

JavaScript Angular. Make it better.

Revisions History

Date:	Version:	Description:	Author:
<18/08/2016>	<1.0>	JavaScript Angular. Make it better.	Sajera

Table of contents

Introduction

AngularJS — JavaScript-фреймворк с открытым исходным кодом. Предназначен для разработки одностраничных приложений. спроектирован с убеждением, что декларативное программирование лучше всего подходит для построения пользовательских интерфейсов и описания программных компонентов, в то время как императивное программирование отлично подходит для описания бизнес-логики .

Основы Angular:

Фреймворк Angular.js действительно ускоряет и упрощает разработку, расширение и тестирование одно страничных приложений. Возникает вопрос - «Как это получается ?» и «Почему именно Ангуляр ?». Ведь существует большое количество и библиотек и фреймворков которые решают такие же задачи. Давайте попробуем на них ответить попроще. Каждый разработчик JavaScript слышал и скорее всего делал «**велосипеды**» и «**костыли**». Естественно в большинстве случаев именно такой код (вне зависимости от его качества) является **проблемой** как при создании так и для тестирования и расширения приложений. На самом деле даже использования фреймворков и библиотек часто приводит к созданию всё тех же костылей и велосипедов только уже с использованием фреймворков и библиотек. С этой точки зрения Ангуляр являеться хорошим помощником, так как в своей базовой поставке включает в себя полный комплект абстракций необходимых для создания приложения и позволяет добавлять свои уникальные абстракции не нарушая общего концепта построения приложения. Естественно как следствие этих самых абстракций гораздо больше чем в других open source фреймворках. Что в свою очередь благотворно сказывается на читаемости и количество «проблемного» кода в приложении. И существенно повышает «точку входа» в Ангуляр так как перед тем как начать более менее серьёзную разработку необходимо разобраться с большей частью этих абстракций.

Базовая поставка:

Как мы любим говорить «из коробки» Ангуляр нам предлагает(можно сказать даже диктует)

1. Инициализацию приложения — которая реализует инициализацию всех частей приложения в правильной последовательности.
2. Абстракция модуля — для создания отдельных частей приложения и указания зависимостей каждого модуля/элемента/компонента приложения.
3. Ajax архитектуру — для получения и отправки данных приложения.
4. MVC архитектуру — для управления клиентскими данными приложения
5. Шаблонизатор — для построения представлений(view)
6. Компоненты – как для представления так и для работы с данными.

Исходя из вышеперечисленного мы видим что Ангуляр затрагивает, в той или иной мере, все стороны разработки. Вы можете подумать, что это далеко не все что необходимо для создания приложения и будите абсолютно правы. Ведь фреймворк не может(и не должен) делать за нас нашу работу. Для реализации и добавления модулей, элементов, компонентов, библиотек и даже других фреймворков к нашему приложению такого набора вполне достаточно(идеальных не существует).

При создании приложений вы столкнетесь с тем что Ангуляр добавляет ряд абстракций, и в ту же очередь исключает привычные способы на прямую влиять на html DOM. Что достаточно часто ставит в тупик разработчиков начинающих работать с Ангуляром. Тем не менее в Ангуляре предполагается возможность управления DOM элементами при помощи абстракции **directive**. На ранней стадии осознать ценность такого подхода достаточно сложно - поэтому мы просто запомним что манипуляции с DOM элементами только при помощи **directive.**

Так же мы столкнемся с понятием «angular way» - концепция декларативного программирования если коротко то - для создания любого модуля любого типа с любыми задачами мы должны указать его зависимости — в свою очередь созданный модуль может быть тоже чьей то зависимостью.

1. **Декларация** (указание зависимостей модуля)
2. **Создание модуля** (с использованием только указанных зависимостей)
3. **Использование** (указание модуля как зависимость другого модуля)

Готовое приложение являет собой основной модуль **app** который инициализирует все свои зависимости каждая из которых инициализирует свои и так далее. Благо не обязательно все это делать вручную — достаточно создать «angular way» модули и указывать в виде зависимости где мы их будем использовать. Все остальное Ангуляр сделает самостоятельно.

Стоит добавить, что все и каждый из вышеперечисленных частей Ангуляра могут быть дополнительно настроены под ваши специфические нужды, правда в подавляющем большинстве случаев в такой настройке нет необходимости. Так же, на момент написания этого пособия, на базе фреймворка Ангуляр написано действительно огромное количество абсолютно разнообразных компонентов, затрагивающих аспекты разработки которые не входят в базовую поставку - представляя дополнительные абстракции или расширяя/заменяя уже существующие абстракции. Подавляющее большинство сторонних компонентов имеют свою документацию, что благотворно сказывается на поддержке приложения в дальнейшем.

Injector (указание зависимостей):

Injector — одна из основных абстракций Ангуляра. Как модуль, **Injector** в своем приложении мы не используем но возможности которые он предоставляет как раз и связывают все части нашего приложения в единое целое. Если вы знакомы с модульной или LMD архитектурой то понять его назначение будет совсем просто. Если нет то попробуем ответить на вопрос «Как это работает ?» для чайников. Современное приложение состоит из большого количества частей (даже ваш «Hello world !») Состоит как минимум из нескольких файлов. Соответственно при создании даже небольших приложений возникают вопросы связанные с архитектурой приложения, начиная от организации структуры каталогов(папок) с файлами , именование папок/файлов, извечное желание пере использовать какой то код и конечно же вопрос беспокоящий весь мир веб разработки «Как разделять приложения на части ?». Есть ряд успешных и признанных фреймворков решающих такие вопросы на пример Require или Common. Мы не беремся судить кто лучше. Мы поговорим о том как это работает в Ангуляр. И так Ангуляр injector – в первую очередь не принципиально как вы называете файл, но хорошей практикой считается:

{name}.{type}.{extension} - pattern

1. **my_best_event.{type}.{extension}** – имя модуля в нижнем регистре, если используются несколько слов, то они соединяются через подчеркивание
2. **{name}.module.{extension}** – название Ангуляр абстракции к которой относиться файл (controller, directive, service, ….)
3. **{name}.{type}.js** – расширение файла модуля (.ts, .coffee)

Задача инжектора знать о всех кусочках вашего приложения благодаря чему любой описанный компонент можно вызвать по его имени указанном при создании компонента. Обратите внимание, Injector не занимается подключением файлов приложения(обычно за этим следит система автоматизации разработки проекта). Давайте рассмотрим пример. Создадим модуль без зависимостей, ведь именно с них все начинается. Модуль можно воспринимать как объект-обертка который содержит в себе все что мы посчитаем нужным добавить в него.

```
/**
* функция принимает 1 или 2 аргумента если 1 аргумент то получить модуль по указанному имени
* если 2 аргумента то создать модуль с указанным именем и списком зависимостей которые должны
* быть инициированы раньше этого модуля
* @param: { String } - имя модуля (имя должно быть уникальным в рамках приложения)
* @param: { Array } -  массив строк с именами модулей от которых зависит этот (может быть пустым)
* @returns: { Object } - объект модуля example
*/
angular.module('example', []);
```

После создания мы начинаем наполнять модуль необходимыми элементами. Для примера мы сделаем сервис который будет выполнять важный для приложения расчет примера. Вызываем ранее созданный модуль, а у него вызываем метод создания сервиса(который кстати вернет наш модуль). Таким образом когда Ангуляр приложение будет стартовать и у любого модуля может появится необходимость

производить данный расчет примера нам просто нужно будет указать в массиве зависимостей имя модуля содержащий наш сервис. Тогда сервис станет доступен для использование.

```
angular
 // с одним аргументом – вызываем ранее созданный модуль
      .module('example')
/**
* функция принимает 2 аргумента
* первый аргумент  имя сервиса, а второй аргумент функция результат работы которой в
* дальнейшем и будет передаваться как exampleService
* @param: { String } - имя модуля (имя должно быть уникальным в рамках приложения)
* @param: { Function } -  метод создания сервиса
* @returns: { Object } - объект модуля example
*/
      .service('exampleService', function ( $log, $window ) {
            // $log и $window это зависимости нашего сервиса
            // под этими именами предоставляется самый обыкновенный window и
            // обертка над обычной console с той лишь разницей что они уже
            // описаны как компоненты Ангуляр внутри самого фреймворка

            // приватный метод сервиса
            function calc( value ) {

            };

            // приватный метод сервиса
            function buildCounter ( data ) {

            };

            // приватные данные сервисса
            var counter = buildCounter();

            // что вернет эта функция и будет нашим сервисом  exampleService
            return {
                  // публичный метод сервиса
                  clear: function () {

                  },
                  // публичный метод сервиса
                  calculate: function ( value ) {

                  },
                  // публичный метод сервиса
                  get: function () {

                  },
            };
      });
```

Модуль может содержать в себе любое количество сервисов либо других компонентов Ангуляр (кроме модулей) модули указываются в зависимостях модулей, а компоненты в зависимостях у компонентов. Как пример давайте создадим другой модуль который будет содержать контроллер в котором мы и хотим использовать наш сервис.

```
// обратите внимание что этот модуль имеет зависимость к предыдущему модулю
// значит сначала должен инициализироваться модуль example
angular.module('module', ['example']);
```

Теперь создадим контроллер. Схема создания вам уже знакома — сначала вызываем модуль потом вызываем у него метод создания контроллера. С методами которые есть у модулей мы можем ознакомиться [тут](). Перечень популярных filter, controller, service, factory, directive, constant, value.

```
angular
      .module('module')
      /**
       * функция принимает 2 аргумента
       * первый аргумент  имя контролера, а второй аргумент функция в которой как аргумент
       * доступен специальный объект $scope. Из этой функции возвращать ничего ненужно
       * @param: { String } - имя модуля (имя должно быть уникальным в рамках приложения)
       * @param: { Function } -  собственно сам контроллер
       * @returns: { Object } - объект модуля module
      */
      .controller('simpleController', function ( $scope, exampleService ) {
            // exampleService – это созданный нами ранее сервис
            // и мы теперь можем его использовать.
            //  подобным образом можно инъецировать любые части приложения
             exampleService.calculate( 15 );
      });
```

Не смотря на то что мы рассматривали Ангуляр Injector, как модуль, мы его не использовали. Но пользуемся возможностями которые он предоставляет. Ведь система указания зависимостей работает именно благодаря этому модулю. Конечно его можно так же указать как зависимость и использовать как компонент, но в этом нет необходимости. А вот знать о его существовании и понимать принципы работы необходимо. Инжектор обеспечивает каждому нашему компоненту свой контекст и свою область видимости [инкапсуляция](). При этом это не требует создания своих велосипедных фабрик и оберток, а сам Injector документированный и хорошо зарекомендовавший себя подход к разработке приложений.

Bootstraping (инициализация приложения):

Инициализация или старт приложения в Ангуляр преимущественно осуществляется фреймворком самостоятельно. Ангуляр предоставляет возможность инициализировать приложение самостоятельно, но в подавляющем большинстве случаев в этом нет необходимости, а иногда даже вредит приложению. «Как же так, а как же тогда действия необходимые перед загрузкой приложения ?» Действительно хороший вопрос, и Ангуляр предоставляет очень интересное решение данного вопроса. Каждый из модулей которые мы создаем при инициализации приложения проходит две стадии сначала стадия конфигурации, потом стадия запуска модуля. Предположим, что exampleService должен перезаписывать localStorage все свои результаты. Логично будет при инициализации проверить есть ли на данном компьютере записи с предыдущих запусков и если нет то создать пустой массив(что бы не выполнять данную проверку при каждой записи). Для конкретно примера нам не принципиально какую стадию инициализации использовать. Возьмём для примера «run» запуск.

```
angular
      // вызываем модуль
      .module('example')
      // передаем функцию - что нужно выполнить при запуске
      .run( function ( $window ) {
            // проверяем наличие записи
            if ( !angular.isArray( $window.sessionStorage['example'] ) ) {
                  // записи нет создаем пустой массив
                  $window.sessionStorage['example'] = JSON.stringify( '[]' );
            }

      });
```

Прелесть, такого подхода в том, что нам нет необходимости описывать гигантский файл инициализации — каждый модуль знает только о том что ему нужно инициализировать и даже если компоненты модуля описаны в разных файлах и некоторые из них требуют каких то подготовительных действий при инициализации мы можем передавать в метод модуля «run» функцию столько раз сколько нужно, передать только те действия которые необходимы для инициализации конкретно этой части модуля, а в другом файле передать еще одну функцию с действиями необходимыми для другой части модуля. В добавок, как мы понимаем из главы об инжекторе, в нашем приложении будет не один модуль, и каждый из них будет проходить стадию конфигурации — после того как пройдет стадия конфигурации у всех его зависимостей, а потом стадию запуска, все так же, после того как пройдет стадия запуска у всех его зависимостей. Что дает возможность, не прибегая к изменению правил инициализации по умолчанию у фреймворка Ангуляр, выполнить подготовительные действия для каждого модуля и каждого компонента модуля по отдельности. Что в свою очередь, опять же, существенно упростит структуру сигнализационного кода, поможет структурировать компоненты начиная от стадии инициализации и уменьшит количество велосипедов.

Хочется обратить внимание, что не смотря на то что для данного примера нам было неважно какую стадию использовать — разница все же имеется. На стадии конфигурации как зависимости в аргументы функции не получится указать описанные нами сервисы или модели, на стадии конфигурации нам доступны только constant, value и providers. На стадии запуска уже доступны service и factory.

AJAX (получение данных):

Современное одно страничное приложение делится на две части. Каждая из которых строиться максимально независимо друг от друга. Такой подход позволяет повторно использовать и расширять функциональность уже работающего(возможно даже приносящего деньги проекта). Позволяет создавать разные представления/отображения для серверных данных или наоборот использовать похожие представления для разных серверов/данных.

1. **app —** клиентская часть (логика отображения данных в браузере)
2. **api —** серверная часть (бизнес логика и безопасность данных)

Соответственно возникает ряд сложностей с реализацией такого подхода и конечно же ассортимент вариантов их решения. Основные сложности состоят в том, что нужно создать методы запросов на сервер, которые могут получать, изменять и удалять данные. При этом методы должны быть безопасными для бизнес логики — соответственно клиентская часть должна быть в состоянии передавать любые дополнительные данные необходимые для безопасности сервера. Так же важным моментом является удобство и понятность использования методов при отображении данных. Ну и конечно же возможность обращаться к разным серверам. В связи с тем, что невозможно предугадать как именно будет работать сервер и какие требования безопасности будут приняты/расширены/добавлены на серверной части приложения — неправильная подготовка к работе с данными может превратиться в ад для разработчика.

Тут Ангуляр тоже может порадовать нас принятыми решениями. С помощью технологии ajax реализованной в Ангуляре по средствами сервиса $http, и инкапсуляции частей приложения мы можем реализовать методы отвечающие нашим требованиям. «Как этого можно добиться ?». Давайте рассмотрим на примере. Допустим у нас на сервере есть важная модель **event**, нам необходимо реализовать создание получение, изменение данных и удаления события. Для примера возьмём созданный нами ранее модуль **example** и добавим к нему еще один сервис(почему нет ?) который реализует нашу работу с данными соглано описанных выше требований.

```
angular
      .module('example')
      .service('eventService', function ( $q, $http ) {
            /**
             * приватные данные сервиса к ним мы отнесем
             * адрес сервера к которому обращаемся
             * специальный заголовок как один из вариантов по безопасности
             * лично для этой модели, так как дополнительные заголовки для всего
             * приложение лучше описывать вместе с основной конфигурацией
             * внутри сервиса мы стараемся конфигурировать только то, что
             * относиться относиться к этому конкретному сервису
             */
            var baseApiPath = 'api/example/';
            var additionHeader = 'adition-header';

            // приватный метод
            function generate () {
                  /**
                   * предположим что этот метод генерирует
                   * обязательное для сервера значение
                   */
            },
```

```
        // публичные методы
        return {
            /**
             *
             *
             * @public
             * @param: { Number } - id of event
             * @returns: { Object } - promise
             */
            getEvent: function ( eventId ) {
                var deferred = $q.defer();
                $http({
                    method: 'GET',
                    url:  baseApiPath+'event',
                    headers: {
                        [additionHeader]:  generate()
                    }
                }).then(
                    function ( success ) {
                        // дополнительные дествие при успехе
                        ...
                        // передаем предобработанные данные
                         deferred.resolve(success);
                    },
                    function ( error ) {
                        // дополнительные дествие при ошибке
                        ...
                        // передаем предобработанные данные
                         deferred.reject(error);
                    }
                );
                // возвращаем промис док
                return deferred.promise;
            },
            createEvent: function ( event ) {
                // данный пример всего лишь один из возможных вариантов
            },
            updateEvent: function ( eventId, update ) {
                // ознакомится более подробно с возможностями $http
            },
            deleteEvent: function ( eventId ) {
                // можно в официальной документации док
            }
        };
    });
```

С одной стороны может показаться, что много кода для реализации, а с другой стороны теперь наш сервис можно использовать в любой части приложения и если измениться серверная логика или правила безопасности это не повлечет изменений в местах где сервис был использован, достаточно будет внести корректировки в сам сервис. Так же, такой подход позволяет внедрять в сервис конфигурационные данные проекта без изменений по всему приложению. Что очень помогает при построении представлений данных — так как все нюансы реализации запросов остаются сокрыты внутри сервиса.

Шаблонизатор (HTML):

Работа с html в Ангуляре производиться с помощью встроенного шаблонизатора. Для тех кто работал с JavaScript шаблонизаторами такими как **ejs**, **underscore**, **handlebars** и подобными, разобраться с шаблонизатором Ангуляр не составит труда. Прежде всего, основная задача шаблонизатора это вставить динамические данные в статическую верстку. Angular templates не исключение. Для этого вводиться специальный "тег" - **{{ }}.** В отличии от обычных тегов этот тег не используется браузером, внутри этого тега вы можете вставлять JavaScript переменные и выполнять простые JavaScript действия, так же можете использовать Ангуляр фильтры и контроллеры формы. Так же в рамках работы с шаблонами Ангуляр предоставляет сервис $templateCache для оптимизации загрузки шаблонов с сервера на клиент(если в этом есть необходимость). Давайте рассмотрим пример, у нас есть блок и кнопка которая его скрывает и показывает «Как это выглядит в шаблоне ?»

```
<div class="example-wrapper " >
        <div class="example container" data-ng-show="open ? true : false " >
                Я супер-крутой скрывающийся элемент
        </div>
        <button
                type="button"
                class="btn example-btn"
                data-ng-class="{{ open ? 'btn-default' : 'btn-primary' }}"
                data-ng-click="open = !open"
                        >
                {{ open ? 'Close' : 'Open' }}
        </button>
</div>
```

Интересно то что для данного шаблона ненужны сторонние данные. Но глядя на пример скорее всего появилось больше вопросов чем ответов. Начнем с оформления. Формат HTML вполне поддерживает такое заполнение атрибутов(хотя оно может быть непривычным) я использую именно такой подход в связи с тем что при описании шаблонов Ангуляр у тегов может быть довольно много атрибутов(как вы могли заметить).

```
<button                                                         // открывающий тег
        type="button"                                           // стандартный атрибут
        class="btn example-btn"                                 // стандартный атрибут
        data-ng-class="{{ open ? 'btn-default' : 'btn-primary' }}"  // data атрибут
        data-ng-click="open = !open"                            // data атрибут
                >                                               // закрывающий знак тега
        {{ open ? 'Close' : 'Open' }}                           // контент
</button>                                                        // закрывающий тег
```

При чем на ряду с стандартными атрибутами html присутствуют и нестандартные. В данном случае нестандартные атрибуты это стандартные директивы Ангуляр. Согласно стандарта HTML 5 теги могут содержать нестандартные атрибуты, они считаются несущими дополнительную информацию, но согласно тег же правил семантики что бы дополнительный атрибут считался валидным он должен содержать приставку «data-». Ангуляр поддерживает все атрибуты (directive) как с приставкой так же и без нее. Давайте писать валидный код.

Внутри спец тега мы использовали простой JavaScript код , а именно «тернарный оператор». И вся магия у нас происходит вокруг единственной переменной которая необходима для отслеживания состояния скрывающегося элемента. Возникает вопрос «где мы взяли эту переменную ?» Вот если совсем правильно ответить то шаблонизатор её создал когда прочитал шаблон так как мы ее больше нигде не определяли то и начальное значение у не undefined. Значение этой переменной изменится когда мы клацнем на кнопку. «Почему ?» Шаблонизатор ангуляр помимо специльного тега **{{ }}** умеет работать с директивами. Ангуляр, в своей базовой поставке имеет набор простых в использовании директив док (а еще их можно создавать самостоятельно — об этом позже) когда шаблонизатор читает шаблон он ищет в нем директивы и если находит то «заменяет/выполняет» их на то, что они должны делать. Проще говоря **data-ng-click** означает, что на этот элемент нужно повесить обработчик события **onclick** который выполнит код указанный в значении атрибута, то есть «**open = !open;**» присвоение значения переменной(вспомним что изначально она равна undefined). При следующем клике этот код снова выполниться но переменная уже будет иметь значение **true**. В свою очередь директива **data-ng-show** покажет элемент если ее значение приводится к **true**. И скрывает элемент если значение приводится к **false**. Давайте возьмём другой пример. С динамическими данными. У нас есть блок текста на пример с приветствием. Нужно сделать приветствие пользователя на нашем событии с учетом типа события «Как это выглядит в шаблоне ?»

```
<div class="example-wrapper " >
        <div class="example text-center">
                Добро пожаловать на
                {{ event.type == 'male' ? 'a' : '' }}
                ! Надеемся вы посетите наш
                {{ event.name }}
                .
        </div>
</div>
```

В данном примере мы используем объект event у которого как минимум 2 свойства (имя и тип). Конечно же для того что бы шаблон отработал корректно ему нужно передать эти данные. «Как это сделать ?» Для этого нам нужен контроллер который будет работать с этим шаблоном(не обязательно только с этим). Давайте рассмотрим как это делается в следующем разделе — MVC в Ангуляре.

MVC в angular.js (организация данных для представления):

Концепция разделения клиентских данных на три основных типа — модель, представление и контроллер MVC (wiki , tutorialspoint) на данный момент не является новинкой. Ангуляр предоставляет возможности для реализации такого подхода. «Как это реализовать ?». В прошлом разделе мы рассмотрели как можно реализовать представления в html по средствам Ангуляр шаблонизатора. Моделью считаются данные полученные с сервера, как вы помните получение данных организовывается с помощью сервисов. Теперь давайте разбираться какие возможности нам предоставляет Ангуляр для работы с данными которые представляются клиенту и обработки клиентских действий. Рассмотрим пример на котором мы остановились в прошлом разделе слегка его изменив для наглядности (**пример-1**).

Мы ожидаем что сервер предоставит нам данные события. Продолжаем разбираться, как же нам его представить. Нам необходимо передать данные с сервера в шаблон. Это реализуется с помощью контроллера. Создадим модуль приветствия. Укажем ему зависимость к модулю с сервисом для работы с событиями, к которому, по уже знакомой нам схеме, добавим контроллер приветствия (**пример-2**). Исходя из модифицированного шаблона мы имеем представление на тот случай если событие еще нет(в процессе загрузки или ошибки запроса/сервера). Когда шаблон(**пример-1**) и контроллер(**пример-2**) запустятся выполнится запрос на сервер а шаблон покажет сообщение о загрузке. Когда сервер пришлет данные о событии, мы передадим эти данные в переменную шаблона и укажем, что загрузка завершена, после чего выполним отрисовку шаблона с новыми данными.

Сейчас можно подумать что мало кода для реализации такого функционала. Но ведь приложению еще нужно реагировать на действия пользователя. А описание как реагировать на действия в основном и находится в контроллере. На пример рассмотренная нами ранее директива **data-ng-click** и подобные ей может вызывать методы описанные в контроллере как переменные, все того же, специального объекта **$scope** док .

Пример-1: пример шаблона к которому мы применим контроллер

```
<div class="example-wrapper " >
        <div class="example text-center" data-ng-show="vm.loaded" >
                Добро пожаловать на {{ vm.event.type }} !
                Надеемся вы посетите наш {{ vm.event.name }} .
        </div>
        <div class="example text-center" data-ng-hide="vm.loaded" >
                Ожидаем загрузку события...
        </div>
</div>
```

Пример-2: пример модуля с контролером для передачи данных в шаблон

```
angular
      // создаем модуль
      .module('module',['example'])

      .controller('greetingsController', function ( $scope, eventService ) {
            /**
             * контроллер это функция которая подготавливает данные для
             * представления. Каждому контроллеру доступен
             * специальный объект $scope поля этого объект как раз и являются
             * переменными к которым мы обращаемся в шаблоне
             * для удобства я использую следующий подход
             */
            var vm = $scope.vm = {
                  // создадим событие по умолчанию
                  evnent: null,
                  // создадим флаг (запросы на сервер это асинхронный код)
                  loded: false
            };
            // мы будем получать в данном случае одно и то же событие с id 1
            eventService
                  .getEvent( 1 )
                  .then(
                        function ( event ) {
                              // говорим что нужно выполнить это действие и
                              // перерисовать представление(шаблон)
                              $scope.evalAsync( function () {
                                    // передаем событие в представление
                                    vm.event = event;
                                    // сообщаем представлению о загрузке
                                    vm.loded = true;
                              });
                        }
                        // в данном примере не будем обрабатывать ошибку
                  );
      });
```

Компоненты в angular.js (расширение функционала):

Как бы много нам не давали библиотеки и фреймворки, всегда найдется задача которая требует «особенного» подхода. Ангуляр представляет широкий ассортимент возможностей для расширения функционала. И на момент написания этого пособия, можно найти много компонентов, уже написанных и документированных, фактически на любой вкус, которые расширяют возможности предоставляемые Ангуляром. Как мы знаем большой ассортимент возможностей «из коробки» ускоряет разработку и уменьшает количество велосипедов. Давайте попробуем разобраться какие компоненты бывают и почему это удобно/востребовано.

Сторонние компоненты Ангуляр поставляются на основе его базовых типов данных. Вы подключаете маленькую библиотеку которая добавляет к вашему приложению Ангуляр модули наполненные сервисами, провайдерами, директивами и т. д. Которые мы сможем указывать как зависимости в своих компонентах.

«Почему это удобно ?» Давайте возьмем для примера наш модуль **example.** Допустим, что наши сервисы (напомню exampleService и eventService) получились очень удачные и нам хочется их использовать в другом проекте. Так как сам модуль, кроме как от базовой поставки angular.js, больше ни отчего не зависит, мы можем взять все файлы с его частями, собрать в один файл и вот, пожалуйста, модуль готов к экспортированию. Кончено же было бы чудненько, написать к нему документацию и добавить файл с минимизированной версией того же кода, но это больше относится к правилам «хорошего тона» при создании библиотек, чем непосредственно к ре использованию кода.

Так как с созданием сервисов и контроллеров мы уже сталкивались, в этом разделе стоит коснуться типов данных которые мы еще не рассматривали, таких как **filter** и **directive.** Так как это востребованные и удобные возможности поставляемые Ангуляр.

filter:

Фильтры, так же как большинство типом данных Ангуляр, принадлежат к модулю. И так же после подключения модуля с фильтром он становится доступным и в шаблонах и в JavaScript коде. Фильтры в JavaScript коде нельзя указать как зависимость по имени. Для них есть специальный поставщик $filter который и указывается как зависимость, и содержит все доступные в приложении фильтры.(**пример-1**) В шаблонах, фильтры доступны на прямую, под своими именами, после указания специального символа - **|.**(**пример-2**) Так же давайте рассмотрим основные моменты создания фильтров. При создании фильтров стоит учитывать что они могут выполнять как задачи фильтрации массива,(**пример-3**) так и задачи преобразования единичного элемента.(**пример-4**) Естественно если мы будем использовать фильтр не по назначению, то и результаты его работы нас вряд ли устроят.

Пример-1: получения специального метода $filter в контроллере

```
angular
	.module('module')
	.controller('greetingsController', function ( $scope, $filter ) {
		/**
		 * сам поставщик это функция принимающая один аргумент - имя фильтра
		 * и возвращает функцию фильтра
		 * в данном случае мы вызовем стандартный фильтр Ангуляр
		 * который входит в базовую поставку
		 * результатом работы фильтра будет массив элементов
		 * с подходящими под условия фильтрации параметрами
		 */
		var filterResults = $filter('filter')([...],function ( item ) {
			// произведем нестандартное сравнение
			if ( ... ) {
				// вернем элемент если он удовлетворяет условиям
				return item;
			}
		});
	});
```

Пример-2: **использование** фильтров в шаблонах

```
// эта директива позволят создать переменную (поле объекта $scope) прямо в шаблоне
<div data-ng-init="friends = [{name:'John', phone:'555-1276'},
	{name:'Mary', phone:'800-BIG-MARY'},
	{name:'mike', phone:'555-4321'},
	{name:'Adam', phone:'555-5678'},
	{name:'julie', phone:'555-8765'},
	{name:'Juliette', phone:'555-5678'}]"
	>
</div>
<ul>
	// эта директива создаст столько тегов ли(с содержимым) сколько элементов в массиве friends
	<li data-ng-repeat="friend in friends | orderBy: 'name'">
		// orderBy – это стандартный фильтр сортировки который выполнит сортировку массива
		// по полю name каждого элемента массива (сравнит их значения)
		Имя: {{ friend.name }}
		<br>
		Телефон: {{ friend.phone }}
	</li>
</ul>
<ul>
	<li data-ng-repeat="friend in friends | filter: {name: 'a'}">
		// filter – это стандартный фильтр отфильтрует элементы массива
		// по полю name каждого элемента массива (значение должно содержать букву 'a')
		Имя: {{ friend.name }}
		<br>
		Телефон: {{ friend.phone }}
	</li>
</ul>
```

Пример-3: **создание** фильтров для фильтрации

```
angular
	.module('module')
	/**
	 * функция принимает 2 аргумента
	 * первый аргумент  имя фильтра, а второй аргумент функция создания фильтра
	 * @param: { String } - имя модуля (имя должно быть уникальным в рамках приложения)
	 * @param: { Function } -  функция которая должна вернуть то что будет фильтром
	 * @returns: { Object } - объект модуля module
	 */
	.filter('firstFilter', function ( $log ){// зависимости (если нужно)
		//
		$log.debug('filter was created');
		// возвращаем функцию фильтрации
		return function ( items, count ) {
			// вернем первые элементы
			return items.slice(0, count||3);
		};
	});
```

пример использования созданного фильтра в шаблоне

```
// как мы можем понять фильтр вернет первые 5 элементов массива
<li data-ng-repeat="friend in friends | firstFilter:5 "> {{ friend.name }} </li>
```

Пример-4: **создание** фильтров для преобразования

```
angular
	.module('module')
	.filter('capitalize', function ( $log ){// зависимости (если нужно)
		$log.debug('filter was created');
		// возвращаем функцию преобразования
		return function ( string ) {
			// вернем преобразованную строку
			return string.replace( /^.{1,1}/, function ( sibling ) {
				return sibling.toUpperCase();
			});
		};
	});
```

пример использования созданного фильтра в шаблоне с предыдущим фильтром

```
// фильтр фильтрации, все так же, вернет первые 5 элементов массива
// а фильтр преобразования будет применен к каждой строке имени (сделает с заглавной)
<li data-ng-repeat="friend in friends | firstFilter:5 "> {{ friend.name | capitalize }} </li>
```

Так же можете почерпнуть дополнительной информации о создании пользовательских фильтров. Прелесть такого подхода в том что позволяет вынести логику сортировки и корректировки данных в представлении в отдельное место и без труда применять там где нужно, а так же корректировать сразу для всего приложения (где используется фильтр) в случае ошибок или изменения правил отображения.

directive:

Ранее мы с вами видели какие-то, на первый взгляд, непонятные атрибуты в html тегах. Они называются директивами. Мы использовали их в шаблонах не очень понимая «как» они устроены, из документации мы можем узнать о том какие директивы входят в базовую поставку, что с их помощью можно делать, зачем они нужны. Ангуляр в своей базовой поставке предоставляет достаточное большое количество директив. Для решения простых задач их более чем достаточно. Но хотелось бы обратить внимание, на то, что директивы могут быть не только атрибутами тегов, но и самими тегами и их можно создавать самостоятельно. Директивы позволяют создавать инкапсулированные компоненты. Которые могут реализовывать в себе как маленькие так и большие части представления. Абстракция директивы в Ангуляре позволяет связывать логические действия javaScript с html разметкой. В том числе и манипуляции элементами DOM(при помощи методов jQuery light или оригинального jQuery если он подключен к проекту). У директивы может быть свой $scope, свой контроллер, свои модели и сервисы. Это мощный инструмент angular.js. Соответственно приступать к созданию своих директив не стоит если вы только начали знакомится данным фреймворком. Для начала стоит освоить уже существующий функционал как из базовой поставки самого Ангуляра так и с популярными директивами сторонних разработчиков. Благо их много, и они покрывают своими возможностями очень приличный кусок потребностей веб разработки. Тем не менее, рано или поздно мы столкнемся с необходимостью создать свою уникальную директиву, для решения уникальной задачи приложения(да и вообще полезно понимать общие принципы работы директив).

Для начала, стоит обратить внимание на правила именования директив. (**пример-1**) При создании имя директивы указывается по правилам «верблюжий горб». В шаблонах используется это же имя, но слова пишутся через дефис. В случае если директива является атрибутом к атрибуту можно добавлять префикс «data-» этот префикс ненужно указывать в имени при создании директивы. Так же директивы с префиксом «ng-» считаются директивами базовой поставки, в связи с чем не стоит использовать этот префикс при создании директив. Так же как и в javaScipt стоит именовать директивы осмысленно и коротко, но не в ущерб ясности ее назначения.

Пример-1: **правила** именования директив

```
angular

	.module('module')
	// пример именования директивы при создании
	.directive('testExampleNaming',function () { // список зависимостей
		// код создания директивы
	});

// пример указания директивы в шаблоне как атрибут
<div data-test-example-naming=""> контент </div>

// пример указания директивы в шаблоне как тег
<test-example-naming> контент </test-example-naming>
```

Раз уж возникла необходимость создать свою директиву, стоит понять, что это мощный и многофункциональный инструмент. Тем не менее, ее стоит делать как можно более простой, что приведет к возможности ре использовать этот код. Давайте рассмотрим «как это работает ?» на примере.(**пример-2**) Создадим директиву, которая выполняет те же задачи что и ng-show, но наша директива будет тегом.

Пример-2: директива аналог ng-show

```
angular
	.module('module')
	/**
	 * функция принимает 2 аргумента
	 * первый аргумент  имя директивы, а второй аргумент функция создания директивы
	 * @param: { String } - имя модуля (имя должно быть уникальным в рамках приложения)
	 * @param: { Function } -  функция которая должна вернуть то что будет фильтром
	 * @returns: { Object } - объект модуля module
	 */
	.directive('customShow',function ( $log ) {
		$log.debug('directive was created');

		/**
		 * функция создания директивы должна вернуть объект или функцию
		 * если она вернет функцию, то это функция будет считаться функцией link
		 * как если бы мы возвращали объект
		 *  в данном примере мы вернем объект
		 */

		return {

			// шаблоны могут быть переданы как url или как шаблон(строкой)
			// templateUrl: 'template.html',
			// можно использовать только один варриант
			template: '<div style="display:{{show ? 'block': 'none'}}">\
				<ng-transclude></ng-transclude>\
			</div>',
			// указываем как мы планируем использовать директиву
			restrict: 'E', // используем как тег
			// при использовании директивы как тега указываем заменить
			// или дополнить наш нестандартный тег
			replace: true, // заменяем шаблоном
			// при использовании директивы как тега указываем
			// нужен ли его контент или его можно удалить
			// если контент не удаляется его можно делегировать с помощью
			// специальной директивы ng-transclude как в нашем случае
			// либо указать как зависимость $transclude
			// и обработать его содержимое при помощи javaScript
			transclude: true, // нужен
			// так же мы можем указать контроллер для этой директивы
			// имя контроллера или
			// функцию как если бы использовали метод создания у модуля
			// controller: function ( $scope ) {},
			// можем создать специальный объект $scope
			// более подробно тут, тут и тут
			scope: {
				show: '=?show'
			},
```

```
                        /**
                        * функция  link единственный обязательный параметр любой директивы
                        * единственное исключение она может быть заменена на compile
                        * (это еще более низкий уровень работы с шаблоном)
                        * в ней выполняется связывние шаблона с данными и выполняются любые
                        * дополнительные действия необходимые при инициализации директивы
                        */
                        link: function ( $scope, element, attrs ) {
                                /**
                                * исходя из нашей задачи нам нужно отображать или скрывать контент
                                * в зависимости от внешнего значения, мы будем его получать через
                                * дополнительный атрибут show нашего волшебного тега
                                */
                                $scope.watch('show', function ( newVal ) {
                                        // это не обязательно но для примера вызовем
                                        // перерисовку шаблона при изменении значения show
                                        $scope.evalAsync();
                                });
                        }
                };
        });
```

пример использования в шаблоне нашей старой новой директивы

```
<custom-show data-show="flagForShowing"> контент </custom-show>
```

Общие понятия:

JS — JavaScript язык програмирования по спецификации es5

Angular | Ангуляр — Библиотека angular.js версии 1.4.3. док

GIT | Гит — распределенная система управления версиями док

Node | Нода — программная платформа для выполнения JS за пределами браузера. док

NPM | НПМ — менеджер пакетов входящий в состав Node.js версии 3.10.5. док

Bower | Бовер — менеджер пакетов на базе НПМ оптимизированный для клиентской части док

Gulp | галп — Система потоковой сборки на Node.js док

build | билд — пре-процессинг кода — такой как — склеивание, минимизация, авто тесты, и т.д.

Ангуляр действительно позволяет создавать замечательные приложения. Но от “Hello world” до приложения в случае с «Ангуляром» целая пропасть. Так как Ангуляр добавляет целый каскад абстракций связанных друг с другом и этим влияет на построение структуры проекта и билд.

Задачи:

Что требуется от каждого web проекта вне зависимости от технологий которые выбраны для реализации. Конечно же это скорость работы, масштабируемость, версионный контроль, автотесты, и качество исходного кода.

Скорость — ускорить приложение помогают склеивание файлов в идеале это один-два файла js и css на всё приложение, затем минимизация этих файлов. Естественно с преобразованным подобным образом кодом продолжать разработку невозможно. Для этого следует настроить билд процесс таким образом что бы можно было в любой момент «сбилдить» ваш проект в конечную оптимизированную версию (**prodaction**)

Масштабируемость — простоту расширения текущего функционала может предоставить ангуляр и целый каскад разработанных на его основе плагинов (директив) как от разработчиков самого ангуляра так и от сторонних разработчиков в том числе и ваших.

версионный контроль — система контроля версий GIT помогает решать такие вопросы

автотесты —

качество кода — качество вашего кода на прямую зависит от понимания базовых абстракций ангуляра таких как модуль, фабрика, сервис, дирректива и другие.

Подготовка:

Хорошая практикой считается иметь документ с описанием действий необходимых для запуска проекта на пример в таком виде.

```
// download project
git pull  https://gitSome.com/someUser/someProject.git

// get a devDependencies
npm install

// get a dependencies
bower install

// set a configuration

// start command to launch a project instance in browser
gulp serve-dev
```

В большинстве случаев такой последовательности действий достаточно для того что бы запустить web проект на любом компьютере. В случае правильной подготовки проекта что как и почему происходит вы даже и не спросите — но это если все работает.

Подробнее:

1. git pull — либо любое приложение работающее с серверами гит скачивает проект(либо его ветку) из репозитория. В репозиториях никогда не храниться (убить и сжечь если не так) сопутствующие пакеты необходимые для процессинга (билда) — хранятся только конфигурационные файлы.

2. npm install — скачивает и устанавливает пакеты необходимые для процессинга (билда) с учетом версий описанных в конфигурационном файле **package.json**.

3. bower install — (**bower** уже установленный npm) скачивает библиотеки необходимые для проекта с учетом версий описанных в конфигурационном файле **bower.json** .

4. конфигурация проекта — в большинстве случаев проект нуждается в дополнительной настройке согласно вашей системы (windows / linux /mac), расположения серверов для разработки, ключей доступа к сторонним серверам и т. д.. С этой целью создаются файлы конфигурации приложения в нескольких вариантах наиболее часто можно встретить prod – для prodaction версии и dev – для разработчика.

5. gulp — (**gulp** уже установленный npm) запускает процессы слежения за изменениями файлов, локальные сервера (если есть необходимость), препроцессинги JS типа **typescript, coffeescript** или css типа **sass, less, stylus**, минимизация и оптимизация проекта для **production**.

Задачи «build» процесса:

Build production происходит по следующей схеме:

1. Происходит замена файла конфигурации согласно типа сборки в данном примере это **prod**
2. Выполняются препроцессинги для css библиотек. Складываются в папку .temp как vendor.css
3. Склеиваются минимизированные версии JS библиотек проекта. Складываются в папку .temp как vendor.js
4. Выполняются препроцессинги — sass, less, stylus и тому подобных. Затем склеиваются и складываются в папку .temp. как style.css
5. Выполняется проверка на «code style» как правило это **jshint** док
6. Выполняются авто тесты
7. Выполняются препроцессинги — преобразования файлов из typescript, coffeescript и тому подобных. Затем склеиваются и складываются в папку .temp. как scripts.js
8. Собираются все html файлы преобразовываются в JS. Склеиваются и складываются в папку .temp как templatecash.js
9. Минимизируются файлы css и перекладываются в dist как vendor.css(библиотеки) и style.css(стили проекта).
10. Минимизируются файлы js и перекладываются в dist как vendor.js(библиотеки) и scripts.js все js файлы приложения - в случае с англяром сначала склеиваются scripts.js и templatecash.js затем анотируються с помощью **ng-annotate** и только потом минимизируются и складываются в папку dist.
11. Минимизируются картинки и складываются сразу в папку dist.
12. Копируются шрифты сразу в папку dist.
13. Копируется **index.html** – согласно разметке по комментариям (**аккуратнее с комментариями в index.html**) заменяются подключенные скрипты и стили на вновь сформированные файлы vendor.css, style.css, vendor.js и scripts.js.
14. Ко всем именам файлов проекта дописывается cash(соль) такая же соль дописывается во всех указателях ресурсов в файлах в том числе и index.html.
15. Последним минимизируется **index.html**.

Для разработки конечно же выполнять все эти действия смысла нет, поэтому варианты билда для **dev** отличаются. Создание еще одного проекта с минимизированным кодом заменяется на «wathcer ы» изменений файлов, для запуска соответствующих препроцессоров. И проект запускается с JS и css файлами без минимизации.

Ангуляр в силу своих особенностей так же нуждается в дополнительных модулях процессинга на которых хотелось бы остановиться подробнее.

ng-annotate:

Перед тем как наш код будет минимизирован стоит отметить особенность указания зависимостей ангуляра.

пример:

```
angular
	.module('module')
	.controller('Controller', function ( $scope, $timeout ) {

	});
```

В данном коде мы указываем зависимости контроллера. После минимизации имена аргументов функции будут минимизированы $scope — a, $timeout – b. Еще один подход указания зависимостей ангуляр более явный для кода и менее читаемый. Преобразуем наш код для примера.

преобразованный пример:

```
angular
	.module('module')
	.controller('Controller', [ '$scope', '$timeout',
		function ( $scope, $timeout ) {

	}]);
```

ng-annotate док – попытается сделает это за нас. Но иногда приходиться преобразовывать самостоятельно.

templatecash:

Благодаря встроенному в ангуляр шаблонизатору появилась возможность инъецировать html шаблоны в JS файлы. Если потратить немного времени и разобраться с $templateCash док то можно понять что это легко, просто, надежно и быстро (быстрее чем каждый раз загружать шаблоны аяксом) возникает вопрос как это автоматизировать — вы же не хотите все шаблоны в js файлах писать =). **Gulp-ng-template –** нам поможет. И добавит еще один повод **не пихать лишних тегов** в шаблоны.

Итоги по «build» процессу:

Если у вас правильно настроенный **build** то скорее всего вы даже не узнаете нюансы его реализации и на выходе заказчик получит быстрый масштабируемый и конфигурируемый web проект. Который можно будет настроить на любой сервер запускать из любой обертки да и просто таскать где и как угодно. А вам как разработчику не придется сидеть ночи на пролет клепая хотфиксы «перед» или «в» день релиза или обновления продакшена. Если здесь вы не почерпнули ничего нового можете попробовать создать свой генератор проектов с помощью **yeoman** док.

DIR:

Папка с проектом выглядит примерно так. **такие** файлы и папки **не комитят !**

- **Root dir => mYapp**
  - **app**
    - **assets**
      - **images**
      - **fonts**
    - **styles**
      - **less** (либо другой препроцессор)
      - **all.less** (либо другой препроцессор)
      - **all.css**
    - **scripts**
      - **directives**
      - **filters**
      - **services**
      - **states** (состояния или страници приложения)
      - **app.js** (основной модуль)
      - **config.js** (переменные окружения)
    - **index.html**
  - **environment**
    - **development.json**
    - **production.json**
    - **config.template.js**
  - **node_modules**
  - **bower_components**
  - **tests**
  - **dist**
  - **.tmp**
  - **.gitignore**
  - **.npmignore**
  - **bower.json**
  - **package.json**
  - **gulpfile.js**

Архитектура Angular приложения:

С помощью Ангуляр в рамках «angular way» можно построить разные архитектура. Разрешите поделиться некоторыми соображениями по этому поводу.

Модули:

Одна из основных абстракций Ангуляра это **angular.module.** Все приложение состоит из модулей. И начинается все с создания основного модуля от которого будет строиться приложение и указателя html элемента с которого оно начнет строиться в DOM.

```
// создания от получения отличается только наличием/отсутствием 2го аргумента
var app = angular.module('appName', ['dependency1','dependency2']);
app === angular.module('appName');
```

Можно провести аналогию как если бы писали «jQuery way» приложение и создали глобальный объект в window для ограничения пространства имен. С той лишь разницей что модуль имеет более сложную структуру специально подготовленную для внедрения зависимостей и инициализации.

Далее мы указываем список модулей(частей) нашего приложения. Как правило это модуль `cammon/util` в котором в качестве зависимостей указаны модули/сервисы/фабрики/директивы общего назначения как написанные нами так и заимствованные (из других наших приложений или разработанные сторонними разработчиками).

Используя библиотеку **ui-router** док предоставляющую возможность удобного описания маршрутизации проекта подключаются модули обработки маршрутов (states) отвечающие за разные части представления. Архитектуру зависимостей стараются максимально повторить в структуре каталогов папки скрипт. Давайте более подробно рассмотрим вариант папки **scripts**.

DIR:

- **Root dir => mYapp => app => scripts**
  - **services** (реализуют запросы к разным частям api)
  - **models** (модели приложения)
  - **directives** (сложные елементы представления)
  - **interceptors** (прослушивание всех запросов как пример ошибки авторизации)
  - **constants** (глобальные переменные в рамках приложения)
  - **states** (состояния/страници/маршруты приложения)
    - **home** (представление по умолчанию **otherwise** и **redirect**)
      - **home.html**
      - **home.controller.js**
      - **home.module.js**
    - **event** (представление элемента навигации)
      - **event-create** (дочерний элемент представления)
        - **event-create.html**
        - **event-create.controller.js**
        - **event-create.module.js** (декларация зависимостей config & run)
      - **event-update** (дочерний элемент представления)
        - **event-update.html**
        - **event-update.controller.js**
        - **event-update.module.js** (декларация зависимостей config & run)
      - **event-show**
        - **event-show-spec.html**
        - **event-show-spec.directive.js** (специфическая личная директива для show)
        - **event-show.html**
        - **event-show.controller.js**
        - **event-show.module.js** (декларация зависимостей config & run)
      - **event-date.service.js** (специфический личный сервис для event)
      - **event.html**
      - **event.controller.js**
      - **event.module.js** (декларация зависимостей config & run)
    - **layout.html** (описание основного layout приложения)
    - **layout.module.js**
    - **layout.controller.js**
  - **config.js** (файл с конфигурацией приложения заменяется согласно **gulp local/dev/prod**)
  - **app.js** (декларация зависимостей config & run)

Сервисы:

Сервис выступает в Ангуляре как поставщики функционала. С таким описанием в него можно завернуть все что угодно. Поэтому давайте разберемся какого рода функционал нуждается в таких обертках. Во первых это касается сторонних библиотек (пример-1) и AJAX запросов (пример-2) конечно в сервисы можно заворачивать что угодно. Для создания сервисов используют **angular.module('some').service** или **angular.module('some').factory** в принципе своей работы они очень похожи. Оба выполняются один раз при загрузке приложения и дальше во всех местах вызова (как зависимостей модуля) будут возвращать результат функции.

Пример-1: сервис для поставки библиотеки moment.js

```
angular
	.module('module')
	.service('moment', function ( $log, $window ) {
		// если библиотека не подключена или еще не инициализирована
		if ( typeof $window.moment == 'undefined' ) {
			// сервис выведет не блокирующую ошибку
			$log.error( 'moment.js is undefined', $window.moment );
		}
		return $window.moment;
	});
```

Пример-2: сервис для поставки функционала работы с сервером касательно событий.

```
angular
	.module('module')
	.service('eventService', function ( $q, Restangular, config ) {

		// инициализация сервиса
		var baseApiPath =  config.apiPath || 'api/1';

		return {

			getEvent: function ( eventId ) {
				/**
				 * GET: baseApiPath/event/:eventId
				 * get by id data event from server
				 * @public
				 * @param: { Number } - id of event
				 * @returns: { Object } - promise
				 */
			},

			createEvent: function ( event ) {
				/**
				 * POST: baseApiPath/event
				 * create an event on the server
				 * @public
				 * @param: { Object } - data of event
				 * @returns: { Object } - promise
				 */
			},
```

```
                updateEvent: function ( eventId, update ) {
                        /**
                         * PUT: baseApiPath/event/:eventId
                         * update event by id on the server
                         * @public
                         * @param: { Number } - id of event
                         * @param: { Object } - data of event
                         * @returns: { Object } - promise
                         */
                },
                deleteEvent: function ( eventId ) {
                        /**
                         * DELETE: baseApiPath/event/:eventId
                         * delete event by id on the server
                         * @public
                         * @param: { Number } - id of event
                         * @returns: { Object } - promise
                         */
                }
        };
});
```

И что дальше с ними делать ? - Все так же использовать в качестве зависимостей другого модуля приложения. На пример нам нужно реализовать контроллер с возможностью работать с датами и управлять событиями. Как раз и пригодятся оба наших сервиса (пример-3).

Пример-3: получение и использование зависимостей

```
angular
        .module('module')
        .controller('Controller', function ( eventService, moment ) {
                // функционал предоставленный нашим сервисом событий
                eventService.getEvent
                eventService.createEvent
                eventService.updateEvent
                eventService.deleteEvent
                // функционал предоставленный библиотекой moment.js
                moment - док
        });
```

По схожему примеру сервисы могут добавить в ваше приложение некий функционал. Но так же как и с именованием переменных хорошей практикой является правильное именование сервисов например если бы наш eventService обладал бы методами типа «logout» или «updateUserGeneralPhoto» - это по меньшей мере было бы «неожиданно» если не сказать больше. Таким образом настоятельно рекомендуется ограничивать функционал предоставляемый сервисом исключительно юрисдикцией этого сервиса и конечно именовать соответственно.

Модели:

Вопреки популярному мнению - «ангуляру не нужны модели» или «зачем все так усложнять» в ангуляре предусмотрен метод **.factory** как раз для создания классов (*class/constructor*) который на подобие **.service** Инициализируеться при старте приложения и возвращает результат работы но у нее есть особенность — после инициализации у результата работы функции (собственно наша будущая зависимость) остаеться не тронутым прототип (*prototype*), в отличии от других модулей ангуляра, благодаря чему **.factory** (пример-4) отлично подходит для создания конструкторов и инстанциации (*instance/model*) их в любой части приложения. Соответственно именование конструктора стоит начинать с «большой» буквы.

Пример-4: модель события(event) использующая сервис работы с событиями сервера

```
angular
	.module('module')
		.factory('EventModel', function ( eventService, $q ) {
			// приватный метод валидации события
			function validate ( event ) {
				/** validate event
				 * @privat
				 * @param: { Object } - event data
				 * @returns: { Boolean }
				 */
			}
			/** Event
			 * @publick
			 * @param: { Object } - event data
			 * @constructor
			 */
			function Event ( event ) {
				// расширяем заготовку копией поступивших данных
				angular.extend(this, angular.copy( event ));
			}
			Event.prototype = {
				constructor: Event,
				/** update
				 * @public
				 * @param: { Object } - data of event
				 * @returns: { Object } - promise
				 */
				update: function ( data ) {
					if ( vlidate( data ) ) {
						if ( // from server ) {
					return eventService.updateEvent( this.id, data );
						} else {
					return eventService.createEvent( data );
						}
					}
				}
			};
			// далее как зависимость будет приходить именно конструктор
			// с встроеным методом обновления
			return Event;
		});
```

Состояние(state):

Давайте чуть подробнее рассмотрим использование состояний которое предоставляет **ui-router**. Состояние (state) — абстракция предоставляемая **ui-router** с помощью которой мы можем по данным хранящимся в **url** попытаться восстановить состояние приложения для этого **url.** Таким образом необходима инициализировать модули соответствующие состоянию. Для этого создается(указывается) имя состояния и только потом описывается какой вункционал и какое представление ему соответствует. Звучит сложно … В коде гораздо проще =). Давайте рассмотрим пример состояния приложения **(пример-5)** для нашего события (event).

Все начинается как вы уже догадались с создания модуля **(angular.module)** если есть зависимости указываем их в массиве. Затем на стадии конфигурации **(все модули angular во время инициализации проходят 2 стадии 1-конфигурации(config) и 2-запуска(run))** указывается зависимость $stateProvider предоставляемый **ui-router** которому и передаем объект с описание состояния к которому относить этот модуль. В объект конфигурации указываем имя состояния образец **(pattern) url** к которому его относить. Шаблон который показать контроллер который будет подготавливать данные для этого шаблона и список действий **(в том числе и асинхронных результаты которых можно получить в контроллере указав их как зависимости контроллера)**

Пример-5: описание простого состояния приложения

```
angular

	.module('app.event', [])

	.config( function ( $stateProvider ) {

		$stateProvider.state('event', { // имя состояния
			url: '/event/:eventId', // pattern
			controller: 'EventController', // имя контроллера
			templateUrl: 'event.html', // путь к шаблону
			resolve: { // список действий
				// имя действия по которому можно указывать зависимости
				eventData: function ( $stateParams, eventService ) {
		// используя наш сервис который возвращает промис
		// и указав зависимость в контроллере
		// ui-router инициализирует контроллер
		// только после получения ответа с сервера
					return eventService.getEvent($stateParams.eventId);
				}
		// $stateParams - тоже модуль ui-router с данными выбранными из url
			}
		});

	});
```

Таким образом мы подготовили модуль который будет инициализироваться если **url** будет соответствовать образцу **(pattern)** и клиент увидит восстановленное по **url** состояние приложения. Пример **url** которому будет соответствовать этот паттерн `https://myApp.com/#/event/7504` либо любой другой `eventId`.

Контроллеры:

Контроллеры позволяют управлять «двухсторонним датабиндингом» в ангуляре. Как это происходит ?

Все так же мы указываем зависимости контроллера это могут быть сервисы для обращения к серверу за данными, могут быть экземпляры моделей или уже полученные заранее данные. С помощью **ui-router** мы можем удобно и просто указать как зависимости контроллера - выполнение запросов к серверу либо любую другую предварительную обработку данных таким образом что в самом контроллере мы не будем отвлекаться на нюансы получения данных благодаря чему при их описании почти все внимание посвящается вопросу «**в как ом виде**» данные попадают в html для представления.

Пример-6-1: использование зависимостей для формирования представления

```
angular
	.module('module')
	.controller('EventController',
		function ( $scope, eventData, EventModel, moment ) {
			// $scope – встроенный в ангуляр объект
			// он переносит данные для шаблона этого контроллера
			var vm = $scope.vm = {
				// предобработка
				event: new  EventModel( eventData )
			};
			/*

				в рамках шаблона этого контроллера
				нам будет доступен объект vm.event
				со всеми свойствами пришедшими с сервера
				и методом vm.event.update() добавленным моделью
			*/
	});
```

Пример-6-2: внутри html шаблона

```
<div id="event-state-body" class="container">

	<p> {{ vm.event.name }} </p>

	<input type="text"  class="event-name" ng-model="vm.event.name">

	<button type="button"  ng-click="vm.event.update()" > Update  </button>

</div>
```

Зачем так много кода для того что можно сделать двумя **ajax** запросами **jquery** и двумя вызовами метода **.html('...')** того же **jquery** ? Скорее всего потому что подобный подход позволяет дополнительно обрабатывать и расширять функционал приложения на любой стадии и теоретически в неограниченном количестве не теряя ясности кода и общих логических концептов, до пока событие (**event**) является частью серверного **api** и вашего приложения.