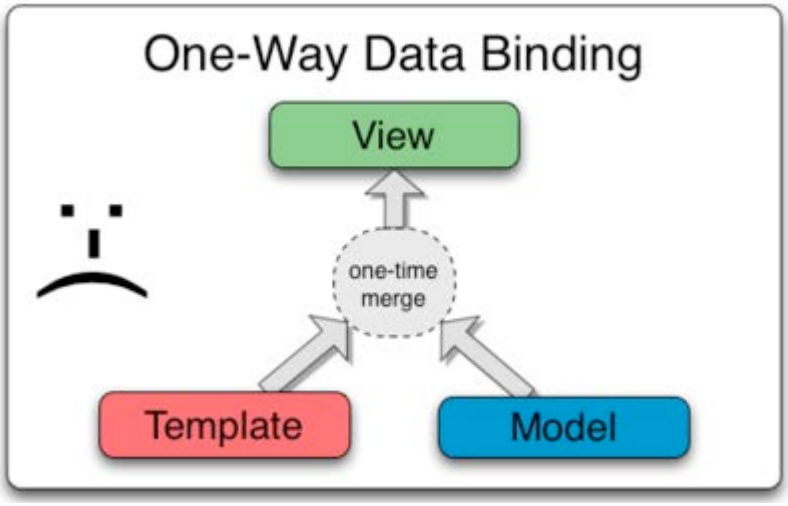
- `ng-invalid`, `ng-valid`
  - **Использование:** angular применяет эти классы к элементам ввода данных, если значение этих элементов не валидно. (смотрите `input` директиву).
- `ng-pristine`, `ng-dirty`
  - **Использование:** angular директива `input` помечается классом `ng-pristine` в случае если она новая и еще не взаимодействовала с пользователем. Сразу после взаимодействия с пользователем, класс будет изменен на `ng-dirty`.

Привязка данных

В веб-приложении Angular привязанные данные автоматически синхронизируются между данными модели и представления. Эта возможность Angular расширяется использование привязок данных к моделям, которые существуют в одном экземпляре в приложении. Представление следит за

изменением состояния модели все время. Когда модель изменяется, представление обновляет свое содержимое.

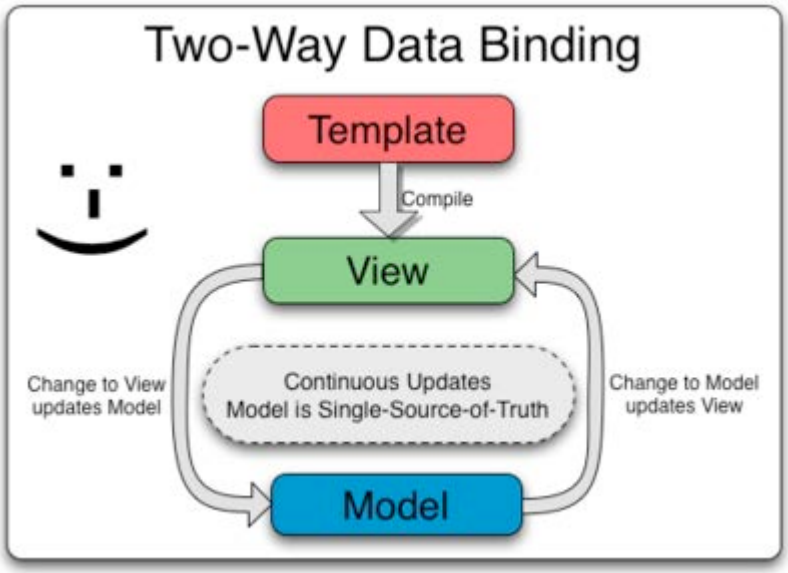
Привязка данных в классических системах шаблонов



Большинство шаблонных систем привязывают данные только одним способом: они объединяют шаблон и модель вместе в представление, что показано на рисунке. После объединения, изменения в модели или связанной секции в представлении не автоматически отображаются на представление. Такая же картина и при изменении, внесенном пользователем в представлении, оно не отображается автоматически на модель. В результате разработчику приходится писать код, который синхронизирует представление с моделью и наоборот.

Привязка данных в шаблонах Angular

Angular шаблоны работают по другому, что показано на рисунке. Это потому, что сначала шаблон (который является не скомпилированным HTML расширяется дополнительными выражениями или директивами) компилируется в браузере, и на следующем этапе создается актуальное представление. Любые изменения в представлении немедленно отображаются на модели, и наоборот. Это делает модель единственным источником данных для состояния приложения, что очень легко программируется разработчиками. Вы можете значение для представления просто задать в вашей модели.



Потому что представление это только проекция модели, контроллер полностью отделен от представления и ничего о нем не знает. Это делает тестирование простым, т.к. легко тестировать контроллер в изоляции от представления и связанных DOM/браузерных зависимостей.

Объяснение фильтров

Angular фильтры форматируют выходные данные для пользователя.

К примеру, вы имеете данные, которые нужно отформатировать в соответствии с текущими локальными настройками пользователя. Вы можете передать фильтр внутр выражения, используя синтаксис, как здесь:

```
    name | uppercase
```

Значение выражения name просто передается в `фильтр uppercase` и выводится результат.

Создание фильтров Angular

Писать собственные фильтры очень просто: только нужно зарегистрировать новый фабричный метод, создающий фильтр в вашем модуле. Фабричная функция должна возвращать функцию фильтра, которая принимает входное значение в первом параметре. Любые аргументы фильтра передаются в дополнительных аргументах функции.

В следующем примере простой фильтр переворачивает строку задом на перед. Дополнительный параметр позволяет перевести текст в верхний регистр.

Index.html

```
<!doctype html>
<html ng-app="MyReverseModule">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl">
      <input ng-model="greeting" type="greeting"><br>
      No filter: {{greeting}}<br>
      Reverse: {{greeting|reverse}}<br>
      Reverse + uppercase: {{greeting|reverse:true}}<br>
    </div>
  </body>
</html>
```

Script.js

```
angular.module('MyReverseModule', []).
  filter('reverse', function() {
    return function(input, uppercase) {
      var out = "";
      for (var i = 0; i < input.length; i++) {
        out = input.charAt(i) + out;
      }
      // conditional based on optional argument
      if (uppercase) {
        out = out.toUpperCase();
      }
      return out;
    }
  });

function Ctrl($scope) {
  $scope.greeting = 'hello';
}
```

End to end test

```
it('should reverse greeting', function() {
  expect(binding('greeting|reverse')).toEqual('olleh');
  input('greeting').enter('ABC');
  expect(binding('greeting|reverse')).toEqual('CBA');
});
```

Использование фильтров Angular

Фильтры могут применяться и в коде используя `api/ng.$rootScope.Scope`, но обычно используются для форматирования выражений в ваших шаблонах:

```
{{ expression | filter }}
```

Обычно фильтры трансформируют данные в новый тип данных, применяя форматирование в этом процессе. Фильтры могут соединятся в цепочку, а также принимать параметры.

Вы можете соединять фильтры, используя это синтаксис:

```
{{ expression | filter1 | filter2 }}
```

Вы можете также передать через разделитель «:» аргумент в фильтр, для примера, отобразим число 123 с двумя знаками после запятой:

```
123 | number:2
```

Этот же синтаксис используется и для нескольких аргументов:

```
myArray | orderBy:'timestamp':true
```

Вот несколько примеров, которые показывают значение перед и после применением различных фильтров в выражении привязки данных:

- Нет фильтра: `{{1234.5678}}` => `1234.5678`
- Фильтр число: `{{1234.5678|number}}` => `1,234.57`. Заметьте "," - разделитель тысяч и округление до двух знаков после запятой.
- Фильтр с аргументами: `{{1234.5678|number:5}}` => `1,234.56780`. Фильтры могут принимать необязательные аргументы, разделенные двоеточием. К примеру, фильтр "number" принимает числовой аргумент, который указывает количество цифр в дробной части числа.

Angular сервисы

Сервис – это функция, которая угловой приносит к стороне клиента веб-приложений со стороны сервера, где услуги широко использовались в течение длительного времени. Услуги в угловых apps являются взаимозаменяемыми объектами, которые соединены вместе с помощью внедрения зависимости (DI).

Сервис – это возможность, которую Angular добавляет на стороне клиента для связи с сервером, когда данные возвращаются в течении длительного времени. Сервисы в приложении Angular являются взаимозаменяемыми объектами, которые соединяются вместе с помощью внедрения зависимости (DI).

Использование $location

Что он делает?

Сервис `$location` разбирает URL в адресной строке браузера (базируется на window.location) и делает URL доступным для вашего приложения. Изменение URL в адресной строке браузера отображается в сервисе $location и изменение $location отображается в адресной строке браузера.

Сервис $location:

- Используя текущий URL в адресной строке браузера, также вы можете
  - Следить и наблюдать за URL.
  - Изменять URL.
- Синхронизирует URL в браузере когда пользователь
  - Изменить адресную строку
  - Кликает на кнопки вперед или назад в браузере (или кликает на ссылку история).
  - Кликает на ссылки.
- Изменяет объект URL когда методы изменяют его части (protocol, host, port, path, search, hash).

Сравнение $location с window.location

	window.location	$location service
цель	Применить чтения/запись доступ к текущему браузера расположению	То же
API	Представляет объект "raw" со свойствами, которые могут применяться для управления изменениями	Представляет гетеры и сетеры в стиле jQuery
Интеграция с жизненным циклом приложений angular	нет	Знает о всех внутренних фазах жизненного цикла, интегрирован с $watch, ...
Легкость интеграции с HTML5 API	нет	да (только при поддержке в браузере)
Задание корневого документа / контекста из которого загружено приложение	нет - window.location.path возвращает "/docroot/actual/path"	да - $location.path() возвращает "/actual/path"

Когда я должен использовать $location?

В любое время когда ваше приложение нуждается в реакции на изменение текущего URL или если вы должны изменять текущий URL в браузере.

Когда не нужно его использовать?

Его использование не вызывает полной перезагрузки страницы в браузере. Чтобы перезагрузить страницу после изменений URL, используйте метод API более низкого уровня `$window.location.href`.

Главный обзор API

Сервис `$location` может быть разделен, в зависимости от текущей конфигурации, когда она создается. Конфигурация по умолчанию подходит для большинства приложений, для включения других возможностей нужно создать другую конфигурацию.

Сразу после создания службы `$location`, вы можете взаимодействовать с ней с использование гетеров и сетеров в стиле jQuery, которые позволяют получить или изменить текущий URL в браузере.

Конфигурация службы $location

Конфигурация службы `$location`, извлекает `$locationProvider` и устанавливает параметры как ниже:

- **html5Mode(mode)**: {boolean}
  `true` – использовать стиль HTML5
  `false` – использовать стиль с хэшем
  default: `false`
- **hashPrefix(prefix)**: {string}
  префикс используемый для построения хэша в URLs (используйте стиль с хэшем, а для поддерживающих браузеров можно использовать стиль HTML5)
  default: `""`

Пример конфигурации

```
1. $locationProvider.html5Mode(true).hashPrefix('!');
```

Методы установки и получения

`Служба` `$location` предоставляет метод получения для чтения частей URL (absUrl, protocol, host, port) и методы получения / установки для url, path, search, hash:

```
1. // получить текущий путь
2. $location.path();
3.
4. // изменить текущий путь
5. $location.path('/newValue')
```

Все методы установки возвращают объект `$location` поддерживающий изменения. Например, чтобы изменить несколько сегментов в одном действии, вызовите установщики по очереди:

```
1. $location.path('/newValue').search({key: value});
```

Есть специальный метод `replace` который может быть использован для указания службе $location что необходимо синхронизировать свое состояние с браузером, последняя запись в истории посещений перемещается на предпоследнюю позицию, а на ее место добавляется новая запись. Это используется когда вам нужно сделать переадресацию, с возможностью вернуться назад по

нажатию кнопки назад (навигация назад снова вызывает переадресацию). Для изменения текущего URL с сохранением изменения в истории браузера выполните следующий код:

```
1. $location.path('/someNewPath');
2. $location.replace();
3. // or you can chain these as: $location.path('/someNewPath').replace();
```

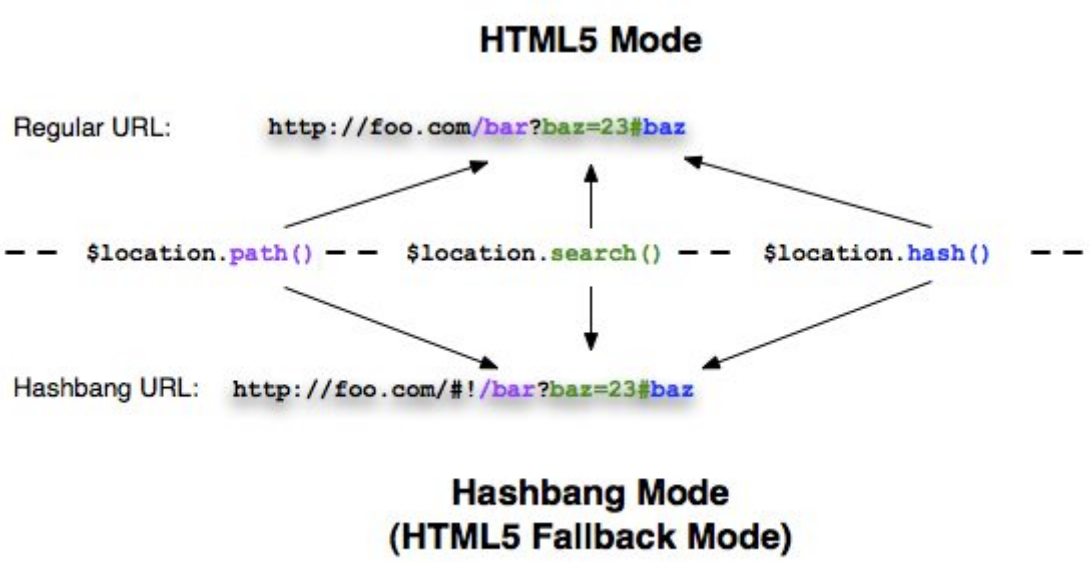
Заметьте что установка свойств сразу не обновляет `window.location`. Следовательно, сервис `$location` знает о жизненном цикле области видимости и накапливает множество изменений `$location` для однократного применения к объекту `window.location` в фазе `$digest` области видимости. Чтобы изменения состояния $location были применены в браузере, нужно вызвать метод `replace()`, только после этого изменения будут применены и обновиться история в браузере. Когда браузер обновлен, служба $location выключает флаг установленный методом `replace()` и другие изменения будут создавать новую историю браузера, снова вызывая метод `replace()`.

Установщики и кодировка символов
Вы может передать спец. символы сервису `$location` и они будут закодированы с использованием спецификации RFC 3986. Когда это доступно:

- Для всех значений, передаваемых в методы установки сервиса `$location`, `path()`, `search()`, `hash()`, могут быть кодированными.
- Получение (вычисления методов с параметрами) возвращает декодированные значения для следующих методов `path()`, `search()`,`hash()`.
- Когда вы вызываете метод `absUrl()` возвращается значение полного url с закодированными сегментами.
- Когда вы вызываете метод `url()`, возвращается значение пути, строки поиска и хэш, в форме `/path?search=a&b=c#hash`. Эти сегменты уже будут декодированы.

Режим хэшей и HTML5

Служба `$location` имеет два варианта конфигурации, которые отвечают за формат URL в адресной строке браузера:**С использование хэшей** (по умолчанию) и **HTML5 режим** который базируется на History API. Приложения используют один и тот же API в обоих режимах и служба `$location` работает с соответствующими сегментами URL используя облегченное APIs браузера чтобы изменить URL и управлять историей.



	Хэш режим	HTML5 режим
конфигурация	По умолчанию	{ html5Mode: true }
Формат URL	URLs с хэшем во всех браузерах	красивое URLs в современных браузерах, URLs с хэшем в старых.
<a href=""> переписывание ссылок	нет	да
Обязательная конфигурация сервера	нет	да

Хэш режим (по умолчанию)
В этом режиме, `$location` использует хэш в URLs для всех браузеров

Пример

```
it('should show example', inject(
  function($locationProvider) {
    $locationProvider.html5Mode(false);
    $locationProvider.hashPrefix = '!';
  },
  function($location) {
    // open http://host.com/base/index.html#!/a
    $location.absUrl() == 'http://host.com/base/index.html#!/a'
    $location.path() == '/a'

    $location.path('/foo')
```

```
    $location.absUrl() == 'http://host.com/base/index.html#!/foo'

    $location.search() == {}
    $location.search({a: 'b', c: true});
    $location.absUrl() == 'http://host.com/base/index.html#!/foo?a=b&c'

    $location.path('/new').search('x=y');
    $location.absUrl() == 'http://host.com/base/index.html#!/new?x=y'
  }
));
```

Обход вашего приложения поисковыми ботами

Чтобы разрешить индексирование вашего AJAX приложения, вам нужно добавить специальный мете-тег в секцию head вашего документа:

```
<meta name="fragment" content="!" />
```

Это приведет к обходу ботом ссылки, которая содержится `_escaped_fragment_` param так что ваш сервер может разрешить запрос бота и отдать ему требуемый фрагмент HTML. Для получения более детальной информации об этой технике смотрите Создание индексируемых AJAX приложений.

Режим HTML5

В режиме HTML5, сервис `$location`, взаимодействуя с браузером, получает и изменяет адрес URL через HTML5 истории API, которое позволяет использовать постоянный путь URL и сегмент поиска, который эквивалентен использованию хэша. Если HTML5 Истории API не поддерживается в браузере, сервис `$location` начнет использовать режим хэшей автоматически. Это избавляет вас от необходимости беспокоится, поддерживает ли браузер API истории или нет; сервис `$location` самостоятельно выберет нужную опцию.

- Открытие постоянной URL в устаревших браузерах -> переадресация на URL с хэшем
- Отркрытие URL с хэшем в новых браузерах -> переписывает его в постоянное URL

Пример

```
it('should show example', inject(
  function($locationProvider) {
    $locationProvider.html5Mode(true);
    $locationProvider.hashPrefix = '!';
  },
  function($location) {
    // в браузерах с поддержкой HTML5 истории:
    // открытие http://host.com/#!/a -> переписывается в http://host.com/a
    // (в истории сохраняется как http://host.com/#!/a)
    $location.path() == '/a'

    $location.path('/foo');
    $location.absUrl() == 'http://host.com/foo'

    $location.search() == {}
    $location.search({a: 'b', c: true});
```

```
    $location.absUrl() == 'http://host.com/foo?a=b&c'

    $location.path('/new').search('x=y');
    $location.url() == 'new?x=y'
    $location.absUrl() == 'http://host.com/new?x=y'

    // в браузерах с поддержкой html5 истории:
    // открытие http://host.com/new?x=y -> переходит к http://host.com/#!/new?x=y
    // (затем снова заменяется в истории на http://host.com/new?x=y history)
    $location.path() == '/new'
    $location.search() == {x: 'y'}

    $location.path('/foo/bar');
    $location.path() == '/foo/bar'
    $location.url() == '/foo/bar?x=y'
    $location.absUrl() == 'http://host.com/#!/foo/bar?x=y'
  }
));
```

Резерв для устаревших браузеров
Для браузеров с поддержкой HTML5 history API, сервис `$location` использует его для записи пути и строки поиска. Если history API не поддерживается браузером, `$location` применит режим с хэшами в URL. Это позволяет вам не беспокоится о поддержке браузером history API; сервис `$location` сам позаботится об этом.

Переписка Html ссылок
Когда вы используете режим HTML5 history API, вам будут нужны различные ссылки для различных браузеров, все что нужно сделать, это написать специальное постоянное URL ссылки, такое как: `<a href="/some?foo=bar">link</a>`

Когда пользователь кликнет на этой ссылке,

- В старых браузерах URL изменится на `/index.html#!/some?foo=bar`
- В современных браузерах URL изменится на `/some?foo=bar`

В следующих случаях ссылки не переписываются; вместо этого браузер будет польностью перезагружать первоначальную страницу с оригинальной ссылкой.

- Ссылка содержит атрибут `target`
  Пример: `<a href="/ext/link?a=b" target="_self">link</a>`
- Абсолютные ссылки, которые ведут на другой домен
  Пример: `<a href="http://angularjs.org/">link</a>`
- Ссылки начинающиеся с '/' ведущие к другому базовому пути при определенной базе
  Пример: `<a href="/not-my-base/link">link</a>`

На стороне сервера
Используя этот режим обязательно перепишите URL на сервере, который используется как точка входа для всех ваших ссылок внутри приложения (обычно это index.html)

Относительные ссылки
Часто требуются относительные ссылки, картинки, скрипты и т.д. Можно указать любое url как базовый в секции head в файле вашей главной страницы (`<base href="/my-base">`) или вы

можете использовать абсолютные url (начинающиеся с /) который разрешается относительно корневого элемента приложения.

Запуск приложения Angular с включенной поддержкой History API относительно корневого элемента настоятельно рекомендуется, т.к. это берет на себя разрешение всех относительных ссылок.

Перенаправление ссылок в различных браузерах
Так как ссылки переписываются при включенном режиме HTML5, пользователи могут открывать оба вида ссылок в различных браузерах:

- Современные браузеры переписывают URL к виду постоянных URL.
- Старые браузеры перенаправляются с постоянного URL на URL с хэшами.

Пример
Здесь вы увидите два экземпляра `$location`, но в режиме **Html5**, при открытии в различных браузерах, вы можете увидеть различия.

Сервисы `$location` соединяются со своими браузерами. Каждый элемент input представляет адресную строку вашего браузера.

Заметьте, что когда вы используете тип url с хэшами в первом браузере это не переписывает / перенаправляет на постоянное /hashbang url, как это преобразуется при разборе стартового URL = странице загрузки.

В этом примере мы использовали `<base href="/base/index.html" />`

`Index.html`

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-non-bindable class="html5-hashbang-example">
      <div id="html5-mode" ng-controller="Html5Cntl">
        <h4>Browser with History API</h4>
        <div ng-address-bar browser="html5"></div><br><br>
        $location.protocol() = {{$location.protocol()}}<br>
        $location.host() = {{$location.host()}}<br>
        $location.port() = {{$location.port()}}<br>
        $location.path() = {{$location.path()}}<br>
        $location.search() = {{$location.search()}}<br>
        $location.hash() = {{$location.hash()}}<br>
        <a href="http://www.host.com/base/first?a=b">/base/first?a=b</a> |
        <a href="http://www.host.com/base/sec/ond?flag#hash">sec/ond?flag#hash</a>
 |
        <a href="/other-base/another?search">external</a>
      </div>

      <div id="hashbang-mode" ng-controller="HashbangCntl">
        <h4>Browser without History API</h4>
        <div ng-address-bar browser="hashbang"></div><br><br>
```

```
        $location.protocol() = {{$location.protocol()}}<br>
        $location.host() = {{$location.host()}}<br>
        $location.port() = {{$location.port()}}<br>
        $location.path() = {{$location.path()}}<br>
        $location.search() = {{$location.search()}}<br>
        $location.hash() = {{$location.hash()}}<br>
        <a href="http://www.host.com/base/first?a=b">/base/first?a=b</a> |
        <a href="http://www.host.com/base/sec/ond?flag#hash">sec/ond?flag#hash</a>
|
        <a href="/other-base/another?search">external</a>
      </div>
    </div>
  </body>
</html>
```

Script.js

```
function FakeBrowser(initUrl, baseHref) {
  this.onUrlChange = function(fn) {
    this.urlChange = fn;
  };

  this.url = function() {
    return initUrl;
  };

  this.defer = function(fn, delay) {
    setTimeout(function() { fn(); }, delay || 0);
  };

  this.baseHref = function() {
    return baseHref;
  };

  this.notifyWhenOutstandingRequests = angular.noop;
}

var browsers = {
  html5: new FakeBrowser('http://www.host.com/base/path?a=b#h', '/base/index.html'
),
  hashbang: new FakeBrowser('http://www.host.com/base/index.html#!/path?a=b#h', '/
base/index.html')
};

function Html5Cntl($scope, $location) {
  $scope.$location = $location;
}

function HashbangCntl($scope, $location) {
  $scope.$location = $location;
}

```

```
function initEnv(name) {
  var root = angular.element(document.getElementById(name + '-mode'));
  angular.bootstrap(root, [function($compileProvider, $locationProvider, $provide)
{
    $locationProvider.html5Mode(true).hashPrefix('!');

    $provide.value('$browser', browsers[name]);
    $provide.value('$document', root);
    $provide.value('$sniffer', {history: name == 'html5'});

    $compileProvider.directive('ngAddressBar', function() {
      return function(scope, elm, attrs) {
        var browser = browsers[attrs.browser],
            input = angular.element('<input type="text">').val(browser.url()),
            delay;

        input.bind('keypress keyup keydown', function() {
          if (!delay) {
            delay = setTimeout(fireUrlChange, 250);
          }
        });

        browser.url = function(url) {
          return input.val(url);
        };

        elm.append('Address: ').append(input);

        function fireUrlChange() {
          delay = null;
          browser.urlChange(input.val());
        }
      };
    });
  }]);
  root.bind('click', function(e) {
    e.stopPropagation();
  });
}

initEnv('html5');
initEnv('hashbang');
```

Предостережения

Перезагрузка страницы

Сервис `$location` позволяет изменять только URL; он не позволяет вам перезагружать страницу. Когда вам нужно изменить URL и перезагрузить страницу или перейти на другую страницу, используйте более низкоуровневое API, `$window.location.href`.

Использование $location вне жизненного цикла области видимости

`$location` знает о жизненном цикле области видимости Angular. Когда URL изменяется в браузере, это обновляет `$location` и вызывает функцию `$apply` для того чтобы уведомить все $watchers / $observers. Когда вы изменяете `$location` из фазы `$digest` также все хорошо; `$location` будет распространять свои изменения в браузер и уведомлять о них всех $watchers / $observers. Когда вам нужно изменить `$location` из кода вне Angular (к примеру, через события DOM или при тестировании) – вы должны вызвать функцию `$apply` для распространения ваших изменений.

$location.path() и префикс ! или /

Путь всегда начинается со слеша (/); и `$location.path()` установщик будет его добавлять во всех случаях.

Заметьте, что префикс `!` в режиме использования хэшей не является частью `$location.path()`; это актуально для любого.

Тестирование сервиса $location

Когда используется сервис `$location` в тестировании, вы находитесь вне жизненного цикла области видимости. Это возлагает на вас ответственность за вызов `scope.$apply()`.

```
1. describe('serviceUnderTest', function() {
2.   beforeEach(module(function($provide) {
3.     $provide.factory('serviceUnderTest', function($location){
4.       // whatever it does...
5.     });
6.   });
7.
8.   it('should...', inject(function($location, $rootScope, serviceUnderTest) {
9.     $location.path('/new/path');
10.    $rootScope.$apply();
11.
12.    // test whatever the service should do...
13.
14.  }));
15. });
```

Переход из более раних релизов AngularJS

В более ранних версиях Angular, `$location` использует методы `hashPath` или `hashSearch` для получения пути и строки поиска. В этой версии, сервис `$location` получает путь и строку поиска с помощью методов path и search и затем использует полученную информацию для составления адреса с использование режима хэшей URL (таких как `http://server.com/#!/path?search=a`), когда это нужно.

Изменения в ваш код

Навигация внутри приложения	Измените на
$location.href = value $location.hash = value	$location.path(path).search(search)

Навигация внутри приложения	**Измените на**
$location.update(value) $location.updateHash(value)	
$location.hashPath = path	$location.path(path)
$location.hashSearch = search	$location.search(search)
Навигация вне приложения	**Используйте API более низкого уровня**
$location.href = value $location.update(value)	$window.location.href = value
$location[protocol | host | port | path | search]	$window.location[protocol | host | port | path | search]
Доступ для чтения	**Измените на**
$location.hashPath	$location.path()
$location.hashSearch	$location.search()
$location.href $location.protocol $location.host $location.port $location.hash	$location.absUrl() $location.protocol() $location.host() $location.port() $location.path() + $location.search()
$location.path $location.search	$window.location.path $window.location.search

Двунаправленная привязка к $location

Компилятор Angular сейчас не поддерживает двунаправленную привязку для следующих методов (см. здесь). Если требуется друнаправленная привязка к объекту $location (используя директиву `ngModel` для полей ввода), вам нужно будет создать дополнительные свойства модели (например `locationPath`) с двумя наблюдателями, которые следят за изменением $location в обоих направлениях. Вот пример:

```
1. <!-- html -->
```

```
<input type="text" ng-model="locationPath" />
```

```
// js - controller
$scope.$watch('locationPath', function(path) {
  $location.path(path);
});

$scope.$watch('$location.path()', function(path) {
  scope.locationPath = path;
});
```

Создание сервисов

Хотя Angular представляет некоторые сервисы, для любого не тривиального приложения вы найдете полезным написать собственные сервисы. Чтобы сделать это, начните с регистрации фабричной функции для создания вашего сервиса в модуле используя `Module#factory api` или внутри функции конфигурации модуля, используя сервис `$provide`.

Все службы Angular могут использоваться для внедрение зависимостей (DI), для этого используется имя сервиса, указанное при его регистрации, которое также указывается при внедрении зависимости. Возможность замены зависимостей на макеты / заглушки / заменители в тестах позволяет легко тестировать сервисы.

Регистрация служб

Чтобы зарегистрировать службу, нужно иметь модуль, частью которого и станет служба. Вы можете зарегистрировать службу с использование `Module api` или используя сервис `$provide` в функции конфигурации модуля. Следующий псевдо код это демонстрирует:

Использование angular.Module api:

```
var myModule = angular.module('myModule', []);
myModule.factory('serviceId', function() {
  var shinyNewServiceInstance;
  //тело фабричной функции, которая конструирует shinyNewServiceInstance
  return shinyNewServiceInstance;
});
```

Используя сервис $provide:

```
angular.module('myModule', [], function($provide) {
  $provide.factory('serviceId', function() {
    var shinyNewServiceInstance;
    // тело фабричной функции, которая конструирует shinyNewServiceInstance
    return shinyNewServiceInstance;
  });
});
```

Заметьте, вы не регистрируете экземпляр сервиса, но предоставляете фабричную функцию, которая создает этот экземпляр, когда она вызывается.

Зависимости

Службы не только могут вставляться в зависимости, но и могут иметь собственные. Они должны быть указанны в аргументах фабричной функции. Почитать больше о внедрении зависимостей (DI) в Angular. Лучше используйте нотацию, когда свойство $inject является массивом, в этом случае ваш код легко пройдет через минификаторы.

Следующий пример показывает простой сервис. Он зависти от сервиса `$window` (который передается в качестве параметра в фабричную функцию) и является функцией. Сервис просто сохраняет все пояснения, а затем выводит их используя window alert.

```
1. angular.module('myModule', [], function($provide) {
2.   $provide.factory('notify', ['$window', function(win) {
3.     var msgs = [];
4.     return function(msg) {
5.       msgs.push(msg);
6.       if (msgs.length == 3) {
7.         win.alert(msgs.join("\n"));
8.         msgs = [];
9.       }
10.    };
11.  }]);
12. });
```

Создание экземпляра сервиса Angular

Все сервисы Angular создаются в «ленивом» стиле. Сервис создается только тогда, когда его экземпляр нужен другому сервису или компонент приложения зависит от него. Другими словами, Angular не создает экземпляр сервиса сразу после регистрации, а делает это после получения запроса или указания от приложения.

Сервисы являются одиночками

И наконец, все сервисы Angular реализуют паттерн «одиночка». Существует только один экземпляр каждого сервиса на инжектор. У Angular смертельная аллергия на глобальные состояния, но позволительно создать несколько инжекторов, каждый из которых будет иметь собственный экземпляр службы, хотя это редко бывает нужным, за исключением тестирования.

Инъекция сервисов в контроллер

Использование сервисов как зависимости для контроллера похоже на использование сервиса как зависимости для другого сервиса.

Так как JavaScript является динамически типизируемым языком, DI не может выяснить какой сервис вставляется в статичекие типы (но не в статически типизированных языках). Однако можно указать имена сервисов, используя свойство `$inject` , которое является массивом, содержащим строки имен сервисов для инъекции. Имя должно быть одинаковым с ID сервиса, которое было заданно при его регистрации. Найденный сервис будет создан и вставлен в качестве параметра. Порядок значений имен в массиве имеет значение, а имена значений не имеют, и могут быть любыми.

```
1. function myController($loc, $log) {
2. this.firstMethod = function() {
3.  // использование сервиса $location
4.  $loc.setHash();
```

```
};
this.secondMethod = function() {
 // использование сервиса $log
 $log.info('...');
};
}
// какие сервисы нужны?
myController.$inject = ['$location', '$log'];
```

index.html

```
<!doctype html>
<html ng-app="MyServiceModule">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="myController">
      <p>Let's try this simple notify service, injected into the controller...</p>
      <input ng-init="message='test'" ng-model="message" >
      <button ng-click="callNotify(message);">NOTIFY</button>
    </div>
  </body>
</html>
```

script.js

```
angular.
 module('MyServiceModule', []).
 factory('notify', ['$window', function(win) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        win.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  }]);

function myController(scope, notifyService) {
  scope.callNotify = function(msg) {
    notifyService(msg);
  };
}

myController.$inject = ['$scope','notify'];
```

ent to end test

```
it('should test service', function() {
  expect(element(':input[ng\\:model="message"]').val()).toEqual('test');
});
```

Неявная инъекция зависимостей

Новая возможность Angular DI позволяет определить зависимости по имени параметра. Давайте перепишем предыдущий пример с использование неявной инъекции зависимостей для сервисов `$window`, `$scope` и `notify`:

Index.html

```
<!doctype html>
<html ng-app="MyServiceModuleDI">
  <head>
    <script src="http://code.angularjs.org/1.0.6/angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="myController">
      <p>Let's try the notify service, that is implicitly injected into the contro
ller...</p>
      <input ng-init="message='test'" ng-model="message">
      <button ng-click="callNotify(message);">NOTIFY</button>
    </div>
  </body>
</html>
```

Script.js

```
angular.
 module('MyServiceModuleDI', []).
 factory('notify', function($window) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        $window.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  });

function myController($scope, notify) {
  $scope.callNotify = function(msg) {
    notify(msg);
  };
}
```

Однако, если вы планируете минифицировать ваш код, ваши имена переменных будут изменены, в этом случае используйте спецификацию зависимостей с использование свойства `$inject`.

Управление зависимостями для сервисов

Angular позволяет сервисам декларировать другие сервисы как зависимости, нужные для построения экземпляра текущего сервиса.

Чтобы задекларировать зависимость, вы указываете их в сигнатуре фабричной функции. Можно просто указать имена, а можно передать в свойство $inject массив с именами сервисов. Если инъекция не нужна, тогда определение свойства `$inject` не обязательно.

Использование нотации с массивом:

```
function myModuleCfgFn($provide) {
  $provide.factory('myService', ['dep1', 'dep2', function(dep1, dep2) {}]);
}
```

Использование свойсва $inject:

```
function myModuleCfgFn($provide) {
  var myServiceFactory = function(dep1, dep2) {};
  myServiceFactory.$inject = ['dep1', 'dep2'];
  $provide.factory('myService', myServiceFactory);
}
```

Использование интерференции DI (не поддерживает минификации):

```
function myModuleCfgFn($provide) {
  $provide.factory('myService', function(dep1, dep2) {});
}
```

В этом примере два сервиса, один зависит от другого и оба зависят от третьего, который предоставляется Angular:

```
/**
 * batchLog service allows for messages to be queued in memory and flushed
 * to the console.log every 50 seconds.
 *
 * @param {*} message Message to be logged.
 */
  function batchLogModule($provide){
    $provide.factory('batchLog', ['$timeout', '$log', function($timeout, $log) {
      var messageQueue = [];

      function log() {
        if (messageQueue.length) {
          $log('batchLog messages: ', messageQueue);
          messageQueue = [];
        }
        $timeout(log, 50000);
```

```
    }

    // start periodic checking
    log();

    return function(message) {
      messageQueue.push(message);
    }
  }]);

  /**
   * routeTemplateMonitor monitors each $route change and logs the current
   * template via the batchLog service.
   */
  $provide.factory('routeTemplateMonitor',
                   ['$route', 'batchLog', '$rootScope',
                    function($route,   batchLog,   $rootScope) {
    $rootScope.$on('$routeChangeSuccess', function() {
      batchLog($route.current ? $route.current.template : null);
    });
  }]);
}

// get the main service to kick of the application
angular.injector([batchLogModule]).get('routeTemplateMonitor');
```

В этом примере:

- сервис `batchLog` зависит от сервисов `$timeout` и `$log`, и позволяет выводить в консоль сообщения.
- сервис `routeTemplateMonitor` зависит от сервиса `$route` а также от вашего `batchLog`service.
- Оба эти сервиса используют сигнатуру фабричной функции и нотацию массива для инъекции зависимостей. Важно, чтобы порядок имен сервисов в массиве соответствовал порядку аргументов в сигнатуре фабричной функции. При использовании нотации массива инжектор определяет по сигнатуре фабричной функции, какой сервис ей передать (порядок параметров в функции соответствует порядку в передаваемом массиве имен).

Тестирование сервисов Angular

Слудующий юнит тест для сервиса 'notify' из примера 'Dependencies' в создании сервиса Angular. Этот пример использует Jasmine spy (mock) вместо вызова функции реального браузера alert.

```
var mock, notify;

beforeEach(function() {
  mock = {alert: jasmine.createSpy()};

  module(function($provide) {
    $provide.value('$window', mock);
  });

```

```
  inject(function($injector) {
    notify = $injector.get('notify');
  });
});

it('should not alert first two notifications', function() {
  notify('one');
  notify('two');

  expect(mock.alert).not.toHaveBeenCalled();
});

it('should alert all after third notification', function() {
  notify('one');
  notify('two');
  notify('three');

  expect(mock.alert).toHaveBeenCalledWith("one\ntwo\nthree");
});

it('should clear messages after alert', function() {
  notify('one');
  notify('two');
  notify('third');
  notify('more');
  notify('two');
  notify('third');

  expect(mock.alert.callCount).toEqual(2);
  expect(mock.alert.mostRecentCall.args).toEqual(["more\ntwo\nthird"]);
});
```

Объяснение сервисов Angular

Сервис Angular – это одиночка, который выполняет специфичные задачи для веб-приложения, такие как `$http service` который предоставляет высокоуровневый доступ к объекту браузера `XMLHttpRequest`.

Чтобы использовать сервис Angular, вы идентифицируете его как зависимость для потребителей зависимости (контроллеров или других сервисов). В Angular при инъекции зависимости система заботится обо всем остальном. В Angular инжектор отвечает за создание экземпляра сервиса, разрешение его зависимостей, и вызова фабричной функции когда это требуется.

Angular вводит зависимости используя сервис "constructor" (сервис передается через параметры фабричной функции). Так как JavaScript динамически типизированный язык, в Angular системе внедрения зависимости нельзя использовать статические типы для выявления зависимостей. По этой причине зависимости нужно явно определять с использованием свойства `$inject.` Пример:

```
    myController.$inject = ['$location'];
```

Angular веб-фрамеворк предоставляет сервисы для большинства обычных операций. Как и другие возможности движка Angular, переменные и идентификаторы, название сервисов всегда начинается с $ (так же, как рассмотренный ранее `$http`). Вы можете также создать собственные сервисы.

Юнит тестирование

JavaScript – это динамически типизированный язык, который может выполнить любые выражения, но при этом практически нет помощи от компилятора. По этой причине код, написанный на JavaScript должен пройти через набор тестов. В Angular есть множество функций, которые облегчают тестирование приложений Angular. Так что нет причин для того чтобы не тестировать ваши приложения.

Все о НЕ смешивании проблемм

Юнит тестирование предназначено для тестирования блоков кода, как следует из его названия. Юнит тесты пытаются задавать вопросы, такие как "Работает ли данный блок кода корректно?" или "Делается ли сортировка в правильном порядке?"

Для того, чтобы иметь возможность ответить на эти вопросы, имеет большое значение изоляция кода. Это потому, что когда мы тестируем функцию сортировки, мы не хотим учитывать другие детали, такие как элементы DOM или делать любые манипуляции с XHR для получения данных.

Хотя это может показаться очевидным, обычно бывает трудно вызывать отдельные функции на типичном объекте. Проблемма в том, что разработчики часто смешивают все в одно куске кода, который делает все. Он читает данные с помощью XHR, сортирует их, а затем манипулирует DOM.

В Angular мы попытались сделать тестирование легким для использования, для чего мы предоставляем внедрение зависимостей для XHR (который можно имитировать) и мы создали абстракции, которые позволяют вам сортировать модель без необходимости прибегать к манипуляции с DOM. В итоге легко написать функцию сортировки, которая сортирует некоторые данные, и при этом ваш тест может создать эти данные, применить функцию, и проверить утверждения.

С большими возможностями больше ответственности

Angular писалась с учетом потребностей тестирования, но все же требуется чтобы и вы делали все правильно. Мы попытались облегчить вам работу, и если вы будете делать все правильно, вы в итоге получите легко тестируемое приложение.

Внедрение зависимостей

Существуют несколько способов для внедрения зависимостей: 1. Можно создать ее с помощью оператора new. 2. Вы можете найти его в известном месте, например в глобальном объекте. 3. Вы можете запросить ее у реестра (также известной как служба реестра, но как вы ее получите. Скорее всего это вариант 2). 4. Вы можете попросить передать ее вам.

Из четырех вариантов, только последний является тестируемым. Давайте посмотрим почему:

Использование оператора ***new***

Хотя нет ничего плохого, в применении оператора new принципиальным является вопрос, что вызывая его каждый раз, вы привязываете вызов функции к типу. Например, можно сказать, что каждый раз когда мы хотим получить данные с сервера, мы создаем экземпляр XHR.

```
1. function MyClass() {
2.   this.doWork = function() {
3.     var xhr = new XHR();
4.     xhr.open(method, url, true);
5.     xhr.onreadystatechange = function() {...}
6.     xhr.send();
7.   }
8. }
```

Этот вопрос встает при тестировании, мы очень хотели бы создать экземпляр заменитель `MockXHR` который позволит нам использовать нужные нам данные или имитировать сбой сети. Путем создания `new XHR()` мы постоянно связанны с фактическим XHR, и нет хорошего способа заменить его. Да, можно сделать это третьими путями, но это плохая идея по многим причинам, которые выходят за рамки даннаго документа.

Класс выше трудно проверить, поэтому нужно использовать обходные пути:

```
var oldXHR = XHR;
XHR = function MockXHR() {};
var myClass = new MyClass();
myClass.doWork();
// assert that MockXHR got called with the right arguments
XHR = oldXHR; // if you forget this bad things will happen
```

Глобальный просмотр:

Другой вариант решения проблемы это просмотр сервисов, расположение которых уже известно.

```
function MyClass() {
  this.doWork = function() {
    global.xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
  }
}
```

Хотя создается новый экземпляр зависимости, это принципиально то же, что и создание нового с помощью new, так как нет хорошего способа для перехвата вызова `global.xhr` в целях тестирования, опять придется чего то придумывать. Основной вопрос при тестировании, это то, что глобальную переменную нужно изменить для того чтобы заменить ее макетом.Для объяснение, почему это плохо, смотрите : Хрупкое глобальное состояние и одиночки

Класс выше трудно тестировать, так как мы должны изменить глобальное состояние:

```
var oldXHR = global.xhr;
global.xhr = function mockXHR() {};
var myClass = new MyClass();
myClass.doWork();
// assert that mockXHR got called with the right arguments
global.xhr = oldXHR; // if you forget this bad things will happen
```

сервис реестр:

Как может показаться, проблема может быть решена путем ведения реестра для всех сервисов, а затем в тестах заменять ими сервисы при необходимости.

```
function MyClass() {
  var serviceRegistry = ????;
  this.doWork = function() {
    var xhr = serviceRegistry.get('xhr');
```

```
    xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
}
```

Однако, откуда взять сервис serviceRegistry? Если это: * new объект, тогда тест не имеет ни какой возможности для сброса служб тестирования * глобального просмотра, то также возвращается глобальная служба (но сброс легче, так как есть только одна глобальная переменная для сброса).

Класс выше трудно проверить, поскольку мы должны изменить глобальное состояние:

```
var oldServiceLocator = global.serviceLocator;
global.serviceLocator.set('xhr', function mockXHR() {});
var myClass = new MyClass();
myClass.doWork();
// assert that mockXHR got called with the right arguments
global.serviceLocator = oldServiceLocator; // if you forget this bad things will h
appen
```

Передача зависимостей:

И последнее, зависимость может быть передана в метод.

```
function MyClass(xhr) {
  this.doWork = function() {
    xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
}
```

Это предпочтительный способ, так как код не делает ни каких предположений о том, что за объект `xhr` пришел, кто его создал и передал во внутрь. Так как его создает другой код, а не использующий его класс, он отделяет ответсвтенность создания от логики, и вот это и есть внедрение зависимостей в двух словах.

Класс выше легко тестируется, поскольку мы можем написать:

```
function xhrMock(args) {...}
var myClass = new MyClass(xhrMock);
myClass.doWork();
// assert that xhrMock got called with the right arguments
```

Заметьте, глобальные переменные не изменялись при написании этого теста.

Angular поставляется с внедрением зависимостей в результате делать правильные вещи становится легче, но все же их нужно делать.

Контроллеры

В любом приложении уникальным является то что оно делает, т.е. логика, вот ее мы и хотим протестировать. Если логика вашего приложения содержит манипуляции с DOM, его будет трудно протестировать, это показано ниже:

```
function PasswordCtrl() {
  // get references to DOM elements
  var msg = $('.ex1 span');
  var input = $('.ex1 input');
  var strength;

  this.grade = function() {
    msg.removeClass(strength);
    var pwd = input.val();
    password.text(pwd);
    if (pwd.length > 8) {
      strength = 'strong';
    } else if (pwd.length > 3) {
      strength = 'medium';
    } else {
      strength = 'weak';
    }
    msg
     .addClass(strength)
     .text(strength);
  }
}
```

Приведенный выше код является проблемным с точки зрения пригодности для тестирования, так как тесту требуется иметь правильное представление DOM, перед тем, как выполнить тест. Тест может выглядеть следующим образом:

```
var input = $('<input type="text"/>');
var span = $('<span>');
$('body').html('<div class="ex1">')
  .find('div')
    .append(input)
    .append(span);
var pc = new PasswordCtrl();
input.val('abc');
pc.grade();
expect(span.text()).toEqual('weak');
$('body').html('');
```

В контроллерах angular логика строго отделяется от манипуляций с DOM, что в итоге делает его более пригодным для тестирования, посмотрите на пример ниже::

```
function PasswordCtrl($scope) {
```

```
  $scope.password = '';
  $scope.grade = function() {
    var size = $scope.password.length;
    if (size > 8) {
      $scope.strength = 'strong';
    } else if (size > 3) {
      $scope.strength = 'medium';
    } else {
      $scope.strength = 'weak';
    }
  };
}
```

И на тест для него

```
var pc = new PasswordCtrl();
pc.password('abc');
pc.grade();
expect(pc.strength).toEqual('weak');
```

Обратите внимание, что тест не только намного короче, но и легче понять что происходит. Мы говорим, что тест рассказывает историю, а затем проверяет утверждения случайных свойств, которые не связаны между собой.

Фильтры

Фильтры это функции, которые преобразуют данные в требуемый формат для чтения пользователем. Они важны, поскольку освобождают логику приложения от необходимости форматирования данных, что ее упрощает еще больше.

```
myModule.filter('length', function() {
  return function(text){
    return (''+(text||'')).length;
  }
});

var length = $filter('length');
expect(length(null)).toEqual(0);
expect(length('abc')).toEqual(3);
```

Директивы

Директивы в angular отвечают за инкапсуляцию комплексной функциональности внутри пользовательского тега HTML, атрибута, класса или комментария. Модульные тесты очень важны для директив, поскольку компоненты, которые вы создаете с помощью директив, могут быть использованы во всем приложении и во многих различных контекстах.

Простая директива HTML

Давайте начнем с определение зависимостей в angular приложении.

```
var app = angular.module('myApp', []);
```

Сейчас можно добавить директиву в ваше приложение.

```
app.directive('aGreatEye', function () {
    return {
        restrict: 'E',
        replace:  true,
        template: '<h1>lidless, wreathed in flame</h1>'
    };
});
```

Эта директива использует тег `<a-great-eye></a-great-eye>`. Она заменяет входной тег на шаблон `<h1>lidless, wreathed in flame</h1>`. Сейчас напишем юнит тест на jasmine для тестирования функциональности.

```
describe('Unit testing great quotes', function() {
    var $compile;
    var $rootScope;

    // Загрузка модуля myApp, который содержит директиву
    beforeEach(module('myApp'));

    // Сохранение ссылок на $rootScope и $compile
    // так что они доступны для всех тество внутри блока describe
    beforeEach(inject(function(_$compile_, _$rootScope_){
      // Инжектор распаковывает параметры с (_) для соответствующих параметров
      $compile = _$compile_;
      $rootScope = _$rootScope_;
    }));

    it('Replaces the element with the appropriate content', function() {
        // Компиляция HTML содержимого для директивы
        var element = $compile("<a-great-eye></a-great-eye>")($rootScope);
        // Проверка, что элемент после компиляции содержит требуемое значение
        expect(element.html()).toContain("lidless, wreathed in flame");
    });
});
```

Мы внедрили службы $compile и $rootScope перед каждым тестом jasmine. Сервис $compile используется для отображения директивы aGreatEye. После визуализации директивы мы проверяем что директива заменила свое содержимое на "lidless, wreathed in flame".

Mocks

oue

Global State Isolation

oue

Preferred way of Testing

uo

JavaScriptTestDriver

ou

Jasmine

ou

Sample project

uoe