Contacts Application

Table of Contents

[Project Overview 2](#_Toc520379109)

[Introduction 2](#_Toc520379110)

[SiteMap 2](#_Toc520379111)

[Project Architecture 5](#_Toc520379112)

[Folder Structure 5](#_Toc520379113)

[UI Architecture 6](#_Toc520379114)

[Sanitize HTML 7](#_Toc520379115)

[UI Validation (phone number validation code snippet) 7](#_Toc520379116)

[Typical JSModel 7](#_Toc520379117)

[Server Structure 8](#_Toc520379118)

[Mock Object Framework for Unit Testing 9](#_Toc520379121)

[Assertion Methods 9](#_Toc520379122)

[Sample Expectations 9](#_Toc520379123)

[DataBase 10](#_Toc520379124)

[SP spGetContactDetails 10](#_Toc520379125)

**PREPARED BY:**

Mini Thomas

# Project Overview

## Introduction

Contacts application is for maintaining contact information.This includes features like:

* List Contacts,
* Add Contact,
* Edit Contact and
* Delete/Inactivate Contact

**Technology and Tools**

* AngularJS
* .NET WEB API with Visual C# 2017
* ADO.Net Model
* SQL 2012
* Moq.Tests

**Architectural patterns and coding practices:**

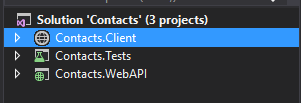
Repository pattern and Dependency Injection pattern, test-driven development, SOLID priniciples, OWASP10 security practices, AngularJS: API: $sanitize and Stored Procedures.

## SiteMap

|  |  |
| --- | --- |
| **Page** | **Screen** |
| *List Contacts*  *(Default)* |  |
| *List Contacts*  *(Filter by Name)* |  |
| [*Add*](https://www.tutorialspoint.com/backbonejs/event_trigger.htm) *Contact*  *(Validation)* |  |
| [*Add*](https://www.tutorialspoint.com/backbonejs/event_listento.htm) *Contact*  *(Save)* |  |
| Edit Contact |  |
| *Delete Contact*  *(Before clicking on Delete)* |  |
| *Delete Contact*  *(After clicking on Delete)* |  |

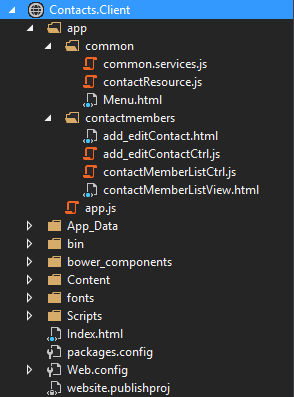
# Project Architecture

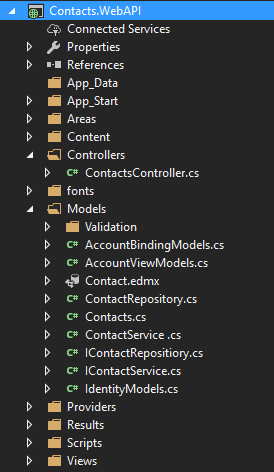
## Folder Structure



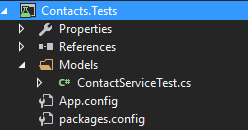
There are 3 projects Contacts.Client, Contacts.WebAPI and Contacts.Tests

Run the application by setting Contacts.Client website as startup project.





Below is the Unit Test project that uses the test-driven development approach.



## UI Architecture

Page

View

Controller

***Controller:*** The Controller represents the layer that has the business logic. User events trigger the functions which are stored inside your controller. The user events are part of the controller.

***Views:*** View contains HTML templates which can be loaded and transformed, and rendered as the view. A View is a visual representation of its Model that represented the current state of a Model.

We are creating a model binding using the ng-model directive.

## Sanitize HTML

The ngSanitize module provides functionality to sanitize HTML.

Load the module in application by adding it as a dependent module:

|  |
| --- |
| var app = angular.module("contact", ["ODataResources", "ngRoute", "ngSanitize"]). |

Use ng-bind-html in your Html:

|  |
| --- |
| <td ng-bind-html="contact.firstName"></td> |

## UI Validation (phone number validation code snippet)

|  |
| --- |
| <label class="col-sm-2 col-form-label" for="phoneNumber">Phone Number:</label>  <input id="phoneNumber"  type="text"  name="phoneNumber"  data-ng-model="add\_editContact.contact[0].phoneNumber"  ng-minlength="10"  ng-maxlength="10"  placeholder="Phone"  ng-pattern="/^[\+]?[(]?[0-9]{3}[)]?[-\s\.]?[0-9]{3}[-\s\.]?[0-9]{4,6}$/"  required />  <div class="error-body" ng-show="contactForm.phoneNumber.$invalid">  <div ng-message="required">Please enter valid 10 digit phone number</div>  </div> |

## Typical JSModel

|  |
| --- |
| [  {  "contactID": 1,  "firstName": "<b>Mini</b>",  "lastName": "<b>Thomas</b>",  "email": "mini.ceepi@gmail.com",  "phoneNumber": "9822821987",  "status": "Active"  },  {  "contactID": 16,  "firstName": "Mike",  "lastName": "Molly",  "email": "mike@molly.com",  "phoneNumber": "1234567890",  "status": "Active"  }  ] |

## Server Structure

## 

UI (AngularJS)

()

Controller

Service Layer

(Validation Logic)

Repository

Database

ADO.Net Model

* Browser will send request in Http/JSon to Web API which is having controller.
* Our application is more loosely coupled. It uses the Repository pattern and the Dependency Injection pattern.
* We will take advantage of a software design pattern called Dependency Injection to use the Repository class in our controller. The Contacts controller class uses Constructor Dependency Injection.
* The service layer is a separate layer that we can insert between our controller and repository classes. The service layer contains our business logic including all of our validation logic. Each layer is isolated as much as possible from other layers.
* Our application runs on the Single Responsibility Principle (SRP). The Contact controller is responsible only for controlling the flow of application execution. All the validation logic has been pushed into the service layer. All of the database logic has been pushed into the repository layer.
* When we created the service layer, we took advantage of the Decorator pattern to isolate ModelState from our service layer. In our service layer, we programmed against the IValidationDictionary interface instead of ModelState.

UI (AngularJS)

()

Controller

IValidationDictionary

IContactService

ContactService

IContactRepositiory

# 

ContactRepositiory

# Mock Object Framework for Unit Testing

A Mock Object framework enables you to create mocks and stubs for the classes in your application.

Moq enables us in creating a set of unit tests for our Contacts application service layer. We'll use these tests to verify our validation logic.

In our unit tests, we instantiate the ContactService by passing a mock implementation of the IContactRepository class.

## Assertion Methods

* *SaveContactDetails() - Tests that SaveContactDetails() returns the value true when a valid Contact is passed to the method.*
* *SaveContactRequiredFirstName() - Tests that an error message is added to model state when a Contact with a missing first name is passed to the SaveContactDetails() method.*
* *SaveContactRequiredLastName() - Tests that an error message is added to model state when a Contact with a missing last name is passed to the SaveContactDetails() method.*
* *SaveContactInvalidPhone() - Tests that an error message is added to model state when a Contact with an invalid phone number is passed to the SaveContactDetails() method.*
* *SaveContactInvalidEmail() - Tests that an error message is added to model state when a Contact with an invalid email address is passed to the SaveContactDetails() method.*

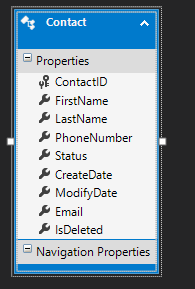
## Sample Expectations

|  |
| --- |
| [TestMethod]  public void SaveContactInvalidEmail()  {  // Arrange  int ID = 0;  var contact = new Contact.Models.Contacts();  contact.ContactID = 1;  contact.FirstName = "Mini";  contact.LastName = "Thomas";  contact.PhoneNumber = "1234567890";  contact.Email = "test";  // Act  var result = \_service.SaveContactDetails(ID, contact);  // Assert  Assert.IsFalse(result);  var error = \_modelState["Email"].Errors[0];  Assert.AreEqual("Invalid email address.", error.ErrorMessage);  } |

# DataBase

There is 1 table and 3 stored procedures as below:

* Table: Contact
* SP: spDeleteContact, spGetContactDetails and spSaveContact



## SP spGetContactDetails

|  |
| --- |
| USE [Contact]  GO  /\*\*\*\*\*\* Object: StoredProcedure [dbo].[spGetContactDetails] Script Date: 26-07-2018 14:32:42 \*\*\*\*\*\*/  SET ANSI\_NULLS ON  GO  SET QUOTED\_IDENTIFIER ON  GO  CREATE PROCEDURE [dbo].[spGetContactDetails] @ContactID int  AS  SET NOCOUNT ON  IF (@ContactID=0)  BEGIN  SELECT [ContactID]  ,[FirstName]  ,[LastName]  ,[PhoneNumber]  ,[Email]  ,[Status]  ,[CreateDate]  ,[ModifyDate]  FROM [dbo].[Contact]  where [IsDeleted]=0  END  ELSE  BEGIN  SELECT [ContactID]  ,[FirstName]  ,[LastName]  ,[PhoneNumber]  ,[Email]  ,[Status]  ,[CreateDate]  ,[ModifyDate]  FROM [dbo].[Contact]  WHERE [ContactID]=@ContactID  END  RETURN  GO |

