Building Single Page Applications

using Web API and AngularJS

By Chris Sakellarios

chsakell's Blog

ANYTHING AROUND ASP.NET MVC, WEB API, WCF, Entity Framework & C#

Building Single Page Applications

using Web API and AngularJS

By

Chris Sakellarios

chsakell's Blog

ANYTHING AROUND

ASP.NET MVC, WEB API, WCF, Entity Framework & C#

Table of contents

Contents

Introduction

Single Page Applications are getting more and more attractive nowadays for two basic reasons. Website users have always preferred a fluid user experience than one with page reloads and the incredible growth of several JavaScript frameworks such as **angularJS**. This growth in conjunction with all the powerful server side frameworks makes Single Page Application development a piece of cake. This book is the pdf version of the online post in *chsakell's Blog* and describes step by step how to build a production-level SPA using ASP.NET Web API 2 and angularJS. There are a lot of stuff to build in this application so I will break the book in the following basic sections:

1. **What we are going to build**: Describe the purpose and the requirements of our SPA
2. **What we are going to use**: The list of technologies and all server and front-end side libraries and frameworks
3. **SPA architecture**: The entire design of our SPA application from the lowest to the highest level
4. **Domain Entities and Data repositories**: Build the required Domain Entities and Data repositories using the generic repository pattern
5. **Membership**: Create a custom authentication mechanism which will be used for Basic Authentication through Web API
6. **Single Page Application**: Start building the core SPA components step by step
7. **Discussion**: We 'll discuss the choices we made in the development process and everything you need to know to scale up the SPA

Who is this book for

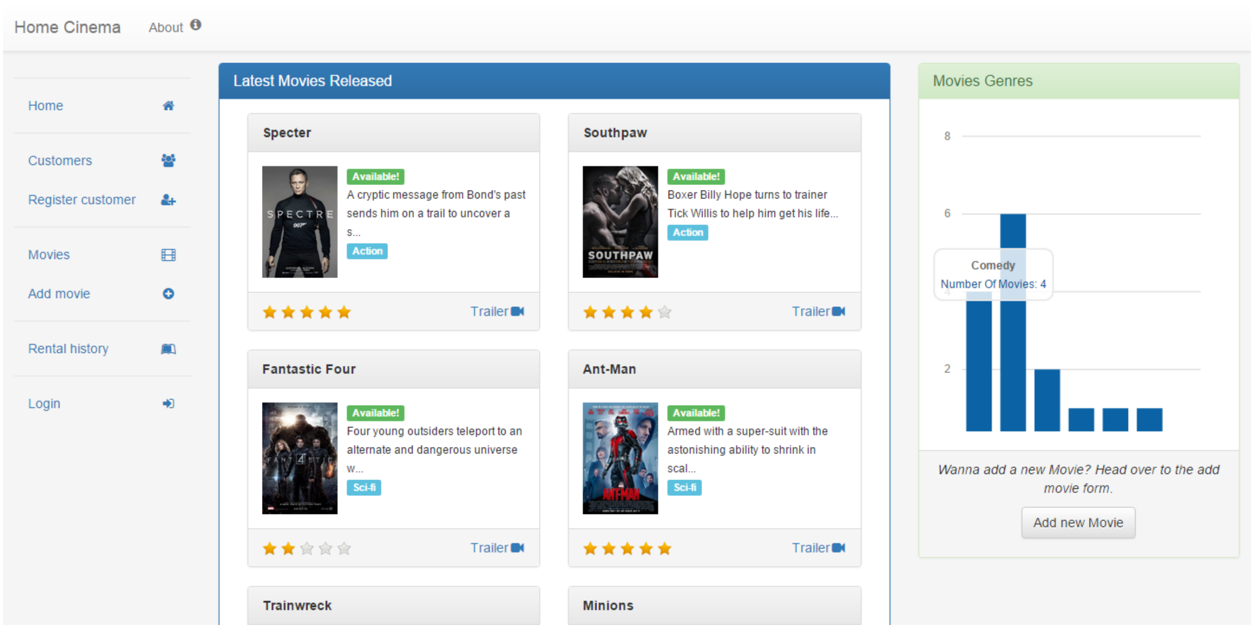
This book is for .NET Web devopers who want to learn how to build *Single Page Applications* using ASP.NET Web API and angularJS. Basic knowledge for both of these frameworks is assumed but not required. The book describes step by step how to implement an *SPA* web application regarding the test case of a *Video Club.* This means that if you came here to learn how an MVC message handler works behind the scenes you probably are on the wrong road. If however, you want to learn how to build a real, production level *Single Page Application* from scratch, by integrating an amazing server-side framework such as *Web API* and a spectacular *JavaScript* framework such as *angularJS* then this is your book you were looking for.

What we are going to build

We are going to build a **Singe Page Application** to support the requirements of a Video Rental store that is a store that customers visit, pick and rent DVDs. Days later they come back and return what they have borrowed. This Web application is supposed to be used only by the rental store's employees and that's a requirement that will affect mostly the front-end application's architecture. Let's see the requirements along with their respective screenshots:

Requirement 1:

1. Latest DVD movies released added to the system must be displayed
2. For each DVD, relevant information and functionality must be available, such as display *availability*, YouTube trailer and its rating
3. On the right side of the page, genre statistics are being displayed
4. This page should be accessible to unauthenticated users



A cryptic message from Bond's past sends him on a trail to uncover a s...

Boxer Billy Hope turns to trainer Tick Willis to help him get his life...

Four young outsiders teleport to an alternate and dangerous universe w...

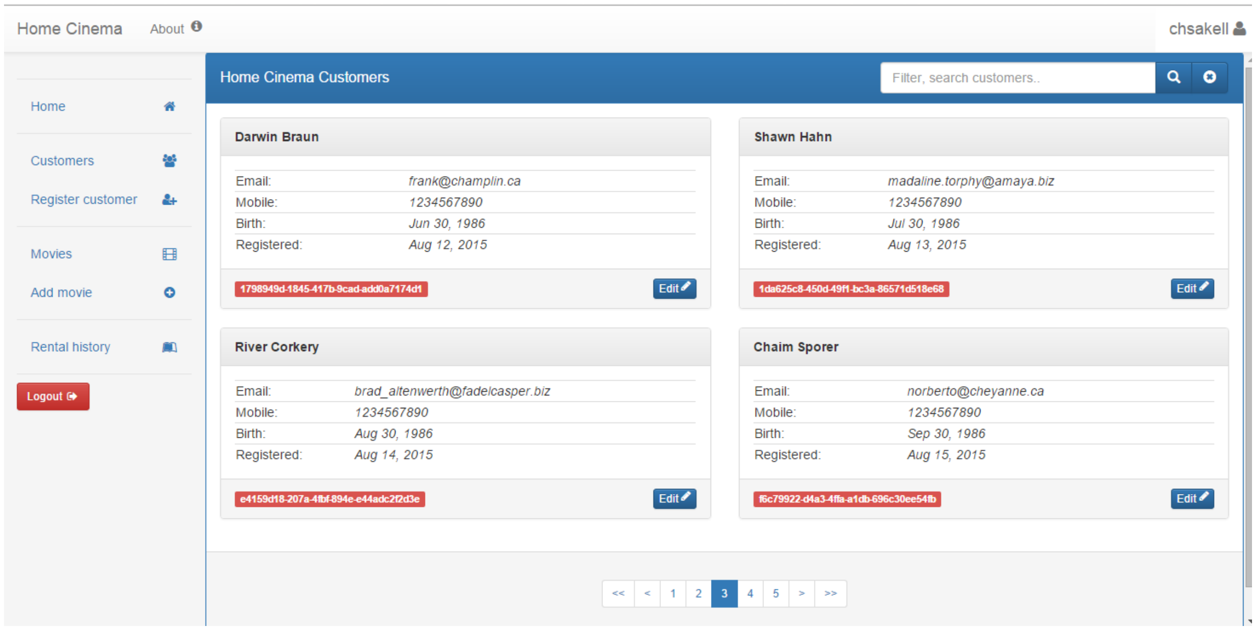
Armed with a super-suit with the astonishing ability to shrink in scal...

Wanna add a new Movie? Head over to the add movie form.

Figure 1. Home page

Requirement 2 - Customers:

1. There will be 2 pages related to customers. One to view and edit them and another for registration
2. Both of the pages must be accessible only to authenticated users
3. The page where all customers are being displayed should use pagination for faster results. A search textbox must be able to filter the already displayed customers and start a new server side search as well
4. Customer information should be editable in the same view through a modal popup window


Home Cinema · About · chsakell

Home · Customers · Register customer · Movies · Add movie · Rental history · Logout

Home Cinema Customers — Filter, search customers..

Darwin Braun

Email:	*frank@champlin.ca*
Mobile:	*1234567890*
Birth:	*Jun 30, 1986*
Registered:	*Aug 12, 2015*

1798949d-1845-417b-9cad-add0a7174d1 · Edit

Shawn Hahn

Email:	*madaline.torphy@amaya.biz*
Mobile:	*1234567890*
Birth:	*Jul 30, 1986*
Registered:	*Aug 13, 2015*

1da625c8-450d-49f1-bc3a-86571d518e68 · Edit

River Corkery

Email:	*brad_altenwerth@fadelcasper.biz*
Mobile:	*1234567890*
Birth:	*Aug 30, 1986*
Registered:	*Aug 14, 2015*

e4159d18-207a-4fbf-894e-e44adc2f2d3e · Edit

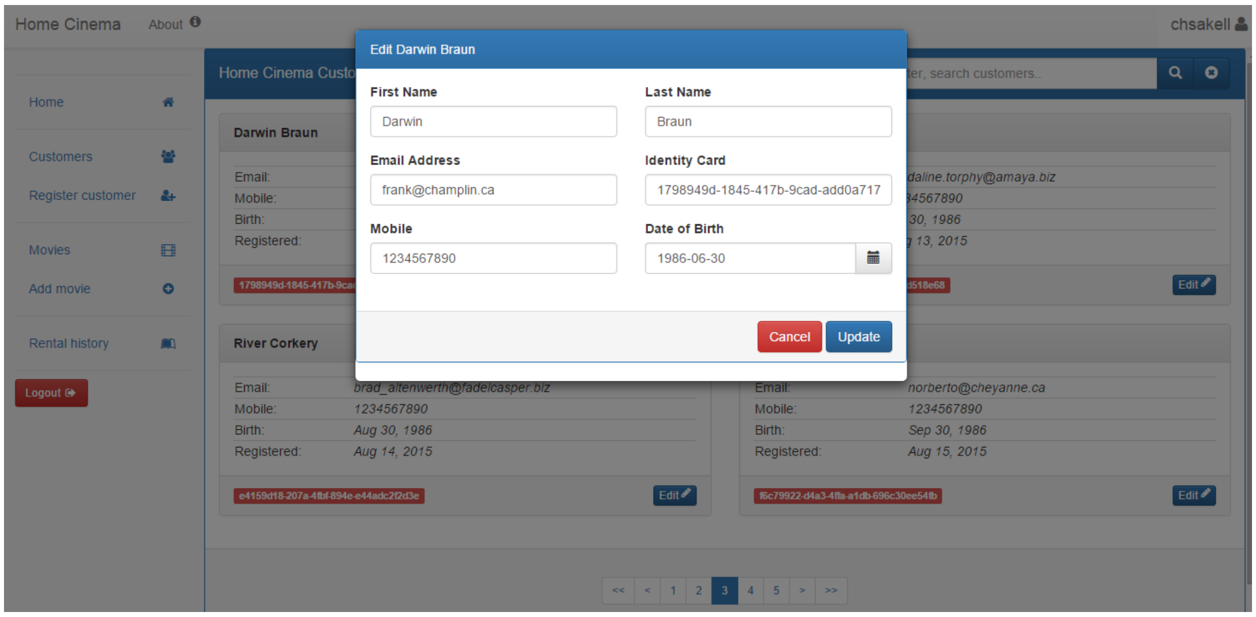
Chaim Sporer

Email:	*norberto@cheyanne.ca*
Mobile:	*1234567890*
Birth:	*Sep 30, 1986*
Registered:	*Aug 15, 2015*

f6c79922-d4a3-4ffa-a1db-696c30ee54fb · Edit

<< < 1 2 3 4 5 > >>

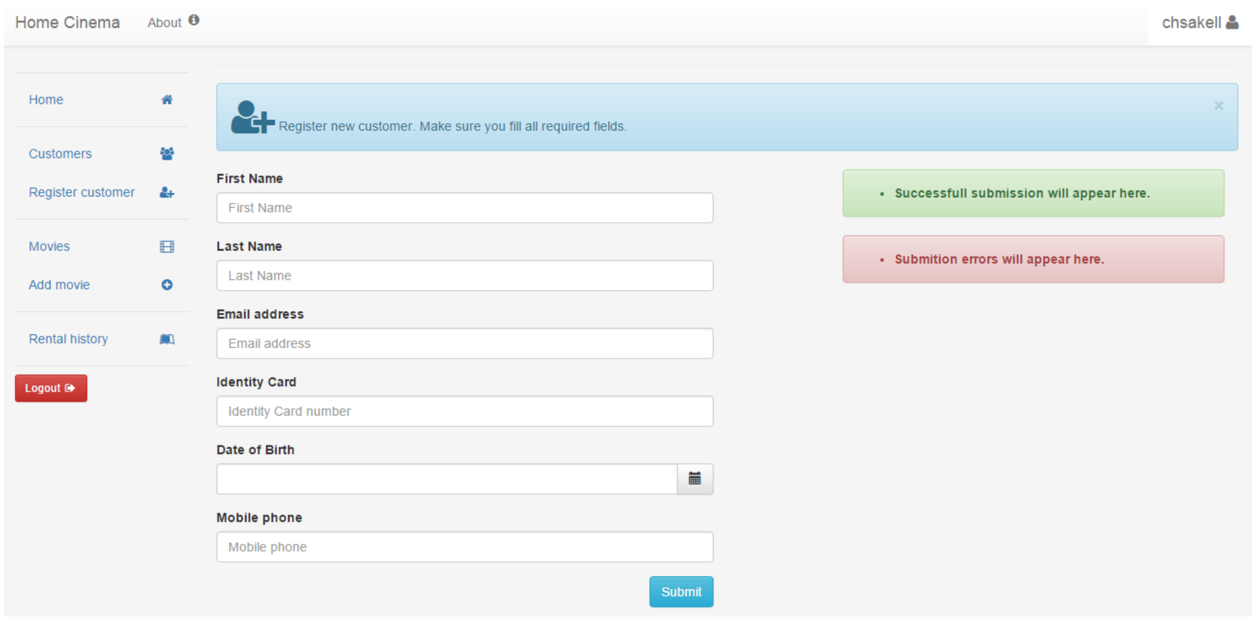
Figure 2. Customers view


Edit Darwin Braun

First Name	**Last Name**
Darwin	Braun
Email Address	**Identity Card**
frank@champlin.ca	1798949d-1845-417b-9cad-add0a717
Mobile	**Date of Birth**
1234567890	1986-06-30

Cancel · Update

Figure 3. Edit Customer popup



Register new customer. Make sure you fill all required fields.

First Name

Last Name

Email address

Identity Card

Date of Birth

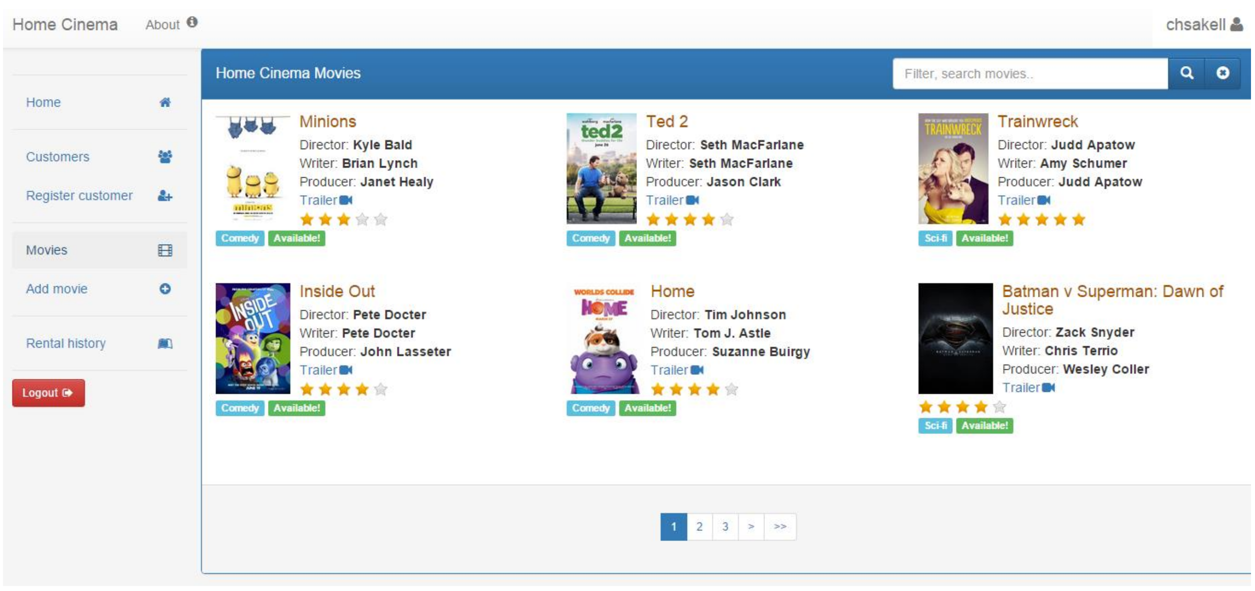
Mobile phone

- Successfull submission will appear here.
- Submition errors will appear here.

Figure 4. Customer registration view

Requirement 3 - Movies:

1. All movies must be displayed with their relevant information (*availability, trailer etc..*)
2. Pagination must be used for faster results, and user can either filter the already displayed movies or search for new ones
3. Clicking on a DVD image must show the movie's **Details** view where user can either **edit** the movie or **rent** it to a specific customer if available. This view is accessible only to authenticated users
4. When employee decides to rent a specific DVD to a customer through the Rent view, it should be able to search customers through an auto-complete textbox
5. The details view displays inside a panel, rental-history information for this movie, that is the dates rentals and returnings occurred. From this panel user can search a specific rental and mark it as **returned**
6. Authenticated employees should be able to add a new entry to the system. They should be able to upload a relevant image for the movie as well



Minions
Director: **Kyle Bald**
Writer: **Brian Lynch**
Producer: **Janet Healy**
Trailer
Comedy Available!

Ted 2
Director: **Seth MacFarlane**
Writer: **Seth MacFarlane**
Producer: **Jason Clark**
Trailer
Comedy Available!

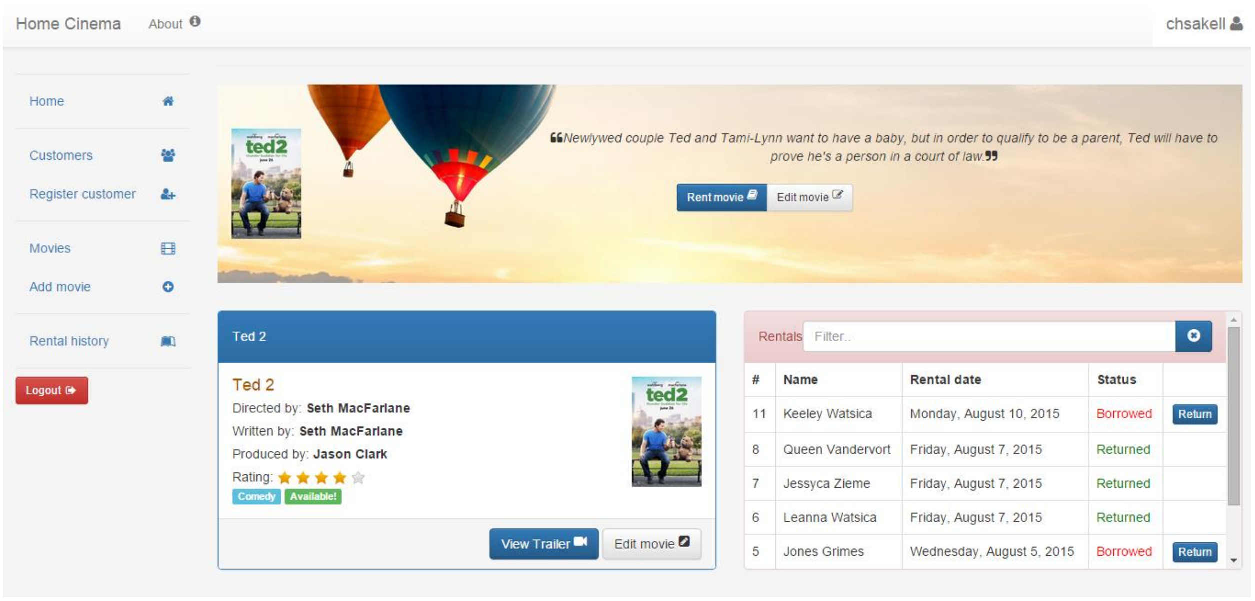
Trainwreck
Director: **Judd Apatow**
Writer: **Amy Schumer**
Producer: **Judd Apatow**
Trailer
Sci-fi Available!

Inside Out
Director: **Pete Docter**
Writer: **Pete Docter**
Producer: **John Lasseter**
Trailer
Comedy Available!

Home
Director: **Tim Johnson**
Writer: **Tom J. Astle**
Producer: **Suzanne Buirgy**
Trailer
Comedy Available!

Batman v Superman: Dawn of Justice
Director: **Zack Snyder**
Writer: **Chris Terrio**
Producer: **Wesley Coller**
Trailer
Sci-fi Available!

Figure 5. Movies home view with filter-search capabilities



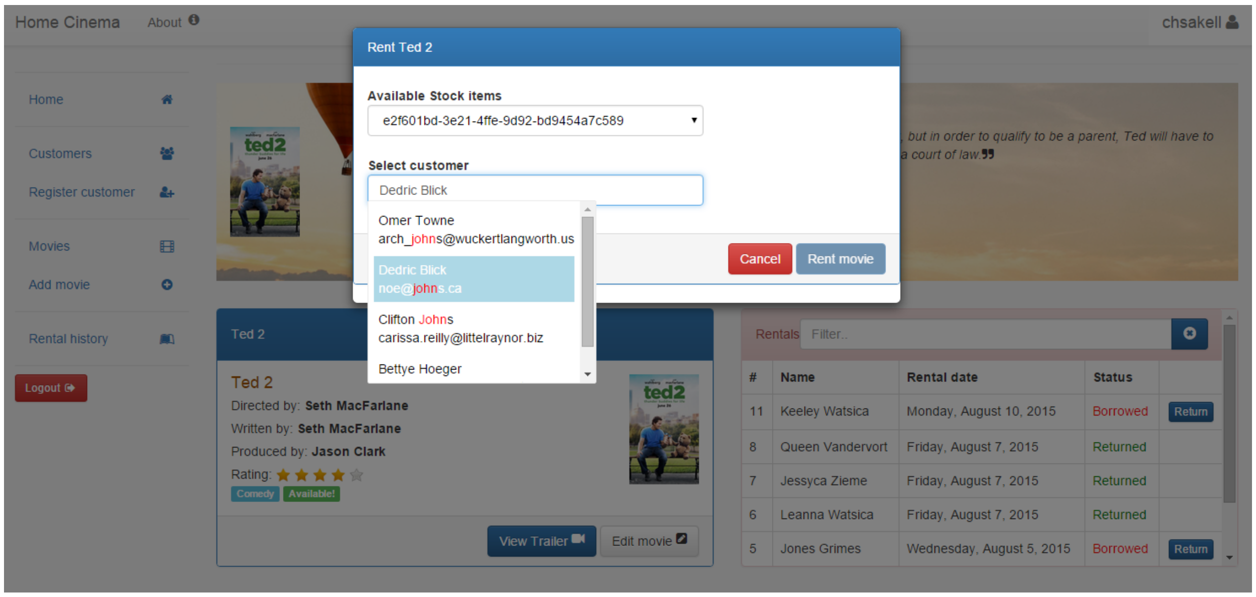
Newlywed couple Ted and Tami-Lynn want to have a baby, but in order to qualify to be a parent, Ted will have to prove he's a person in a court of law.

Ted 2

Ted 2
Directed by: **Seth MacFarlane**
Written by: **Seth MacFarlane**
Produced by: **Jason Clark**
Rating:
Comedy Available!

#	Name	Rental date	Status	
11	Keeley Watsica	Monday, August 10, 2015	Borrowed	Return
8	Queen Vandervort	Friday, August 7, 2015	Returned	
7	Jessyca Zieme	Friday, August 7, 2015	Returned	
6	Leanna Watsica	Friday, August 7, 2015	Returned	
5	Jones Grimes	Wednesday, August 5, 2015	Borrowed	Return

Figure 6. Movie details view with rental statistics



Available Stock items: e2f601bd-3e21-4ffe-9d92-bd9454a7c589

Select customer: Dedric Blick

Omer Towne
arch_johns@wuckertlangworth.us

Dedric Blick
noe@johns.ca

Clifton Johns
carissa.reilly@littelraynor.biz

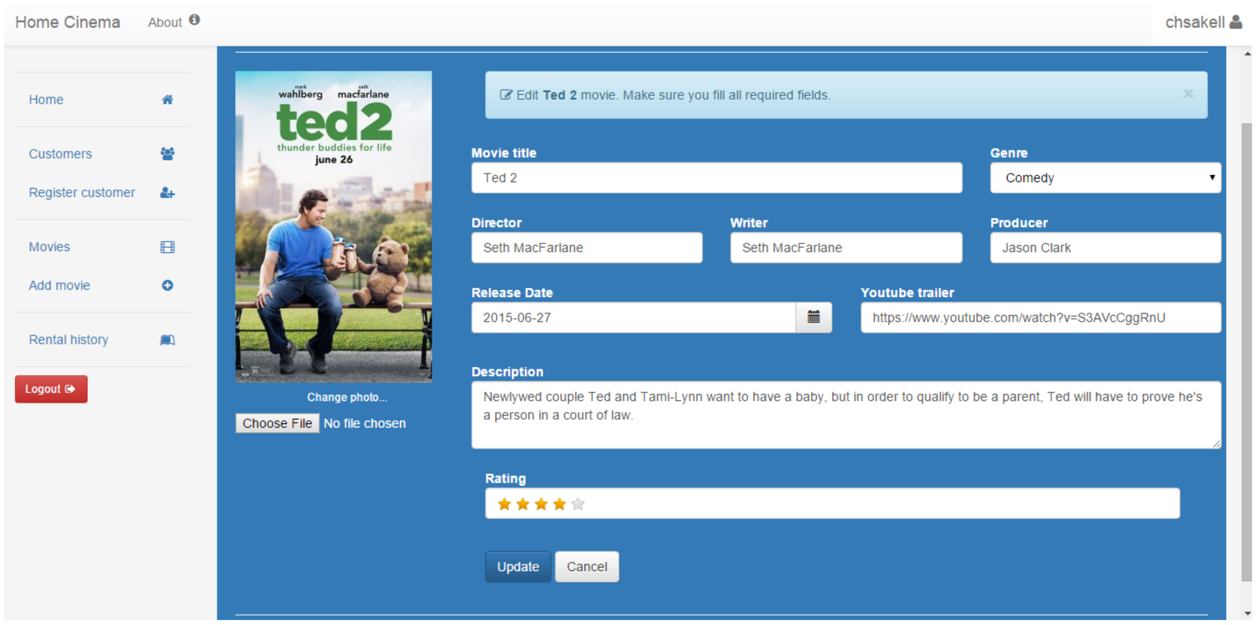
Bettye Hoeger

... but in order to qualify to be a parent, Ted will have to ... a court of law.

Ted 2
Directed by: **Seth MacFarlane**
Written by: **Seth MacFarlane**
Produced by: **Jason Clark**
Rating: ★★★★☆
Comedy Available!

#	Name	Rental date	Status	
11	Keeley Watsica	Monday, August 10, 2015	Borrowed	Return
8	Queen Vandervort	Friday, August 7, 2015	Returned	
7	Jessyca Zieme	Friday, August 7, 2015	Returned	
6	Leanna Watsica	Friday, August 7, 2015	Returned	
5	Jones Grimes	Wednesday, August 5, 2015	Borrowed	Return

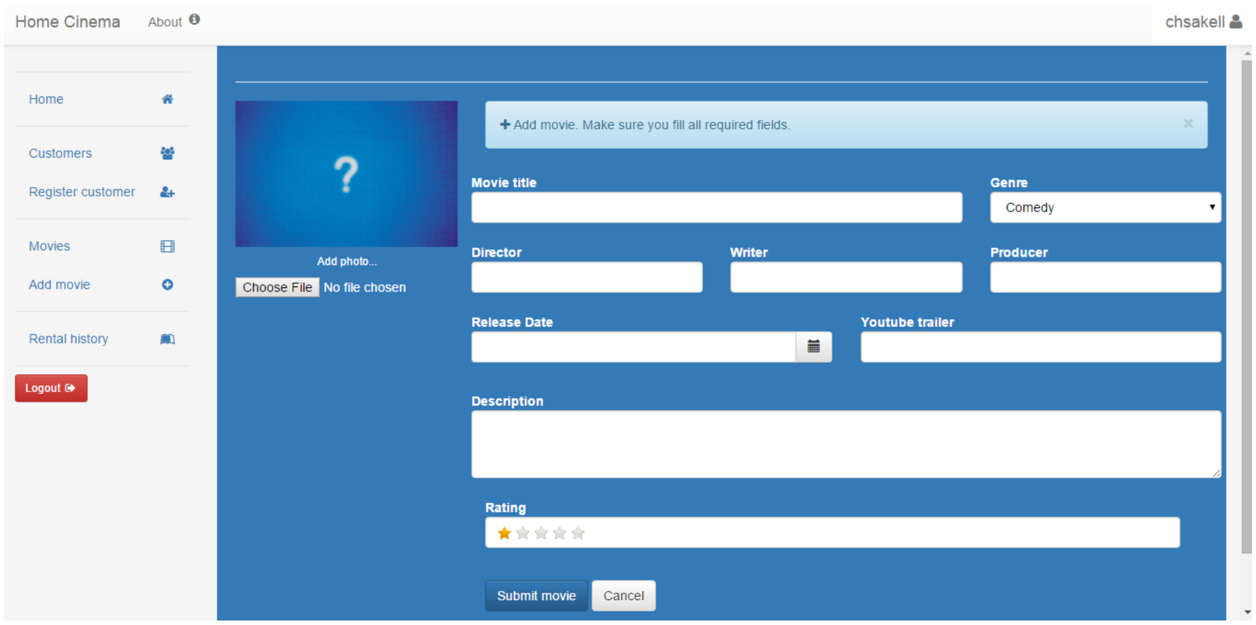
Figure 7. Rent movie to customer popup



Edit **Ted 2** movie. Make sure you fill all required fields.

Movie title: Ted 2
Genre: Comedy
Director: Seth MacFarlane
Writer: Seth MacFarlane
Producer: Jason Clark
Release Date: 2015-06-27
Youtube trailer: https://www.youtube.com/watch?v=S3AVcCggRnU
Description: Newlywed couple Ted and Tami-Lynn want to have a baby, but in order to qualify to be a parent, Ted will have to prove he's a person in a court of law.
Rating: ★★★★☆

Figure 8. Edit movie view



Add movie. Make sure you fill all required fields.

Genre: Comedy

Figure 9. Add movie view

Requirement 4 – Movie Rental History:

1. There should be a specific view for authenticated users where rental history is being displayed for all system's movies. History is based on total rentals per date and it's being displayed through a line chart

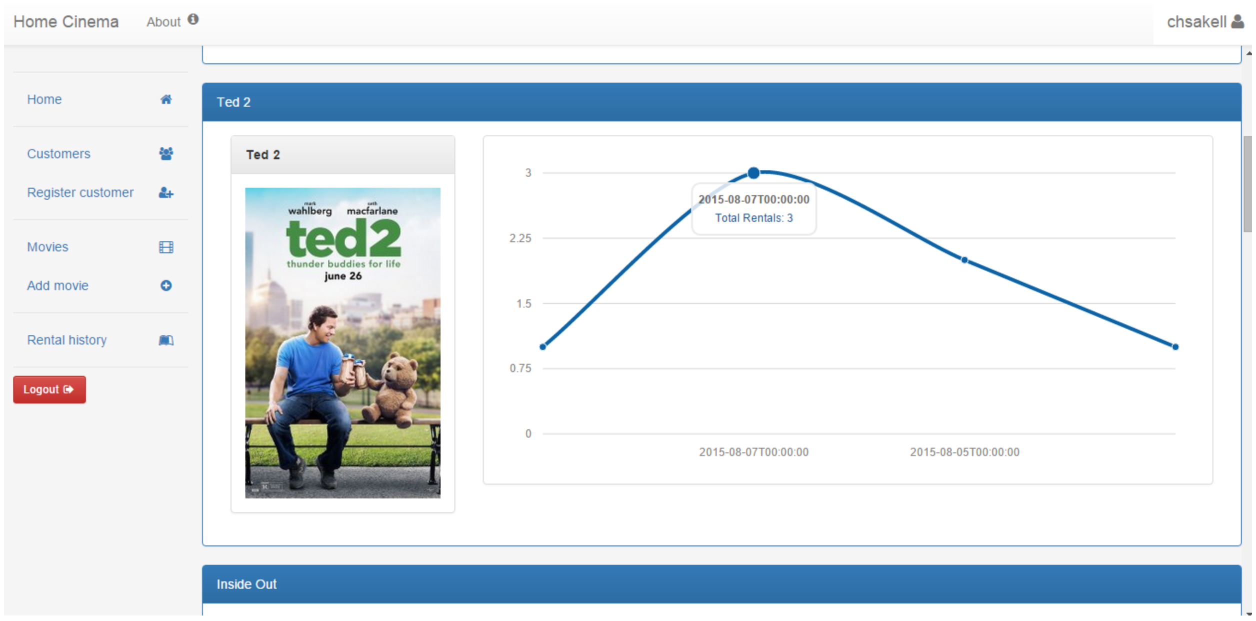


Figure 10. Movie rental statistics chart

Requirement 5 – Accounts:

1. There should be views for employees to either login or register to system. For start employees are being registered as Administrator

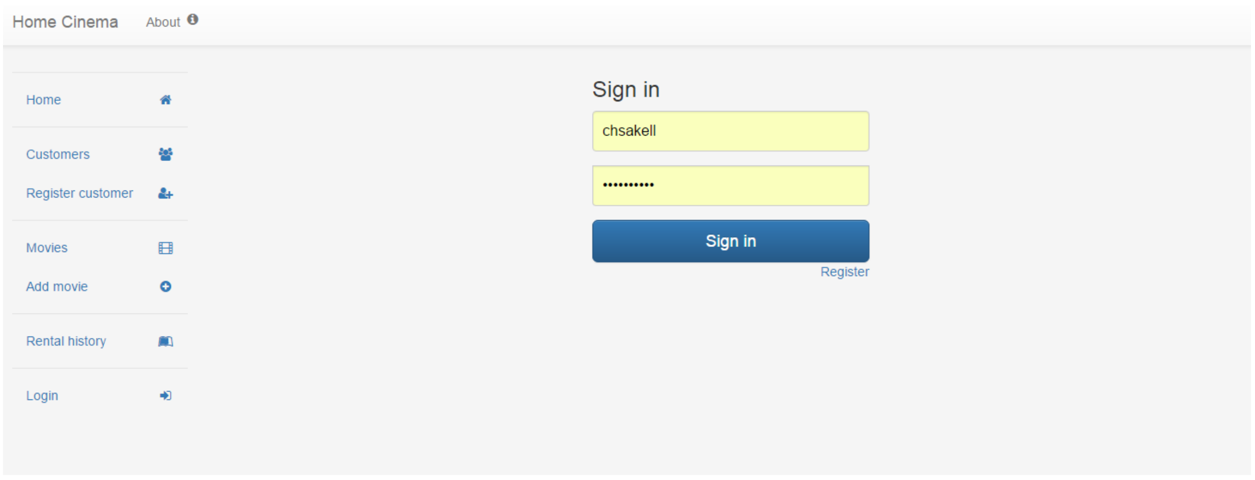
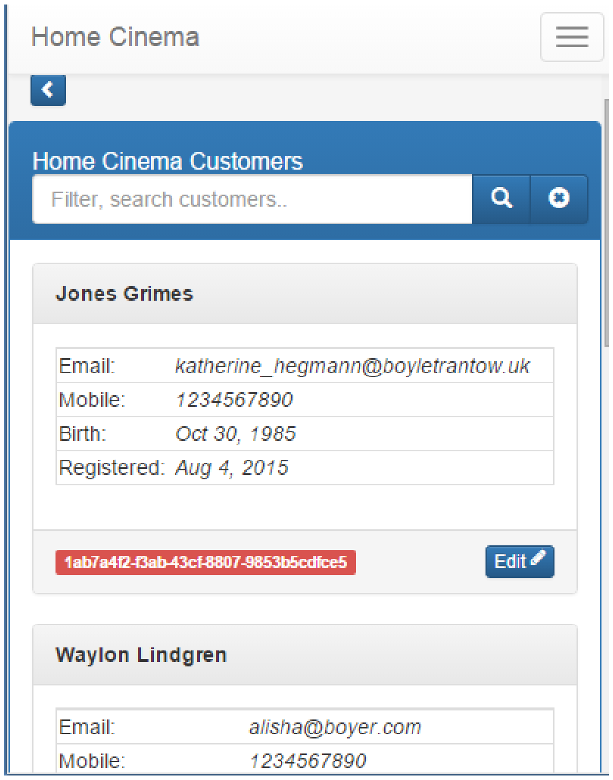


Figure 11. Sign in view

General requirements:

1. All views should be displayed smoothly even to mobile devices. For this bootstrap and collapsible components will be used (*sidebar*, *topbar*)



Jones Grimes

Email:	*katherine_hegmann@boyletrantow.uk*
Mobile:	*1234567890*
Birth:	*Oct 30, 1985*
Registered:	*Aug 4, 2015*

1ab7a4f2-f3ab-43cf-8807-9853b5cdfce5

Waylon Lindgren

Email:	*alisha@boyer.com*
Mobile:	*1234567890*

Figure 12. Collapsible top and side bars

What we are going to use

We have all the requirements, now we need to decide the technologies we are going to use in order to build our SPA application.

Server side:

- *ASP.NET Web API* for serving data to Web clients (browsers)
- *Entity Framework* as *Object-relational Mapper* for accessing data (SQL Server)
- *Autofac* for *Inversion of Control Container* and resolving dependencies
- *Automapper* for mapping Domain entities to *ViewModels*
- *FluentValidation* for validating *ViewModels* in Web API Controllers

Client side:

- *AngularJS* as the core JavaScript framework
- *Bootstrap 3* as the CSS framework for creating a fluent and mobile compatible interface
- *3rd party* libraries

SPA architecture

We have seen both application's requirements and the technologies we are going to use, now it's time to design a decoupled, testable and scalable solution. There are two different designs we need to provide here. The first one has to do with the entire project's solution structure and how is this divided in independent components. The second one has to do with the *SPA* structure itself that is how **angularJS** folders and files will be organized.

Application Design – 1

- At the lowest level we have the **Database**. We'll use *Entity Framework Code First* but this doesn't prevent us to design the database directly from the SQL Server. In fact that's what I did, I created all the tables and relationships in my SQL Server and then added the respective model Entities. That way I didn't have to work with **Code First Migrations** and have more control on my database and entities. Though, I have to note that when development processes finished, I enabled code first migrations and added a seed method to initialize some data, just to help you kick of the project. We'll see more about this in the installation section
- The next level is the domain Entities. These are the classes that will map our database tables. One point I want to make here is that when I started design my entities, none of them had virtual references or collections for lazy loading. Those virtual properties were added during the development and the needs of the application
- Entity Framework configurations, **DbContext** and **Generic Repositories** are the next level. Here we 'll configure EF and we 'll create the base classes and repositories to access database data
- **Service layer** is what comes next. For this application there will be only one service the *membership service*. This means that data repositories are going to be injected directly to our Web API Controllers. I've made that decision because there will be no complex functionality as far the data accessing. If you wish though you can use this layer to add a middle layer for data accessing too
- Last but not least is the Web application that will contain the **Web API Controllers** and the SPA itself. This project will start as an Empty ASP.NET Web Application with both Web API and MVC references included

Let's take a look at the Database design.

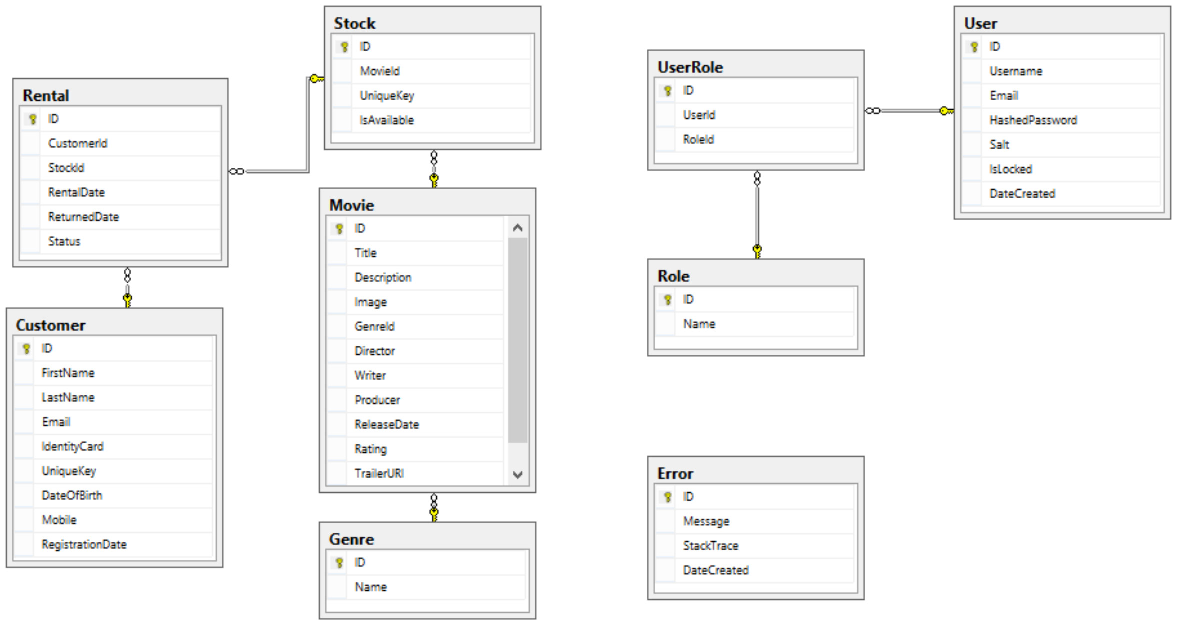


Figure 13. HomeCinema database diagram

Notice that for each Movie multiple stock items can exist that may be available or not. Think this as there are many DVDs of the same movie. The movie itself will be categorized as available or not depending on if there is any available stock item or not. Each customer rents a stock item and when he/she does, this stock item becomes unavailable until he/she returns it back to store. The tables used to accomplish this functionality are *Customer*, *Rental*, *Stock*. You can also see that there are 3 membership tables, *User*, *UserRole and Role* which are quite self-explanatory. Upon them we'll build the custom membership mechanism. I have also created an Error table just to show you how to avoid polluting you code with *Try*, *Catch* blocks and have a centralized logging point in your application.

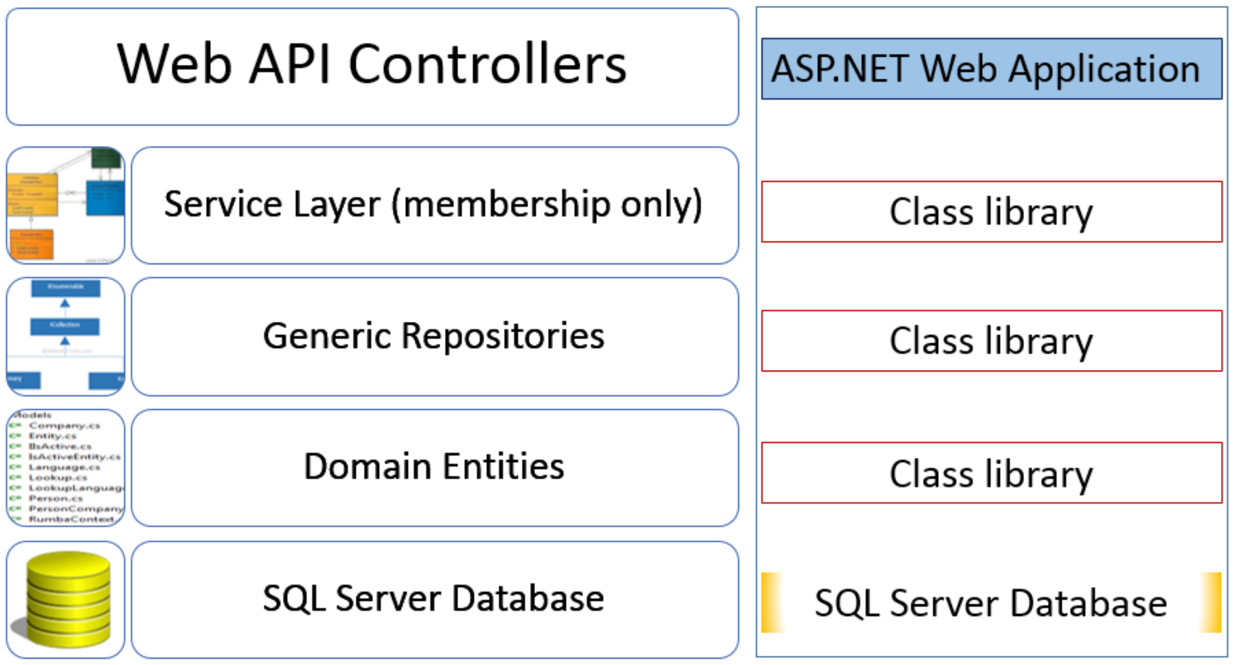


Figure 14. Solution's architecture

Application Design – 2 (angular components)

- Folders are organized by **Feature** in our SPA. This means that you 'll see folders such as *Customers*, *Movies* and *Rental*
- Each of those folders may have angularJS controllers, directives or templates
- There is a folder *Modules* for hosting reusable components-modules. Those modules use common directives or services from the respective common folders
- **3rd party libraries** are inside a folder *Vendors*. Here I want to point something important. You should (if not already yet) start using Bower for installing-downloading web dependencies, packages etc... After you download required packages through Bower, you can then either include them in your project or simple simply reference them from their downloaded folder. In this application though, you will find all the required vendors inside this folder and just for reference, I will provide you with the bower installation commands for most of those packages

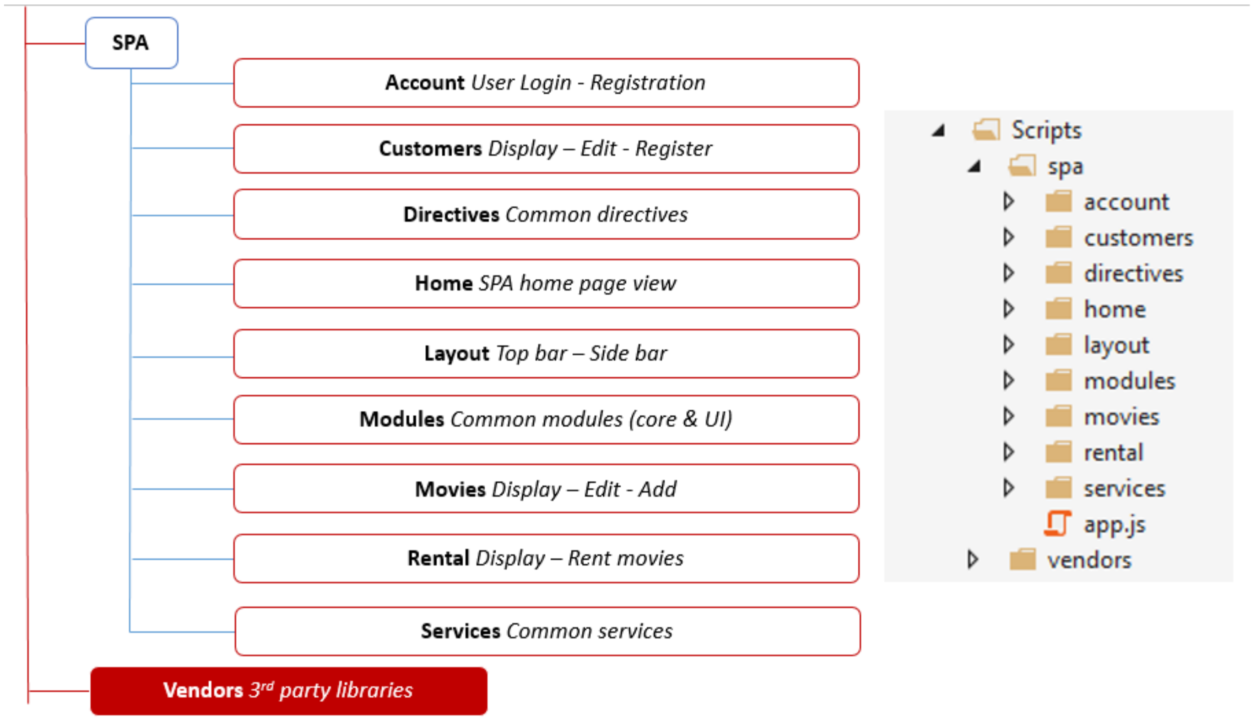

SPA
Account *User Login - Registration*
Customers *Display – Edit - Register*
Directives *Common directives*
Home *SPA home page view*
Layout *Top bar – Side bar*
Modules *Common modules (core & UI)*
Movies *Display – Edit - Add*
Rental *Display – Rent movies*
Services *Common services*
Vendors *3rd party libraries*
Scripts
spa
account
customers
directives
home
layout
modules
movies
rental
services
app.js
vendors

Figure 15. SPA architecture (angularJS)

Domain Entities

Time to start building our Single Page Application. Create a new empty solution named *HomeCinema* and add new class library project named *HomeCinema.Entities*. We'll create those first. All of our entities will implement an **IEntityBase** interface which means that will have an *ID* property mapping to their primary key in the database. Add the following interface:

IEntityBase.cs

```
public interface IEntityBase
    {
        int ID { get; set; }
    }
```

Each movie belongs to a specific Genre (*Comedy, Drama, Action, etc.*.. If we want to be able to retrieve all movies through a Genre instance, then we need to add a virtual collection of Movies property.

Genre.cs

```
public class Genre : IEntityBase
    {
        public Genre()
        {
            Movies = new List<Movie>();
        }
        public int ID { get; set; }
        public string Name { get; set; }
        public virtual ICollection<Movie> Movies { get; set; }
    }
```

The most important Entity of our application is the **Movie**. A movie holds information such as title, director, release date, trailer URL (Youtube) or rating. As we have already mentioned, for each movie there are several stock items and hence for this entity we need to add a collection of Stock.

Movie.cs

```
public class Movie : IEntityBase
    {
        public Movie()
        {
            Stocks = new List<Stock>();
        }
        public int ID { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Image { get; set; }
        public int GenreId { get; set; }
        public virtual Genre Genre { get; set; }
        public string Director { get; set; }
        public string Writer { get; set; }
        public string Producer { get; set; }
        public DateTime ReleaseDate { get; set; }
        public byte Rating { get; set; }
        public string TrailerURI { get; set; }
        public virtual ICollection<Stock> Stocks { get; set; }
    }
```

Each stock actually describes a DVD by itself. It has a reference to a specific movie and a unique key (code) that uniquely identifies it. For example, when there are three available DVDs for a specific movie then 3 unique codes identify those DVDs. The employee will choose among those codes which could probably be written on the DVD to rent a specific movie to a customer. Since a movie rental is directly connected to a stock item, Stock entity may have a collection of Rental items that is all rentals for this stock item.

Stock.cs

```
public class Stock : IEntityBase
    {
        public Stock()
        {
            Rentals = new List<Rental>();
        }
        public int ID { get; set; }
        public int MovieId { get; set; }
        public virtual Movie Movie { get; set; }
        public Guid UniqueKey { get; set; }
        public bool IsAvailable { get; set; }
        public virtual ICollection<Rental> Rentals { get; set; }
    }
```

The customer Entity is self-explanatory.

Customer.cs

```
public class Customer : IEntityBase
    {
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
        public string IdentityCard { get; set; }
        public Guid UniqueKey { get; set; }
        public DateTime DateOfBirth { get; set; }
        public string Mobile { get; set; }
        public DateTime RegistrationDate { get; set; }
    }
```

The Rental entity which finally describes a DVD rental for a specific customer holds information about the customer, the stock item he/she picked (DVD and its code), the rentals date, its status (*Borrowed or Returned* and the date the customer returned it.

Rental.cs

```
public class Rental : IEntityBase
    {
        public int ID { get; set; }
        public int CustomerId { get; set; }
        public int StockId { get; set; }
        public virtual Stock Stock { get; set; }
        public DateTime RentalDate { get; set; }
        public Nullable<DateTime> ReturnedDate { get; set; }
        public string Status { get; set; }
    }
```

Now let's see all Entities related Membership. The first one is the *Role* that describes logged in user's role. For our application there will be only the Admin role (employees) but we will discuss later the scalability options we have in case we want customers to use the application as well. Let me remind you that we are going to use Basic Authentication for Web API Controllers and many controllers and their actions will have an *Authorize* attribute and a list of roles authorized to access their resources.

Role.cs

```
public class Role : IEntityBase
    {
        public int ID { get; set; }
        public string Name { get; set; }
    }
```

User entity holds basic information for the user and most important the salt and the encrypted by this salt, password.

User.cs

```
public class User : IEntityBase
    {
        public User()
        {
            UserRoles = new List<UserRole>();
        }
        public int ID { get; set; }
        public string Username { get; set; }
        public string Email { get; set; }
        public string HashedPassword { get; set; }
        public string Salt { get; set; }
        public bool IsLocked { get; set; }
        public DateTime DateCreated { get; set; }

        public virtual ICollection<UserRole> UserRoles { get; set; }
    }
```

A user may have more than one roles so we have a *UserRole* Entity as well.

UserRole.cs

```
public class UserRole : IEntityBase
    {
        public int ID { get; set; }
        public int UserId { get; set; }
        public int RoleId { get; set; }
        public virtual Role Role { get; set; }
    }
```

One last entity I have added is the **Error**. It's always good to log your application's errors and we'll use a specific repository to do this. I decided to add error logging functionality in order to show you a nice trick that will prevent you from polluting you controllers with *Try Catch* blocks all over the place. We'll see it in action when we reach Web API Controllers.

Error.cs

```
public class Error : IEntityBase
    {
        public int ID { get; set; }
        public string Message { get; set; }
        public string StackTrace { get; set; }
        public DateTime DateCreated { get; set; }
    }
```

Data repositories

Add a new class library project named *HomeCinema.Data* and add reference to *HomeCinema.Entities* project. Make sure you also install **Entity Framework** through Nuget Packages. For start we will create EF Configurations for our Entities. Add a new folder named *Configurations* and add the following configuration to declare the primary key for our Entities:

EntityBaseConfiguration.cs

```
public class EntityBaseConfiguration<T> : EntityTypeConfiguration<T> where T : class,
IEntityBase
    {
        public EntityBaseConfiguration()
        {
            HasKey(e => e.ID);
        }
    }
```

Entity Framework either way assumes that a property named "*ID*" is a primary key but this is a nice way to declare it in case you give this property different name. Following are one by one all other configurations. I will highlight the important lines (if any) to notice for each of these.

GenreConfiguration.cs

```
public class GenreConfiguration : EntityBaseConfiguration<Genre>
    {
        public GenreConfiguration()
        {
            Property(g => g.Name).IsRequired().HasMaxLength(50);
        }
    }
```

MovieConfiguration.cs

```
public class MovieConfiguration : EntityBaseConfiguration<Movie>
    {
        public MovieConfiguration()
        {
            Property(m => m.Title).IsRequired().HasMaxLength(100);
            Property(m => m.GenreId).IsRequired();
            Property(m => m.Director).IsRequired().HasMaxLength(100);
            Property(m => m.Writer).IsRequired().HasMaxLength(50);
            Property(m => m.Producer).IsRequired().HasMaxLength(50);
            Property(m => m.Writer).HasMaxLength(50);
            Property(m => m.Producer).HasMaxLength(50);
            Property(m => m.Rating).IsRequired();
            Property(m => m.Description).IsRequired().HasMaxLength(2000);
            Property(m => m.TrailerURI).HasMaxLength(200);
            HasMany(m => m.Stocks).WithRequired().HasForeignKey(s => s.MovieId);
        }
    }
```

StockConfiguration.cs

```
public class StockConfiguration : EntityBaseConfiguration<Stock>
    {
        public StockConfiguration()
        {
            Property(s => s.MovieId).IsRequired();
            Property(s => s.UniqueKey).IsRequired();
            Property(s => s.IsAvailable).IsRequired();
            HasMany(s => s.Rentals).WithRequired(r=> r.Stock).HasForeignKey(r =>
r.StockId);
        }
    }
```

CustomerConfiguration.cs

```
public class CustomerConfiguration : EntityBaseConfiguration<Customer>
    {
        public CustomerConfiguration()
        {
            Property(u => u.FirstName).IsRequired().HasMaxLength(100);
            Property(u => u.LastName).IsRequired().HasMaxLength(100);
            Property(u => u.IdentityCard).IsRequired().HasMaxLength(50);
            Property(u => u.UniqueKey).IsRequired();
            Property(c => c.Mobile).HasMaxLength(10);
            Property(c => c.Email).IsRequired().HasMaxLength(200);
            Property(c => c.DateOfBirth).IsRequired();
        }
    }
```

RentalConfiguration.cs

```
public class RentalConfiguration : EntityBaseConfiguration<Rental>
    {
        public RentalConfiguration()
        {
            Property(r => r.CustomerId).IsRequired();
            Property(r => r.StockId).IsRequired();
            Property(r => r.Status).IsRequired().HasMaxLength(10);
            Property(r => r.ReturnedDate).IsOptional();
        }
    }
```

RoleConfiguration.cs

```
public class RoleConfiguration : EntityBaseConfiguration<Role>
    {
        public RoleConfiguration()
        {
            Property(ur => ur.Name).IsRequired().HasMaxLength(50);
        }
    }
```

UserRoleConfiguration.cs

```
public class UserRoleConfiguration : EntityBaseConfiguration<UserRole>
    {
        public UserRoleConfiguration()
        {
            Property(ur => ur.UserId).IsRequired();
            Property(ur => ur.RoleId).IsRequired();
        }
    }
```

UserConfiguration.cs

```
public class UserConfiguration : EntityBaseConfiguration<User>
    {
        public UserConfiguration()
        {
            Property(u => u.Username).IsRequired().HasMaxLength(100);
            Property(u => u.Email).IsRequired().HasMaxLength(200);
            Property(u => u.HashedPassword).IsRequired().HasMaxLength(200);
            Property(u => u.Salt).IsRequired().HasMaxLength(200);
            Property(u => u.IsLocked).IsRequired();
            Property(u => u.DateCreated);
        }
    }
```

Those configurations will affect how the database tables will be created. Add a new class named **HomeCinemaContext** at the root of the project. This class will inherit from DbContext and will be the main class for accessing data from the database.

HomeCinemaContext.cs

```
public class HomeCinemaContext : DbContext
    {
        public HomeCinemaContext()
            : base("HomeCinema")
        {
            Database.SetInitializer<HomeCinemaContext>(null);
        }

        #region Entity Sets
        public IDbSet<User> UserSet { get; set; }
        public IDbSet<Role> RoleSet { get; set; }
        public IDbSet<UserRole> UserRoleSet { get; set; }
        public IDbSet<Customer> CustomerSet { get; set; }
        public IDbSet<Movie> MovieSet { get; set; }
        public IDbSet<Genre> GenreSet { get; set; }
        public IDbSet<Stock> StockSet { get; set; }
        public IDbSet<Rental> RentalSet { get; set; }
        public IDbSet<Error> ErrorSet { get; set; }
        #endregion

        public virtual void Commit()
        {
            base.SaveChanges();
        }
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

            modelBuilder.Configurations.Add(new UserConfiguration());
            modelBuilder.Configurations.Add(new UserRoleConfiguration());
            modelBuilder.Configurations.Add(new RoleConfiguration());
            modelBuilder.Configurations.Add(new CustomerConfiguration());
            modelBuilder.Configurations.Add(new MovieConfiguration());
            modelBuilder.Configurations.Add(new GenreConfiguration());
            modelBuilder.Configurations.Add(new StockConfiguration());
            modelBuilder.Configurations.Add(new RentalConfiguration());
        }
    }
```

Notice that I've made a decision to name all of the Entity Sets with a *Set* prefix. Also I turned off the default pluralization convention that Entity Framework uses when creating the tables in database. This will result in tables having the same name as the Entity. You need to add an **App.config** file if not exists and create the following connection string:

App.config's Connection string

```
<connectionStrings>
    <add name="HomeCinema" connectionString="Data Source=(LocalDb)\v11.0;Initial
Catalog=HomeCinema;Integrated Security=SSPI; MultipleActiveResultSets=true"
providerName="System.Data.SqlClient" />
  </connectionStrings>
```

You can alter the server if you wish to match your development environment. Let us proceed with the **UnitOfWork** pattern implementation. Add a folder named *Infrastructure* and paste the following classes and interfaces:

Disposable.cs

```
public class Disposable : IDisposable
    {
        private bool isDisposed;

        ~Disposable()
        {
            Dispose(false);
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
        private void Dispose(bool disposing)
        {
            if (!isDisposed && disposing)
            {
                DisposeCore();
            }

            isDisposed = true;
        }

        // Ovveride this to dispose custom objects
        protected virtual void DisposeCore()
        {
        }
    }
```

IDbFactory.cs

```
public interface IDbFactory : IDisposable
    {
        HomeCinemaContext Init();
    }
```

DbFactory.cs

```
public class DbFactory : Disposable, IDbFactory
    {
        HomeCinemaContext dbContext;

        public HomeCinemaContext Init()
        {
            return dbContext ?? (dbContext = new HomeCinemaContext());
        }

        protected override void DisposeCore()
        {
            if (dbContext != null)
                dbContext.Dispose();
        }
    }
```

IUnitOfWork.cs

```
public interface IUnitOfWork
    {
        void Commit();
    }
```

UnitOfWork.cs

```
public class UnitOfWork : IUnitOfWork
    {
        private readonly IDbFactory dbFactory;
        private HomeCinemaContext dbContext;

        public UnitOfWork(IDbFactory dbFactory)
        {
            this.dbFactory = dbFactory;
        }
        public HomeCinemaContext DbContext
        {
            get { return dbContext ?? (dbContext = dbFactory.Init()); }
        }

        public void Commit()
        {
            DbContext.Commit();
        }
    }
```

Time for the Generic Repository Pattern. We have seen this pattern many times in this blog but this time I will make a slight change. One of the blog's readers asked me if he had to create a specific repository class that implements the generic repository T each time a need for a new type of repository is needed. Reader's question was really good if you think that you may have hundreds of Entities in a large scale application. The answer is NO and we will see it on action in this project where will try to inject repositories of type T as needed. Create a folder named *Repositories* and add the following interface with its implementation class:

IEntityBaseRepository.cs

```
public interface IEntityBaseRepository<T> : IEntityBaseRepository where T : class,
IEntityBase, new()
    {
        IQueryable<T> AllIncluding(params Expression<Func<T, object>>[]
includeProperties);
        IQueryable<T> All { get; }
        IQueryable<T> GetAll();
        T GetSingle(int id);
        IQueryable<T> FindBy(Expression<Func<T, bool>> predicate);
        void Add(T entity);
        void Delete(T entity);
        void Edit(T entity);
    }
```

EntityBaseRepository.cs

```
public class EntityBaseRepository<T> : IEntityBaseRepository<T>
            where T : class, IEntityBase, new()
    {

        private HomeCinemaContext dataContext;

        #region Properties
        protected IDbFactory DbFactory
        {
            get;
            private set;
        }

        protected HomeCinemaContext DbContext
        {
            get { return dataContext ?? (dataContext = DbFactory.Init()); }
        }
        public EntityBaseRepository(IDbFactory dbFactory)
        {
            DbFactory = dbFactory;
        }
        #endregion
        public virtual IQueryable<T> GetAll()
        {
            return DbContext.Set<T>();
        }
        public virtual IQueryable<T> All
        {
            get
            {
                return GetAll();
            }
        }
        public virtual IQueryable<T> AllIncluding(params Expression<Func<T, object>>[]
includeProperties)
        {
            IQueryable<T> query = DbContext.Set<T>();
            foreach (var includeProperty in includeProperties)
            {
                query = query.Include(includeProperty);
            }
            return query;
        }
        public T GetSingle(int id)
        {
            return GetAll().FirstOrDefault(x => x.ID == id);
        }
        public virtual IQueryable<T> FindBy(Expression<Func<T, bool>> predicate)
        {
            return DbContext.Set<T>().Where(predicate);
        }

        public virtual void Add(T entity)
        {
            DbEntityEntry dbEntityEntry = DbContext.Entry<T>(entity);
            DbContext.Set<T>().Add(entity);
        }
        public virtual void Edit(T entity)
        {
            DbEntityEntry dbEntityEntry = DbContext.Entry<T>(entity);
            dbEntityEntry.State = EntityState.Modified;
        }
        public virtual void Delete(T entity)
        {
            DbEntityEntry dbEntityEntry = DbContext.Entry<T>(entity);
            dbEntityEntry.State = EntityState.Deleted;
        }
    }
```

Before leaving this project and proceed with the *Services* one, there is only one thing remained to do. As I said when I was developing this application I was designing the database on my SQL Server and adding the respective Entities at the same time (yeap, you can do this as well...). No migrations where enabled.

Yet, I thought that I should enable them in order to help you kick of the project and create the database automatically. For this you should do the same thing if you follow along with me. Open Package Manager Console, make sure you have selected the *HomeCinema.Data* project and type the following command:

Enable migrations

```
enable-migrations
```

This will add a **Configuration** class inside a Migrations folder. This Configuration class has a seed method that is invoked when you create the database. The seed method I've written is a little bit large for pasting it here so please find it here. What I did is add data for Genres, Movies, Roles, Customers and Stocks. For customers I used a Nuget Package named **MockData** so make sure you install it too. More over I added a user with username chsakell and password homecinema. You can use them in order to login to our SPA. Otherwise you can just register a new user and sign in with those credentials. If you want to create the database right now, run the following commands from the Package Manager Console:

Create initial migration

```
add-migration "initial_migration"
```

Update database

```
update-database -verbose
```

We'll come up again to *HomeCinema.Data* project later to add some extension methods.

Membership

There is one middle layer between the Web application and the Data repositories and that's the **Service** layer. In this application though, we will use this layer only for the membership's requirements leaving all data repositories being injected as they are directly to API Controllers. Add a new class library project named HomeCinema.Services and make sure you add references to both of the other projects, HomeCinema.Data and HomeCinema.Entities. First, we'll create a simple Encryption service to create salts and encrypted passwords and then we'll use this service to implement a custom membership mechanism. Add a folder named Abstract and create the following interfaces.

EncryptionService.cs

```
public interface IEncryptionService
    {
        string CreateSalt();
        string EncryptPassword(string password, string salt);
    }
```

IMembershipService.cs

```
public interface IMembershipService
    {
        MembershipContext ValidateUser(string username, string password);
        User CreateUser(string username, string email, string password, int[] roles);
        User GetUser(int userId);
        List<Role> GetUserRoles(string username);
    }
```

At the root of this project add the EncryptionService implementation. It's a simple password encryption based on a salt and the **SHA256** algorithm from System.Security.Cryptography namespace. Of course you can always use your own implementation algorithm.

EncryptionService.cs

```
public class EncryptionService : IEncryptionService
    {
        public string CreateSalt()
        {
            var data = new byte[0x10];
            using (var cryptoServiceProvider = new RNGCryptoServiceProvider())
            {
                cryptoServiceProvider.GetBytes(data);
                return Convert.ToBase64String(data);
            }
        }

        public string EncryptPassword(string password, string salt)
        {
            using (var sha256 = SHA256.Create())
            {
                var saltedPassword = string.Format("{0}{1}", salt, password);
                byte[] saltedPasswordAsBytes = Encoding.UTF8.GetBytes(saltedPassword);
                return
Convert.ToBase64String(sha256.ComputeHash(saltedPasswordAsBytes));
            }
        }
    }
```

Let's move to the Membership Service now. For start let's see the base components of this class. Add the following class at the root of the project as well.

MembershipService.cs

```
public class MembershipService : IMembershipService
    {
        #region Variables
        private readonly IEntityBaseRepository<User> _userRepository;
        private readonly IEntityBaseRepository<Role> _roleRepository;
        private readonly IEntityBaseRepository<UserRole> _userRoleRepository;
        private readonly IEncryptionService _encryptionService;
        private readonly IUnitOfWork _unitOfWork;
        #endregion
        public MembershipService(IEntityBaseRepository<User> userRepository,
IEntityBaseRepository<Role> roleRepository,
        IEntityBaseRepository<UserRole> userRoleRepository, IEncryptionService
encryptionService, IUnitOfWork unitOfWork)
        {
            _userRepository = userRepository;
            _roleRepository = roleRepository;
            _userRoleRepository = userRoleRepository;
            _encryptionService = encryptionService;
            _unitOfWork = unitOfWork;
        }
    }
```

Here we can see for the first time the way the generic repositories are going to be injected through the Autofac Inversion of Control Container. Before moving to the core implementation of this service we will have to add a User extension method in *HomeCinema.Data* and some helper methods in the previous class. Switch to *HomeCinema.Data* and add a new folder named **Extensions**. We will use this folder for adding Data repository extensions based on the Entity Set. Add the following User Entity extension method which retrieves a User instance based on its username.

UserExtensions.cs

```
public static class UserExtensions
    {
        public static User GetSingleByUsername(this IEntityBaseRepository<User>
userRepository, string username)
        {
            return userRepository.GetAll().FirstOrDefault(x => x.Username == username);
        }
    }
```

Switch again to *MembershipService* class and add the following helper private methods.

MembershipService helper methods

```
private void addUserToRole(User user, int roleId)
        {
            var role = _roleRepository.GetSingle(roleId);
            if (role == null)
                throw new ApplicationException("Role doesn't exist.");

            var userRole = new UserRole()
            {
                RoleId = role.ID,
                UserId = user.ID
            };
            _userRoleRepository.Add(userRole);
        }

        private bool isPasswordValid(User user, string password)
        {
            return string.Equals(_encryptionService.EncryptPassword(password,
user.Salt), user.HashedPassword);
        }

        private bool isUserValid(User user, string password)
        {
            if (isPasswordValid(user, password))
            {
                return !user.IsLocked;
            }

            return false;
        }
```

Let's view the *CreateUser* implementation method. The method checks if username already in use and if not creates the user.

CreateUser

```
public User CreateUser(string username, string email, string password, int[] roles)
        {
            var existingUser = _userRepository.GetSingleByUsername(username);

            if (existingUser != null)
            {
                throw new Exception("Username is already in use");
            }

            var passwordSalt = _encryptionService.CreateSalt();

            var user = new User()
            {
                Username = username,
                Salt = passwordSalt,
                Email = email,
                IsLocked = false,
                HashedPassword = _encryptionService.EncryptPassword(password,
passwordSalt),
                DateCreated = DateTime.Now
            };

            _userRepository.Add(user);

            _unitOfWork.Commit();

            if (roles != null || roles.Length > 0)
            {
                foreach (var role in roles)
                {
                    addUserToRole(user, role);
                }
            }

            _unitOfWork.Commit();

            return user;
        }
```

The *GetUser* and *GetUserRoles* implementations are quite simple.

GetUser - GetUserRoles

```
public User GetUser(int userId)
        {
            return _userRepository.GetSingle(userId);
        }

public List<Role> GetUserRoles(string username)
        {
            List<Role> _result = new List<Role>();

            var existingUser = _userRepository.GetSingleByUsername(username);

            if (existingUser != null)
            {
                foreach (var userRole in existingUser.UserRoles)
                {
                    _result.Add(userRole.Role);
                }
            }

            return _result.Distinct().ToList();
        }
```

I left the ValidateUser implementation last because is a little more complex than the others. You will have noticed from its interface that this service make use of a class named **MembershipContext**. This custom class is the one that will hold the IPrincipal object when authenticating users. When a valid user passes his/her credentials the service method will create an instance of GenericIdentity for user's username. Then it will set the IPrincipal property using the *GenericIdentity* created and user's roles. User roles information will be used to authorize API Controller's actions based on logged in user's roles. Let's see the MembershipContext class and then the *ValidateUser* implementation.

MembershipContext.cs

```
public class MembershipContext
    {
        public IPrincipal Principal { get; set; }
        public User User { get; set; }
        public bool IsValid()
        {
            return Principal != null;
        }
    }
```

ValidateUser method

```
public MembershipContext ValidateUser(string username, string password)
        {
            var membershipCtx = new MembershipContext();

            var user = _userRepository.GetSingleByUsername(username);
            if (user != null && isUserValid(user, password))
            {
                var userRoles = GetUserRoles(user.Username);
                membershipCtx.User = user;

                var identity = new GenericIdentity(user.Username);
                membershipCtx.Principal = new GenericPrincipal(
                    identity,
                    userRoles.Select(x => x.Name).ToArray());
            }

            return membershipCtx;
        }
```

MembershipContext class may hold additional information for the User. Add anything else you wish according to your needs.

Web Application Configuration

Web API Configuration

We are done building the core components for the HomeCinema Single Page Application, now it's time to create the Web Application that will make use all of the previous parts we created. Add a new Empty Web Application project named *HomeCinema.Web* and make sure to check both the Web API and MVC check buttons. Add references to all the previous projects and install the following Nuget Packages:

Nuget Packages:

1. Entity Framework
2. Autofac ASP.NET Web API 2.2 Integration
3. Automapper
4. FluentValidation

First thing we need to do is create any configurations we want to apply to our application. We 'll start with the Autofac Inversion of Control Container to work along with Web API framework. Next we will configure the bundling and last but not least, we will add the MessageHandler to configure the **Basic Authentication**. Add the following class inside the *App_Start* project's folder.

AutofacWebapiConfig.cs

```
public class AutofacWebapiConfig
    {
        public static IContainer Container;
        public static void Initialize(HttpConfiguration config)
        {
            Initialize(config, RegisterServices(new ContainerBuilder()));
        }

        public static void Initialize(HttpConfiguration config, IContainer container)
        {
            config.DependencyResolver = new AutofacWebApiDependencyResolver(container);
        }

        private static IContainer RegisterServices(ContainerBuilder builder)
        {
            builder.RegisterApiControllers(Assembly.GetExecutingAssembly());

            // EF HomeCinemaContext
            builder.RegisterType<HomeCinemaContext>()
                   .As<DbContext>()
                   .InstancePerRequest();

            builder.RegisterType<DbFactory>()
                .As<IDbFactory>()
                .InstancePerRequest();

            builder.RegisterType<UnitOfWork>()
                .As<IUnitOfWork>()
                .InstancePerRequest();

            builder.RegisterGeneric(typeof(EntityBaseRepository<>))
                   .As(typeof(IEntityBaseRepository<>))
                   .InstancePerRequest();

            // Services
            builder.RegisterType<EncryptionService>()
                .As<IEncryptionService>()
                .InstancePerRequest();

            builder.RegisterType<MembershipService>()
                .As<IMembershipService>()
                .InstancePerRequest();

            Container = builder.Build();

            return Container;
        }
    }
```

This will make sure dependencies will be injected in constructors as expected. Since we are in the *App_Start* folder let's configure the Bundling in our application. Add the following class in *App_Start* folder as well.

BundleConfig.cs

```
 public static void RegisterBundles(BundleCollection bundles)
        {
            bundles.Add(new ScriptBundle("~/bundles/modernizr").Include(
                "~/Scripts/Vendors/modernizr.js"));

            bundles.Add(new ScriptBundle("~/bundles/vendors").Include(
                "~/Scripts/Vendors/jquery.js",
                "~/Scripts/Vendors/bootstrap.js",
                "~/Scripts/Vendors/toastr.js",
                "~/Scripts/Vendors/jquery.raty.js",
                "~/Scripts/Vendors/respond.src.js",
                "~/Scripts/Vendors/angular.js",
                "~/Scripts/Vendors/angular-route.js",
                "~/Scripts/Vendors/angular-cookies.js",
                "~/Scripts/Vendors/angular-validator.js",
                "~/Scripts/Vendors/angular-base64.js",
                "~/Scripts/Vendors/angular-file-upload.js",
                "~/Scripts/Vendors/angucomplete-alt.min.js",
                "~/Scripts/Vendors/ui-bootstrap-tpls-0.13.1.js",
                "~/Scripts/Vendors/underscore.js",
                "~/Scripts/Vendors/raphael.js",
                "~/Scripts/Vendors/morris.js",
                "~/Scripts/Vendors/jquery.fancybox.js",
                "~/Scripts/Vendors/jquery.fancybox-media.js",
                "~/Scripts/Vendors/loading-bar.js"
                ));

            bundles.Add(new ScriptBundle("~/bundles/spa").Include(
                "~/Scripts/spa/modules/common.core.js",
                "~/Scripts/spa/modules/common.ui.js",
                "~/Scripts/spa/app.js",
                "~/Scripts/spa/services/apiService.js",
                "~/Scripts/spa/services/notificationService.js",
                "~/Scripts/spa/services/membershipService.js",
                "~/Scripts/spa/services/fileUploadService.js",
                "~/Scripts/spa/layout/topBar.directive.js",
                "~/Scripts/spa/layout/sideBar.directive.js",
                "~/Scripts/spa/layout/customPager.directive.js",
                "~/Scripts/spa/directives/rating.directive.js",
                "~/Scripts/spa/directives/availableMovie.directive.js",
                "~/Scripts/spa/account/loginCtrl.js",
                "~/Scripts/spa/account/registerCtrl.js",
                "~/Scripts/spa/home/rootCtrl.js",
                "~/Scripts/spa/home/indexCtrl.js",
                "~/Scripts/spa/customers/customersCtrl.js",
                "~/Scripts/spa/customers/customersRegCtrl.js",
                "~/Scripts/spa/customers/customerEditCtrl.js",
                "~/Scripts/spa/movies/moviesCtrl.js",
                "~/Scripts/spa/movies/movieAddCtrl.js",
                "~/Scripts/spa/movies/movieDetailsCtrl.js",
                "~/Scripts/spa/movies/movieEditCtrl.js",
                "~/Scripts/spa/controllers/rentalCtrl.js",
                "~/Scripts/spa/rental/rentMovieCtrl.js",
                "~/Scripts/spa/rental/rentStatsCtrl.js"
                ));

            bundles.Add(new StyleBundle("~/Content/css").Include(
                "~/content/css/site.css",
                "~/content/css/bootstrap.css",
                "~/content/css/bootstrap-theme.css",
                 "~/content/css/font-awesome.css",
                "~/content/css/morris.css",
                "~/content/css/toastr.css",
                "~/content/css/jquery.fancybox.css",
                "~/content/css/loading-bar.css"));

            BundleTable.EnableOptimizations = false;
        }
```

You may wonder where the heck I will find all these files. Do not worry about that, I will provide you links for all the static JavaScript libraries inside the ~/bundles/vendors bundle and the CSS stylesheets in the ~/Content/css one as well. All JavaScript files in the ~/bundles/spa bundle are the angularJS components we are going to build. Don't forget that as always the source code for this application will be available at the end of this post. Add the Bootstrapper class in the App_Start folder. We will call its *Run* method from the **Global.asax** *ApplicationStart* method. For now let the Automapper's part commented out and we will un-comment it when the time comes.

Bootstrapper.cs

```
 public class Bootstrapper
    {
        public static void Run()
        {
            // Configure Autofac
            AutofacWebapiConfig.Initialize(GlobalConfiguration.Configuration);
            //Configure AutoMapper
            //AutoMapperConfiguration.Configure();
        }
    }
```

Change Global.asax.cs (*add it if not exists*) as follow:

Global.asax.cs

```
public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            var config = GlobalConfiguration.Configuration;

            AreaRegistration.RegisterAllAreas();
            WebApiConfig.Register(config);
            Bootstrapper.Run();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            GlobalConfiguration.Configuration.EnsureInitialized();
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
```

Not all pages will be accessible to unauthenticated users as opposed from application requirements and for this reason we are going to use Basic Authentication. This will be done through a **Message Handler** whose job is to search for an *Authorization* header in the request. Create a folder named *Infrastructure* at the root of the web application project and add a sub-folder named *MessageHandlers*. Create the following handler.

Infranstructure/MessageHandlers/HomeCinemaAuthHandler.cs

```
public class HomeCinemaAuthHandler : DelegatingHandler
    {
        IEnumerable<string> authHeaderValues = null;
        protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request, CancellationToken cancellationToken)
        {
            try
            {
                request.Headers.TryGetValues("Authorization",out authHeaderValues);
                if(authHeaderValues == null)
                    return base.SendAsync(request, cancellationToken); // cross fingers

                var tokens = authHeaderValues.FirstOrDefault();
                tokens = tokens.Replace("Basic","").Trim();
                if (!string.IsNullOrEmpty(tokens))
                {
                    byte[] data = Convert.FromBase64String(tokens);
                    string decodedString = Encoding.UTF8.GetString(data);
                    string[] tokensValues = decodedString.Split(':');
                    var membershipService = request.GetMembershipService();
```

```
                        var membershipCtx = membershipService.ValidateUser(tokensValues[0],
tokensValues[1]);
                        if (membershipCtx.User != null)
                        {
                            IPrincipal principal = membershipCtx.Principal;
                            Thread.CurrentPrincipal = principal;
                            HttpContext.Current.User = principal;
                        }
                        else // Unauthorized access - wrong crededentials
                        {
                            var response = new
HttpResponseMessage(HttpStatusCode.Unauthorized);
                            var tsc = new TaskCompletionSource<HttpResponseMessage>();
                            tsc.SetResult(response);
                            return tsc.Task;
                        }
                    }
                    else
                    {
                        var response = new HttpResponseMessage(HttpStatusCode.Forbidden);
                        var tsc = new TaskCompletionSource<HttpResponseMessage>();
                        tsc.SetResult(response);
                        return tsc.Task;
                    }
                    return base.SendAsync(request, cancellationToken);
                }
                catch
                {
                    var response = new HttpResponseMessage(HttpStatusCode.Forbidden);
                    var tsc = new TaskCompletionSource<HttpResponseMessage>();
                    tsc.SetResult(response);
                    return tsc.Task;
                }
            }
        }
```

The *GetMembershipService()* is an HttpRequestMessage extension which I will provide right away. Notice that we've made some decisions in this handler. When a request dispatches, the handler searches for an *Authorization* header. If it doesn't find one then we cross our fingers and let the next level decide if the request is accessible or not. This means that if a Web API Controller's action has the *AllowAnonymous* attribute the request doesn't have to hold an Authorization header. One the other hand if the action did have an *Authorize* attribute then an ***Unauthorized*** response message would be returned in case of empty Authorization header. If Authorization header is present, the membership service decodes the based64 encoded credentials and checks their validity. User and role information is saved in the HttpContext.Current.User object.

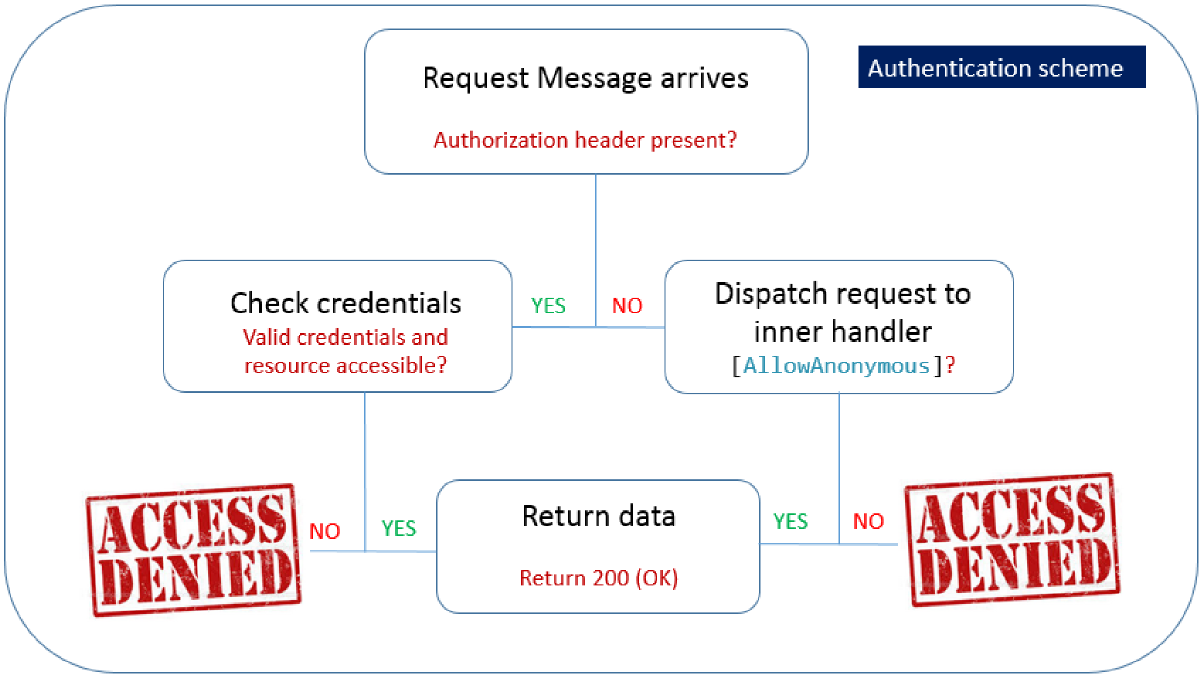


Figure 16. Authentication

As far as the HttpRequestMessage extension add a folder named *Extensions* inside the Infrastructure folder and create the following class.

Infrastructure/Extensions/RequestMessageExtensions.cs

```
public static class RequestMessageExtensions
    {
        internal static IMembershipService GetMembershipService(this HttpRequestMessage
request)
        {
            return request.GetService<IMembershipService>();
        }

        private static TService GetService<TService>(this HttpRequestMessage request)
        {
            IDependencyScope dependencyScope = request.GetDependencyScope();
            TService service = (TService)dependencyScope.GetService(typeof(TService));

            return service;
        }
    }
```

Now switch to the WebApiConfig.cs inside the App_Start folder and register the authentication message handler.

WebApiConfig.cs

```
public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services
            config.MessageHandlers.Add(new HomeCinemaAuthHandler());

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
```

Don't forget to add the same connection string you added in the *HomeCinema.Data* App.config file, in the Web.config configuration file.

Static CSS files and images

We configured previously the bundling but at this point you don't have all the necessary files and their respective folders in your application. Add a *Content* folder at the root of the web application and create the following sub-folders in it.

CSS - Fonts - Images:

1. ***Content/css***: Bootstrap, Font-awesome, fancy-box, morris, toastr, homecinema relative css files
2. ***Content/fonts***: Bootstrap and font-awesome required fonts
3. ***Content/images***
4. ***Content/images/movies***: Hold's application's movies
5. ***Content/images/raty***: Raty.js required images

You can find and download all these static files from here.

Vendors - 3rd party libraries

Create a *Scripts* folder in application's root and a two sub-folders, ***Scripts***/spa and ***Scripts/vendors***. Each of those libraries solves a specific requirement and it was carefully picked. Before providing you some basic information for most of them, let me point out something important.

Always pick 3rd libraries carefully. When searching for a specific component you may find several implementations for it out on the internet. One thing to care about is do not end up with a bazooka while trying to kill a mosquito. For example, in this application we need modal popups. If you search on the internet you will find various implementations but most of these are quite complex. All we need is a modal window so for this I decided that the $modal service of angularJS **UI-bootstrap** is more than enough to do the job. Another thing that you should do is always check for any opened issues for the 3rd library. Those libraries are usually hosted on *Github* and there is an Issues section that you can view.

Check if any performance or memory leaks issue exists before decide to attach any library to your project.

3rd party libraries:

1. toastr.js: A non-blocking notification library
2. jquery.raty.js</a>: A star rating plug-in
3. angular-validator.js: A light weighted validation directive
4. angular-base64.js: Base64 encode-decode library
5. angular-file-upload.js: A file-uploading library
6. angucomplete-alt.min.js: Auto-complete search directive
7. ui-bootstrap-tpls-0.13.1.js: Native AngularJS (Angular) directives for Bootstrap
8. morris.js: Charts library
9. jquery.fancybox.js: Tool for displaying images

These are the most important libraries we will be using but you may find some other files too. Download all those files from here. As I mentioned, generally you should use Bower and **Grunt** or **Gulp** for resolving web dependencies so let me provide you some installation commands for the above libraries:

- bower install angucomplete-alt --save
- bower install angular-base64
- bower install angular-file-upload
- bower install tg-angular-validator
- bower install bootstrap
- bower install raty
- bower install angular-loading-bar
- bower install angular-bootstrap

AngularJS Setup

The ng-view

Now that we are done configurating the Single Page Application it's time to create its initial page, the page that will be used as the *parent* of all different views and templates in our application. Inside the Controllers folder, create an MVC Controller named *HomeController* as follow. Right click inside its Index method and create a new view with the respective name.

HomeController.cs

```
public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
```

Alter the *Views/Home/Index.cshtml* file as follow:

Views/Home/Index.cshtml

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html ng-app="homeCinema">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home Cinema</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body ng-controller="rootCtrl">
    <top-bar></top-bar>
    <div class="row-offcanvas row-offcanvas-left">
        <side-bar></side-bar>
        <div id="main">
            <div class="col-md-12">
                <p class="visible-xs">
                    <button type="button" class="btn btn-primary btn-xs" data-
toggle="offcanvas"><i class="glyphicon glyphicon-chevron-left"></i></button>
                </p>
                <div class="page {{ pageClass }}" ng-view></div>
            </div>
        </div>
    </div><!--/row-offcanvas -->
    @Scripts.Render("~/bundles/vendors")
    @Scripts.Render("~/bundles/spa")

</body>
</html>
```

Let's explain one by one the highlighted lines, from top to bottom. The main module in our spa will be named *homeCinema*. This module will have two other modules as dependencies, the Common.core and the Common.UI which we will create later. Next we render the CSS bundles we created before. There will be one root-parent controller in the application named *rootCtrl*. The top-bar and side-bar elements are custom angularJS directives what will build for the top and side bar collapsible menus respectively. We use a *page-class* $scope variable to change the style in a rendered template. This is a nice trick to render different styles in ng-view templates. Last but not least, we render the *vendors* and spa JavaScript bundles.

From what we have till now, it's obvious that first we should create the **homeCinema** module, the **rootCtrl** controller then the two custom directives for this initial page to be rendered. Instead of doing this, we'll take a step back and prepare some basic angularJS components, the Common.core and Common.ui modules. Add a *Modules* folder in the *Scripts/spa* folder and create the following two files.

spa/modules/common.ui.js

```
(function () {
    'use strict';

    angular.module('common.ui', ['ui.bootstrap', 'chieffancypants.loadingBar']);

})();
```

spa/modules/cmmon.core.js

```
(function () {
       'use strict';

       angular.module('common.core', ['ngRoute', 'ngCookies', 'base64',
'angularFileUpload', 'angularValidator', 'angucomplete-alt']);

})();
```

Let Common.ui module be a UI related reusable component through our SPA application and Common.core a core functional one. We could just inject all those dependencies directly to the *homeCinema* module but that would require to do the same in case we wanted to scale the application as we'll discuss later. Pay some attention the way we created the modules. We will be using this pattern a lot for not polluting the global JavaScript's namespace. At the root of the spa folder add the following app.js file and define the core **homeCinema** module and its routes.

spa/app.js

```
(function () {
    'use strict';
    angular.module('homeCinema', ['common.core', 'common.ui'])
        .config(config);

    config.$inject = ['$routeProvider'];
    function config($routeProvider) {
        $routeProvider
            .when("/", {
                templateUrl: "scripts/spa/home/index.html",
                controller: "indexCtrl"
            })
            .when("/login", {
                templateUrl: "scripts/spa/account/login.html",
                controller: "loginCtrl"
            })
            .when("/register", {
                templateUrl: "scripts/spa/account/register.html",
                controller: "registerCtrl"
            })
            .when("/customers", {
                templateUrl: "scripts/spa/customers/index.html",
                controller: "customersCtrl"
            })
            .when("/customers/register", {
                templateUrl: "scripts/spa/customers/register.html",
                controller: "customersRegCtrl"
            })
            .when("/movies", {
                templateUrl: "scripts/spa/movies/index.html",
                controller: "moviesCtrl"
            })
            .when("/movies/add", {
                templateUrl: "scripts/spa/movies/add.html",
                controller: "movieAddCtrl"
            })
            .when("/movies/:id", {
                templateUrl: "scripts/spa/movies/details.html",
                controller: "movieDetailsCtrl"
            })
            .when("/movies/edit/:id", {
                templateUrl: "scripts/spa/movies/edit.html",
                controller: "movieEditCtrl"
            })
            .when("/rental", {
                templateUrl: "scripts/spa/rental/index.html",
                controller: "rentStatsCtrl"
            }).otherwise({ redirectTo: "/" });
    }
})();
```

Once again take a look at the **explicit service** injection using the angularJS property annotation $inject which allows the minifiers to rename the function parameters and still be able to inject the right services. You probably don't have all the referenced files in the previous script (*except if you have downloaded the source code*) but that's OK. We'll create all of those one by one. It is common practice to place any angularJS components related to application's layout in a layout folder so go ahead and create this folder. For the **side-bar** element directive we used we need two files, one for the directive definition and another for its template. Add the following two files to the layout folder you created.

spa/layout/sideBar.directive.js

```
(function(app) {
    'use strict';

    app.directive('sideBar', sideBar);

    function sideBar() {
        return {
            restrict: 'E',
            replace: true,
            templateUrl: '/scripts/spa/layout/sideBar.html'
        }
    }

})(angular.module('common.ui'));
```

spa/layout/sidebar.html

```
<div id="sidebar" class="sidebar-offcanvas">
    <div class="col-md-12">
        <h3></h3>
        <ul class="nav nav-pills nav-stacked">
            <li class="nav-divider"></li>
            <li class=""><a ng-href="#/">Home<i class="fa fa-home fa-fw pull-
right"></i></a></li>
            <li class="nav-divider"></li>
            <li><a ng-href="#/customers/">Customers<i class="fa fa-users fa-fw pull-
right"></i></a></li>
            <li><a ng-href="#/customers/register">Register customer<i class="fa fa-
user-plus fa-fw pull-right"></i></a></li>
            <li class="nav-divider"></li>
            <li><a ng-href="#/movies/">Movies<i class="fa fa-film fa-fw pull-
right"></i></a></li>
            <li><a ng-href="#/movies/add">Add movie<i class="fa fa-plus-circle fa-fw
pull-right"></i></a></li>
            <li class="nav-divider"></li>
            <li><a ng-href="#/rental/">Rental history<i class="fa fa-leanpub fa-fw
pull-right"></i></a></li>
            <li class="nav-divider"></li>
            <li><a ng-href="#/login" ng-if="!userData.isUserLoggedIn">Login<i class="fa
fa-sign-in fa-fw pull-right"></i></a></li>
            <li><button type="button" class="btn btn-danger btn-sm" ng-
click="logout();" ng-if="userData.isUserLoggedIn">Logout<i class="fa fa-sign-out fa-fw
pull-right"></i></a></li>
        </ul>
    </div>
</div>
```

This directive will create a collapsible side-bar such as the following when applied.

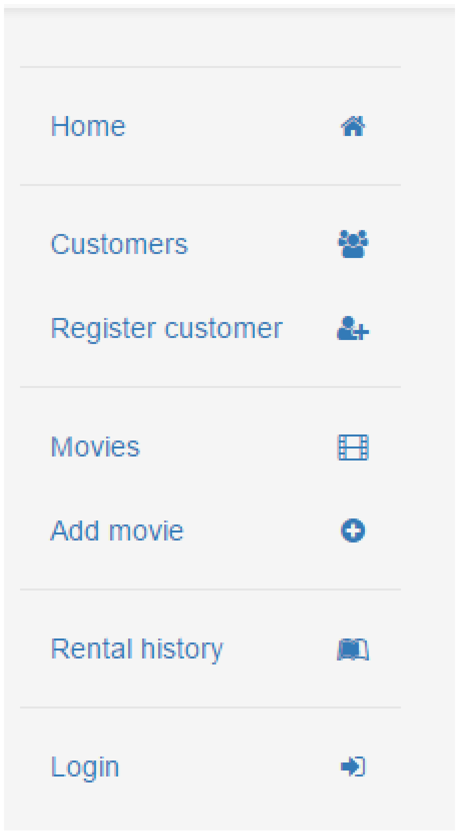


Figure 17. side-bar

Let's see the top-bar directive and its template as well.

spa/layout/topBar.directive.js

```
(function(app) {
    'use strict';

    app.directive('topBar', topBar);

    function topBar() {
        return {
            restrict: 'E',
            replace: true,
            templateUrl: '/scripts/spa/layout/topBar.html'
        }
    }

})(angular.module('common.ui'));
```

spa/layout/topBar.html

```
<div class="navbar navbar-default navbar-fixed-top">
    <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
            <span class="icon-bar"></span>
```

```
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand active" href="#/">Home Cinema</a>
        </div>
        <div class="collapse navbar-collapse">
            <ul class="nav navbar-nav">
                <li><a href="#about">About<i class="fa fa-info-circle fa-fw pull-
right"></i></a></li>
            </ul>
            <ul class="nav navbar-nav navbar-right" ng-if="userData.isUserLoggedIn">
                <li class="userinfo"><a href="#/">{{username}}<i class="fa fa-user fa-
fw"></i></a></li>
            </ul>
        </div>
</div>
```

You may have noticed I highlighted two lines in the side-bar template's code. Those lines are responsible to show or hide the login and log-off buttons respectively depending if the user is logged in or not. The required functionality will be place at the rootCtrl Controller inside a home folder.

spa/home/rootCtrl.js

```
(function (app) {
    'use strict';

    app.controller('rootCtrl', rootCtrl);

    function rootCtrl($scope) {

        $scope.userData = {};

        $scope.userData.displayUserInfo = displayUserInfo;
        $scope.logout = logout;


        function displayUserInfo() {
        }

        function logout() {
        }

        $scope.userData.displayUserInfo();
    }

})(angular.module('homeCinema'));
```

We will update its contents as soon as we create the membership service. All of our views require to fetch data from the server and for that a specific apiService will be used through our application. This service will also be able to display some kind of notifications to the user so let's build a notificationService as well. Create a *services* folder under the *spa* and add the following angularJS **factory** services.

spa/services/notificationService.js

```
(function (app) {
    'use strict';

    app.factory('notificationService', notificationService);

    function notificationService() {

        toastr.options = {
            "debug": false,
            "positionClass": "toast-top-right",
            "onclick": null,
            "fadeIn": 300,
            "fadeOut": 1000,
            "timeOut": 3000,
            "extendedTimeOut": 1000
        };

        var service = {
            displaySuccess: displaySuccess,
            displayError: displayError,
            displayWarning: displayWarning,
            displayInfo: displayInfo
        };

        return service;

        function displaySuccess(message) {
            toastr.success(message);
        }

        function displayError(error) {
            if (Array.isArray(error)) {
                error.forEach(function (err) {
                    toastr.error(err);
                });
            } else {
                toastr.error(error);
            }
        }

        function displayWarning(message) {
            toastr.warning(message);
        }

        function displayInfo(message) {
            toastr.info(message);
        }

    }

})(angular.module('common.core'));
```

The notificationService is based on the toastr.js notification library. It displays different type (*style class*) of notifications depending on the method invoked, which is success, error, warning and info.

spa/services/apiService.js

```
(function (app) {
    'use strict';

    app.factory('apiService', apiService);

    apiService.$inject = ['$http', '$location', 'notificationService','$rootScope'];

    function apiService($http, $location, notificationService, $rootScope) {
        var service = {
            get: get,
            post: post
        };

        function get(url, config, success, failure) {
            return $http.get(url, config)
                    .then(function (result) {
                        success(result);
                    }, function (error) {
                        if (error.status == '401') {
                            notificationService.displayError('Authentication
required.');
                            $rootScope.previousState = $location.path();
                            $location.path('/login');
                        }
                        else if (failure != null) {
                            failure(error);
                        }
                    });
        }

        function post(url, data, success, failure) {
            return $http.post(url, data)
                    .then(function (result) {
                        success(result);
                    }, function (error) {
                        if (error.status == '401') {
                            notificationService.displayError('Authentication
required.');
                            $rootScope.previousState = $location.path();
                            $location.path('/login');
                        }
                        else if (failure != null) {
                            failure(error);
                        }
                    });
        }

        return service;
    }

})(angular.module('common.core'));
```

The **apiService** is quite straight forward. It defines a factory with two basic methods, GET and POST. Both of these methods can handle 401 errors by redirecting the user at the login view and saving the previous state so that after a successful login, the user gets back where he/she was. They also accept a required *success* callback to invoke and an optional *failure* one in case of a failed request. Our spa application is pretty much ready to fetch and post data from the server so this is the right time to write the first Web API controller.

Web API Controllers Setup

Base Web API Controller class

We’ll try to apply some basic rules for all of Web API Controllers in our application. The first one is that all of them will inherit from a base class named ApiControllerBase. The basic responsibility of this class will be handling the Error logging functionality. That's the only class where instances of IEntityBaseRepository<Error> will be injected with the centralized *Try, Catch* point we talked about at the start of this post. This class of course will inherit from the ApiController. Create a folder named *core* inside the *Infrastructure* and create the base class for our controllers.

ApiControllerBase.cs

```
public class ApiControllerBase : ApiController
    {
        protected readonly IEntityBaseRepository<Error> _errorsRepository;
        protected readonly IUnitOfWork _unitOfWork;

        public ApiControllerBase(IEntityBaseRepository<Error> errorsRepository,
IUnitOfWork unitOfWork)
        {
            _errorsRepository = errorsRepository;
            _unitOfWork = unitOfWork;
        }

        protected HttpResponseMessage CreateHttpResponse(HttpRequestMessage request,
Func<HttpResponseMessage> function)
        {
            HttpResponseMessage response = null;

            try
            {
                response = function.Invoke();
            }
            catch (DbUpdateException ex)
            {
                LogError(ex);
                response = request.CreateResponse(HttpStatusCode.BadRequest,
ex.InnerException.Message);
            }
            catch (Exception ex)
            {
                LogError(ex);
                response = request.CreateResponse(HttpStatusCode.InternalServerError,
ex.Message);
            }

            return response;
        }

        private void LogError(Exception ex)
        {
            try
            {
                Error _error = new Error()
                {
                    Message = ex.Message,
                    StackTrace = ex.StackTrace,
                    DateCreated = DateTime.Now
                };

                _errorsRepository.Add(_error);
                _unitOfWork.Commit();
            }
            catch { }
        }
    }
```

You will surprised how powerful the *CreateHttpResponse* function can be when we reach the discussion section. Notice that this method can handle a DbUpdateException exception as well. You can omit this type if you want and write more custom methods such as this. Each controller's action will start by calling this base method.

ViewModels & Validators

If you recall, the home's page displays the latest movies released plus some genre statistics on the right. Let's start from the latest movies. First thing we need to do is create a ViewModel for *Movie* entities. For each type of Entity we'll create the respective ViewModel for the client. All ViewModels will have the relative validation rules based on the FluentValidation Nuget package. Add the MovieViewModel and GenreViewModel classes inside the *Models* folder.

MovieViewModel.cs

```
    [Bind(Exclude = "Image")]
    public class MovieViewModel : IValidatableObject
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Image { get; set; }
        public string  Genre { get; set; }
        public int GenreId { get; set; }
        public string Director { get; set; }
        public string Writer { get; set; }
        public string Producer { get; set; }
        public DateTime ReleaseDate { get; set; }
        public byte Rating { get; set; }
        public string TrailerURI { get; set; }
        public bool IsAvailable { get; set; }

        public IEnumerable<ValidationResult> Validate(ValidationContext 
validationContext)
        {
            var validator = new MovieViewModelValidator();
            var result = validator.Validate(this);
            return result.Errors.Select(item => new ValidationResult(item.ErrorMessage, 
new[] { item.PropertyName }));
        }
    }
```

We excluded the Image property from **MovieViewModel** binding cause we will be using a specific FileUpload action to upload images.

GenreViewModel.cs

```
public class GenreViewModel
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public int NumberOfMovies { get; set; }
    }
```

Create a *Validators* folder inside the *Infrastructure* and the *MovieValidator*.

MovieViewModelValidator.cs

```
public class MovieViewModelValidator : AbstractValidator<MovieViewModel>
    {
        public MovieViewModelValidator()
        {
            RuleFor(movie => movie.GenreId).GreaterThan(0)
                .WithMessage("Select a Genre");

            RuleFor(movie => movie.Director).NotEmpty().Length(1,100)
                .WithMessage("Select a Director");

            RuleFor(movie => movie.Writer).NotEmpty().Length(1,50)
                .WithMessage("Select a writer");

            RuleFor(movie => movie.Producer).NotEmpty().Length(1, 50)
                .WithMessage("Select a producer");

            RuleFor(movie => movie.Description).NotEmpty()
                .WithMessage("Select a description");

            RuleFor(movie => movie.Rating).InclusiveBetween((byte)0, (byte)5)
                .WithMessage("Rating must be less than or equal to 5");

            RuleFor(movie => movie.TrailerURI).NotEmpty().Must(ValidTrailerURI)
                .WithMessage("Only Youtube Trailers are supported");
        }

        private bool ValidTrailerURI(string trailerURI)
        {
            return (!string.IsNullOrEmpty(trailerURI) && 
trailerURI.ToLower().StartsWith("https://www.youtube.com/watch?"));
        }
    }
```

Automapper configuration

Now that we have our first ViewModels and its validator setup, we can configure the Automapper mappings as well. Add a *Mappings* folder inside the *Infrastructure* and create the following DomainToViewModelMappingProfile Profile class.

DomainToViewModelMappingProfile.cs

```
public class DomainToViewModelMappingProfile : Profile
    {
        public override string ProfileName
        {
            get { return "DomainToViewModelMappings"; }
        }

        protected override void Configure()
        {
            Mapper.CreateMap<Movie, MovieViewModel>()
                .ForMember(vm => vm.Genre, map => map.MapFrom(m => m.Genre.Name))
                .ForMember(vm => vm.GenreId, map => map.MapFrom(m => m.Genre.ID))
                .ForMember(vm => vm.IsAvailable, map => map.MapFrom(m => m.Stocks.Any(s
=> s.IsAvailable)));

            Mapper.CreateMap<Genre, GenreViewModel>()
                .ForMember(vm => vm.NumberOfMovies, map => map.MapFrom(g =>
g.Movies.Count()));
        }
    }
```

Notice how we set if a Movie (ViewModel) is available or not by checking if any of its stocks is available. Add Automapper's configuration class and make sure to comment out the respective line in the *Bootstrapper* class.

AutoMapperConfiguration.cs

```
public class AutoMapperConfiguration
    {
        public static void Configure()
        {
            Mapper.Initialize(x =>
            {
                x.AddProfile<DomainToViewModelMappingProfile>();
            });
        }
    }
```

Bootstrapper.cs

```
public class Bootstrapper
    {
        public static void Run()
        {
            // Configure Autofac
            AutofacWebapiConfig.Initialize(GlobalConfiguration.Configuration);
            //Configure AutoMapper
            AutoMapperConfiguration.Configure();
        }
    }
```

Features

We have done so much preparation and now it is time to implement and view all application requirements in practice. We will start with the Home page.

The Home page

The home page displays information about the latest DVD movie released plus some Genre statistics. For the first feature will start from creating the required Web API controller, **MoviesController**. Add this class inside the *Controllers* folder.

MoviesController.cs

```
    [Authorize(Roles = "Admin")]
    [RoutePrefix("api/movies")]
    public class MoviesController : ApiControllerBase
    {
        private readonly IEntityBaseRepository<Movie> _moviesRepository;
        private readonly IEntityBaseRepository<Rental> _rentalsRepository;
        private readonly IEntityBaseRepository<Stock> _stocksRepository;
        private readonly IEntityBaseRepository<Customer> _customersRepository;

        public MoviesController(IEntityBaseRepository<Movie> moviesRepository,
            IEntityBaseRepository<Rental> rentalsRepository,
IEntityBaseRepository<Stock> stocksRepository,
            IEntityBaseRepository<Customer> customersRepository,
            IEntityBaseRepository<Error> _errorsRepository, IUnitOfWork _unitOfWork)
            : base(_errorsRepository, _unitOfWork)
        {
            _moviesRepository = moviesRepository;
            _rentalsRepository = rentalsRepository;
            _stocksRepository = stocksRepository;
            _customersRepository = customersRepository;
        }

        [AllowAnonymous]
        [Route("latest")]
        public HttpResponseMessage Get(HttpRequestMessage request)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;
                var movies = _moviesRepository.GetAll().OrderByDescending(m =>
m.ReleaseDate).Take(6).ToList();

                IEnumerable<MovieViewModel> moviesVM = Mapper.Map<IEnumerable<Movie>,
IEnumerable<MovieViewModel>>(movies);

                response =
request.CreateResponse<IEnumerable<MovieViewModel>>(HttpStatusCode.OK, moviesVM);

                return response;
            });
        }
    }
```

Let's explain the highlighted lines from top to bottom. All actions for this Controller required the user not only to be authenticated but also belong to *Admin* role, except if AllowAnonymous attribute is applied. All requests to this controller will start with a prefix of ***api/movies***. The error handling as already explained is handled from the base class ApiControllerBase and its method CreateHttpResponse. Here we can see for the first time how this method is actually called. Add the GenresController as well.

GenresController.cs

```
    [Authorize(Roles = "Admin")]
    [RoutePrefix("api/genres")]
    public class GenresController : ApiControllerBase
    {
        private readonly IEntityBaseRepository<Genre> _genresRepository;

        public GenresController(IEntityBaseRepository<Genre> genresRepository,
             IEntityBaseRepository<Error> _errorsRepository, IUnitOfWork _unitOfWork)
            : base(_errorsRepository, _unitOfWork)
        {
            _genresRepository = genresRepository;
        }

        [AllowAnonymous]
        public HttpResponseMessage Get(HttpRequestMessage request)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;
                var genres = _genresRepository.GetAll().ToList();

                IEnumerable<GenreViewModel> genresVM = Mapper.Map<IEnumerable<Genre>,
 IEnumerable<GenreViewModel>>(genres);

                response =
 request.CreateResponse<IEnumerable<GenreViewModel>>(HttpStatusCode.OK, genresVM);

                return response;
            });
        }
    }
```

We prepared the server side part, let's move on to its JavaScript one now. If you recall we'll follow a structure by feature in our spa, so we will place the two required files for the home page inside the spa/home folder. We need two files, one template and the respective controller. Let's see the template first.

spa/home/index.html

```
<hr />
<div class="row">
    <div class="col-md-8">
        <div class="panel panel-primary" id="panelLatestMovies">
            <div class="panel-heading">
```

```
                <h3 class="panel-title">Latest Movies Released</h3>
            </div>

            <div class="panel-body">
                <div ng-if="loadingMovies">
                    <div class="col-xs-4"></div>
                    <div class="col-xs-4">
                        <i class="fa fa-refresh fa-5x fa-spin"></i> <label class="label
label-primary">Loading movies...</label>
                    </div>
                    <div class="col-xs-4"></div>
                </div>
                <div class="col-xs-12 col-sm-6 movie" ng-repeat="movie in
latestMovies">
                    <div class="panel panel-default">
                        <div class="panel-heading">
                            <strong>{{movie.Title}} </strong>
                        </div>
                        <div class="panel-body">
                            <div class="media">
                                <a class="pull-left" href="#">
                                    <a class="fancybox pull-left" rel="gallery1" ng-
href="../../Content/images/movies/{{movie.Image}}" title="{{movie.Description |
limitTo:200}}">
                                        <img class="media-object" height="120" ng-
src="../../Content/images/movies/{{movie.Image}}" alt="" />
                                    </a>

                                </a>
                                <div class="media-body">
                                    <available-movie is-
available="{{movie.IsAvailable}}"></available-movie>
                                    <div><small>{{movie.Description | limitTo:
70}}...</small></div>
                                    <label class="label label-
info">{{movie.Genre}}</label><br />
                                </div>
                                <br />
                            </div>
                        </div>
                        <div class="panel-footer">
                            <span component-rating="{{movie.Rating}}"></span>
                            <a class="fancybox-media pull-right" ng-
href="{{movie.TrailerURI}}">Trailer<i class="fa fa-video-camera fa-fw"></i></a>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="panel panel-success" id="panelMovieGenres">
            <div class="panel-heading">
                <h3 class="panel-title">Movies Genres</h3>
            </div>

            <div class="panel-body">
                <div ng-if="loadingGenres">
```

```
                    <div class="col-xs-4"></div>
                    <div class="col-xs-4"><i class="fa fa-refresh fa-5x fa-spin"></i>
<label class="label label-primary">Loading Genres..</label></div>
                    <div class="col-xs-4"></div>
                </div>
                <div id="genres-bar"></div>
            </div>
            <div class="panel-footer">
                <p class="text-center"><em>Wanna add a new Movie? Head over to the add
movie form.</em></p>
                <p class="text-center"><a ng-href="#/movies/add" class="btn btn-
default">Add new Movie</a></p>
            </div>
        </div>
    </div>
</div>
```

I made a convention that the first view rendered for each template will be named *index.html*. This means that you will see later the *movies/index.html*, *rental/index.html* etc... We use ng-if angularJS **directive** to display a loader (spinner if you prefer) till server side data retrieved from the server. Let's see now the controller that binds the data to the template, the indexCtrl. Add the following file to the *home* folder as well.

spa/home/indexCtrl.js

```
(function (app) {
    'use strict';

    app.controller('indexCtrl', indexCtrl);

    indexCtrl.$inject = ['$scope', '$location', 'apiService','notificationService'];

    function indexCtrl($scope, $location, apiService, notificationService) {
        $scope.pageClass = 'page-home';
        $scope.loadingMovies = true;
        $scope.loadingGenres = true;
        $scope.isReadOnly = true;

        $scope.latestMovies = [];

        apiService.get('/api/movies/latest', null,
            moviesLoadCompleted,
            moviesLoadFailed);

        apiService.get("/api/genres/", null,
            genresLoadCompleted,
            genresLoadFailed);

        function moviesLoadCompleted(result) {
            $scope.latestMovies = result.data;
            $scope.loadingMovies = false;
        }

        function genresLoadFailed(response) {
            notificationService.displayError(response.data);
        }
        function moviesLoadFailed(response) {
            notificationService.displayError(response.data);
        }

        function genresLoadCompleted(result) {
            var genres = result.data;
            Morris.Bar({
                element: "genres-bar",
                data: genres,
                xkey: "Name",
                ykeys: ["NumberOfMovies"],
                labels: ["Number Of Movies"],
                barRatio: 0.4,
                xLabelAngle: 55,
                hideHover: "auto",
                resize: 'true'
            });

            $scope.loadingGenres = false;
        }
    }

})(angular.module('homeCinema'));
```

The ***loadData()*** function requests movie and genre data from the respective Web API controllers we previously created. For the movie data only thing needed to do is bind the requested data to a $scope.latestMovies variable. For the genres data thought, we used genres retrieved data and a specific div element, *genres-bar* to create a Morris bar.

Movie Directives

In case you noticed, the index.html template has two custom directives. One to render if the movie is available and another one to display its rating through the raty.js library. Those two directives will be used over and over again through our application so let's take a look at them.

The first one is responsible to render a *label* element that may be red or green depending if the movie is available or not. Since those directives can be used all over the application we'll place their components inside a *directives* folder, so go ahead and add this folder under the spa. Create an **availableMovie.html** file which will be the template for the new directive.

spa/directives/availableMovie.html

```
<label ng-class="getAvailableClass()">{{getAvailability()}}</label>
```

Now add the directive definition, **availableMovie.directive.js**

spa/directives/availableMovie.directive.js

```
(function (app) {
       'use strict';

       app.directive('availableMovie', availableMovie);
```

```
        function availableMovie() {
                return {
                        restrict: 'E',
                        templateUrl: "/Scripts/spa/directives/availableMovie.html",
                        link: function ($scope, $element, $attrs) {
                                $scope.getAvailableClass = function () {
                                        if ($attrs.isAvailable === 'true')
                                                return 'label label-success'
                                        else
                                                return 'label label-danger'
                                };
                                $scope.getAvailability = function () {
                                        if ($attrs.isAvailable === 'true')
                                                return 'Available!'
                                        else
                                                return 'Not Available'
                                };
                        }
                }
        }

})(angular.module('common.ui'));
```

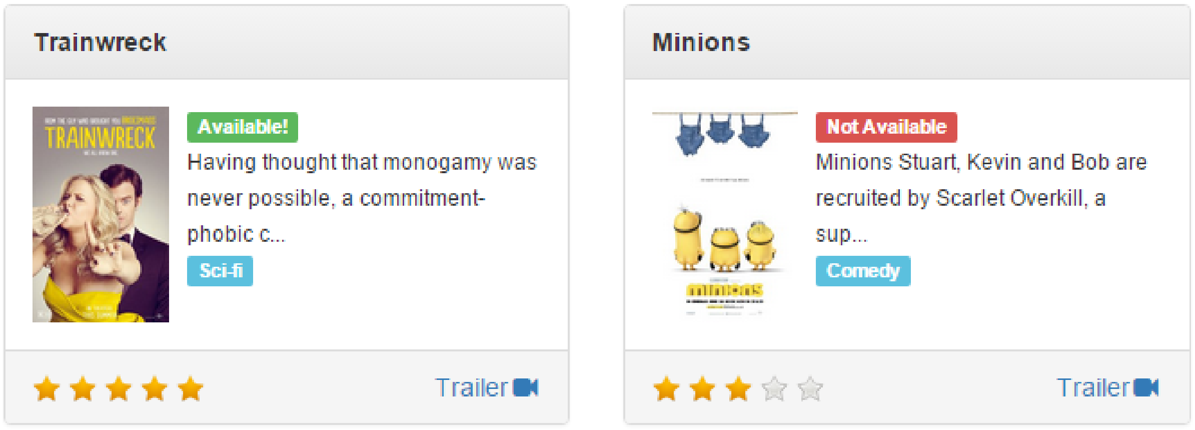

Trainwreck
Available!
Having thought that monogamy was never possible, a commitment-phobic c...
Sci-fi
Trailer
Minions
Not Available
Minions Stuart, Kevin and Bob are recruited by Scarlet Overkill, a sup...
Comedy
Trailer

Figure 18. Available movie directives

The component-rating which displays a star based rating element, is slightly different in terms of restriction, since its used as an element, not as an attribute. I named it *component-rating* cause you may want to use it to rate entities other than movies. When you want to render the rating directive all you have to do is create the following element.

componentRating.directive.js

```
(function(app) {
    'use strict';

    app.directive('componentRating', componentRating);

    function componentRating() {
        return {
            restrict: 'A',
            link: function ($scope, $element, $attrs) {
                $element.raty({
                    score: $attrs.componentRating,
                    halfShow: false,
                    readOnly: $scope.isReadOnly,
                    noRatedMsg: "Not rated yet!",
                    starHalf: "../Content/images/raty/star-half.png",
                    starOff: "../Content/images/raty/star-off.png",
                    starOn: "../Content/images/raty/star-on.png",
                    hints: ["Poor", "Average", "Good", "Very Good", "Excellent"],
                    click: function (score, event) {
                        //Set the model value
                        $scope.movie.Rating = score;
                        $scope.$apply();
                    }
                });
            }
        }
    }

})(angular.module('common.ui'));
```

One important thing the directive needs to know is if the rating element will be editable or not and that is configured through the **readOnly:** $scope.isReadOnly definition. For the *home/index.html* we want the rating to be read-only so the controller has the following declaration:

$scope.isReadOnly *= true;*

Any other controller that requires to edit movie's rating value will set this value to false.

Account

One of the most important parts in every application is how users are getting authenticated in order to access authorized resources. We certainly built a custom membership schema and add a Basic Authentication message handler in Web API, but we haven't yet created either the required Web API AccountController or the relative angularJS component. Let's start with the server side first and view the AccountController.

AccountController.cs

```
    [Authorize(Roles="Admin")]
    [RoutePrefix("api/Account")]
    public class AccountController : ApiControllerBase
    {
        private readonly IMembershipService _membershipService;

        public AccountController(IMembershipService membershipService,
            IEntityBaseRepository<Error> _errorsRepository, IUnitOfWork _unitOfWork)
            : base(_errorsRepository, _unitOfWork)
        {
            _membershipService = membershipService;
        }

        [AllowAnonymous]
        [Route("authenticate")]
        [HttpPost]
        public HttpResponseMessage Login(HttpRequestMessage request, LoginViewModel
user)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                if (ModelState.IsValid)
                {
                    MembershipContext _userContext =
_membershipService.ValidateUser(user.Username, user.Password);

                    if (_userContext.User != null)
                    {
                        response = request.CreateResponse(HttpStatusCode.OK, new {
success = true });
                    }
                    else
                    {
                        response = request.CreateResponse(HttpStatusCode.OK, new {
success = false });
                    }
                }
                else
                    response = request.CreateResponse(HttpStatusCode.OK, new { success
= false });

                return response;
            });
        }

        [AllowAnonymous]
        [Route("register")]
        [HttpPost]
        public HttpResponseMessage Register(HttpRequestMessage request, 
RegistrationViewModel user)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                if (!ModelState.IsValid)
                {
                    response = request.CreateResponse(HttpStatusCode.BadRequest, new { 
success = false });
                }
                else
                {
                    Entities.User _user = _membershipService.CreateUser(user.Username, 
user.Email, user.Password, new int[] { 1 });

                    if (_user != null)
                    {
                        response = request.CreateResponse(HttpStatusCode.OK, new { 
success = true });
                    }
                    else
                    {
                        response = request.CreateResponse(HttpStatusCode.OK, new { 
success = false });
                    }
                }

                return response;
            });
        }
    }
```

User sends a **POST** request to *api/account/authenticate* with their credentials (we'll view the LoginViewModel soon) and the controller validates the user though the MemebershipService. If user's credentials are valid then the returned MembershipContext will contain the relative user's User entity. The registration process works pretty much the same. This time the user posts a request to *api/account/register* (we'll view the RegistrationViewModel later) and if the *ModelState* is valid then the user is created through the MemebershipService's **CreateUser** method. Let's see now both the LoginViewModel and the RegistrationViewModel with their respective validators. Add the ViewModel classes in the *Models* folder and a AccountViewModelValidators.cs file inside the Infrastructure/Validators folder to hold both their validators.

LoginViewModel.cs

```
public class LoginViewModel : IValidatableObject
    {
        public string Username { get; set; }
        public string Password { get; set; }

        public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
        {
            var validator = new LoginViewModelValidator();
            var result = validator.Validate(this);
            return result.Errors.Select(item => new ValidationResult(item.ErrorMessage,
new[] { item.PropertyName }));
        }
    }
```

RegistrationViewModel.cs

```
public class RegistrationViewModel : IValidatableObject
    {
        public string Username { get; set; }
        public string Password { get; set; }
        public string Email { get; set; }

        public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
        {
            var validator = new RegistrationViewModelValidator();
            var result = validator.Validate(this);
            return result.Errors.Select(item => new ValidationResult(item.ErrorMessage,
new[] { item.PropertyName }));
        }
    }
```

Infrastructure/Validators/AccountViewModelValidators.cs

```
public class RegistrationViewModelValidator : AbstractValidator<RegistrationViewModel>
    {
        public RegistrationViewModelValidator()
        {
            RuleFor(r => r.Email).NotEmpty().EmailAddress()
                .WithMessage("Invalid email address");

            RuleFor(r => r.Username).NotEmpty()
                .WithMessage("Invalid username");

            RuleFor(r => r.Password).NotEmpty()
                .WithMessage("Invalid password");
        }
    }

    public class LoginViewModelValidator : AbstractValidator<LoginViewModel>
    {
        public LoginViewModelValidator()
        {
            RuleFor(r => r.Username).NotEmpty()
                .WithMessage("Invalid username");

            RuleFor(r => r.Password).NotEmpty()
                .WithMessage("Invalid password");
        }
    }
```

In the **Front-End** side now, we need to build a *MembershipService* to handle the following:

<u>MembershipService factory:</u>

- Authenticate user through the Login view
- Register a user through the Register view
- Save user's credentials after successful login or registration in a session cookie (***$cookieStore***)
- Remove credentials when user log-off from application
- Checks if user is logged in or not through the relative ***$cookieStore*** repository value

This factory service is mostly depending in the **$cookieStore** service (*ngCookies module*) and in a 3rd party module named '$base64' able to encode - decode strings in base64 format. Logged in user's credentials are saved in a **$rootScope** variable and added as *Authorization* header in each http request. Add the *membershipService*.js file inside the spa/*services* folder.

membershipService.js

```
(function (app) {
    'use strict';

    app.factory('membershipService', membershipService);

    membershipService.$inject = ['apiService', 'notificationService','$http',
'$base64', '$cookieStore', '$rootScope'];

    function membershipService(apiService, notificationService, $http, $base64,
$cookieStore, $rootScope) {

        var service = {
            login: login,
            register: register,
            saveCredentials: saveCredentials,
            removeCredentials: removeCredentials,
            isUserLoggedIn: isUserLoggedIn
        }

        function login(user, completed) {
            apiService.post('/api/account/authenticate', user,
            completed,
            loginFailed);
        }

        function register(user, completed) {
            apiService.post('/api/account/register', user,
            completed,
            registrationFailed);
        }

        function saveCredentials(user) {
            var membershipData = $base64.encode(user.username + ':' + user.password);

            $rootScope.repository = {
                loggedUser: {
                    username: user.username,
                    authdata: membershipData
                }
            };

            $http.defaults.headers.common['Authorization'] = 'Basic ' + membershipData;
            $cookieStore.put('repository', $rootScope.repository);
        }

        function removeCredentials() {
            $rootScope.repository = {};
            $cookieStore.remove('repository');
            $http.defaults.headers.common.Authorization = '';
        };

        function loginFailed(response) {
            notificationService.displayError(response.data);
        }

        function registrationFailed(response) {

            notificationService.displayError('Registration failed. Try again.');
        }

        function isUserLoggedIn() {
            return $rootScope.repository.loggedUser != null;
        }

        return service;
    }


})(angular.module('common.core'));
```

Now that we built this service we are able to handle page refreshes as well. Go ahead and add the **run** configuration for the main module homeCiname inside the app.js file.

part of app.js

```
 (function () {
     'use strict';

     angular.module('homeCinema', ['common.core', 'common.ui'])
         .config(config)
         .run(run);

     // routeProvider code omitted

     run.$inject = ['$rootScope', '$location', '$cookieStore', '$http'];

     function run($rootScope, $location, $cookieStore, $http) {
         // handle page refreshes
```

```
        $rootScope.repository = $cookieStore.get('repository') || {};
        if ($rootScope.repository.loggedUser) {
            $http.defaults.headers.common['Authorization'] =
$rootScope.repository.loggedUser.authdata;
        }

        $(document).ready(function () {
            $(".fancybox").fancybox({
                openEffect: 'none',
                closeEffect: 'none'
            });

            $('.fancybox-media').fancybox({
                openEffect: 'none',
                closeEffect: 'none',
                helpers: {
                    media: {}
                }
            });

            $('[data-toggle=offcanvas]').click(function () {
                $('.row-offcanvas').toggleClass('active');
            });
        });
    }

    isAuthenticated.$inject = ['membershipService', '$rootScope','$location'];

    function isAuthenticated(membershipService, $rootScope, $location) {
        if (!membershipService.isUserLoggedIn()) {
            $rootScope.previousState = $location.path();
            $location.path('/login');
        }
    }

})();
```

I found the opportunity to add same *fancy-box* related initialization code as well. Now that we have the membership functionality configured both for the server and the front-end side, let's proceed to the login and register views with their controllers. Those angularJS components will be placed inside an *account* folder in the *spa*. Here is the login.html template:

spa/account/login.html

```
<div class="row">
    <form class="form-signin" role="form" novalidate angular-validator
name="userLoginForm" angular-validator-submit="login()">
        <h3 class="form-signin-heading">Sign in</h3>
        <div class="form-group">
            <label for="inputUsername" class="sr-only">Username</label>
            <input type="text" id="inputUsername" name="inputUsername" class="form-
control" ng-model="user.username" placeholder="Username"
                   validate-on="blur" required required-message="'Username is
required'">
        </div>
        <div class="form-group">
```

```
            <label for="inputPassword" class="sr-only">Password</label>
            <input type="password" id="inputPassword" name="inputPassword" class="form-
control" ng-model="user.password" placeholder="Password"
                   validate-on="blur" required required-message="'Password is
required'">
        </div>
        <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
        <div class="pull-right">
            <a ng-href="#/register" class="control-label">Register</a>
        </div>
    </form>
</div>
```

Here you can see (*highlighted lines*) for the first time a new library we will be using for validating form controls, the angularValidator.

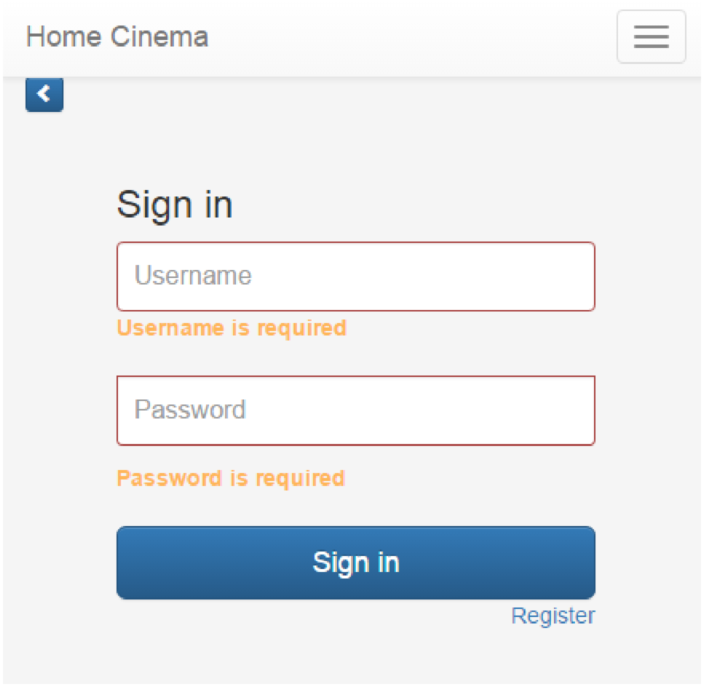


Figure 19. User login form

And now the *loginCtrl* controller.

spa/account/loginCtrl.js

```
(function (app) {
    'use strict';

    app.controller('loginCtrl', loginCtrl);

    loginCtrl.$inject = ['$scope', 'membershipService',
'notificationService','$rootScope', '$location'];

    function loginCtrl($scope, membershipService, notificationService, $rootScope,
$location) {
        $scope.pageClass = 'page-login';
        $scope.login = login;
        $scope.user = {};

        function login() {
            membershipService.login($scope.user, loginCompleted)
        }

        function loginCompleted(result) {
            if (result.data.success) {
                membershipService.saveCredentials($scope.user);
                notificationService.displaySuccess('Hello ' + $scope.user.username);
                $scope.userData.displayUserInfo();
                if ($rootScope.previousState)
                    $location.path($rootScope.previousState);
                else
                    $location.path('/');
            }
            else {
                notificationService.displayError('Login failed. Try again.');
            }
        }
    }

})(angular.module('common.core'));
```

The login function calls the **membershipService's** login and passes a success callback. If the login succeed it does three more things: First, it saves user's credentials through membershipService and then displays logged-in user's info through the *rootCtrl* controller. Finally, checks if the user ended in login view cause authentication required to access another view and if so, redirects him/her to that view. Let me remind you a small part of the apiService.

part of apiService.js

```
if (error.status == '401') {
        notificationService.displayError('Authentication required.');
        $rootScope.previousState = $location.path();
        $location.path('/login');
  }
```

The register.html template and its respective controller work in exactly the same way.

spa/account/register.html

```
<div class="row">
    <form class="form-signin" role="form" novalidate angular-validator
name="userRegistrationForm" angular-validator-submit="register()">
        <h3 class="form-signin-heading">Register <label class="label label-
danger">Admin</label></h3>
        <div class="form-group">
            <label for="inputUsername" class="">Username</label>
            <input type="text" name="inputUsername" class="form-control" ng-
model="user.username"
                   placeholder="Username" validate-on="blur" required required-
message="'Username is required'">
```

```
        </div>
        <div class="form-group">
            <label for="inputPassword">Password</label>
            <input type="password" name="inputPassword" class="form-control" ng-
model="user.password" placeholder="Password"
                    validate-on="blur" required required-message="'Password is
required'">
        </div>
        <div class="form-group">
            <label for="inputEmail">Email</label>
            <input type="email" name="inputEmail" class="form-control" ng-
model="user.email" placeholder="Email address"
                   validate-on="blur" required required-message="'Email is required'">
        </div>
        <button class="btn btn-lg btn-primary btn-block"
type="submit">Register</button>
    </form>
</div>
```

spa/account/registerCtrl.js

```
(function (app) {
    'use strict';

    app.controller('registerCtrl', registerCtrl);

    registerCtrl.$inject = ['$scope', 'membershipService', 'notificationService',
'$rootScope', '$location'];

    function registerCtrl($scope, membershipService, notificationService, $rootScope,
$location) {
        $scope.pageClass = 'page-login';
        $scope.register = register;
        $scope.user = {};

        function register() {
            membershipService.register($scope.user, registerCompleted)
        }

        function registerCompleted(result) {
            if (result.data.success) {
                membershipService.saveCredentials($scope.user);
                notificationService.displaySuccess('Hello ' + $scope.user.username);
                $scope.userData.displayUserInfo();
                $location.path('/');
            }
            else {
                notificationService.displayError('Registration failed. Try again.');
            }
        }
    }

})(angular.module('common.core'));
```

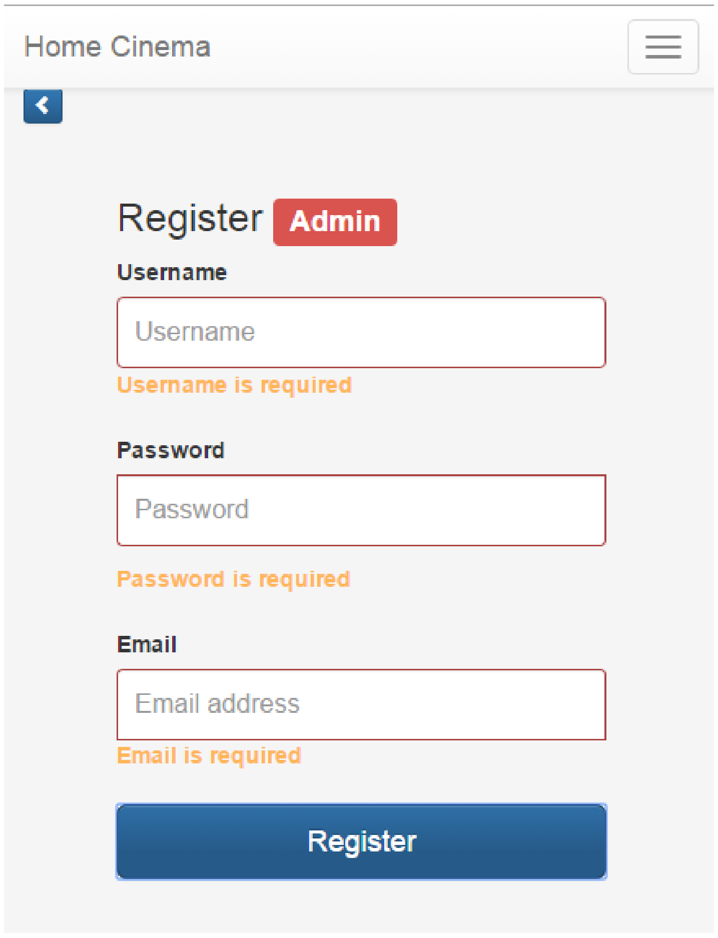


Figure 20. User registration form

Customers

The *Customers* feature is consisted by 2 views in our SPA application. The first view is responsible to display all customers. It also supports pagination, filtering current view data and start a new server search. The second one is the registration view where an employee can register a new customer. We will start from the server side required components first and then with the front-end as we did before. Add a new CustomersController Web API controller inside the *controllers* folder.

CustomersController.cs

```
    [Authorize(Roles="Admin")]
    [RoutePrefix("api/customers")]
    public class CustomersController : ApiControllerBase
    {
        private readonly IEntityBaseRepository<Customer> _customersRepository;

        public CustomersController(IEntityBaseRepository<Customer> customersRepository,
            IEntityBaseRepository<Error> _errorsRepository, IUnitOfWork _unitOfWork)
            : base(_errorsRepository, _unitOfWork)
        {
            _customersRepository = customersRepository;
        }
    }
```

First feature we want to support is the pagination with an optional filter search parameter. For this to work, the data returned by the Web API action must also include some pagination related information so that the front-end components can re-build the paginated list. Add the following generic PaginationSet class inside the *Infrastructure/core* folder.

PaginationSet.cs

```
 public class PaginationSet<T>
    {
        public int Page { get; set; }

        public int Count
        {
            get
            {
                return (null != this.Items) ? this.Items.Count() : 0;
            }
        }

        public int TotalPages { get; set; }
        public int TotalCount { get; set; }

        public IEnumerable<T> Items { get; set; }
    }
```

This class holds the list of items we want to render plus all the pagination information we need to build a paginated list at the front side. Customer entity will have its own **ViewModel** so let's create the

CustomerViewModel and its validator. I assume that at this point you know where to place the following files.

CustomerViewModel.cs

```
    [Bind(Exclude = "UniqueKey")]
    public class CustomerViewModel : IValidatableObject
    {
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
        public string IdentityCard { get; set; }
        public Guid UniqueKey { get; set; }
        public DateTime DateOfBirth { get; set; }
        public string Mobile { get; set; }
        public DateTime RegistrationDate { get; set; }

        public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
        {
            var validator = new CustomerViewModelValidator();
            var result = validator.Validate(this);
            return result.Errors.Select(item => new ValidationResult(item.ErrorMessage,
new[] { item.PropertyName }));
        }
    }
```

I have excluded the *UniqueKey* property from binding since that's a value to be created on server side.

CustomerViewModelValidator.cs

```
public class CustomerViewModelValidator : AbstractValidator<CustomerViewModel>
    {
        public CustomerViewModelValidator()
        {
            RuleFor(customer => customer.FirstName).NotEmpty()
                .Length(1, 100).WithMessage("First Name must be between 1 - 100
characters");

            RuleFor(customer => customer.LastName).NotEmpty()
                .Length(1, 100).WithMessage("Last Name must be between 1 - 100
characters");

            RuleFor(customer => customer.IdentityCard).NotEmpty()
                .Length(1, 100).WithMessage("Identity Card must be between 1 - 50
characters");

            RuleFor(customer => customer.DateOfBirth).NotNull()
                .LessThan(DateTime.Now.AddYears(-16))
                .WithMessage("Customer must be at least 16 years old.");

            RuleFor(customer => customer.Mobile).NotEmpty().Matches(@"^\d{10}$")
                .Length(10).WithMessage("Mobile phone must have 10 digits");

            RuleFor(customer => customer.Email).NotEmpty().EmailAddress()
                .WithMessage("Enter a valid Email address");
        }
    }
```

Don't forget to add the **Automapper mapping** from Customer to CustomerViewModel so switch to DomainToViewModelMappingProfile and add the following line inside the *Configure()* function.

```
Mapper.CreateMap<Customer, CustomerViewModel>();
```

Now we can go to CustomersController and create the **Search** method.

CustomersController Search action

```
        [HttpGet]
        [Route("search/{page:int=0}/{pageSize=4}/{filter?}")]
        public HttpResponseMessage Search(HttpRequestMessage request, int? page, int?
pageSize, string filter = null)
        {
            int currentPage = page.Value;
            int currentPageSize = pageSize.Value;

            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;
                List<Customer> customers = null;
                int totalMovies = new int();

                if (!string.IsNullOrEmpty(filter))
                {
                    filter = filter.Trim().ToLower();

                    customers = _customersRepository.GetAll()
                        .OrderBy(c => c.ID)
                        .Where(c => c.LastName.ToLower().Contains(filter) ||
                            c.IdentityCard.ToLower().Contains(filter) ||
                            c.FirstName.ToLower().Contains(filter))
                        .ToList();
                }
                else
                {
                    customers = _customersRepository.GetAll().ToList();
                }

                totalMovies = customers.Count();
                customers = customers.Skip(currentPage * currentPageSize)
                        .Take(currentPageSize)
                        .ToList();

                IEnumerable<CustomerViewModel> customersVM =
Mapper.Map<IEnumerable<Customer>, IEnumerable<CustomerViewModel>>(customers);

                PaginationSet<CustomerViewModel> pagedSet = new
PaginationSet<CustomerViewModel>()
                {
                    Page = currentPage,
                    TotalCount = totalMovies,
                    TotalPages = (int)Math.Ceiling((decimal)totalMovies /
currentPageSize),
                    Items = customersVM
                };

                response =
request.CreateResponse<PaginationSet<CustomerViewModel>>(HttpStatusCode.OK, pagedSet);

                return response;
            });
        }
```

We will continue with the front-end required **angularJS** components. As opposed from the routes we defined in the *app.js*, we need two files to display the customers view, the customers.html template and a customersCtrl controller inside a *customersCtrl.js* file.

part of app.js

```
.when("/customers", {
                templateUrl: "scripts/spa/customers/customers.html",
                controller: "customersCtrl"
            })
```

Go ahead and add a **customers** folder inside the spa and create the following customers.html template.

spa/customers/customers.html

```
<div class="row">
    <div class="panel panel-primary">
        <div class="panel-heading clearfix">
            <h4 class="panel-title pull-left" style="padding-top: 7.5px;">Home Cinema
Customers</h4>
            <div class="input-group">
                <input id="inputSearchCustomers" type="search" ng-
model="filterCustomers" class="form-control shortInputSearch" placeholder="Filter,
search customers..">
                <div class="input-group-btn">
                    <button class="btn btn-primary" ng-click="search();"><i
class="glyphicon glyphicon-search"></i></button>
                    <button class="btn btn-primary" ng-click="clearSearch();"><i
class="glyphicon glyphicon-remove-sign"></i></button>
                </div>
            </div>
        </div>
        <div class="panel-body">
            <div class="row">
                <div class="col-sm-6" ng-repeat="customer in Customers |
filter:filterCustomers">
                    <div class="panel panel-default">
                        <div class="panel-heading">
                            <strong>{{customer.FirstName}}
{{customer.LastName}}</strong><br/>

                        </div>
                        <div class="panel-body">
                            <div class="table-responsive">
                                <table class="table table-condensed shortMargin">
                                    <tr>
                                        <td class="shortPadding">Email:</td>
                                        <td
class="shortPadding"><i>{{customer.Email}}</i></td>
                                    </tr>
                                    <tr>
                                        <td class="shortPadding">Mobile:</td>
                                        <td
class="shortPadding"><i>{{customer.Mobile}}</i></td>
                                    </tr>
                                    <tr>
                                        <td class="shortPadding">Birth:</td>
                                        <td
class="shortPadding"><i>{{customer.DateOfBirth | date:'mediumDate'}}</i></td>
                                    </tr>
                                    <tr>
                                        <td class="shortPadding">Registered:</td>
                                        <td
class="shortPadding"><i>{{customer.RegistrationDate | date:'mediumDate'}}</i></td>
                                    </tr>
                                </table>
                            </div>
                        </div>
                        <div class="panel-footer clearfix">
                            <label class="label label-
danger">{{customer.IdentityCard}}</label>
                            <div class="pull-right">
                                <buton class="btn btn-primary btn-xs" ng-
click="openEditDialog(customer);">Edit <i class="fa fa-pencil pull-right"></i></buton>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        </div>
        <div class="panel-footer">
            <div class="text-center">
                <custom-pager page="{{page}}" pages-count="{{pagesCount}}" total-
count="{{totalCount}}" search-func="search(page)"></custom-pager>
            </div>
        </div>
    </div>
</div>
```

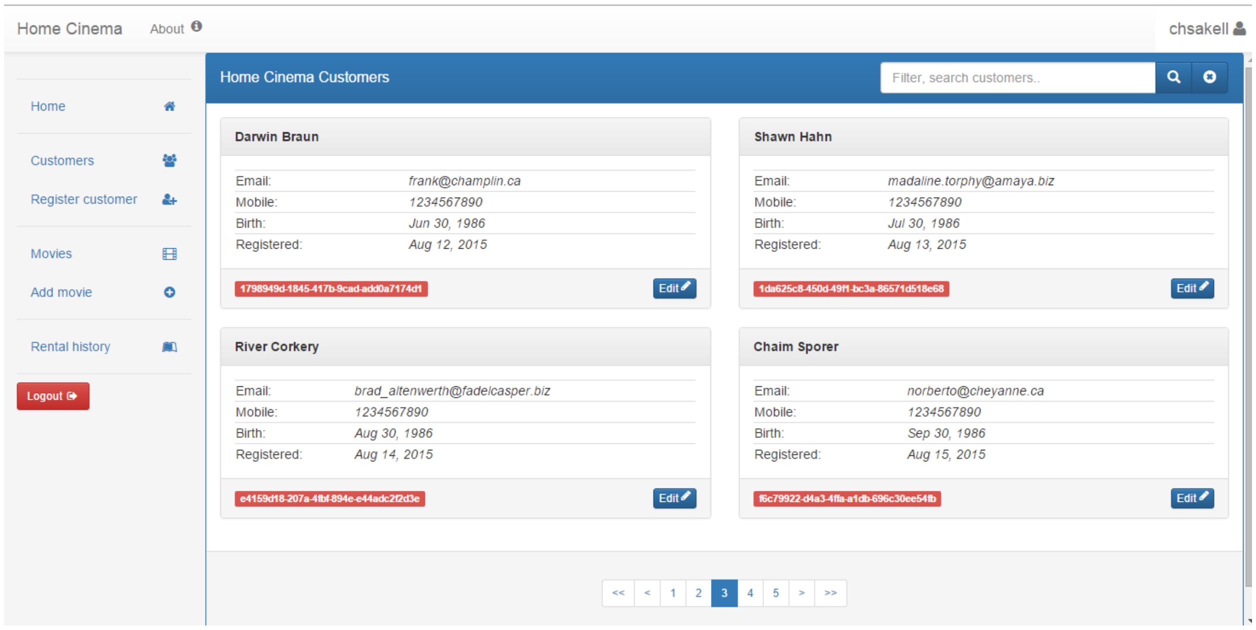
The most important highlighted line is the last one where we build a custom pager element. The directive we are going to add is responsible to render a paginated list depending on pagination information retrieved from the server (*page, pages-count, total-count*). Add the following pager.html template and its definition directive inside the *layout* folder.

spa/layout/pager.html

```
<div>
    <div ng-hide="(!pagesCount || pagesCount < 2)" style="display:inline">
        <ul class="pagination pagination-sm">
            <li><a ng-hide="page == 0" ng-click="search(0)"><<</a></li>
            <li><a ng-hide="page == 0" ng-click="search(page-1)"><</a></li>
            <li ng-repeat="n in range()" ng-class="{active: n == page}">
                <a ng-click="search(n)" ng-if="n != page">{{n+1}}</a>
                <span ng-if="n == page">{{n+1}}</span>
            </li>
            <li><a ng-hide="page == pagesCount - 1" ng-
click="search(pagePlus(1))">></a></li>
            <li><a ng-hide="page == pagesCount - 1" ng-click="search(pagesCount -
1)">>></a></li>
        </ul>
    </div>
</div>
```

spa/layout/customPager.directive.js

```
(function(app) {
    'use strict';

    app.directive('customPager', customPager);

    function customPager() {
        return {
            scope: {
                page: '@',
                pagesCount: '@',
                totalCount: '@',
                searchFunc: '&',
                customPath: '@'
            },
            replace: true,
            restrict: 'E',
            templateUrl: '/scripts/spa/layout/pager.html',
            controller: ['$scope', function ($scope) {
                $scope.search = function (i) {
                    if ($scope.searchFunc) {
                        $scope.searchFunc({ page: i });
                    }
                };

                $scope.range = function () {
                    if (!$scope.pagesCount) { return []; }
                    var step = 2;
                    var doubleStep = step * 2;
                    var start = Math.max(0, $scope.page - step);
                    var end = start + 1 + doubleStep;
                    if (end > $scope.pagesCount) { end = $scope.pagesCount; }

                    var ret = [];
                    for (var i = start; i != end; ++i) {
                        ret.push(i);
                    }

                    return ret;
                };

                $scope.pagePlus = function(count)
                {
                    return +$scope.page + count;
                }
            }]
        }
    }

})(angular.module('common.ui'));
```



Darwin Braun

Email: frank@champlin.ca
Mobile: 1234567890
Birth: Jun 30, 1986
Registered: Aug 12, 2015

1798949d-1845-417b-9cad-add0a7174d1

Shawn Hahn

Email: madaline.torphy@amaya.biz
Mobile: 1234567890
Birth: Jul 30, 1986
Registered: Aug 13, 2015

1da625c8-450d-49f1-bc3a-86571d518e68

River Corkery

Email: brad_altenwerth@fadelcasper.biz
Mobile: 1234567890
Birth: Aug 30, 1986
Registered: Aug 14, 2015

e4159d18-207a-4fbf-894e-e44adc2f2d3e

Chaim Sporer

Email: norberto@cheyanne.ca
Mobile: 1234567890
Birth: Sep 30, 1986
Registered: Aug 15, 2015

f6c79922-d4a3-4ffa-a1db-696c30ee54fb

Figure 21. Pagination example

Now let's see the customersCtrl controller. This controller is responsible to retrieve data from Web API and start a new search if the user presses the *magnify* button next to the textbox. Moreover it's the one that will open a modal popup window when the employee decides to edit a specific customer. For this popup window we will use the **angular-ui $modal** service.

spa/customers/customersCtrl.js

```
(function (app) {
    'use strict';

    app.controller('customersCtrl', customersCtrl);

    customersCtrl.$inject = ['$scope','$modal', 'apiService', 'notificationService'];

    function customersCtrl($scope, $modal, apiService, notificationService) {

        $scope.pageClass = 'page-customers';
        $scope.loadingCustomers = true;
        $scope.page = 0;
        $scope.pagesCount = 0;
        $scope.Customers = [];

        $scope.search = search;
        $scope.clearSearch = clearSearch;
        $scope.search = search;
        $scope.clearSearch = clearSearch;
        $scope.openEditDialog = openEditDialog;

        function search(page) {
            page = page || 0;

            $scope.loadingCustomers = true;

            var config = {
                params: {
                    page: page,
                    pageSize: 4,
                    filter: $scope.filterCustomers
                }
            };

            apiService.get('/api/customers/search/', config,
            customersLoadCompleted,
            customersLoadFailed);
        }

        function openEditDialog(customer) {
            $scope.EditedCustomer = customer;
            $modal.open({
                templateUrl: 'scripts/spa/customers/editCustomerModal.html',
                controller: 'customerEditCtrl',
                scope: $scope
            }).result.then(function ($scope) {
                clearSearch();
            }, function () {
            });
        }

        function customersLoadCompleted(result) {
            $scope.Customers = result.data.Items;

            $scope.page = result.data.Page;
            $scope.pagesCount = result.data.TotalPages;
            $scope.totalCount = result.data.TotalCount;
            $scope.loadingCustomers = false;

            if ($scope.filterCustomers && $scope.filterCustomers.length) {
                notificationService.displayInfo(result.data.Items.length + ' customers
found');
            }

        }

        function customersLoadFailed(response) {
            notificationService.displayError(response.data);
        }

        function clearSearch() {
            $scope.filterCustomers = '';
            search();
        }
        $scope.search();
    }

})(angular.module('homeCinema'));
```

Let's focus on the following part of the *customersCtrl* controller where the modal window pops up.

```
function openEditDialog(customer) {
            $scope.EditedCustomer = customer;
            $modal.open({
                templateUrl: 'scripts/spa/customers/editCustomerModal.html',
                controller: 'customerEditCtrl',
                scope: $scope
            }).result.then(function ($scope) {
                clearSearch();
            }, function () {
            });
        }
```

When we decide to edit a customer we don't have to request data from the server. We have them already and we can pass them to the customerEditCtrl through the **$scope**.

```
$scope.EditedCustomer = customer;
```

The popup window isn't a new view to be rendered but a single pop-up window with a template and a custom controller. Let's view both of those components, the editCustomerModal.html template and the customerEditCtrl<span> controller.

spa/customers/editCustomerModal.html

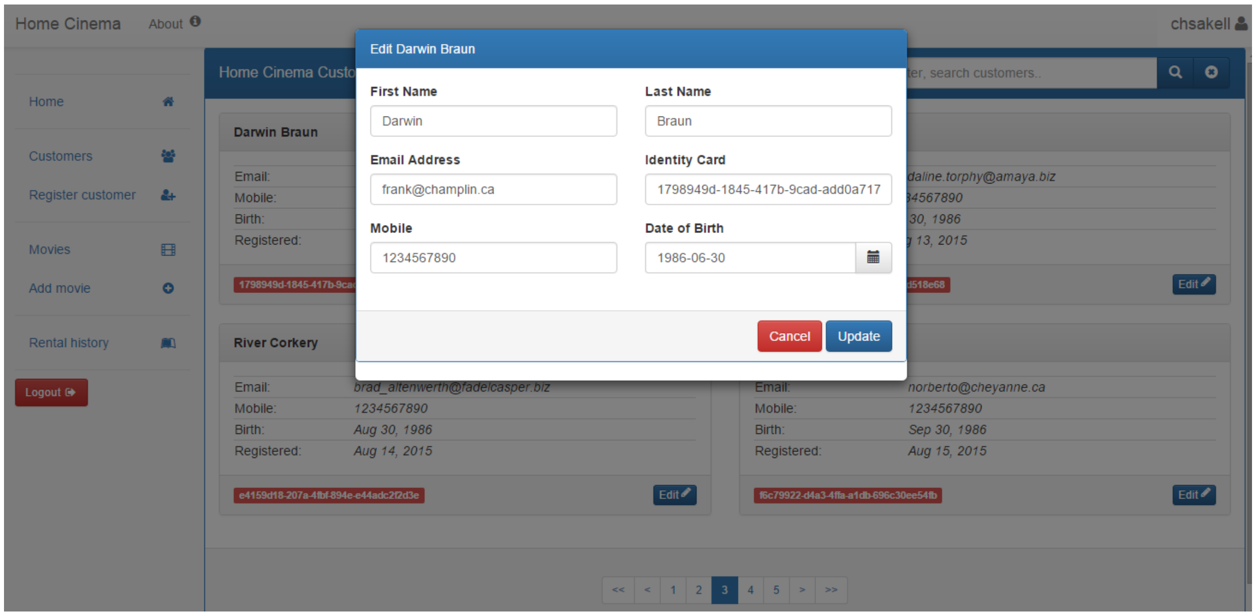
```
<div class="panel panel-primary">
    <div class="panel-heading">
        Edit {{EditedCustomer.FirstName}} {{EditedCustomer.LastName}}
    </div>
    <div class="panel-body">
        <form role="form" novalidate angular-validator name="addCustomerForm" angular-
validator-submit="Register()">
            <div class="">
                <div class="form-group">
                    <div class="row">
                        <div class="col-sm-6">
                            <label class="control-label" for="firstName">First
Name</label>
                            <input type="text" class="form-control" ng-
model="EditedCustomer.FirstName" name="firstName" id="firstName" placeholder="First
Name"
                                   validate-on="blur" required required-message="'First
Name is required'">
                        </div>

                        <div class="col-sm-6 selectContainer">
```

```
                            <label class="control-label" for="lastName">Last
Name</label>
                            <input type="text" class="form-control" ng-
model="EditedCustomer.LastName" name="lastName" id="lastName" placeholder="Last Name"
                                   validate-on="blur" required required-message="'Last
Name is required'">
                        </div>
                    </div>
                </div>
                <div class="form-group">
                    <div class="row">
                        <div class="col-xs-6">
                            <label class="control-label" for="email">Email
Address</label>
                            <input type="email" class="form-control" ng-
model="EditedCustomer.Email" id="email" name="email" placeholder="Email address"
                                   validate-on="blur" required required-message="'Email
is required'">
                        </div>

                        <div class="col-xs-6 selectContainer">
                            <label class="control-label" for="IdentityCard">Identity
Card</label>
                            <input type="text" class="form-control" ng-
model="EditedCustomer.IdentityCard" id="identityCard" name="identityCard"
placeholder="Identity Card number"
                                   validate-on="blur" required required-
message="'Identity Card is required'">
                        </div>
                    </div>
                </div>
                <div class="form-group">
                    <div class="row">
                        <div class="col-xs-6">
                            <label class="control-label"
for="mobilePhone">Mobile</label>
                            <input type="text" ng-model="EditedCustomer.Mobile"
class="form-control" id="mobilePhone" name="mobilePhone" placeholder="Mobile phone"
                                   validate-on="blur" required required-
message="'Mobile phone is required'">
                        </div>
                        <div class="col-xs-6 selectContainer">
                            <label class="control-label" for="dateOfBirth">Date of
Birth</label>
                            <p class="input-group">
                                <input type="text" class="form-control"
name="dateOfBirth" datepicker-popup="{{format}}" ng-model="EditedCustomer.DateOfBirth"
is-open="datepicker.opened"
                                       datepicker-options="dateOptions" ng-
required="true" datepicker-append-to-body="true" close-text="Close"
                                       validate-on="blur" required required-
message="'Date of birth is required'" />
                                <span class="input-group-btn">
                                    <button type="button" class="btn btn-default" ng-
click="openDatePicker($event)"><i class="glyphicon glyphicon-calendar"></i></button>
                                </span>
                            </p>
                        </div>
                    </div>
                </div>
            </div>
        </form>
    </div>
    <div class="panel-footer clearfix">
        <div class="pull-right">
            <button type="button" class="btn btn-danger" ng-
click="cancelEdit()">Cancel</button>
            <button type="button" class="btn btn-primary" ng-
click="updateCustomer()">Update</button>
        </div>
    </div>
</div>
```

customerEditCtrl.js

```
(function (app) {
    'use strict';

    app.controller('customerEditCtrl', customerEditCtrl);

    customerEditCtrl.$inject = ['$scope', '$modalInstance','$timeout', 'apiService',
'notificationService'];

    function customerEditCtrl($scope, $modalInstance, $timeout, apiService,
notificationService) {

        $scope.cancelEdit = cancelEdit;
        $scope.updateCustomer = updateCustomer;

        $scope.openDatePicker = openDatePicker;
        $scope.dateOptions = {
            formatYear: 'yy',
            startingDay: 1
        };
        $scope.datepicker = {};

        function updateCustomer()
        {
            console.log($scope.EditedCustomer);
            apiService.post('/api/customers/update/', $scope.EditedCustomer,
            updateCustomerCompleted,
            updateCustomerLoadFailed);
        }

        function updateCustomerCompleted(response)
        {
            notificationService.displaySuccess($scope.EditedCustomer.FirstName + ' ' +
$scope.EditedCustomer.LastName + ' has been updated');
            $scope.EditedCustomer = {};
            $modalInstance.dismiss();
        }

        function updateCustomerLoadFailed(response)
        {
            notificationService.displayError(response.data);
        }

        function cancelEdit() {
            $scope.isEnabled = false;
            $modalInstance.dismiss();
        }

        function openDatePicker($event) {
            $event.preventDefault();
            $event.stopPropagation();

            console.log('test');
            $timeout(function () {
                $scope.datepicker.opened = true;
            });

        };

    }

})(angular.module('homeCinema'));
```

When the update finishes we ensure that we call the **$modalInstance.dismiss()** function to close the modal popup window.



Home Cinema About chsakell

Home
Customers
Register customer
Movies
Add movie
Rental history
Logout

Edit Darwin Braun

First Name	Last Name
Darwin	Braun
Email Address	Identity Card
frank@champlin.ca	1798949d-1845-417b-9cad-add0a717
Mobile	Date of Birth
1234567890	1986-06-30

Cancel Update

River Corkery

Email:	brad_altenwerth@fadelcasper.biz
Mobile:	1234567890
Birth:	Aug 30, 1986
Registered:	Aug 14, 2015

e4159d18-207a-4fbf-894e-e44adc2f2d3e Edit

Email:	norberto@cheyanne.ca
Mobile:	1234567890
Birth:	Sep 30, 1986
Registered:	Aug 15, 2015

f6c79922-d4a3-4ffa-a1db-696c30ee54fb Edit

<< < 1 2 3 4 5 > >>

You also need to add the **Update** Web API action method to the CustomersController.

CustomersController Update action

```
        [HttpPost]
        [Route("update")]
        public HttpResponseMessage Update(HttpRequestMessage request, CustomerViewModel
customer)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                if (!ModelState.IsValid)
                {
                    response = request.CreateResponse(HttpStatusCode.BadRequest,
                        ModelState.Keys.SelectMany(k => ModelState[k].Errors)
                              .Select(m => m.ErrorMessage).ToArray());
                }
                else
                {
                    Customer _customer = _customersRepository.GetSingle(customer.ID);
                    _customer.UpdateCustomer(customer);

                    _unitOfWork.Commit();

                    response = request.CreateResponse(HttpStatusCode.OK);
                }

                return response;
            });
        }
```

Let's procceed with the customer's registration feature by adding the register.html template and the customersRegCtrl controller.

spa/customers/register.html

```
<hr />
<div class="alert alert-info alert-dismissable">
    <a class="panel-close close" data-dismiss="alert">×</a>
    <i class="fa fa-user-plus fa-3x"></i>
    Register <strong>{{movie.Title}}</strong> new customer. Make sure you fill all
required fields.
</div>
<div class="row">
    <form role="form" novalidate angular-validator name="addCustomerForm" angular-
validator-submit="Register()">
        <div class="col-sm-6">
            <div class="form-group">
                <label for="firstName">First Name</label>
                <div class="form-group">
                    <input type="text" class="form-control" ng-
model="newCustomer.FirstName" name="firstName" id="firstName" placeholder="First Name"
                           validate-on="blur" required required-message="'First Name is
required'">

                </div>
            </div>
            <div class="form-group">
                <label for="lastName">Last Name</label>
                <div class="form-group">
                    <input type="text" class="form-control" ng-
model="newCustomer.LastName" name="lastName" id="lastName" placeholder="Last Name"
                           validate-on="blur" required required-message="'Last Name is
required'">
```

```
                </div>
            </div>
            <div class="form-group">
                <label for="email">Email address</label>
                <div class="form-group">
                    <input type="email" class="form-control" ng-
model="newCustomer.Email" id="email" name="email" placeholder="Email address"
                           validate-on="blur" required required-message="'Email is
required'">
                </div>
            </div>
            <div class="form-group">
                <label for="identityCard">Identity Card</label>
                <div class="form-group">
                    <input type="text" class="form-control" ng-
model="newCustomer.IdentityCard" id="identityCard" name="identityCard"
placeholder="Identity Card number"
                           validate-on="blur" required required-message="'Identity Card
is required'">
                </div>
            </div>
            <div class="form-group">
                <label for="dateOfBirth">Date of Birth</label>
                <p class="input-group">
                    <input type="text" class="form-control" name="dateOfBirth"
datepicker-popup="{{format}}" ng-model="newCustomer.DateOfBirth" is-
open="datepicker.opened" datepicker-options="dateOptions" ng-required="true"
datepicker-append-to-body="true" close-text="Close"
                           validate-on="blur" required required-message="'Date of birth
is required'" />
                    <span class="input-group-btn">
                        <button type="button" class="btn btn-default" ng-
click="openDatePicker($event)"><i class="glyphicon glyphicon-calendar"></i></button>
                    </span>
                </p>
            </div>
            <div class="form-group">
                <label for="mobilePhone">Mobile phone</label>
                <div class="form-group">
                    <input type="text" ng-model="newCustomer.Mobile" class="form-
control" id="mobilePhone" name="mobilePhone" placeholder="Mobile phone"
                           validate-on="blur" required required-message="'Mobile phone
is required'">

                </div>
            </div>
            <input type="submit" name="submit" id="submit" value="Submit" class="btn
btn-info pull-right">
        </div>
    </form>
    <div class="col-sm-5 col-md-push-1">
        <div class="col-md-12">
            <div class="alert alert-success">
                <ul>
                    <li ng-repeat="message in submission.successMessages track by
$index">
                        <strong>{{message}}</strong>
                    </li>
                </ul>
                <strong ng-bind="submission.successMessage"><span class="glyphicon
glyphicon-ok"></span> </strong>
            </div>
            <div class="alert alert-danger">
                <ul>
                    <li ng-repeat="error in submission.errorMessages track by $index">
                        <strong>{{error}}</strong>
                    </li>
                </ul>
            </div>
        </div>
    </div>
</div>
```

spa/customers/customersRegCtrl.js

```
(function (app) {
    'use strict';

    app.controller('customersRegCtrl', customersRegCtrl);

    customersRegCtrl.$inject = ['$scope', '$location', '$rootScope', 'apiService'];

    function customersRegCtrl($scope, $location, $rootScope, apiService) {

        $scope.newCustomer = {};

        $scope.Register = Register;

        $scope.openDatePicker = openDatePicker;
        $scope.dateOptions = {
            formatYear: 'yy',
            startingDay: 1
        };
        $scope.datepicker = {};

        $scope.submission = {
            successMessages: ['Successfull submission will appear here.'],
            errorMessages: ['Submition errors will appear here.']
        };

        function Register() {
            apiService.post('/api/customers/register', $scope.newCustomer,
           registerCustomerSucceded,
           registerCustomerFailed);
        }

        function registerCustomerSucceded(response) {
            $scope.submission.errorMessages = ['Submition errors will appear here.'];
            console.log(response);
            var customerRegistered = response.data;
            $scope.submission.successMessages = [];
            $scope.submission.successMessages.push($scope.newCustomer.LastName + ' has
been successfully registed');
            $scope.submission.successMessages.push('Check ' +
customerRegistered.UniqueKey + ' for reference number');
```

```
            $scope.newCustomer = {};
        }

        function registerCustomerFailed(response) {
            console.log(response);
            if (response.status == '400')
                $scope.submission.errorMessages = response.data;
            else
                $scope.submission.errorMessages = response.statusText;
        }

        function openDatePicker($event) {
            $event.preventDefault();
            $event.stopPropagation();

            $scope.datepicker.opened = true;
        };
    }

})(angular.module('homeCinema'));
```

There's a small problem when rendering the customers registration template. You see there is no authorized resource to call when this template is rendered but the post action will force the user to authenticate himself/herself. We would like to avoid this and render the view only if the user is logged in. What we have used till now (*check the customers view*), is that when the controller bound to the view is activated and requests data from the server, if the server requires the user to be authenticated, then the apiService automatically redirects the user to the login view.

```
if (error.status == '401') {
       notificationService.displayError('Authentication required.');
       $rootScope.previousState = $location.path();
        $location.path('/login');
}
```

On the other hand, in the register customer view the user will be requested to be authenticated when the employee tries to **POST** the data (new customer) to the server. We can overcome this by adding a **resolve** function through the route provider for this route. Switch to the app.js and make the following modification.

part of app.js

```
            // code omitted
            .when("/customers/register", {
                templateUrl: "scripts/spa/customers/register.html",
                controller: "customersRegCtrl",
                resolve: { isAuthenticated: isAuthenticated }
            })
            // code omitted
    isAuthenticated.$inject = ['membershipService', '$rootScope','$location'];

    function isAuthenticated(membershipService, $rootScope, $location) {
        if (!membershipService.isUserLoggedIn()) {
            $rootScope.previousState = $location.path();
```

```
                $location.path('/login');
```

We use a route **resolve** function when we want to check a condition before the route actually changes. In our application we can use it to check if the user is logged in or not and if not redirect to login view. Now add the **Register** Web API action to the CustomersController.

CustomersController Register method

```
        [HttpPost]
        [Route("register")]
        public HttpResponseMessage Register(HttpRequestMessage request,
CustomerViewModel customer)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                if (!ModelState.IsValid)
                {
                    response = request.CreateResponse(HttpStatusCode.BadRequest,
                        ModelState.Keys.SelectMany(k => ModelState[k].Errors)
                              .Select(m => m.ErrorMessage).ToArray());
                }
                else
                {
                    if (_customersRepository.UserExists(customer.Email,
customer.IdentityCard))
                    {
                        ModelState.AddModelError("Invalid user", "Email or Identity
Card number already exists");
                        response = request.CreateResponse(HttpStatusCode.BadRequest,
                        ModelState.Keys.SelectMany(k => ModelState[k].Errors)
                              .Select(m => m.ErrorMessage).ToArray());
                    }
                    else
                    {
                        Customer newCustomer = new Customer();
                        newCustomer.UpdateCustomer(customer);
                        _customersRepository.Add(newCustomer);

                        _unitOfWork.Commit();

                        // Update view model
                        customer = Mapper.Map<Customer,
CustomerViewModel>(newCustomer);
                        response =
request.CreateResponse<CustomerViewModel>(HttpStatusCode.Created, customer);
                    }
                }

                return response;
            });
        }
```

I have highlighted the line where we update the database customer entity using an extension method. We have an Automapper map from Customer entity to CustomerViewModel but not vice-versa. You could do it but I recommend you not to because it doesn't work so well with **Entity Framework**. That's why I created an extension method for Customer entities. Add a new folder named *Extensions* inside the *Infrastructure* and create the following class. Then make sure you include the namespace in the CustomersController class.

EntitiesExtensions.cs

```
public static class EntitiesExtensions
    {
        public static void UpdateCustomer(this Customer customer, CustomerViewModel
customerVm)
        {
            customer.FirstName = customerVm.FirstName;
            customer.LastName = customerVm.LastName;
            customer.IdentityCard = customerVm.IdentityCard;
            customer.Mobile = customerVm.Mobile;
            customer.DateOfBirth = customerVm.DateOfBirth;
            customer.Email = customerVm.Email;
            customer.UniqueKey = (customerVm.UniqueKey == null || customerVm.UniqueKey
== Guid.Empty)
                ? Guid.NewGuid() : customerVm.UniqueKey;
            customer.RegistrationDate = (customer.RegistrationDate == DateTime.MinValue
? DateTime.Now : customerVm.RegistrationDate);
        }
    }
```

Movies

The most complex feature in our application is the *Movies* and that's because several requirements are connected to that feature. Let's recap what we need to do.

1. All movies must be displayed with their relevant information (*availability, trailer etc..*)
2. Pagination must be used for faster results, and user can either filter the already displayed movies or search for new ones
3. Clicking on a DVD image must show the movie's **Details** view where user can either **edit** the movie or **rent** it to a specific customer if available. This view is accessible only to authenticated users
4. When employee decides to rent a specific DVD to a customer through the Rent view, it should be able to search customers through an auto-complete textbox
5. The details view displays inside a panel, rental-history information for this movie, that is the dates rentals and returnings occurred. From this panel user can search a specific rental and mark it as ***Returned***
6. Authenticated employees should be able to add a new entry to the system. They should be able to upload a relevant image for the movie as well

Default view

We will start with the first two of them that that is display all movies with pagination, filtering and searching capabilities. We have seen such features when we created the customers base view. First, let's add the required Web API action method in the MoviesController.

part of MoviesController.cs

```
        [AllowAnonymous]
        [Route("{page:int=0}/{pageSize=3}/{filter?}")]
        public HttpResponseMessage Get(HttpRequestMessage request, int? page, int?
pageSize, string filter = null)
        {
            int currentPage = page.Value;
            int currentPageSize = pageSize.Value;

            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;
                List<Movie> movies = null;
                int totalMovies = new int();

                if (!string.IsNullOrEmpty(filter))
                {
                    movies = _moviesRepository.GetAll()
                        .OrderBy(m => m.ID)
                        .Where(m => m.Title.ToLower()
                        .Contains(filter.ToLower().Trim()))
                        .ToList();
                }
                else
                {
                    movies = _moviesRepository.GetAll().ToList();
                }

                totalMovies = movies.Count();
                movies = movies.Skip(currentPage * currentPageSize)
                        .Take(currentPageSize)
                        .ToList();

                IEnumerable<MovieViewModel> moviesVM = Mapper.Map<IEnumerable<Movie>,
IEnumerable<MovieViewModel>>(movies);

                PaginationSet<MovieViewModel> pagedSet = new
PaginationSet<MovieViewModel>()
                {
                    Page = currentPage,
                    TotalCount = totalMovies,
                    TotalPages = (int)Math.Ceiling((decimal)totalMovies /
currentPageSize),
                    Items = moviesVM
                };

                response =
request.CreateResponse<PaginationSet<MovieViewModel>>(HttpStatusCode.OK, pagedSet);

                return response;
            });
        }
```

As you can see, this view doesn't require the user to be authenticated. Once more we used the PaginationSet class to return additional information for pagination purposes. On the front-end side, create a *movies* folder inside the *spa*, add the movies.html template and the moviesCtrl.js controller as follow.

spa/movies/movies.html

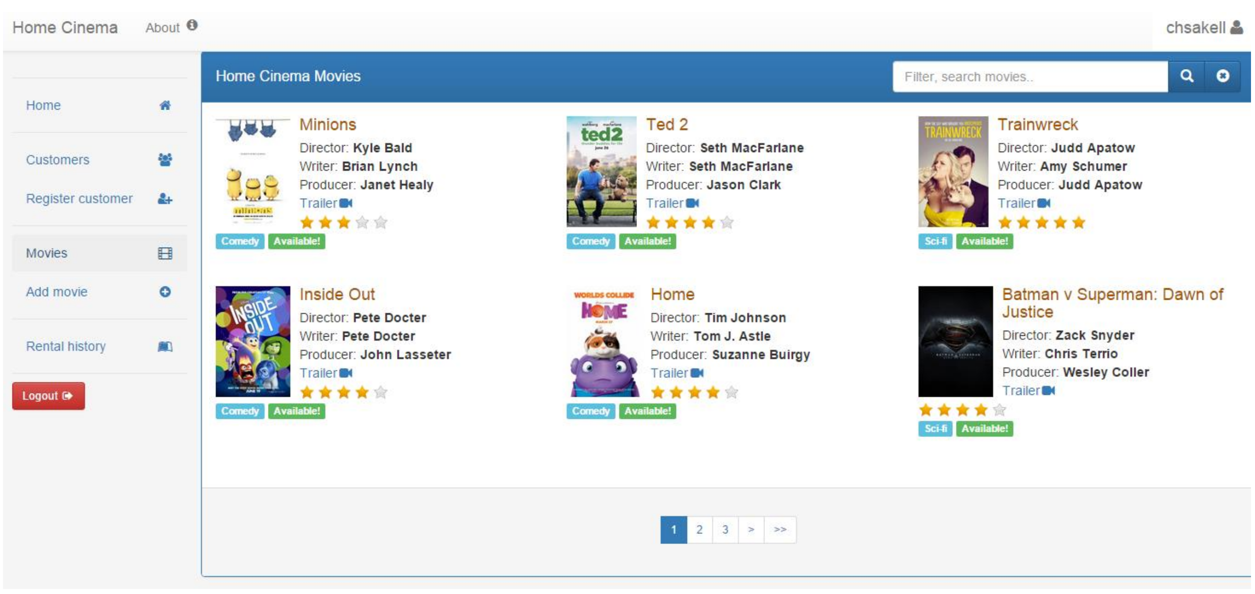
```
<div class="row">
    <div class="panel panel-primary">
        <div class="panel-heading clearfix">
            <h4 class="panel-title pull-left" style="padding-top: 7.5px;">Home Cinema
Movies</h4>
            <div class="input-group">
                <input id="inputSearchMovies" type="search" ng-model="filterMovies"
class="form-control shortInputSearch" placeholder="Filter, search movies..">
                <div class="input-group-btn">
                    <button class="btn btn-primary" ng-click="search();"><i
class="glyphicon glyphicon-search"></i></button>
                    <button class="btn btn-primary" ng-click="clearSearch();"><i
class="glyphicon glyphicon-remove-sign"></i></button>
                </div>
            </div>
        </div>

        <div class="panel-body">
            <div class="row">
```

```
                <div class="col-xs-12 col-sm-6 col-md-4" ng-repeat="movie in Movies |
filter:filterMovies">
                    <div class="media">
                        <a class="pull-left" ng-href="#/movies/{{movie.ID}}"
title="View {{movie.Title}} details">
                            <img class="media-object" height="120" ng-
src="../../Content/images/movies/{{movie.Image}}" alt="" />
                        </a>
                        <div class="media-body">
                            <h4 class="media-heading">{{movie.Title}}</h4>
                            Director: <strong>{{movie.Director}}</strong>
                            <br />
                            Writer: <strong>{{movie.Writer}}</strong>
                            <br />
                            Producer: <strong>{{movie.Producer}}</strong>
                            <br />
                            <a class="fancybox-media" ng-
href="{{movie.TrailerURI}}">Trailer<i class="fa fa-video-camera fa-fw"></i></a>
                        </div>
                        <div class="media-bottom">
                            <span component-rating="{{movie.Rating}}"></span>
                        </div>
                        <label class="label label-info">{{movie.Genre}}</label>
                        <available-movie is-
available="{{movie.IsAvailable}}"></available-movie>
                    </div>
                    <br /><br />
                </div>
            </div>
        </div>
        <div class="panel-footer">
            <div class="text-center">
                <custom-pager page="{{page}}" custom-path="{{customPath}}" pages-
count="{{pagesCount}}" total-count="{{totalCount}}" search-
func="search(page)"></custom-pager>
            </div>
        </div>
    </div>
</div>
```

Once again we used both the **available-movie** and **custom-pager** directives. Moreover, check that when we click on an image we want to change route and display selected movie details.

spa/movies/moviesCtrl.js

```
(function (app) {
    'use strict';

    app.controller('moviesCtrl', moviesCtrl);

    moviesCtrl.$inject = ['$scope', 'apiService','notificationService'];

    function moviesCtrl($scope, apiService, notificationService) {
        $scope.pageClass = 'page-movies';
        $scope.loadingMovies = true;
        $scope.page = 0;
        $scope.pagesCount = 0;
        $scope.Movies = [];

        $scope.search = search;
        $scope.clearSearch = clearSearch;

        function search(page) {
            page = page || 0;

            $scope.loadingMovies = true;

            var config = {
                params: {
                    page: page,
                    pageSize: 6,
                    filter: $scope.filterMovies
                }
            };

            apiService.get('/api/movies/', config,
            moviesLoadCompleted,
            moviesLoadFailed);
        }

        function moviesLoadCompleted(result) {
            $scope.Movies = result.data.Items;
            $scope.page = result.data.Page;
            $scope.pagesCount = result.data.TotalPages;
            $scope.totalCount = result.data.TotalCount;
            $scope.loadingMovies = false;

            if ($scope.filterMovies && $scope.filterMovies.length)
            {
                notificationService.displayInfo(result.data.Items.length + ' movies
found');
            }

        }

        function moviesLoadFailed(response) {
            notificationService.displayError(response.data);
        }

        function clearSearch() {
            $scope.filterMovies = '';
            search();
        }

        $scope.search();
    }

})(angular.module('homeCinema'));
```


Home Cinema About

chsakell

Home
Customers
Register customer
Movies
Add movie
Rental history
Logout

Home Cinema Movies

Filter, search movies..

Minions
Director: Kyle Bald
Writer: Brian Lynch
Producer: Janet Healy
Trailer
Comedy Available!

Ted 2
Director: Seth MacFarlane
Writer: Seth MacFarlane
Producer: Jason Clark
Trailer
Comedy Available!

Trainwreck
Director: Judd Apatow
Writer: Amy Schumer
Producer: Judd Apatow
Trailer
Sci-fi Available!

Inside Out
Director: Pete Docter
Writer: Pete Docter
Producer: John Lasseter
Trailer
Comedy Available!

Home
Director: Tim Johnson
Writer: Tom J. Astle
Producer: Suzanne Buirgy
Trailer
Comedy Available!

Batman v Superman: Dawn of Justice
Director: Zack Snyder
Writer: Chris Terrio
Producer: Wesley Coller
Trailer
Sci-fi Available!

1 2 3 > >>

Figure 22. movies default view

Details View

Let's continue with the movie details page. Think this page as an control panel for selected movie where you can edit or rent this movie to a customer and last but not least view all rental history related to that movie, in other words, who borrowed that movie and its rental status *(borrowed, returned)*. First, we will prepare the server side part so swith to the MoviesController and add the following action that returns details for a specific movie. Check that this action is only available for authenticated users and hence when an employee tries to display the details view he/she will be forced to log in first.

MoviesController details action

```
        [Route("details/{id:int}")]
        public HttpResponseMessage Get(HttpRequestMessage request, int id)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;
                var movie = _moviesRepository.GetSingle(id);

                MovieViewModel movieVM = Mapper.Map<Movie, MovieViewModel>(movie);

                response = request.CreateResponse<MovieViewModel>(HttpStatusCode.OK,
movieVM);

                return response;
            });
        }
```

The movie details page also displays rental-history information so let's see how to implement this functionality. What we mean by movie rental-history is all rentals occurred on stock items related to a specific movie. I remind you that a specific movie may have multiple stock items (*DVDs*) and more over, a rental is actually assigned to the stock item, not the movie entity.

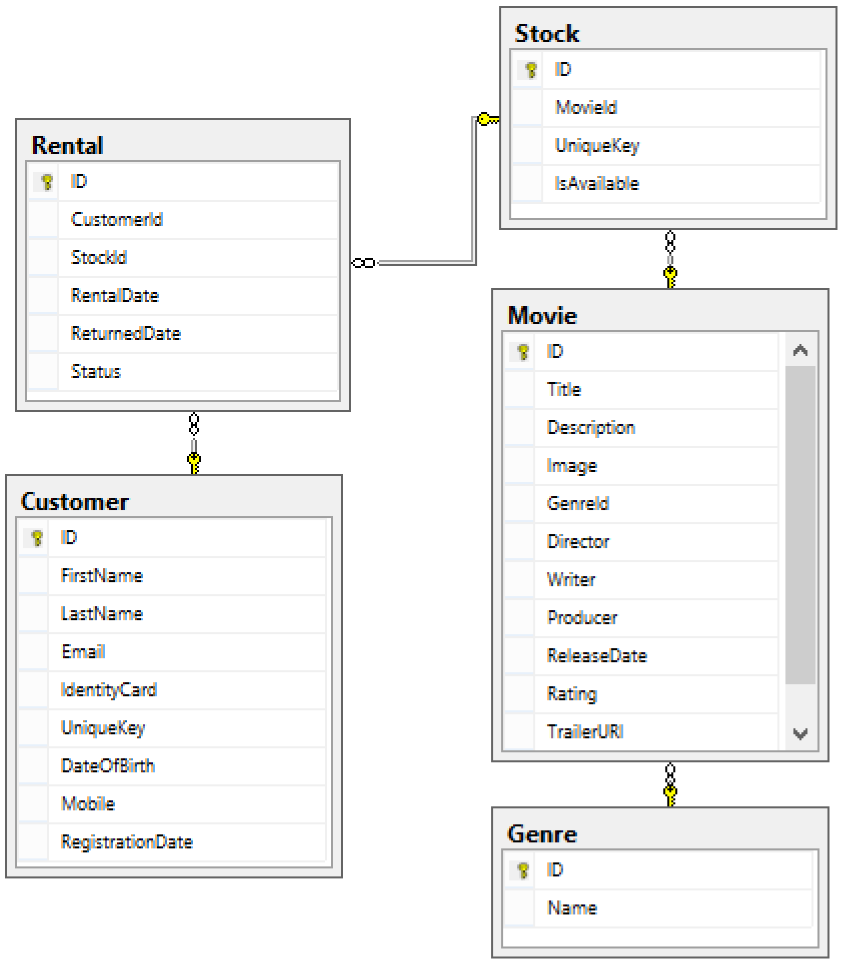


Figure 23. movie - stock - rental relationship

Let's create a new ViewModel named RentalHistoryViewModel to hold the information about a specific rental. Add the following class in the *Models* folder.

RentalHistoryViewModel.cs

```
public class RentalHistoryViewModel
    {
        public int ID { get; set; }
        public int StockId { get; set; }
        public string Customer { get; set; }
        public string Status { get; set; }
        public DateTime RentalDate { get; set; }
        public Nullable<DateTime> ReturnedDate { get; set; }
    }
```

The purpose is to return a list of RentalHistoryViewModel items related to the movie being displayed on the details view. In other words, find all rentals related to stock items that have foreign key the selected movie's ID. Add the following Web API RentalsController controller.

RentalsController.cs

```
    [Authorize(Roles = "Admin")]
    [RoutePrefix("api/rentals")]
    public class RentalsController : ApiControllerBase
    {
        private readonly IEntityBaseRepository<Rental> _rentalsRepository;
        private readonly IEntityBaseRepository<Customer> _customersRepository;
        private readonly IEntityBaseRepository<Stock> _stocksRepository;
        private readonly IEntityBaseRepository<Movie> _moviesRepository;

        public RentalsController(IEntityBaseRepository<Rental> rentalsRepository,
            IEntityBaseRepository<Customer> customersRepository,
IEntityBaseRepository<Movie> moviesRepository,
            IEntityBaseRepository<Stock> stocksRepository,
            IEntityBaseRepository<Error> _errorsRepository, IUnitOfWork _unitOfWork)
            : base(_errorsRepository, _unitOfWork)
        {
            _rentalsRepository = rentalsRepository;
            _moviesRepository = moviesRepository;
            _customersRepository = customersRepository;
            _stocksRepository = stocksRepository;
        }
    }
```

We need a private method in this controller which returns the rental-history items as we previously described.

private method in RentalsController

```
private List<RentalHistoryViewModel> GetMovieRentalHistory(int movieId)
        {
            List<RentalHistoryViewModel> _rentalHistory = new
List<RentalHistoryViewModel>();
            List<Rental> rentals = new List<Rental>();

            var movie = _moviesRepository.GetSingle(movieId);

            foreach (var stock in movie.Stocks)
            {
                rentals.AddRange(stock.Rentals);
            }

            foreach (var rental in rentals)
            {
                RentalHistoryViewModel _historyItem = new RentalHistoryViewModel()
                {
                    ID = rental.ID,
                    StockId = rental.StockId,
                    RentalDate = rental.RentalDate,
                    ReturnedDate = rental.ReturnedDate.HasValue ? rental.ReturnedDate :
null,
                    Status = rental.Status,
                    Customer =
_customersRepository.GetCustomerFullName(rental.CustomerId)
                };
                _rentalHistory.Add(_historyItem);
            }

            _rentalHistory.Sort((r1, r2) => r2.RentalDate.CompareTo(r1.RentalDate));

            return _rentalHistory;
        }
```

And now we can create the Web API action that the client will invoke when requesting rental history information.

```
        [HttpGet]
        [Route("{id:int}/rentalhistory")]
        public HttpResponseMessage RentalHistory(HttpRequestMessage request, int id)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                List<RentalHistoryViewModel> _rentalHistory =
 GetMovieRentalHistory(id);

                response =
 request.CreateResponse<List<RentalHistoryViewModel>>(HttpStatusCode.OK,
 _rentalHistory);

                return response;
            });
        }
```

In case we wanted to request rental history for movie with ID=4 then the request would be in the following form:

api/rentals/4/rentalhistory

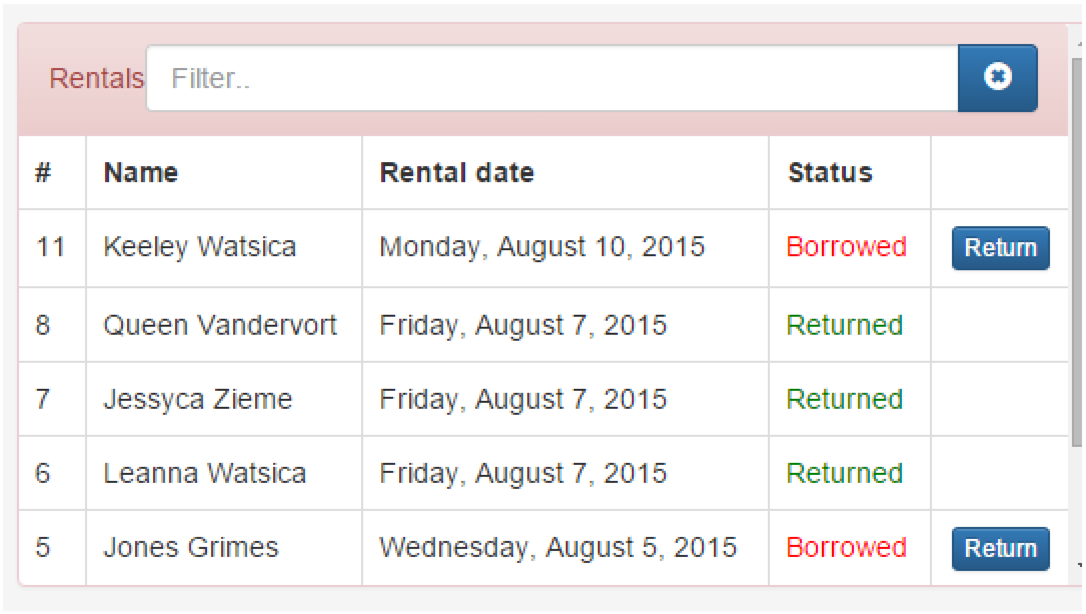
The employee must be able to mark a specific movie rental as *Returned* when the customer returns the DVD so let's add a Return action method as well.

RentalsController return movie method

```
        [HttpPost]
        [Route("return/{rentalId:int}")]
        public HttpResponseMessage Return(HttpRequestMessage request, int rentalId)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                var rental = _rentalsRepository.GetSingle(rentalId);

                if (rental == null)
                    response = request.CreateErrorResponse(HttpStatusCode.NotFound,
 "Invalid rental");
                else
                {
                    rental.Status = "Returned";
                    rental.Stock.IsAvailable = true;
                    rental.ReturnedDate = DateTime.Now;

                    _unitOfWork.Commit();

                    response = request.CreateResponse(HttpStatusCode.OK);
                }

                return response;
            });
        }
```

You can mark a movie with ID=4 as *Returned* with a **POST** request such as:

api/rentals/return/4



Rentals Filter..

#	Name	Rental date	Status	
11	Keeley Watsica	Monday, August 10, 2015	Borrowed	Return
8	Queen Vandervort	Friday, August 7, 2015	Returned	
7	Jessyca Zieme	Friday, August 7, 2015	Returned	
6	Leanna Watsica	Friday, August 7, 2015	Returned	
5	Jones Grimes	Wednesday, August 5, 2015	Borrowed	Return

Figure 24. movie rental history

At this point we can switch to the *front-end* and create the *details.html* template and its relative controller *movieDetailsCtrl*. Add the following files inside the *movies* folder.

spa/movies/details.html

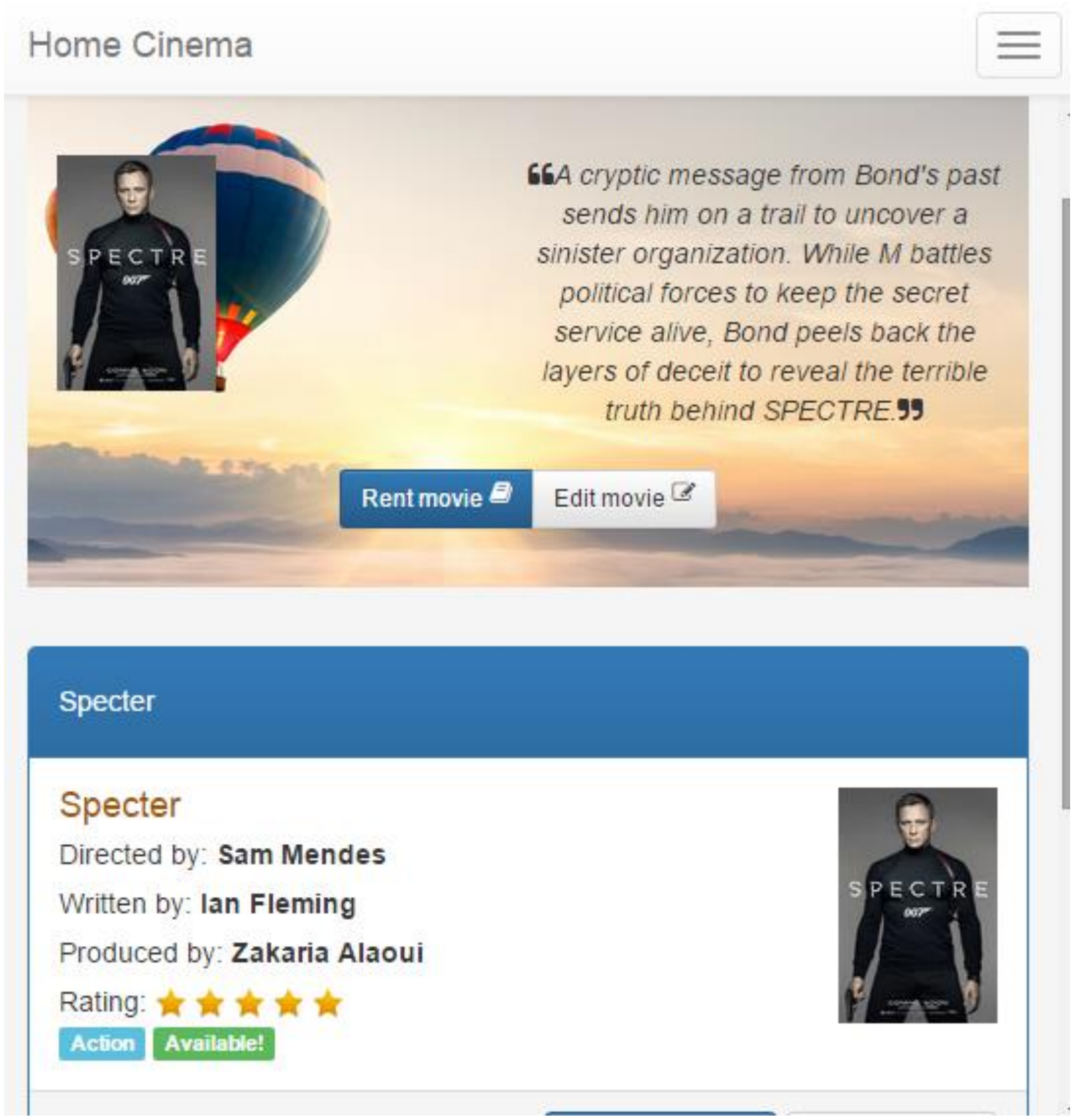
```
<hr />
<div class="jumbotron">
    <div class="container text-center">
        <img alt="{{movie.Title}}" ng-
src="../../../Content/images/movies/{{movie.Image}}" class="pull-left" height="120" />
        <div class="movieDescription"><i><i class="fa fa-quote-
left"></i>{{movie.Description}}<i class="fa fa-quote-right"></i></i></div>
        <br />
        <div class="btn-group">
```

```
            <button ng-if="movie.IsAvailable" type="button" ng-
click="openRentDialog();" class="btn btn-sm btn-primary">Rent movie<i class="fa fa-book
pull-right"></i></button>
            <a href="#/movies/edit/{{movie.ID}}" class="btn btn-sm btn-default">Edit
movie<i class="fa fa-pencil-square-o pull-right"></i></a>
        </div> <!-- end btn-group -->
    </div> <!-- end container -->
</div>

<div class="row">
    <div class="col-md-6">
        <div class="panel panel-primary">
            <div class="panel-heading">
                <h5>{{movie.Title}}</h5>
            </div>
            <div class="panel-body" ng-if="!loadingMovie">
                <div class="media">
                    <a class="pull-right" ng-href="#/movies/{{movie.ID}}" title="View
{{movie.Title}} details">
                        <img class="media-object" height="120" ng-
src="../../Content/images/movies/{{movie.Image}}" alt="" />
                    </a>
                    <div class="media-body">
                        <h4 class="media-heading">{{movie.Title}}</h4>
                        Directed by: <label>{{movie.Director}}</label><br />
                        Written by: <label>{{movie.Writer}}</label><br />
                        Produced by: <label>{{movie.Producer}}</label><br />
                        Rating: <span component-rating='{{movie.Rating}}'></span>
                        <br />
                        <label class="label label-info">{{movie.Genre}}</label>
                        <available-movie is-
available="{{movie.IsAvailable}}"></available-movie>
                    </div>
                </div>
            </div>
            <div class="panel-footer clearfix" ng-if="!loadingMovie">
                <div class="pull-right">
                    <a ng-href="{{movie.TrailerURI}}" class="btn btn-primary fancybox-
media">View Trailer <i class="fa fa-video-camera pull-right"></i></a>
                    <a ng-href="#/movies/edit/{{movie.ID}}" class="btn btn-
default">Edit movie <i class="fa fa-pencil-square pull-right"></i></a>
                </div>
            </div>
            <div ng-if="loadingMovie">
                <div class="col-xs-4"></div>
                <div class="col-xs-4">
                    <i class="fa fa-refresh fa-4x fa-spin"></i> <label class="label
label-primary">Loading movie data...</label>
                </div>
                <div class="col-xs-4"></div>
            </div>
        </div>

    </div>
    <div class="col-md-6">
        <div class="panel panel-danger shortPanel">
            <div class="panel-heading clearfix">
                <h5 class="pull-left">Rentals</h5>
                <div class="input-group">
                    <input id="inputSearchMovies" type="search" ng-
model="filterRentals" class="form-control" placeholder="Filter..">
                    <div class="input-group-btn">
                        <button class="btn btn-primary" ng-click="clearSearch();"><i
class="glyphicon glyphicon-remove-sign"></i></button>
                    </div>
                </div>
            </div>
            <div class="table-responsive" ng-if="!loadingRentals">
                <table class="table table-bordered">
                    <thead>
                        <tr>
                            <th>#</th>
                            <th>Name</th>
                            <th>Rental date</th>
                            <th>Status</th>
                            <th></th>
                        </tr>
                    </thead>
                    <tbody>
                        <tr ng-repeat="rental in rentalHistory | filter:filterRentals">
                            <td>{{rental.ID}}</td>
                            <td>{{rental.Customer}}</td>
                            <td>{{rental.RentalDate | date:'fullDate'}}</td>
                            <td ng-
class="getStatusColor(rental.Status)">{{rental.Status}}</td>
                            <td class="text-center">
                                <button ng-if="isBorrowed(rental)" type="button"
class="btn btn-primary btn-xs" ng-click="returnMovie(rental.ID)">Return</button>
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
            <div ng-if="loadingRentals">
                <div class="col-xs-4"></div>
                <div class="col-xs-4">
                    <i class="fa fa-refresh fa-4x fa-spin"></i> <label class="label
label-primary">Loading rental history...</label>
                </div>
                <div class="col-xs-4"></div>
            </div>
        </div>
    </div>
</div>
```

spa/movies/movieDetailsCtrl.cs

```
(function (app) {
    'use strict';

    app.controller('movieDetailsCtrl', movieDetailsCtrl);

    movieDetailsCtrl.$inject = ['$scope', '$location', '$routeParams', '$modal',
'apiService', 'notificationService'];
```

```
    function movieDetailsCtrl($scope, $location, $routeParams, $modal, apiService,
notificationService) {
        $scope.pageClass = 'page-movies';
        $scope.movie = {};
        $scope.loadingMovie = true;
        $scope.loadingRentals = true;
        $scope.isReadOnly = true;
        $scope.openRentDialog = openRentDialog;
        $scope.returnMovie = returnMovie;
        $scope.rentalHistory = [];
        $scope.getStatusColor = getStatusColor;
        $scope.clearSearch = clearSearch;
        $scope.isBorrowed = isBorrowed;

        function loadMovie() {

            $scope.loadingMovie = true;

            apiService.get('/api/movies/details/' + $routeParams.id, null,
            movieLoadCompleted,
            movieLoadFailed);
        }

        function loadRentalHistory() {
            $scope.loadingRentals = true;

            apiService.get('/api/rentals/' + $routeParams.id + '/rentalhistory', null,
            rentalHistoryLoadCompleted,
            rentalHistoryLoadFailed);
        }

        function loadMovieDetails() {
            loadMovie();
            loadRentalHistory();
        }

        function returnMovie(rentalID) {
            apiService.post('/api/rentals/return/' + rentalID, null,
            returnMovieSucceeded,
            returnMovieFailed);
        }

        function isBorrowed(rental)
        {
            return rental.Status == 'Borrowed';
        }

        function getStatusColor(status) {
            if (status == 'Borrowed')
                return 'red'
            else {
                return 'green';
            }
        }

        function clearSearch()
        {
            $scope.filterRentals = '';
        }

        function movieLoadCompleted(result) {
            $scope.movie = result.data;
            $scope.loadingMovie = false;
        }

        function movieLoadFailed(response) {
            notificationService.displayError(response.data);
        }

        function rentalHistoryLoadCompleted(result) {
            console.log(result);
            $scope.rentalHistory = result.data;
            $scope.loadingRentals = false;
        }

        function rentalHistoryLoadFailed(response) {
            notificationService.displayError(response);
        }

        function returnMovieSucceeded(response) {
            notificationService.displaySuccess('Movie returned to HomeCinema
succeesfully');
            loadMovieDetails();
        }

        function returnMovieFailed(response) {
            notificationService.displayError(response.data);
        }

        function openRentDialog() {
            $modal.open({
                templateUrl: 'scripts/spa/rental/rentMovieModal.html',
                controller: 'rentMovieCtrl',
                scope: $scope
            }).result.then(function ($scope) {
                loadMovieDetails();
            }, function () {
            });
        }

        loadMovieDetails();
    }

})(angular.module('homeCinema'));
```



❝A cryptic message from Bond's past sends him on a trail to uncover a sinister organization. While M battles political forces to keep the secret service alive, Bond peels back the layers of deceit to reveal the terrible truth behind SPECTRE.❞

Specter

Specter

Directed by: **Sam Mendes**

Written by: **Ian Fleming**

Produced by: **Zakaria Alaoui**

Rating: ★★★★★

Figure 25. movie details view

Rent movie

There is one more requirement we need to implement in the *details* view, the **rental**. As you may noticed from the *movieDetailsCtrl* controller, the rental works with a $modal popup window.

part of movieDetailsCtrl.js

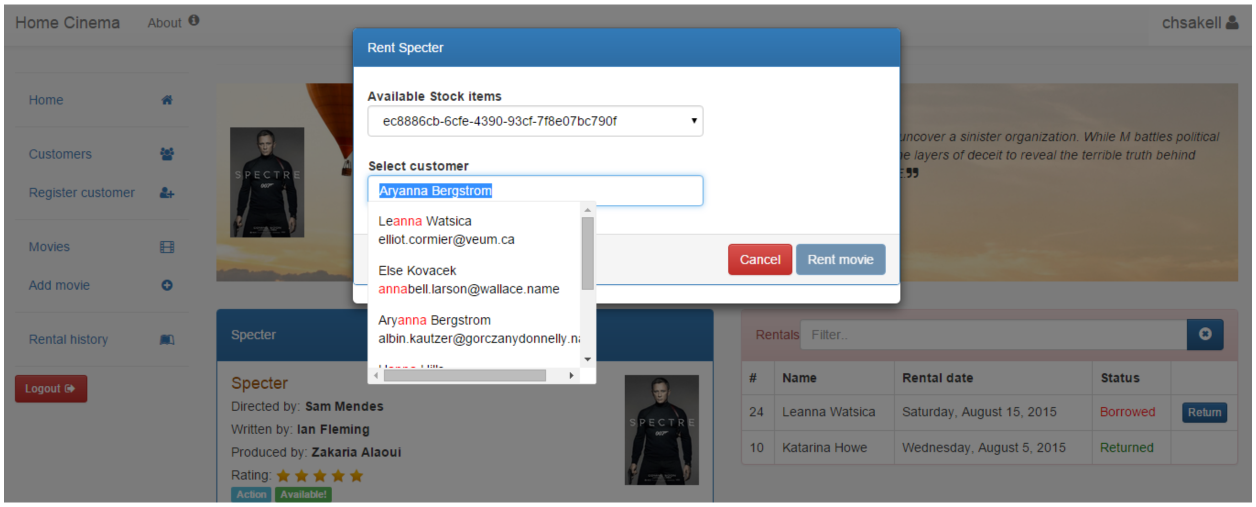
```
 function openRentDialog() {
            $modal.open({
                templateUrl: 'scripts/spa/rental/rentMovieModal.html',
                controller: 'rentMovieCtrl',
                scope: $scope
            }).result.then(function ($scope) {
                loadMovieDetails();
            }, function () {
            });
        }
```

We have seen the *$modal* popup in action when we were at the *edit customer* view. Create the rentMovieModal.html and the rentMovieCtrl controller inside a new folder named *Rental* under the *spa*.

spa/rental/rentMovieModal.html

```
<div class="panel panel-primary">
    <div class="panel-heading">
        Rent {{movie.Title}}
    </div>
    <div class="panel-body">
        <form class="form-horizontal" role="form">
            <div class="form-group">
                <div class="col-xs-8 selectContainer">
                    <label class="control-label">Available Stock items</label>
                    <select ng-model="selectedStockItem" class="form-control black" ng-
options="option.ID as option.UniqueKey for option in stockItems" required></select>
                </div>
            </div>
            <div class="form-group">
                <div class="col-xs-8">
                    <label class="control-label">Select customer</label>
                    <angucomplete-alt id="members"
                                      placeholder="Search customers"
                                      pause="200"
                                      selected-object="selectCustomer"
                                      input-changed="selectionChanged"
                                      remote-url="/api/customers?filter="
                                      remote-url-data-field=""
                                      title-field="FirstName,LastName"
                                      description-field="Email"
                                      input-class="form-control form-control-small"
                                      match-class="red"
                                      text-searching="Searching customers.."
                                      text-no-results="No customers found matching your
filter." />
                </div>
            </div>
        </form>
    </div>
    <div class="panel-footer clearfix">
        <div class="pull-right">
            <button type="button" class="btn btn-danger" ng-
click="cancelRental()">Cancel</button>
            <button type="button" class="btn btn-primary" ng-click="rentMovie()" ng-
disabled="!isEnabled">Rent movie</button>
        </div>
    </div>
</div>
```

One new thing to notice in this template is the use of the angucomplete-alt directive. We use it in order search customers with auto-complete support. In this directive we declared where to request the data from, the fields to display when an option is selected, a text to display till the request is completed, what to do when an option is selected or changed, etc... You can find more info about this awesome auto-complete directive here.



Rent Specter

Available Stock items

ec8886cb-6cfe-4390-93cf-7f8e07bc790f

Select customer

Aryanna Bergstrom

Leanna Watsica
elliot.cormier@veum.ca

Else Kovacek
annabell.larson@wallace.name

Aryanna Bergstrom
albin.kautzer@gorczanydonnelly.n

uncover a sinister organization. While M battles political
e layers of deceit to reveal the terrible truth behind

Specter

Specter
Directed by: **Sam Mendes**
Written by: **Ian Fleming**
Produced by: **Zakaria Alaoui**
Rating:

Rentals Filter..

#	Name	Rental date	Status	
24	Leanna Watsica	Saturday, August 15, 2015	Borrowed	Return
10	Katarina Howe	Wednesday, August 5, 2015	Returned	

Figure 26. Auto-complete customer search

spa/rental/rentMovieCtrl.js

```
(function (app) {
    'use strict';

    app.controller('rentMovieCtrl', rentMovieCtrl);

    rentMovieCtrl.$inject = ['$scope', '$modalInstance', '$location', 'apiService',
'notificationService'];

    function rentMovieCtrl($scope, $modalInstance, $location, apiService,
notificationService) {

        $scope.Title = $scope.movie.Title;
        $scope.loadStockItems = loadStockItems;
        $scope.selectCustomer = selectCustomer;
        $scope.selectionChanged = selectionChanged;
        $scope.rentMovie = rentMovie;
        $scope.cancelRental = cancelRental;
        $scope.stockItems = [];
        $scope.selectedCustomer = -1;
        $scope.isEnabled = false;

        function loadStockItems() {
            notificationService.displayInfo('Loading available stock items for ' +
$scope.movie.Title);

            apiService.get('/api/stocks/movie/' + $scope.movie.ID, null,
            stockItemsLoadCompleted,
            stockItemsLoadFailed);
        }

        function stockItemsLoadCompleted(response) {
            $scope.stockItems = response.data;
            $scope.selectedStockItem = $scope.stockItems[0].ID;
            console.log(response);
        }
        function stockItemsLoadFailed(response) {
            console.log(response);
            notificationService.displayError(response.data);
        }

        function rentMovie() {
            apiService.post('/api/rentals/rent/' + $scope.selectedCustomer + '/' +
$scope.selectedStockItem, null,
            rentMovieSucceeded,
            rentMovieFailed);
        }

        function rentMovieSucceeded(response) {
            notificationService.displaySuccess('Rental completed successfully');
            $modalInstance.close();
        }

        function rentMovieFailed(response) {
            notificationService.displayError(response.data.Message);
        }

        function cancelRental() {
            $scope.stockItems = [];
            $scope.selectedCustomer = -1;
            $scope.isEnabled = false;
            $modalInstance.dismiss();
        }

        function selectCustomer($item) {
            if ($item) {
                $scope.selectedCustomer = $item.originalObject.ID;
                $scope.isEnabled = true;
            }
            else {
                $scope.selectedCustomer = -1;
                $scope.isEnabled = false;
            }
        }

        function selectionChanged($item) {
        }

        loadStockItems();
    }

})(angular.module('homeCinema'));
```

When an employee wants to rent a specific movie to a customer, first he must find the stock item using a code displayed on the DVD the customer requested to borrow. That's why I highlighted the above lines in the *rentMovieCtrl* controller. Moreover, when he finally selects the stock item and the customer, he needs to press the *Rent movie* button and send a request to server with information about the selected customer and stock item as well. With all that said, we need to implement two more Web API actions. The first one will be in a new Web API Controller named StocksController and the second one responsible for movie rentals, inside the RentalsController.

StocksController.cs

```
    [Authorize(Roles="Admin")]
    [RoutePrefix("api/stocks")]
    public class StocksController : ApiControllerBase
    {
        private readonly IEntityBaseRepository<Stock> _stocksRepository;
        public StocksController(IEntityBaseRepository<Stock> stocksRepository,
            IEntityBaseRepository<Error> _errorsRepository, IUnitOfWork _unitOfWork)
            : base(_errorsRepository, _unitOfWork)
        {
            _stocksRepository = stocksRepository;
        }

        [Route("movie/{id:int}")]
        public HttpResponseMessage Get(HttpRequestMessage request, int id)
        {
            IEnumerable<Stock> stocks = null;

            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                stocks = _stocksRepository.GetAvailableItems(id);

                IEnumerable<StockViewModel> stocksVM = Mapper.Map<IEnumerable<Stock>,
 IEnumerable<StockViewModel>>(stocks);

                response =
 request.CreateResponse<IEnumerable<StockViewModel>>(HttpStatusCode.OK, stocksVM);

                return response;
            });
        }
    }
```

We need to create the StockViewModel class with its validator and of course the Automapper mapping.

Models/StockViewModel.cs

```
 public class StockViewModel : IValidatableObject
    {
        public int ID { get; set; }
        public Guid UniqueKey { get; set; }
        public bool IsAvailable { get; set; }

        public IEnumerable<ValidationResult> Validate(ValidationContext
 validationContext)
        {
            var validator = new StockViewModelValidator();
            var result = validator.Validate(this);
            return result.Errors.Select(item => new ValidationResult(item.ErrorMessage,
 new[] { item.PropertyName }));
        }
    }
```

Infrastructure.Validators/StockViewModelValidator.cs

```
public class StockViewModelValidator : AbstractValidator<StockViewModel>
    {
        public StockViewModelValidator()
        {
            RuleFor(s => s.ID).GreaterThan(0)
                .WithMessage("Invalid stock item");

            RuleFor(s => s.UniqueKey).NotEqual(Guid.Empty)
                .WithMessage("Invalid stock item");
        }
    }
```

part of DomainToViewModelMappingProfile.cs

```
protected override void Configure()
        {
            // code omitted
            Mapper.CreateMap<Stock, StockViewModel>();
        }
```

For the rental functionality we need add the following action in the RentalsController.

RentalsController Rent action

```
        [HttpPost]
        [Route("rent/{customerId:int}/{stockId:int}")]
        public HttpResponseMessage Rent(HttpRequestMessage request, int customerId, int
stockId)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                var customer = _customersRepository.GetSingle(customerId);
                var stock = _stocksRepository.GetSingle(stockId);

                if (customer == null || stock == null)
                {
                    response = request.CreateErrorResponse(HttpStatusCode.NotFound,
"Invalid Customer or Stock");
                }
                else
                {
                    if (stock.IsAvailable)
                    {
                        Rental _rental = new Rental()
                        {
                            CustomerId = customerId,
                            StockId = stockId,
                            RentalDate = DateTime.Now,
                            Status = "Borrowed"
                        };

                        _rentalsRepository.Add(_rental);
                        stock.IsAvailable = false;

                        _unitOfWork.Commit();

                        RentalViewModel rentalVm = Mapper.Map<Rental,
RentalViewModel>(_rental);

                        response =
request.CreateResponse<RentalViewModel>(HttpStatusCode.Created, rentalVm);
                    }
                    else
                        response =
request.CreateErrorResponse(HttpStatusCode.BadRequest, "Selected stock is not available
anymore");
                }

                return response;
            });
        }
```

The action accepts the customer id selected from the auto-complete textbox plus the stock's item id. Once more we need to add the required view model and Automapper mapping as follow *(no validator this time..)*.

Models/RentalViewModel.cs

```
public class RentalViewModel
    {
        public int ID { get; set; }
        public int CustomerId { get; set; }
        public int StockId { get; set; }
        public DateTime RentalDate { get; set; }
        public DateTime ReturnedDate { get; set; }
        public string Status { get; set; }
    }
```

part of DomainToViewModelMappingProfile.cs

```
protected override void Configure()
        {
            // code omitted
            Mapper.CreateMap<Rental, RentalViewModel>();
        }
```

Edit movie

From the *Details* view the user has the option to **edit** the movie by pressing the related button. This button redirects to route */movies/edit/:id* where id is selected movie's *ID*. Let's see the related route definition in *app.js*.

part of app.js

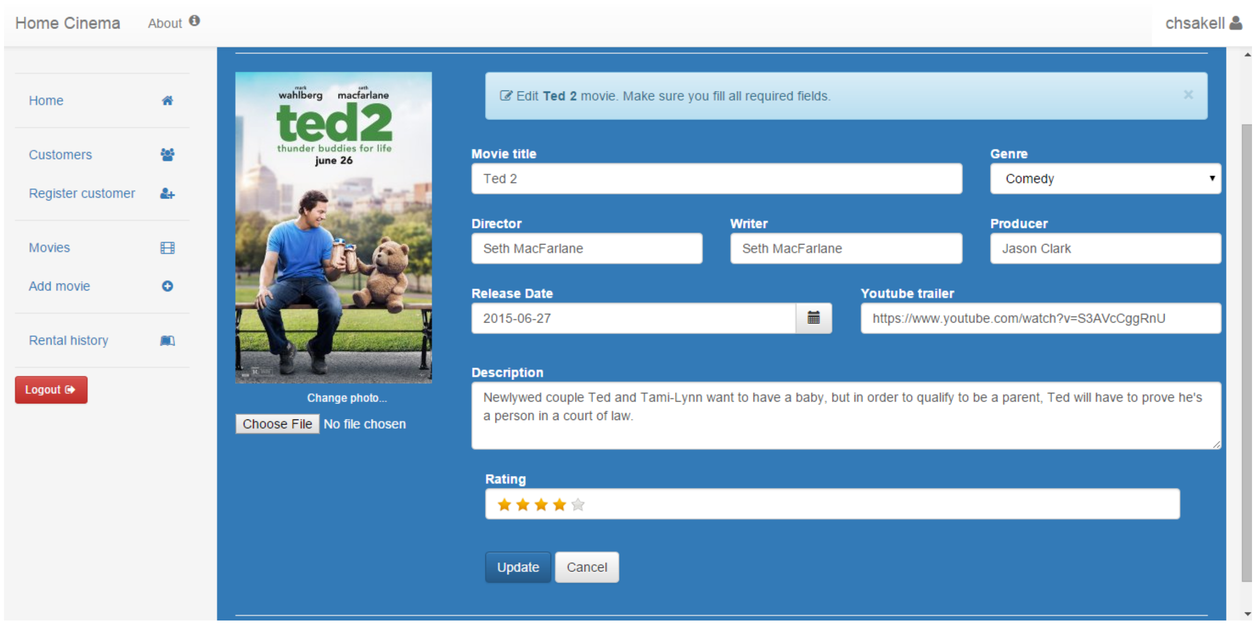
```
.when("/movies/edit/:id", {
```

```
                templateUrl: "scripts/spa/movies/edit.html",
                controller: "movieEditCtrl"
            })
```

Here we'll see for the first time how an **angularJS** controller can capture such a parameter from the route. Add the edit.html and its controller movieEditCtrl inside the *movies* folder.

spa/movies/edit.html

```
<div id="editMovieWrapper">
    <hr>
    <div class="row" ng-if="!loadingMovie">
        <!-- left column -->
        <div class="col-xs-3">
            <div class="text-center">
                <img ng-src="../../Content/images/movies/{{movie.Image}}" class="avatar
img-responsive" alt="avatar">
                <h6>Change photo...</h6>

                <input type="file" ng-file-select="prepareFiles($files)">
            </div>
        </div>

        <!-- edit form column -->
        <div class="col-xs-9 personal-info">
            <div class="alert alert-info alert-dismissable">
                <a class="panel-close close" data-dismiss="alert">×</a>
                <i class="fa fa-pencil-square-o"></i>
                Edit <strong>{{movie.Title}}</strong> movie. Make sure you fill all
required fields.
            </div>
            <form class="form-horizontal" role="form" novalidate angular-validator
name="editMovieForm" angular-validator-submit="UpdateMovie()">
                <div class="form-group">
                    <div class="row">
                        <div class="col-xs-8">
                            <label class="control-label">Movie title</label>
                            <input class="form-control" name="title" type="text" ng-
model="movie.Title"
                                   validate-on="blur" required required-message="'Movie
title is required'">
                        </div>

                        <div class="col-xs-4 selectContainer">
                            <label class="control-label">Genre</label>
                            <select ng-model="movie.GenreId" class="form-control black"
ng-options="option.ID as option.Name for option in genres" required></select>
                            <input type="hidden" name="GenreId" ng-
value="movie.GenreId" />
                        </div>
                    </div>
                </div>

                <div class="form-group">
                    <div class="row">
                        <div class="col-xs-4">
                            <label class="control-label">Director</label>
                            <input class="form-control" type="text" ng-
model="movie.Director" name="director"
                                   validate-on="blur" required required-message="'Movie
director is required'">
                        </div>

                        <div class="col-xs-4">
                            <label class="control-label">Writer</label>
                            <input class="form-control" type="text" ng-
model="movie.Writer" name="writer"
                                   validate-on="blur" required required-message="'Movie
writer is required'">
                        </div>

                        <div class="col-xs-4">
                            <label class="control-label">Producer</label>
                            <input class="form-control" type="text" ng-
model="movie.Producer" name="producer"
                                   validate-on="blur" required required-message="'Movie
producer is required'">
                        </div>
                    </div>
                </div>

                <div class="form-group">
                    <div class="row">
                        <div class="col-xs-6">
                            <label class="control-label">Release Date</label>
                            <p class="input-group">
                                <input type="text" class="form-control"
name="dateReleased" datepicker-popup="{{format}}" ng-model="movie.ReleaseDate" is-
open="datepicker.opened" datepicker-options="dateOptions" ng-required="true"
datepicker-append-to-body="true" close-text="Close"
                                       validate-on="blur" required required-
message="'Date Released is required'" />
                                <span class="input-group-btn">
                                    <button type="button" class="btn btn-default" ng-
click="openDatePicker($event)"><i class="glyphicon glyphicon-calendar"></i></button>
                                </span>
                            </p>
                        </div>

                        <div class="col-xs-6">
                            <label class="control-label">Youtube trailer</label>
                            <input class="form-control" type="text" ng-
model="movie.TrailerURI" name="trailerURI"
                                   validate-on="blur" required required-message="'Movie
trailer is required'" ng-
pattern="/^(https?\:\/\/)?(www\.youtube\.com|youtu\.?be)\/.+$/"
                                   invalid-message="'You must enter a valid YouTube
URL'">
                        </div>
                    </div>
                </div>

                <div class="form-group">
                    <label class="control-label">Description</label>
                    <textarea class="form-control" ng-model="movie.Description"
name="description" rows="3"
                              validate-on="blur" required required-message="'Movie
description is required'" />
                </div>

                <div class="form-group col-xs-12">
                    <label class="control-label">Rating</label>
                    <span component-rating="{{movie.Rating}}" ng-model="movie.Rating"
class="form-control"></span>
                </div>
                <br />
                <div class="form-group col-xs-4">
                    <label class="control-label"></label>
                    <div class="">
                        <input type="submit" class="btn btn-primary" value="Update" />
                        <span></span>
                        <a class="btn btn-default" ng-
href="#/movies/{{movie.ID}}">Cancel</a>
                    </div>
                </div>
            </form>
        </div>
    </div>
    <hr>
</div>
```



Edit **Ted 2** movie. Make sure you fill all required fields.

Movie title: Ted 2
Genre: Comedy
Director: Seth MacFarlane
Writer: Seth MacFarlane
Producer: Jason Clark
Release Date: 2015-06-27
Youtube trailer: https://www.youtube.com/watch?v=S3AVcCggRnU
Description: Newlywed couple Ted and Tami-Lynn want to have a baby, but in order to qualify to be a parent, Ted will have to prove he's a person in a court of law.
Rating: ★★★★☆

Figure 27. Edit movie view

We used an **ng-file-select** directive which in conjunction with the *3rd party* library angular-file-upload will handle the movie image uploading through a Web API controller action. You can read more about file uploading using Web API and angularJS here, where I described the process step by step.

spa/movies/movieEditCtrl.js

```
(function (app) {
    'use strict';

    app.controller('movieEditCtrl', movieEditCtrl);

    movieEditCtrl.$inject = ['$scope', '$location', '$routeParams', 'apiService',
'notificationService', 'fileUploadService'];

    function movieEditCtrl($scope, $location, $routeParams, apiService,
notificationService, fileUploadService) {
        $scope.pageClass = 'page-movies';
        $scope.movie = {};
        $scope.genres = [];
        $scope.loadingMovie = true;
        $scope.isReadOnly = false;
        $scope.UpdateMovie = UpdateMovie;
        $scope.prepareFiles = prepareFiles;
        $scope.openDatePicker = openDatePicker;

        $scope.dateOptions = {
            formatYear: 'yy',
            startingDay: 1
        };
        $scope.datepicker = {};

        var movieImage = null;

        function loadMovie() {

            $scope.loadingMovie = true;

            apiService.get('/api/movies/details/' + $routeParams.id, null,
            movieLoadCompleted,
            movieLoadFailed);
        }

        function movieLoadCompleted(result) {
            $scope.movie = result.data;
            $scope.loadingMovie = false;

            loadGenres();
        }

        function movieLoadFailed(response) {
            notificationService.displayError(response.data);
        }

        function genresLoadCompleted(response) {
            $scope.genres = response.data;
        }

        function genresLoadFailed(response) {
            notificationService.displayError(response.data);
        }

        function loadGenres() {
            apiService.get('/api/genres/', null,
            genresLoadCompleted,
            genresLoadFailed);
        }

        function UpdateMovie() {
            if (movieImage) {
                fileUploadService.uploadImage(movieImage, $scope.movie.ID,
UpdateMovieModel);
            }
            else
                UpdateMovieModel();
        }

        function UpdateMovieModel() {
            apiService.post('/api/movies/update', $scope.movie,
            updateMovieSucceded,
            updateMovieFailed);
        }

        function prepareFiles($files) {
            movieImage = $files;
        }

        function updateMovieSucceded(response) {
            console.log(response);
            notificationService.displaySuccess($scope.movie.Title + ' has been
updated');
            $scope.movie = response.data;
            movieImage = null;
        }

        function updateMovieFailed(response) {
            notificationService.displayError(response);
        }

        function openDatePicker($event) {
            $event.preventDefault();
            $event.stopPropagation();

            $scope.datepicker.opened = true;
        };

        loadMovie();
    }

})(angular.module('homeCinema'));
```

The movieEditController sets a *$scope* variable named *isReadOnly* so that the rating component be editable as we have already discussed. When the user submits the form, if the form is valid it checks if any selected file exists. If so, starts with the image file uploading and continues with the movie details updating. If user hasn't selected an image then only the movie details are being updated. For the image file uploading feature, we injected a **fileUploadService** service in our controller. Let's create that service inside the *services* folder.

spa/services/fileUploadService.js

```
(function (app) {
    'use strict';

    app.factory('fileUploadService', fileUploadService);

    fileUploadService.$inject = ['$rootScope', '$http', '$timeout', '$upload',
'notificationService'];

    function fileUploadService($rootScope, $http, $timeout, $upload,
notificationService) {

        $rootScope.upload = [];

        var service = {
            uploadImage: uploadImage
        }

        function uploadImage($files, movieId, callback) {
            //$files: an array of files selected
            for (var i = 0; i < $files.length; i++) {
                var $file = $files[i];
                (function (index) {
                    $rootScope.upload[index] = $upload.upload({
                        url: "api/movies/images/upload?movieId=" + movieId, // webapi
url
                        method: "POST",
                        file: $file
                    }).progress(function (evt) {
                    }).success(function (data, status, headers, config) {
                        // file is uploaded successfully
                        notificationService.displaySuccess(data.FileName + ' uploaded
successfully');
                        callback();
                    }).error(function (data, status, headers, config) {
                        notificationService.displayError(data.Message);
                    });
                })(i);
            }
        }

        return service;
    }

})(angular.module('common.core'));
```

It is time to switch to server side again and support the image file uploading. We need to create two helper classes. The first one is the class that will contain the file upload result. Add the FileUploadResult class inside the *Infrastructure.Core* folder.

FileUploadResult.cs

```
public class FileUploadResult
    {
        public string LocalFilePath { get; set; }
        public string FileName { get; set; }
        public long FileLength { get; set; }
    }
```

We also need to ensure that the request's content for file uploading is **MIME multipart**. Let's create the following Web API action filter inside the *Infrastructure.Core* folder.

MimeMultipart.cs

```
public class MimeMultipart : System.Web.Http.Filters.ActionFilterAttribute
    {
        public override void OnActionExecuting(HttpActionContext actionContext)
        {
            if (!actionContext.Request.Content.IsMimeMultipartContent())
            {
                throw new HttpResponseException(
                    new HttpResponseMessage(
                        HttpStatusCode.UnsupportedMediaType)
                );
            }
        }

        public override void OnActionExecuted(HttpActionExecutedContext
actionExecutedContext)
        {

        }
    }
```

And here's the *images/upload* Web API action in the MoviesController.

MoviesController images/upload action

```
        [MimeMultipart]
        [Route("images/upload")]
        public HttpResponseMessage Post(HttpRequestMessage request, int movieId)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                var movieOld = _moviesRepository.GetSingle(movieId);
                if (movieOld == null)
                    response = request.CreateErrorResponse(HttpStatusCode.NotFound,
"Invalid movie.");
                else
                {
                    var uploadPath =
HttpContext.Current.Server.MapPath("~/Content/images/movies");

                    var multipartFormDataStreamProvider = new
UploadMultipartFormProvider(uploadPath);

                    // Read the MIME multipart asynchronously

Request.Content.ReadAsMultipartAsync(multipartFormDataStreamProvider);
```

```
                    string _localFileName = multipartFormDataStreamProvider
                        .FileData.Select(multiPartData =>
multiPartData.LocalFileName).FirstOrDefault();

                    // Create response
                    FileUploadResult fileUploadResult = new FileUploadResult
                    {
                        LocalFilePath = _localFileName,

                        FileName = Path.GetFileName(_localFileName),

                        FileLength = new FileInfo(_localFileName).Length
                    };

                    // update database
                    movieOld.Image = fileUploadResult.FileName;
                    _moviesRepository.Edit(movieOld);
                    _unitOfWork.Commit();

                    response = request.CreateResponse(HttpStatusCode.OK,
fileUploadResult);
                }

                return response;
            });
        }
```

For the Update movie operation we need to add an extra extension method as we did with the Customers. Add the following method inside the EntitiesExtensions class.

part of Infrastructure.Extensions.EntitiesExtensions.cs

```
public static void UpdateMovie(this Movie movie, MovieViewModel movieVm)
        {
            movie.Title = movieVm.Title;
            movie.Description = movieVm.Description;
            movie.GenreId = movieVm.GenreId;
            movie.Director = movieVm.Director;
            movie.Writer = movieVm.Writer;
            movie.Producer = movieVm.Producer;
            movie.Rating = movieVm.Rating;
            movie.TrailerURI = movieVm.TrailerURI;
            movie.ReleaseDate = movieVm.ReleaseDate;
        }
```

Check that the *Image* property is missing since this property is changed only when uploading a movie image. Here's the *Update action* in the MoviesController.

MoviesController update action

```
        [HttpPost]
        [Route("update")]
        public HttpResponseMessage Update(HttpRequestMessage request, MovieViewModel
movie)
```

```
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                if (!ModelState.IsValid)
                {
                    response = request.CreateErrorResponse(HttpStatusCode.BadRequest,
ModelState);
                }
                else
                {
                    var movieDb = _moviesRepository.GetSingle(movie.ID);
                    if (movieDb == null)
                        response = request.CreateErrorResponse(HttpStatusCode.NotFound,
"Invalid movie.");
                    else
                    {
                        movieDb.UpdateMovie(movie);
                        movie.Image = movieDb.Image;
                        _moviesRepository.Edit(movieDb);

                        _unitOfWork.Commit();
                        response =
request.CreateResponse<MovieViewModel>(HttpStatusCode.OK, movie);
                    }
                }

                return response;
            });
        }
```

Add movie

The *add movie* feature is pretty much the same as the *edit* one. We need one template named add.html for the add operation and a controller named *movieAddCtrl*. Add the following files inside the *movies* folder.

spa/movies/add.html

```
<div id="editMovieWrapper">
    <hr>
    <div class="row">
        <!-- left column -->
        <div class="col-xs-3">
            <div class="text-center">
                <img ng-src="../../Content/images/movies/unknown.jpg" class="avatar
img-responsive" alt="avatar">
                <h6>Add photo...</h6>

                <input type="file" ng-file-select="prepareFiles($files)">
            </div>
        </div>

        <!-- edit form column -->
        <div class="col-xs-9 personal-info">
            <div class="alert alert-info alert-dismissable">
```

```
                <a class="panel-close close" data-dismiss="alert">×</a>
                <i class="fa fa-plus"></i>
                Add <strong>{{movie.Title}}</strong> movie. Make sure you fill all
required fields.
            </div>

            <form class="form-horizontal" role="form" novalidate angular-validator
name="addMovieForm" angular-validator-submit="AddMovie()">
                <div class="form-group">
                    <div class="row">
                        <div class="col-xs-6 col-sm-4">
                            <label class="control-label">Movie title</label>
                            <input class="form-control" name="title" type="text" ng-
model="movie.Title"
                                   validate-on="blur" required required-message="'Movie
title is required'">
                        </div>

                        <div class="col-xs-6 col-sm-4 selectContainer">
                            <label class="control-label">Genre</label>
                            <select ng-model="movie.GenreId" class="form-control black"
ng-options="option.ID as option.Name for option in genres" required></select>
                            <input type="hidden" name="GenreId" ng-
value="movie.GenreId" />
                        </div>

                        <div class="col-xs-6 col-sm-4">
                            <label class="control-label">Stocks</label>
                            <div class="input-group number-spinner">
                                <span class="input-group-btn">
                                    <button class="btn btn-default" data-
dir="dwn"><span class="glyphicon glyphicon-minus"></span></button>
                                </span>
                                <input type="text" class="form-control text-center"
id="inputStocks" ng-model="movie.NumberOfStocks">
                                <span class="input-group-btn">
                                    <button type="button" class="btn btn-default" ng-
click="changeNumberOfStocks($event)" id="btnSetStocks" data-dir="up"><span
class="glyphicon glyphicon-plus"></span></button>
                                </span>
                            </div>
                        </div>
                    </div>
                </div>

                <div class="form-group">
                    <div class="row">
                        <div class="col-xs-4">
                            <label class="control-label">Director</label>
                            <input class="form-control" type="text" ng-
model="movie.Director" name="director"
                                   validate-on="blur" required required-message="'Movie
director is required'">
                        </div>

                        <div class="col-xs-4">
                            <label class="control-label">Writer</label>
                            <input class="form-control" type="text" ng-
model="movie.Writer" name="writer"
                                   validate-on="blur" required required-message="'Movie
writer is required'">
                        </div>

                        <div class="col-xs-4">
                            <label class="control-label">Producer</label>
                            <input class="form-control" type="text" ng-
model="movie.Producer" name="producer"
                                   validate-on="blur" required required-message="'Movie
producer is required'">
                        </div>
                    </div>
                </div>

                <div class="form-group">
                    <div class="row">
                        <div class="col-xs-6">
                            <label class="control-label">Release Date</label>
                            <p class="input-group">
                                <input type="text" class="form-control"
name="dateReleased" datepicker-popup="{{format}}" ng-model="movie.ReleaseDate" is-
open="datepicker.opened" datepicker-options="dateOptions" ng-required="true"
datepicker-append-to-body="true" close-text="Close"
                                       validate-on="blur" required required-
message="'Date Released is required'" />
                                <span class="input-group-btn">
                                    <button type="button" class="btn btn-default" ng-
click="openDatePicker($event)"><i class="glyphicon glyphicon-calendar"></i></button>
                                </span>
                            </p>
                        </div>

                        <div class="col-xs-6">
                            <label class="control-label">Youtube trailer</label>
                            <input class="form-control" type="text" ng-
model="movie.TrailerURI" name="trailerURI"
                                   validate-on="blur" required required-message="'Movie
trailer is required'" ng-
pattern="/^(https?\:\/\/)?(www\.youtube\.com|youtu\.?be)\/.+$/"
                                   invalid-message="'You must enter a valid YouTube
URL'">
                        </div>
                    </div>
                </div>

                <div class="form-group">
                    <label class="control-label">Description</label>
                    <textarea class="form-control" ng-model="movie.Description"
name="description" rows="3"
                              validate-on="blur" required required-message="'Movie
description is required'" />
                </div>

                <div class="form-group col-xs-12">
                    <label class="control-label">Rating</label>
                    <span component-rating="{{movie.Rating}}" ng-model="movie.Rating"
class="form-control"></span>
                </div>
                <br/>
                <div class="form-group col-xs-4">
                    <label class="control-label"></label>
                    <div class="">
                        <input type="submit" class="btn btn-primary" value="Submit
movie" />
                        <span></span>
                        <a class="btn btn-default" ng-
href="#/movies/{{movie.ID}}">Cancel</a>
                    </div>
                </div>
            </form>
        </div>
    </div>
    <hr>
</div>
```

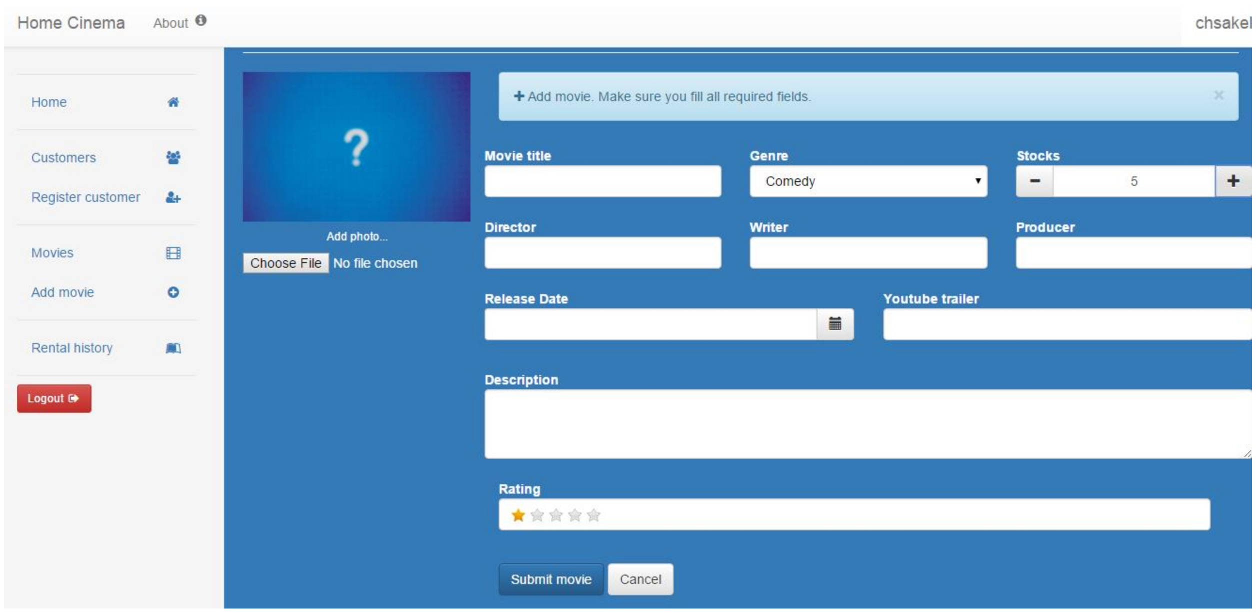


Figure 28. Add movie view

spa/movies/movieAddCtrl.js

```
(function (app) {
    'use strict';

    app.controller('movieAddCtrl', movieAddCtrl);

    movieAddCtrl.$inject = ['$scope', '$location', '$routeParams', 'apiService',
'notificationService', 'fileUploadService'];

    function movieAddCtrl($scope, $location, $routeParams, apiService,
notificationService, fileUploadService) {

        $scope.pageClass = 'page-movies';
        $scope.movie = { GenreId: 1, Rating: 1, NumberOfStocks: 1 };

        $scope.genres = [];
        $scope.isReadOnly = false;
        $scope.AddMovie = AddMovie;
        $scope.prepareFiles = prepareFiles;
        $scope.openDatePicker = openDatePicker;
        $scope.changeNumberOfStocks = changeNumberOfStocks;

        $scope.dateOptions = {
            formatYear: 'yy',
            startingDay: 1
        };
        $scope.datepicker = {};

        var movieImage = null;

        function loadGenres() {
            apiService.get('/api/genres/', null,
            genresLoadCompleted,
            genresLoadFailed);
        }

        function genresLoadCompleted(response) {
            $scope.genres = response.data;
        }

        function genresLoadFailed(response) {
            notificationService.displayError(response.data);
        }

        function AddMovie() {
            AddMovieModel();
        }

        function AddMovieModel() {
            apiService.post('/api/movies/add', $scope.movie,
            addMovieSucceded,
            addMovieFailed);
        }

        function prepareFiles($files) {
            movieImage = $files;
        }

        function addMovieSucceded(response) {
            notificationService.displaySuccess($scope.movie.Title + ' has been 
submitted to Home Cinema');
            $scope.movie = response.data;

            if (movieImage) {
                fileUploadService.uploadImage(movieImage, $scope.movie.ID, 
redirectToEdit);
            }
            else
                redirectToEdit();
        }
        function addMovieFailed(response) {
            console.log(response);
            notificationService.displayError(response.statusText);
        }

        function openDatePicker($event) {
            $event.preventDefault();
            $event.stopPropagation();

            $scope.datepicker.opened = true;
        };

        function redirectToEdit() {
            $location.url('movies/edit/' + $scope.movie.ID);
        }

        function changeNumberOfStocks($vent)
        {
            var btn = $('#btnSetStocks'),
            oldValue = $('#inputStocks').val().trim(),
            newVal = 0;

            if (btn.attr('data-dir') == 'up') {
                newVal = parseInt(oldValue) + 1;
            } else {
                if (oldValue > 1) {
                    newVal = parseInt(oldValue) - 1;
                } else {
                    newVal = 1;
                }
            }
            $('#inputStocks').val(newVal);
            $scope.movie.NumberOfStocks = newVal;
            console.log($scope.movie);
        }

        loadGenres();
    }

})(angular.module('homeCinema'));
```

There are two things here to notice. The first one is that we have to setup somehow the number of stocks for that movie. Normally, each **DVD** stock item has its own code but just for convenience, we will create this code automatically in the controller. The other is that since we need the movie to exist before we upload an image, we ensure that the upload image operation comes second by uploading the image (if exists) after the movie has been added to database. When the operation is completed, we redirect to edit the image. Now let's see the required Add Web API action in the *MoviesController*.

MoviesController Add action

```
        [HttpPost]
        [Route("add")]
        public HttpResponseMessage Add(HttpRequestMessage request, MovieViewModel
movie)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                if (!ModelState.IsValid)
                {
                    response = request.CreateErrorResponse(HttpStatusCode.BadRequest,
ModelState);
                }
                else
                {
                    Movie newMovie = new Movie();
                    newMovie.UpdateMovie(movie);

                    for (int i = 0; i < movie.NumberOfStocks; i++)
                    {
                        Stock stock = new Stock()
                        {
                            IsAvailable = true,
                            Movie = newMovie,
                            UniqueKey = Guid.NewGuid()
                        };
                        newMovie.Stocks.Add(stock);
                    }

                    _moviesRepository.Add(newMovie);

                    _unitOfWork.Commit();

                    // Update view model
                    movie = Mapper.Map<Movie, MovieViewModel>(newMovie);
                    response =
request.CreateResponse<MovieViewModel>(HttpStatusCode.Created, movie);
                }

                return response;
            });
        }
```

Rental history

One last thing remained for our Single Page Application is to display a **stastistic** rental history for movies that have been rented at least one. We want to create a specific chart for each movie rented, that shows for each date how many rentals occurred. Let's start from the server-side. We need the following ViewModel class that will hold the information required for each movie's rental statistics.

Models/TotalRentalHistoryViewModel.cs

```
public class TotalRentalHistoryViewModel
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public string Image { get; set; }
        public int TotalRentals
        {
            get
            {
                return Rentals.Count;
            }
            set { }
        }
        public List<RentalHistoryPerDate> Rentals { get; set; }
    }

    public class RentalHistoryPerDate
    {
        public int TotalRentals { get; set; }
        public DateTime Date { get; set; }
    }
```

Switch to the RentalsController and add the following **GetMovieRentalHistoryPerDates** private method. If you remember, we have already created a private method *GetMovieRentalHistory* to get rental history for a specific movie so getting the rental history for all movies should not be a problem.

RentalsController private method

```
private List<RentalHistoryPerDate> GetMovieRentalHistoryPerDates(int movieId)
        {
            List<RentalHistoryPerDate> listHistory = new List<RentalHistoryPerDate>();
            List<RentalHistoryViewModel> _rentalHistory =
GetMovieRentalHistory(movieId);
            if (_rentalHistory.Count > 0)
            {
                List<DateTime> _distinctDates = new List<DateTime>();
                _distinctDates = _rentalHistory.Select(h =>
h.RentalDate.Date).Distinct().ToList();

                foreach (var distinctDate in _distinctDates)
                {
                    var totalDateRentals = _rentalHistory.Count(r => r.RentalDate.Date
== distinctDate);
                    RentalHistoryPerDate _movieRentalHistoryPerDate = new
RentalHistoryPerDate()
                    {
                        Date = distinctDate,
                        TotalRentals = totalDateRentals
                    };

                    listHistory.Add(_movieRentalHistoryPerDate);
                }

                listHistory.Sort((r1, r2) => r1.Date.CompareTo(r2.Date));
            }

            return listHistory;
        }
```

Only thing required here was to sort the results by ascending dates. And now the action.

RentalsController TotalRentalHistory action

```
        [HttpGet]
        [Route("rentalhistory")]
        public HttpResponseMessage TotalRentalHistory(HttpRequestMessage request)
        {
            return CreateHttpResponse(request, () =>
            {
                HttpResponseMessage response = null;

                List<TotalRentalHistoryViewModel> _totalMoviesRentalHistory = new
List<TotalRentalHistoryViewModel>();

                var movies = _moviesRepository.GetAll();

                foreach (var movie in movies)
                {
                    TotalRentalHistoryViewModel _totalRentalHistory = new
TotalRentalHistoryViewModel()
                    {
                        ID = movie.ID,
                        Title = movie.Title,
                        Image = movie.Image,
                        Rentals = GetMovieRentalHistoryPerDates(movie.ID)
                    };

                    if (_totalRentalHistory.TotalRentals > 0)
                        _totalMoviesRentalHistory.Add(_totalRentalHistory);
                }

                response =
request.CreateResponse<List<TotalRentalHistoryViewModel>>(HttpStatusCode.OK,
_totalMoviesRentalHistory);

                return response;
            });
        }
```

On the front-side we need to create a rental.html template and a rentalStatsCtrl controller. Add the following files inside the *rental* folder.

spa/rental/rental.html

```
<hr />
<div class="row" ng-repeat="movie in rentals" ng-if="!loadingStatistics">
    <div class="panel panel-primary">
        <div class="panel-heading">
            <span>{{movie.Title}}</span>
        </div>
        <div class="panel-body">
            <div class="col-xs-3">
                <div class="panel panel-default">
                    <div class="panel-heading">
                        <strong class="ng-binding">{{movie.Title}} </strong>
                    </div>
                    <div class="panel-body">
                        <div class="media">
```

```
                            <img class="media-object center-block img-responsive
center-block"
                                 ng-src="../../Content/images/movies/{{movie.Image}}"
alt="{{movie.Title}}">
                        </div>
                    </div>
                </div>
            </div>
            <div class="col-xs-9">
                <div class="panel panel-default text-center">
                    <div class="panel-body">
                        <div id="statistics-{{movie.ID}}">

                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div> <!-- end row -->
```

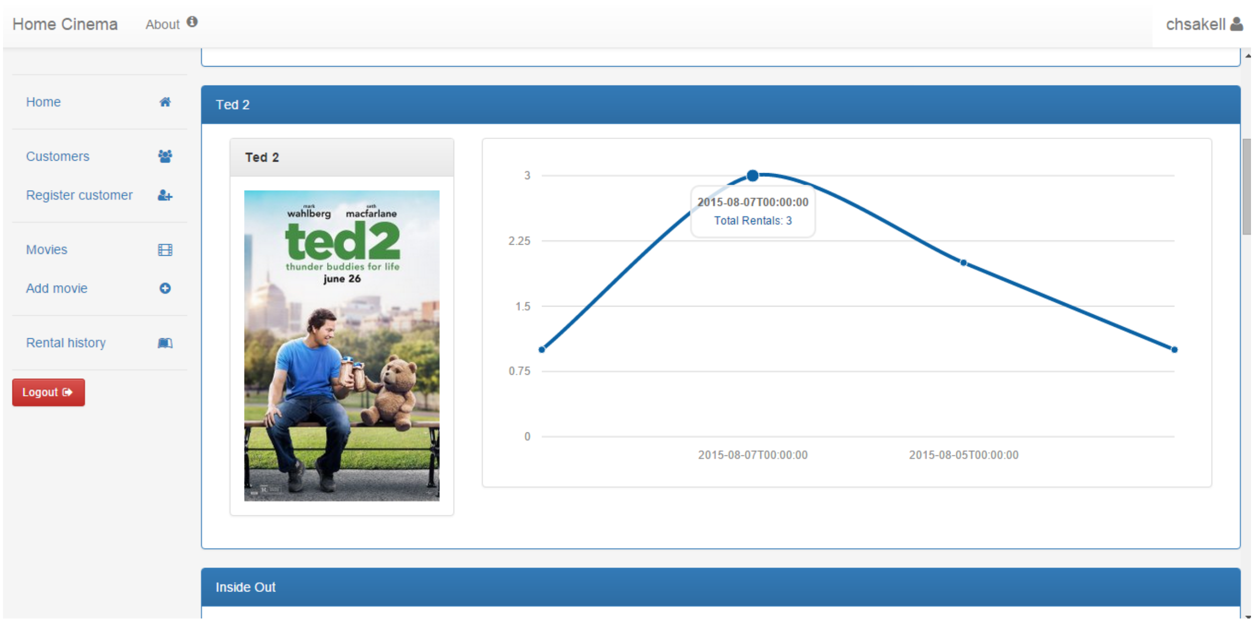


Figure 29. Rental history per dates

spa/rental/rentalStatsCtrl.js

```
(function (app) {
    'use strict';

    app.controller('rentStatsCtrl', rentStatsCtrl);

    rentStatsCtrl.$inject = ['$scope', 'apiService', 'notificationService',
'$timeout'];

    function rentStatsCtrl($scope, apiService, notificationService, $timeout) {
        $scope.loadStatistics = loadStatistics;
        $scope.rentals = [];

        function loadStatistics() {
            $scope.loadingStatistics = true;

            apiService.get('/api/rentals/rentalhistory', null,
            rentalHistoryLoadCompleted,
            rentalHistoryLoadFailed);
        }

        function rentalHistoryLoadCompleted(result) {
            $scope.rentals = result.data;

            $timeout(function () {
                angular.forEach($scope.rentals, function (rental) {
                    if (rental.TotalRentals > 0) {

                        var movieRentals = rental.Rentals;

                        Morris.Line({
                            element: 'statistics-' + rental.ID,
                            data: movieRentals,
                            parseTime: false,
                            lineWidth: 4,
                            xkey: 'Date',
                            xlabels: 'day',
                            resize: 'true',
                            ykeys: ['TotalRentals'],
                            labels: ['Total Rentals']
                        });
                    }
                })
            }, 1000);

            $scope.loadingStatistics = false;
        }

        function rentalHistoryLoadFailed(response) {
            notificationService.displayError(response.data);
        }

        loadStatistics();
    }

})(angular.module('homeCinema'));
```

You need to create the Morris chart after the data have been loaded, otherwise it won't work. Notice that we also needed a different *id* for the element where the chart will be hosted and for that we used the movie's id.

At this point, you should be able to run the HomeCinema Single Page Application. In case you have any problems, you can always download the source code from my **Github** account and follow the installation instructions.

Discussion

Scaling

There are 2 main things I would like to discuss about the application we created. The first one is talk about scaling the application to support more roles. I'm talking about feature-level scaling for example suppose that you have an extra requirement that this web application should be accessible by external users too, the Customers. What would that mean for this application? The first thing popped up to my mind is more templates, more Web API controllers and last but not least even more *JavaScript* files. Guess what, that's all true. The thing is that after such requirement our *HomeCinema* application will not be entirely **Single Page Application**. It could, but it would be wrong.

Let me explain what I mean by that. From the server side and the Web API framework, we have no restrictions at all. We were clever enough to create a custom membership schema with custom roles and then apply **Basic Authentication** through message handlers. This means that if you want to restrict new views only to customers, what you need to do is create a new *Role* named *Customer* and apply an [Authorize(Roles = "Customer")] attribute to the respective controllers or actions. The big problem is on the front-end side. You could add more functionality, templates and controllers upon the same *root* module **homeCinema** but believe me, very soon you wouldn't be able to maintain all those files. Instead, you can simply break your application in two *Single Page Applications*, one responsible for employees and another for customers. This means that you would have two MVC views, not one *(one more action and view in the HomeController*. Each of these views is a *Single Page Application* by itself and has its own module. And if you wish, you can go ever further and adapt a new architecture for the front-end side of your application. Let's say that both the *Admin* and *Customer* roles, require really many views which means you still have the maintainability issue. Then you have to break your SPAs even more and have a specific MVC View (spa) for each sub-feature. All the relative sub-views can share the same **Layout** page which is responsible to render the required *JavaScript* files. Let's see the architecture graphically.

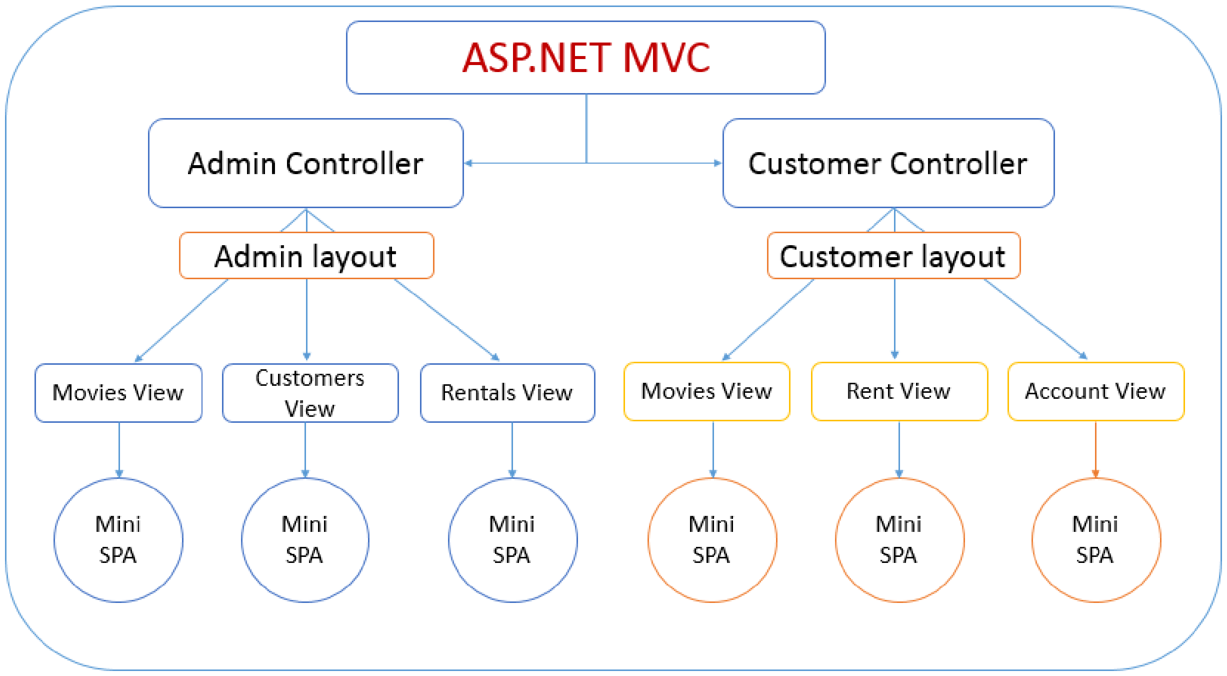


Figure 30. Extended SPA architecture

_*Layout* pages may have a **RenderScript** Razor definitions as follow:

_Layout.cshtml

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title></title>
    <meta name="viewport" content="width=device-width" />
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body data-ng-app="adminApp">
    <div class="wrapper">
        <div class="container">
            <section>
                @RenderBody()
            </section>
        </div>
    </div>
    </div>

    @Scripts.Render("~/bundles/vendors")
    <script src="@Url.Content("~/Scripts/spa/app.js")" type="text/javascript"></script>

    @RenderSection("scripts", required: false)
    <script type="text/javascript">
        @RenderSection("customScript", required: false)
    </script>
</body>
```

```
</html>
```

What we did here is render all the required css and vendor files but all MVC views having this _Layout page can (and must) render their own *JavaScript* files through the **scripts** and **customScript** sections. The first section concerns *angularJS* required components and the latter concerns custom JavaScript code. Remember the following part we added in the *run* method of the *app.js* file?

part of app.js file

```
$(document).ready(function () {
            $(".fancybox").fancybox({
                openEffect: 'none',
                closeEffect: 'none'
            });

            $('.fancybox-media').fancybox({
                openEffect: 'none',
                closeEffect: 'none',
                helpers: {
                    media: {}
                }
            });

            $('[data-toggle=offcanvas]').click(function () {
                $('.row-offcanvas').toggleClass('active');
            });
        });
```

That's the type of code that should be written in the customScript section of an MVC view. Now let's how an MVC View would look like with this architecture.

Custom MVC View

```
@section scripts
{
    <script src="@Url.Content("~/Scripts/spa/customController.js")"
type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/spa/customDirective.js")"
type="text/javascript"></script>
}
@section footerScript
{
    angular.bootstrap(document.getElementById("customApp"),['customApp']);

    $(document).ready(function () {
    //...
    });
}
<div ng-app="customApp" id="customApp">
    <div ng-view></div>
</div>
```

Generic Repository Factory

One last thing I would like to discuss is the way data repositories are injected into Web API controllers. We followed a **generic repository pattern** and all we have to do in order to use a specific repository is inject it to the constructor. But let's take a look the RentalsController constructor.

part of RentalsController

```
public class RentalsController : ApiControllerBase
    {
        private readonly IEntityBaseRepository<Rental> _rentalsRepository;
        private readonly IEntityBaseRepository<Customer> _customersRepository;
        private readonly IEntityBaseRepository<Stock> _stocksRepository;
        private readonly IEntityBaseRepository<Movie> _moviesRepository;

        public RentalsController(IEntityBaseRepository<Rental> rentalsRepository,
            IEntityBaseRepository<Customer> customersRepository,
IEntityBaseRepository<Movie> moviesRepository,
            IEntityBaseRepository<Stock> stocksRepository,
            IEntityBaseRepository<Error> _errorsRepository, IUnitOfWork _unitOfWork)
            : base(_errorsRepository, _unitOfWork)
        {
            _rentalsRepository = rentalsRepository;
            _moviesRepository = moviesRepository;
            _customersRepository = customersRepository;
            _stocksRepository = stocksRepository;
        }
    }
```

It's kind of a mess, isn't it? What if a controller requires much more repositories? Well, there's another trick you can do to keep your controllers cleaner as much as possible and this is via the use of a **Generic DataRepository Factory**. Let's see how we can accomplish it and re-write the RentalsController.

First, we need to add a new HttpRequestMessage extension which will allow us to resolve instances of IEntityBaseRepository<T> so switch to RequestMessageExtensions file and add the following method.

part of RequestMessageExtensions.cs

```
internal static IEntityBaseRepository<T> GetDataRepository<T>(this HttpRequestMessage
request) where T : class, IEntityBase, new()
        {
            return request.GetService<IEntityBaseRepository<T>>();
        }
```

Here we can see how useful the *IEntityBase* interface is in our application. We can use this interface to resolve data repositories for our entities. Now let's create the generic **Factory**. Add the following file inside the *Infrastructure/Core* folder.

DataRepositoryFactory.cs

```
public class DataRepositoryFactory : IDataRepositoryFactory
    {
        public IEntityBaseRepository<T> GetDataRepository<T>(HttpRequestMessage
request) where T : class, IEntityBase, new()
        {
            return request.GetDataRepository<T>();
        }
    }

    public interface IDataRepositoryFactory
    {
        IEntityBaseRepository<T> GetDataRepository<T>(HttpRequestMessage request) where
T : class, IEntityBase, new();
    }
```

This factory has a generic method of type IEntityBase that invokes the extension method we wrote before. We will use this method in the ApiControllerBase class, which the base class for our Web API controllers. Before doing this, switch to AutofacWebapiConfig class where we configured the dependency injection and add the following lines before the end of the **RegisterServices** function.

part of AutofacWebapiConfig class

```
// code omitted
            // Generic Data Repository Factory
            builder.RegisterType<DataRepositoryFactory>()
                .As<IDataRepositoryFactory>().InstancePerRequest();

            Container = builder.Build();

            return Container;
```

I don't want to change the current ApiControllerBase implementation so I will write a new one just for the demonstration. For start let's see its definition along with its variables.

Infrastructure/Core/ApiControllerBaseExtended.cs

```
public class ApiControllerBaseExtended : ApiController
    {
        protected List<Type> _requiredRepositories;

        protected readonly IDataRepositoryFactory _dataRepositoryFactory;
        protected IEntityBaseRepository<Error> _errorsRepository;
        protected IEntityBaseRepository<Movie> _moviesRepository;
        protected IEntityBaseRepository<Rental> _rentalsRepository;
        protected IEntityBaseRepository<Stock> _stocksRepository;
        protected IEntityBaseRepository<Customer> _customersRepository;
        protected IUnitOfWork _unitOfWork;

        private HttpRequestMessage RequestMessage;

        public ApiControllerBaseExtended(IDataRepositoryFactory dataRepositoryFactory,
IUnitOfWork unitOfWork)
        {
            _dataRepositoryFactory = dataRepositoryFactory;
            _unitOfWork = unitOfWork;
        }
        private void LogError(Exception ex)
        {
            try
            {
                Error _error = new Error()
                {
                    Message = ex.Message,
                    StackTrace = ex.StackTrace,
                    DateCreated = DateTime.Now
                };

                _errorsRepository.Add(_error);
                _unitOfWork.Commit();
            }
            catch { }
        }
    }
```

This base class holds references for all types of Data repositories that your application may need. The most important variable is the _requiredRepositories which eventually will hold the types of Data repositories a Web API action may require. Its constructor has only two dependencies one of type IDataRepositoryFactory and another of IUnitOfWork. The first one is required to resolve the data repositories using the new extension method and the other is the one for committing database changes. Now let's see the **esense** of this base class, the extended CreateHttpResponse method. Add the following methods in to the new base class as well.

```
 protected HttpResponseMessage CreateHttpResponse(HttpRequestMessage request, List<Type>
 repos, Func<HttpResponseMessage> function)
        {
            HttpResponseMessage response = null;

            try
            {
                RequestMessage = request;
                InitRepositories(repos);
                response = function.Invoke();
            }
            catch (DbUpdateException ex)
            {
                LogError(ex);
                response = request.CreateResponse(HttpStatusCode.BadRequest,
 ex.InnerException.Message);
            }
            catch (Exception ex)
            {
                LogError(ex);
                response = request.CreateResponse(HttpStatusCode.InternalServerError,
 ex.Message);
            }

            return response;
        }

        private void InitRepositories(List<Type> entities)
        {
            _errorsRepository =
_dataRepositoryFactory.GetDataRepository<Error>(RequestMessage);

            if (entities.Any(e => e.FullName == typeof(Movie).FullName))
            {
                _moviesRepository =
_dataRepositoryFactory.GetDataRepository<Movie>(RequestMessage);
            }

            if (entities.Any(e => e.FullName == typeof(Rental).FullName))
            {
                _rentalsRepository =
_dataRepositoryFactory.GetDataRepository<Rental>(RequestMessage);
            }

            if (entities.Any(e => e.FullName == typeof(Customer).FullName))
            {
                _customersRepository =
_dataRepositoryFactory.GetDataRepository<Customer>(RequestMessage);
            }

            if (entities.Any(e => e.FullName == typeof(Stock).FullName))
            {
                _stocksRepository =
_dataRepositoryFactory.GetDataRepository<Stock>(RequestMessage);
            }

            if (entities.Any(e => e.FullName == typeof(User).FullName))
            {
                _stocksRepository =
_dataRepositoryFactory.GetDataRepository<Stock>(RequestMessage);
            }
        }
```

This method is almost the same as the relative method in the ApiControllerBase class, except that is accepts an extra parameter of type List<Type>. This list will be used to initialized any repositories the caller action requires using the private InitRepositories method. Now let's see how the RentalsController Web API controller could be written using this new base class. I created a new RentalsExtendedController class so that you don't have to change the one you created before. Let's the the new definition. Ready?

RentalsExtendedController.cs

```
    [Authorize(Roles = "Admin")]
    [RoutePrefix("api/rentalsextended")]
    public class RentalsExtendedController : ApiControllerBaseExtended
    {

        public RentalsExtendedController(IDataRepositoryFactory dataRepositoryFactory,
IUnitOfWork unitOfWork)
            : base(dataRepositoryFactory, unitOfWork) { }
    }
```

Yup, that's it, no kidding. Now let's re-write the Rent action that rents a specific movie to a customer. This method requires 3 data repositories of types Customer, Stock and Rental.

Rent action re-written

```
        [HttpPost]
        [Route("rent/{customerId:int}/{stockId:int}")]
        public HttpResponseMessage Rent(HttpRequestMessage request, int customerId, int
stockId)
        {
            _requiredRepositories = new List<Type>() { typeof(Customer), typeof(Stock),
typeof(Rental) };

            return CreateHttpResponse(request, _requiredRepositories, () =>
            {
                HttpResponseMessage response = null;

                var customer = _customersRepository.GetSingle(customerId);
                var stock = _stocksRepository.GetSingle(stockId);

                if (customer == null || stock == null)
                {
                    response = request.CreateErrorResponse(HttpStatusCode.NotFound,
"Invalid Customer or Stock");
                }
                else
                {
                    if (stock.IsAvailable)
                    {
                        Rental _rental = new Rental()
                        {
                            CustomerId = customerId,
                            StockId = stockId,
                            RentalDate = DateTime.Now,
                            Status = "Borrowed"
                        };

                        _rentalsRepository.Add(_rental);

                        stock.IsAvailable = false;

                        _unitOfWork.Commit();

                        RentalViewModel rentalVm = Mapper.Map<Rental,
RentalViewModel>(_rental);

                        response =
request.CreateResponse<RentalViewModel>(HttpStatusCode.Created, rentalVm);
                    }
                    else
                        response =
request.CreateErrorResponse(HttpStatusCode.BadRequest, "Selected stock is not available
anymore");
                }

                return response;
            });
        }
```

The only thing an action needs to do is initialize the list of data repositories needs to perform. If you download the source code of this application from my Github account you will find that I have re-write the RentalsController and the MoviesController classes with the extended base class. It is up to which of the base classes you prefer to use.

Conclusion

That's it, we finally finished this book having described step by step how to build a **Single Page Application** using Web API and AngularJS. Let me remind you that this book has an **online** version as well which you can read here. You can download the source code for this project we built here where you will also find instructions on how to run the *HomeCinema* application. I hope you enjoyed this book as much I did writing it. Please take a moment and post your review at the comments section of the online version.

Connect with chsakell's Blog:

This book is a genuine contribution from chsakell's Blog to its readers.

It's not your skills that make you special but the way you share them

C.S.